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

Student Thesis No. 2394

Integration of Different Aspects of Multi-Tenancy in an Open Source Enterprise Service Bus

Santiago Gómez Sáez

Course of Study: Computer Science

Examiner:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Supervisor:	Dr. Vasilios Andrikopoulos
	Steve Strauch
Commenced:	October 1, 2012
Completed:	November 14, 2012
CR-Classification:	C.2.4, D.2.11, H.3.4, C.2.3

Abstract

The EU project 4CaaSt aims to create an advance PaaS Cloud platform which supports the optimized and elastic hosting of composite Internet-scale multi-tier applications. Cloud computing is essentially changing the way services are built, provided and consumed.

Nowadays applications are composed out of multiple reusable services consisting of newly developed services as well as legacy applications made available as services. These services do not necessarily use the same protocols for communication. So a component for the mediation between various protocols, dynamic service selection and routing based on non-functional requirements is needed. Nowadays an Enterprise Service Bus (ESB) is used in Service-Oriented Architectures (SOAs) to serve precisely these objectives. One important aspect of bringing an ESB as building block into the Cloud is to enable multi-tenancy. This includes multi-tenant aware management and administration of the ESB as well as multi-tenant aware messaging.

In this student thesis we design and implement the extensions of the ESB and the components needed for the integration and evaluation of two approaches to extend an open source ESB for multi-tenancy support: the first covers the multi-tenant aware administration and management and the second covers the multi-tenant aware messaging. Both approaches require the extension of the ESB, which implements the Java Business Integration (JBI). As a result, we provide an integrated prototype based on a scenario emerged from the EU project 4CaaSt and a performance's evaluation of the extended JBI Components in the ESB.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Cloud computing is a recent pattern which describes a new consumption and delivery model for IT Services based on the Internet. It provides dynamically on demand virtualized resources as a service to customers through Web interfaces and standardized protocols with the minimum management effort or service provider interaction. Computer resources (e.g., servers, storage, network) within a Cloud infrastructure are accessed and used by many users. Therefore, functional requirements should be fulfilled at the provider's side in terms of management and usage.

The EU Project 4CaaSt [4Ca] aims to create an advanced Platform-as-a-Service (PaaS) Cloud platform which supports the optimized and elastic hosting of composite internet-scale multi-tier applications. The services supported by these applications should be able to support different protocols. For this mediation between protocols an Enterprise Service Bus (ESB) is used in Service-Oriented Architecture (SOA). However, one requirement must be fulfilled when bringing the ESB into the Cloud: multi-tenancy awareness. The ESB should be modified to be multi-tenant aware in terms of management and messaging.

1.1 Problem Statement

A Cloud infrastructure must serve multiple tenants simultaneously by virtualizing its resources. This requirement allows a Cloud provider to maximize its infrastructure's utilization, lower its infrastructure costs and provide an appropriate Cloud usage market-price to the tenants. The virtualization of resources means that the physical resources will be shared by different tenants. This requires a multi-tenant aware infrastructure which has to be able to isolate tenant data, communication, and computing. The communication to and from the Cloud infrastructure must support different protocols. For this challenge we use an Enterprise Service Bus, which was extended following two main approaches: management and administration, and multi-tenant aware messaging.

Both of the approaches are outputs of the diploma thesis "Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support Focusing on Administration and Management" [Muh12] and master's thesis "Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support" [Ess11].

This student thesis has the goal, in the first place, to design and implement the needed components to integrate the above developed approaches. The management and administration approach [Muh12] specifies and implements a multi-tenant management and administration system for Apache ServiceMix 4.3.0 [ASM]. It provides tenant users interfaces for deploying

and undeploying service assemblies, as well as management and administration artifacts for managing and administrating the tenants and users utilizing the system. The multi-tenant aware messaging approach [Ess11] aims to extend the Apache ServiceMix 4.3.0 [ASM] ESB to provide multi-tenant support with respect to communication to allow its use as a Cloud computing application. This multi-tenant support entails ensuring tenant isolation in the communication to and from the ESB by extending the ESB's JBI components in order to be able to evaluate its functionality based on a scenario originated from the European project 4CaaSt [4Ca].

In the second place, this student thesis must present a testing and evaluation of the extended multi-tenant aware version of the Apache ServiceMix ESB 4.3.0 [ASM]. The separated approaches' implementations have been tested with a low number of tenants and a low message consumption ratio. The goal of the ESB performance's evaluation is to design and implement a benchmark which will provide us with the needed load configurations for running different testing scenarios. The output from these scenarios will be analytical data which will allow us to evaluate the behavior and the performance of the ESB and the system in different scenarios.

In the third place, we must integrate the extended ESB and the management and administration Application JBIMulti2 [Muh12] with the components that build up the Taxi Scenario of the European project 4CaaSt [4Ca] to provide and evaluate an integrated prototype of the taxi application.

1.2 Motivating Scenario

In this section we describe the motivating scenario which will be used for the verification of the outcome of this thesis. The motivating scenario is an outcome from the EU project 4CaaST [4Ca]. It is based on a taxi booking service implemented in a taxi application.

Taxi customers can book a taxi for a transportation from one location to another. In the scenario, each of the taxi companies are tenants and each of the customers are users. The taxi customer contacts the taxi company of their preference to book a taxi and provides the pick up and drop off location (1). The contacted tenant forwards then the taxi request with the information provided by the customer and the customer reply contact preference utilizing one of the following three messaging protocols: Simple Object Access Protocol (SOAP) over HTTP, Java Messaging Service JMS or E-Mail. The taxi request is forwarded through the multi-tenant ESB (2) to the Taxi Service Provider (3), which locates taxi drivers in the area and contacts the corresponding taxi drivers. Once the transport request is accepted by one of the taxi drivers, the confirmation is sent back and a confirmation of the taxi service is sent to the customer (4) (see Figure 1.1).

The taxi service provider activities are implemented as a BPEL process which orchestrates the taxi scenario activities. The BPEL process invokes two external systems through the ESB to obtain the following information: the Context-Management Framework [CCA] provides information about the location of the taxi drivers and their contact details. When receiving

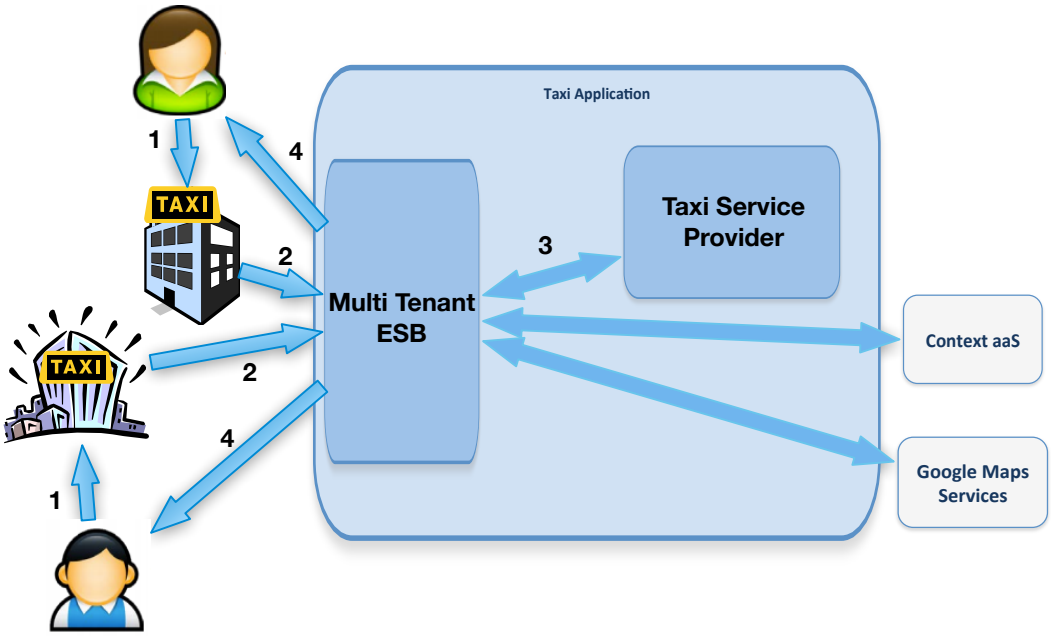

Figure 1.1: Taxi Scenario from the EU 4CaaSt project [4Ca]

the taxi driver location information, the distance information between the taxi driver location and the customer location are calculated in the Google Web Services [GMA].

In Hagin's approach [Hag11] the communication between the taxi service provider and the external systems Context-Management Framework [CCA] and Google Maps Services [GMA] is a point to point communication with two binding applications which mediate the communication between the taxi service provider BPEL process and the external services. In the prototype developed in this student thesis, the point to point communication is replaced by a communication through the Apache ServiceMix 4.3.0 [ASM] multi-tenant aware ESB version. The communication between the these components is not multi-tenant aware, so one of the main goals of this ESB extended version is to provide backward compatibility support for non multi-tenant communication. Both multi-tenant and non multi-tenant connections between the components in the taxi application require the specification and development of JBI components which will be described and evaluated in this thesis.

1.3 Definitions and Conventions

In the following section we will list the definitions and the abbreviations used in this student thesis for understanding the description of the work. These will be used in the document.

Definitions

List of Abbreviations

The following list contains abbreviations used in this document.

Axis2	Apache eXtensible Interaction System v. 2
BC	Binding Component
BPEL	Business Process Execution Language 2.0
EAI	Enterprise Application Integration
EAR	Enterprise Archive
ESB	Enterprise Service Bus
IaaS	Infrastructure-as-a-Service
JBI	Java Business Integration
JDK	Java Development Kit
JMS	Java Message Service
JMX	Java Management Extensions
JVM	Java Virtual Machine
NIST	National Institute of Standards and Technology
NMR	Normalized Message Router
OSGi	Open Services Gateway initiative *(deprecated)*
PaaS	Platform-as-a-Service
POJO	Plain Old Java Object
QoS	Quality of Service
SaaS	Software-as-a-Service
SE	Service Engine
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol *(deprecated)*
UUID	Universally Unique Identifier
WS*	Web Services (Specifications)
WSDL	Web Services Description Language
XML	eXtensible Markup Language

HTTP	Hypertext Transfer Protocol
DCE	Distributed Computing Environment
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
MOM	Message-Oriented Middleware
NM	Normalized Message
NMF	Normalized Message Format
SU	Service Unit
SA	Service Assembly
MEP	Message Exchange Patterns
URI	Uniform Resource Identifier
GAE	Google App Engine
SMPP	Short Message Peer-to-Peer
SNMP	Simple Network Management Protocols
POM	Project Object Model
WAR	Web Application Archive
LRU	Least Recently Used

1.4 Outline

In this section we will list and describe the main points described in this student thesis.

- **Fundamentals, Chapter 2:** we will give the necessary background on the different concepts and technologies used for the realization of this student thesis.
- **Related Works, Chapter 3:** different approaches in the proposed problem have been taken. We will discuss and compare them with the approaches we will take into account.
- **Concept and Specification, Chapter 4:** functional and non-functional requirements will be discussed in this section, for both integration and evaluation analysis.
- **Design, Chapter 5:** we will give a detailed overview of the components which should be modified and the components which should be integrated in order to meet the requirements specified in this student thesis.
- **Implementation, Chapter 6:** the implemented components, as well as the necessary extensions or changes will be detailed in this section from the point of view of coding and configuration.

- **Test, Chapter 7:** in this chapter we will test the final prototype based on the scenario described in this document.
- **Evaluation, Chapter 8:** will perform an evaluation of the extension based on different load scenarios to measure the impact of the outcome of this student thesis on the performance.
- **Outcome and Future Work, Chapter 9:** we will give a conclusion of the developed work and will give some ideas on which further issues can be taken into account in this topic to extend the outcome of this thesis.

2 Fundamentals

In this chapter we give an explanation about the technologies and concepts to provide the background information this student thesis relies on. We start describing the main architecture this student thesis relies on: SOA and its fundamental component, the ESB, and give a review in the patterns followed and in the technologies used.

2.1 Service-Oriented Architecture

The Open Group defines the Service-Oriented Architecture (SOA) as "an architectural style that supports service orientation. Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services" [OPG06].

In the last years enterprises had the need to adapt the SOA architectural paradigm to the their existing IT infrastructure and services to be able to respond to the economic and technological growth they needed. Their distributed system infrastructure was not capable of communicating with external software components wanted to be exposed as a service without a previous agreement in messaging protocols, data types and encoding, and middleware. The same problem could be found in software services inside the company which communicate through its intranet. SOA provides the needed flexibility by building an architectural style with the following benefits: loose coupling, interoperability, efficiency, and standardization. The W3C group defines SOA as a form of distributed system architecture that is typically characterized by the following properties [w3c04]:

- Logical view: the service is defined in terms of what is does and not in terms of how it works.
- Message orientation: the internal structure of an agent is abstracted.
- Description orientation: a service is described by machine-processable meta data.
- Granularity: services tend to use a small number of operations with relatively large and complex messages.
- Network orientation: Services tend to be oriented toward use over a network.
- Platform neutral: Messages are sent in a platform-neutral, standardized format delivered through the interfaces.

The Service Oriented Architecture SOA defines three main roles: requester, provider and broker and the four main operations: publish, find, bind, and invoke. The service provider provides access to services, creates a description of a service and publishes it to the service broker. The service requestor discovers a service by searching through the service descriptions located in the service broker. When the service which best fits to his needs is found, the discovering facility provides the concrete service endpoint and the consumer is responsible for binding to it. With this information, the requestor can then bind to the concrete service and finally execute a business activity [WCL+05].The service broker is responsible for hosting a registry of services descriptions and for linking a requestor to a service provider.

The main component in a SOA is the ESB. The functionalities provided by a service bus can simplify the process (publication, discovery, binding, and invocation) and make it more transparent to provide an ease-to-use experience for a Web service based implementation of SOA [WCL+05]. Chappel defines its function as an intermediate connection provisioning of service providers with service consumers and thereby ensure decoupling of theses [Cha04].

2.1.1 Enterprise Service Bus

The lack of an unique standardized communication middleware between enterprise applications forced numerous distributed computing models (e.g. Distributed Computing Environment (DCE), Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM), Message-Oriented Middleware (MOM), Enterprise Application Integration (EAI), J2EE, Web Services, .NET) to appear. However, some of them have tightly coupled interfaces between applications and services or have high investments costs, and have in the past accomplished less than a 10% of connected enterprise applications [Cha04]. Furthermore, with the increase of the number of applications utilizing a point-to-point connection between them, an increase in the probability of a service failure increased, being each of the applications a single point of failure. There was a need of an integration environment with minimal (or any) integration efforts, which fulfilled SOA.

The ESB is the central component in SOA. It provides a loosely coupled, event-driven SOA with a highly distributed universe of named routing destinations across a multi-protocol message bus [Cha04]. An ESB provides an abstract decoupling between connected applications by creating logical endpoints which are exposed as services and conform a multi-protocol environment, where routing and data transformation are transparent to the service connected to it. Furthermore, when using a ESB, in the first place, services are configured rather than coded, demanding minimal adaptation, implementation and maintenance efforts. The programmer just has to implement the binding to the logical endpoint exposed as a service. In the second place, ESB routing is based on a reliable messaging router. Applications don't need to include message system-failure forwarding mechanisms, to know which data formats are needed in the consumed services, or to care about future changes in applications or services the applications interact with. An ESB hides the complexity of orchestration between services in business processes.

Chappel defines the combination of loosely coupled interfaces and asynchronous interactions as a key concept of the bus terminology [Cha04]. An user of the bus can access every service registered in the bus. For this purpose, it implements the SOA operations in order to make them transparent to the user who can therefore focus on: plugging to the bus and posting and receiving data from the bus.

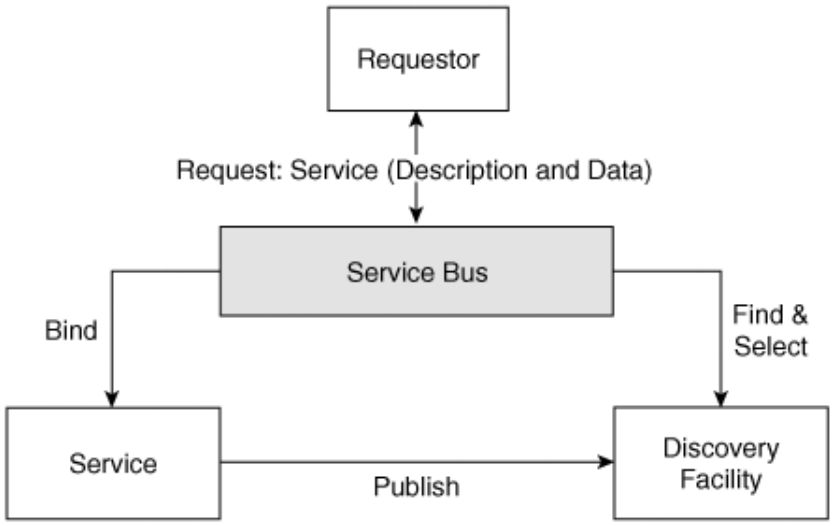

Figure 2.1: The Role of the service bus in SOA [WCL+05]

When receiving a service description (Web Services Description Language (WSDL)) and data from the service requester, the bus (see Figure 2.1) is responsible for selecting the service which best fits to the description requirements, for binding the service requester with the backend service through a route created between the logical endpoints and for making the necessary data transformations to make able the communication between the parts.

One of the challenges in this student thesis is to change developed approach delivered in the first prototype of the taxi scenario [4Ca]. In this the connections between the components of the taxi scenario are point-to-point connection [Hag11]. Using an ESB as mediating component between each of the taxi scenario components eliminates the point-to-point connection and indeed the single point of failure risk of a point-to-point connection. Furthermore, the supported transport protocols in the multi-tenant aware logical endpoints in the ESB are increased in order to be able to provide the taxi scenario as a multi protocol service.

2.2 Cloud Computing

In the last years the term Cloud computing has been widely used in the IT sector. The National Institute of Standards and Technology (NIST) defines Cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable

computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction" [NIS11]. The Cloud computing model is composed of five characteristics:

1. On-demand self-service: a cloud consumer consumes the cloud provider's computing capabilities automatically without the need of human interaction.

2. Broad network access: computing capabilities are available in the network and can be accessed using standard mechanisms.

3. Resource pooling: computing capabilities in the cloud provider side are virtualized to serve multiple consumers simultaneously using a multi-tenant model. The cloud consumer generally has no sense of the provided resources.

4. Rapid Elasticity: computing and storage resources can be dynamically (and in some cases automatically) provisioned and released to respond to the actual consumers' demand.

5. Measured Service: resources' usage is monitored and measured in a transparent way to the cloud consumer and provider for control and optimization purposes.

The control that the cloud consumer has over the computer resources in a cloud provider infrastructure is defined in three service models: *Software-as-a-Service (SaaS)*, *Platform-as-a-Service (PaaS)* and *Infrastructure-as-a-Service (IaaS)*. *SaaS* provides to the cloud consumer access and usage of cloud provider's applications running on a cloud infrastructure. The consumer has no control over the underlying infrastructure where the application he uses is deployed. The customer can only control individual application's configurations during his usage of it. *PaaS* provides the customer with the needed capabilities to deploy applications which's programming language, required libraries, services and tools are supported by the provider. The consumer has no control over the underlying infrastructure where he deploys the application. *IaaS* is the model which gives most control to the consumer. Thus, the consumer is able to deploy and run arbitrary software and has the control over operating systems, storage and deployed applications, but has no management or control on the underlaying cloud infrastructure.

The NIST defines four deployment models in Cloud computing. A private cloud consists in a cloud infrastructure which is provisioned exclusively for one organization and used by the members conforming the organization. It is comparable to processing facilities that are enhanced with the Cloud computing characteristics. A community cloud is a cloud infrastructure where its use is limited to organizations which share the same requirements. A public cloud infrastructure can be accessed and used by the public. It is usually offered by cloud service providers that sell cloud services made for general public or enterprises. Some of the cloud consumers may process and store information which requires more control over the infrastructure in which is located, or consume public Cloud computing resources during peak loads in their private cloud infrastructure. The hybrid cloud model combines two or more deployment models described above and the combination remains as a unique entity.

Cloud computing and SOA are related models at an architectural, solution and service level, according to IBM [OPG11]. Cloud providers expose their Cloud infrastructure as services as part of a SOA solutions and the communication between Clouds in the Hybrid Cloud model described above can be compared to a SOA communication solution between enterprises. Cloud services are services that can be accessed by the cloud consumers. Therefore, we can deduce that the SOA model can be applied in the Cloud computing approach and the need of the ESB in a Cloud computing infrastructure is evident.

2.3 Multi-tenancy

In a Cloud computing environment where the SaaS is offered, the cloud consumer aims to lower its business costs. Using a SaaS solution benefits the customer in the following fields [CC06]:

- Software license is not acquired directly by the cloud consumer, but is owned by the cloud service provider and sold to the customer as a subscription to the software running in the cloud infrastructure.
- More than half of the IT Investments of a company are made in infrastructure and its maintenance. In a SaaS solution this responsibilities are mainly externalized to the cloud provider.
- A Cloud computing environment is based in the utilization of its resources simultaneously by a large number of cloud consumers. For example, a cloud provider that offers a centrally-hosted software service to a large number of customers can serve all of them in a consolidated environment and lower the customer software subscription costs while maintaining or lowering the provider's infrastructure, administration and maintenance costs.
- The cost leverage in the software utilization allows the cloud providers to focus not only on big enterprises capable of large IT budgets, but also on the small business that need access to IT solutions.

The SaaS provider's cost per customer can be maintained or lowered by designing its applications to be multi-tenant aware. With this approach, the provider can grant software which serves multiple customers concurrently while obtaining a better utilization of its resources by reducing no-load running and sharing code base and data. Chong and Carraro [CC06] defines a well designed SaaS application as scalable, multi-tenant-efficient and configurable. With this design patterns, the SaaS model enables the provider to *catch the long tail*. The business software tends to demand the software vendor to give an individual attention to its software customer, and in most of the scenarios an improvement in the customer's infrastructure. This fact leads to have a low reduction interval of the price at the software provider's side, and in consequence a limitation in software access from the software consumer's side. However, if the previous requirements are eliminated and the provider's infrastructure is scaled to combine and centralize customers' hardware and services requirements, the price reduction

limit can be decreased and, in effect, allow a wide range of consumers to be able to access this services.

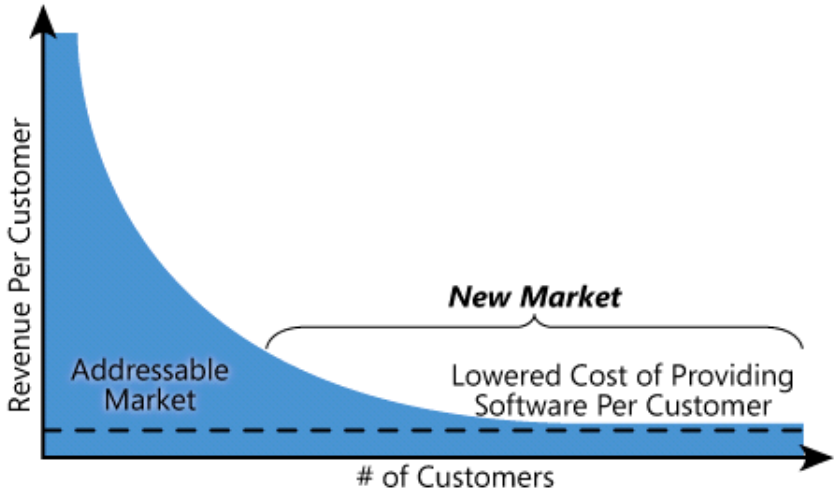

Figure 2.2: New market opened by lower cost of SaaS [CC06]

In the Figure 2.2 the economics of scaling up to a high number of customers while reducing the software price is analyzed. This enables the Cloud computing providers to target a new market formed by software consumers (e.g. small and medium businesses, individual consumers, etc.) that are not able to afford the costing expenses of acquiring individual software.

Multi-tenancy refers to the sharing of the whole technological stack (hardware, operating system, middleware, and application instances) at the same time by different tenants and their corresponding users [SAS+12]. There are several approaches to obtain multi-tenancy in a shared infrastructure.

Chong et al. focuses in multi-tenant data architecture for achieving multi-tenancy in a SaaS solution [FC06]. One of the most important requirement for an architect when creating a SaaS is to grant customers with an optimized data isolation while maintaining the administration and management of the provider's data infrastructure cost-effective. Their model defines an isolation range where the isolation degree depends on how the tenant's data is allocated in their database system. The most simple approach assigns one database per tenant. However, its maintenance cost increases with the number of tenants. The second approach proposed is the allocation of tenant's data in a shared database but in different database schemas. More tenants can be supported in this model while providing a moderate degree of data isolation. The most cost-effective approach is reached by allocating tenant data in shared databases and schemas. In this, the largest number of tenants can be served but an extra security implementation is needed in order to ensure that a tenant cannot access other tenant's data.

Nevertheless, there are more important challenges when deploying multi-tenant aware SaaS solutions which are compositions of multiple services in a SOA environment. Individual SOA services may differ in configuration aspects as well as in the number of tenants sharing the

same service instance. Mietzner et al. classifies services in three instance types: *single instance*, *arbitraty instance* and *multiple instance* [MUTL09]. These differ in the number of instances of the service deployed per tenant and the configurability of the deployed service, as well as the possibility to mix single and multiple instances for fullfilling, for example, a Quality of Service (QoS) pattern. These different types of services should be able to communicate with each other by introducing special integration patterns between them (e.g ESB with multi-tenancy awareness).

In a Cloud computing environment offering a PaaS solution, both of the approaches should be taken into account. The first one relies on the requirements which cover the administration and management concerns in the data stored by the tenants' deployed applications while the second one relies on the mediation needed in the communication between the services. Using an ESB as the mediator of the tenant aware communication profits tenants by eliminating the need of a middleware investment for communicating with their deployed application.

2.4 Java Business Integration

When integrating with external enterprises' applications, the lack of standardized technologies triggered to the use of vendor or self-implemented technologies. The solution to this problem is reached in the Java Business Integration (JBI) defined by the Java Community by maximizing the decoupling between components and defining an interoperation semantic founded on standards-based messaging. This allows components developed by different vendors to interoperate in a multivendor "echosystem" [JBI05].

The integration system is made up of plug-in components which communicate using a standardized internal message exchange format. The decoupling between the components in a JBI environment is obtained by a standardized message format usage and by the usage of a mediator instead of a direct connection between the components. The communication mediator between components in a JBI environment is the Normalized Message Router (NMR). Its main functionality is the routing of the internal standardized Normalized Message (NM) between the components. However, it can perform additional processing during the message exchange. The NMR fields are defined as an eXtensible Markup Language (XML) document format payload, metadata conforming the header and a non XML document format attachment referenced by the payload.

The JBI specification defines two different types of components which are categorized in two types:

- A Service Engine (SE) provides transformation and composition services to other components.
- A BC provides the connectivity between the external services and the JBI environment. They support many different types of protocols and isolate the JBI environment by marshaling and demarshaling the incoming or outgoing message into the internal standardized NM format.

Both components listed above can function as service consumers or service providers following a WSDL-based, service-oriented model. The consumer endpoint provides a service accessible through an endpoint which can be consumed by other components, while the provider endpoint consumes a functionality exposed as a service and accessible through an external endpoint. The routing of NM starts when a message exchange between components is created (bidirectional communication pipe, a *DeliveryChannel*, between the communicating endpoints) and continues with the target of the specified service endpoint for processing (see Figure 2.3). The NMR supports four asynchronous message exchange patterns differing in the reliability and direction of the communication.

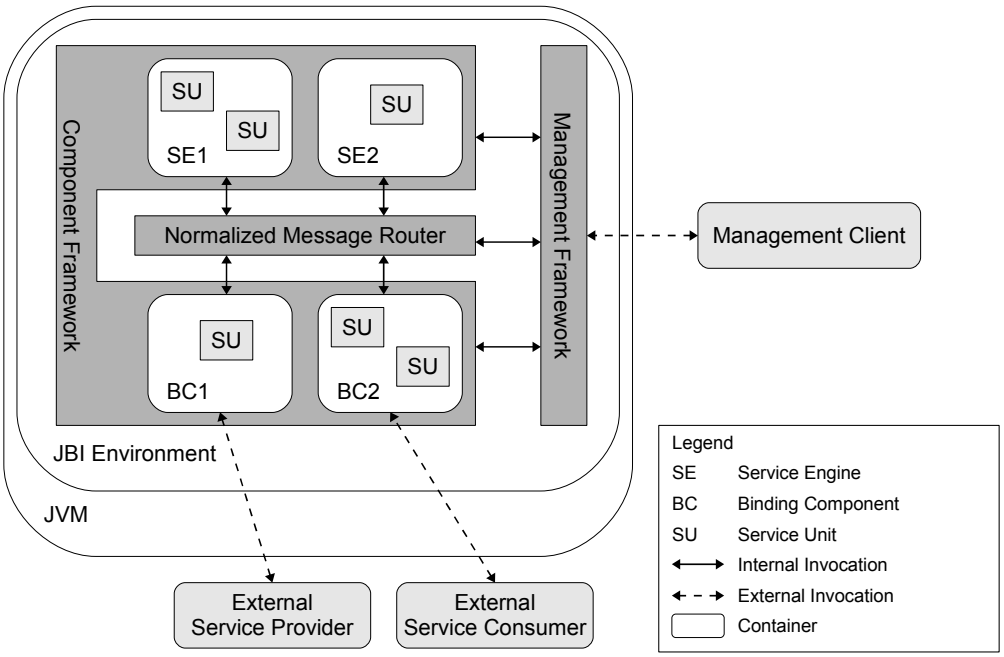

Figure 2.3: Overview of JBI Architecture. Figure 4 in JBI specification document [JBI05].

In Figure 2.3 we can observe that one or more Service Unit (SU) are contained in a BC. The SUs are component-specific artifacts to be installed to a SE or a BC [JBI05]. The service units are packed in a Service Assembly (SA), usually as ZIP files, where it is specified each of the components where each of the SUs should be deployed. The JBI environment provides a Java Management Extension Java Management Extensions (JMX) Framework for installation, life cycle management, addition, and monitoring and control of the components conforming to the environment defined by the JBI specification.

2.5 OSGi Framework

The OSGi defines a framework for deployment support in a Java Virtual Machine (JVM) of downloaded or extended applications known as *bundles*. This framework requires OSGi-friendly devices a minimum system's resources usage by providing dynamic code-loading

and *bundle* lifecycle management. An OSGi *bundle* is the packaging of a group of Java classes and required and provided capabilities' meta-data as a JAR file for providing functionality to end users. OSGi *bundles* can be downloaded, extended and installed remotely or locally in the platform when needed without the need of system reboot. Installation and update of bundles during their lifecycle are also managed by the framework, which uses a service registration for selection, update notifications, or registry of new service objects offered by a deployed bundle. This feature is the main key for connecting bundles whose's services require during runtime capabilities provided by another bundles. The framework defines a bundle's requirement capability as a dependency.

The OSGi framework defines 5 different layers and a bundle's lifecycle [OSG11]. An optional Security Layer provides the infrastructure for deploying and managing applications which must be controlled during runtime. The Module Layer lists the rules for package sharing between the deployed bundles. The lifecycle of a bundle can be modified during runtime through an API provided in the lifecycle layer. The main operations implemented are install, update, start, stop or uninstall.

2.6 Apache ServiceMix

In this student thesis we aim to integrate two approaches to extend an Open Source ESB for multi-tenancy support and to evaluate its performance in different configurable loading scenarios prior and after the extension. Essl evaluates individually different ESB solutions in terms of messaging, integration, runtime, security, QoS, multi-tenancy and extensibility. The absence of technical details on multi-tenancy and the different multi-tenant approach implemented in his highest ranked ESB solution, WSO2 ESB, leads to the selection of the Apache Servicemix ESB [Ess11]. Apache ServiceMix is delivered as a non multi-tenant ESB. This feature has been improved separately in the Apache ServiceMix 4.3.0 [ASM] with the implementation of the two approaches that will be integrated in this thesis. In the document we will refer to Apache ServiceMix 4.3.0 as ServiceMix.

ServiceMix allows multi-vendor components and services to be integrated by complying to the JBI specification. It includes a complete JBI container supporting the JBI specifications [ASM]:

- Routing between logical endpoints through a NMR (see Figure 2.4)
- JBI Management MBeans
- JBI deployment support

ServiceMix is an integration container based on the OSGi Framework implementation Apache Karaf [APA11b]. It provides a light environment in which components and applications can be deployed in a loose coupled way. Apache Karaf provides an extensible management command line console where management of the components lifecycle, such us OSGi bundles, JBI components or SAs, can be done in an user friendly way (see Figure 2.4). Furthermore, a hot deployment directory is shipped with the ESB package where users can deploy OSGi bundles,

JBI components wrapped in SA's, etc. just by copying the file into it. The undeployment is done automatically when the user deletes the file from the *deploy* directory.

Reliable messaging is supported by the integration with an Apache ActiveMQ instance out-of-the-box [AMQ] with a configurable broker, which allows incoming and outgoing connections from several binding components and outside the ESB. ServiceMix ships with different JBI components deployed. In this thesis we will concentrate on the following ones: SOAP over HTTP, JMS, E-Mail and Apache Camel. Apache Camel is a powerful open source integration framework based on EAI [APA11a]. The user can configure logical endpoints between BCs and different routing paths between them by deploying their configuration wrapped in a SA in the *deploy* directory. Different Maven plugins can make the configuration of a JBI or SE as simple as possible by providing different built archetypes which generates the SU files and directories where the developer can configure the component [AMV].

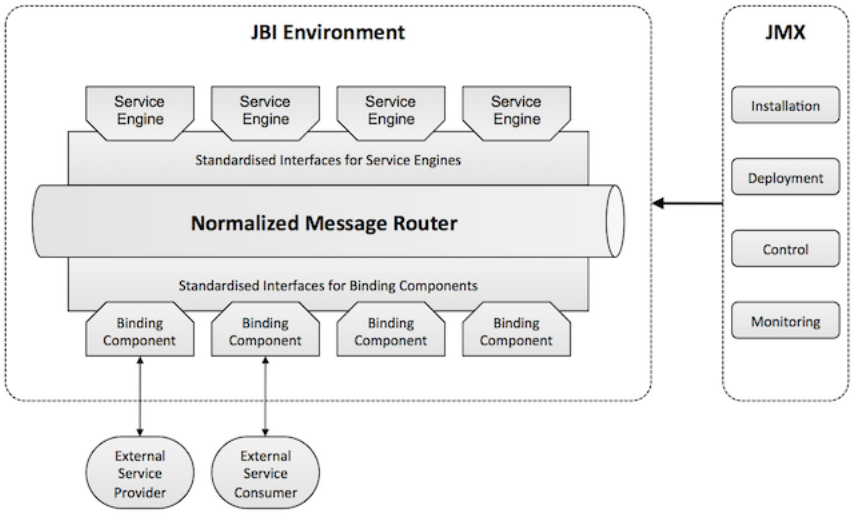

Figure 2.4: Architecture of Apache ServiceMix [ASM]

The NMR routes the messages between the endpoints created by the JBI components (see Figure 2.4). This endpoints are divided in two types: consumers and providers. A consumer endpoint is exposed as a service while a provider endpoint consumes a service. When a message arrives to a consumer endpoint of a JBI component, it is transformed into a Normalized Message Format (NMF). The NMF is the protocol neutral format which is routed in a JBI environment and described in Section 2.4. Muhler has implemented the administration and managing approach in ServiceMix by developing an OSGi bundle which uses management and administration functionalities provided in the extensible management libraries for controlling the lifecycle of the deployed JBI components and has extended the HTTP BC to make it multi-tenant aware [Muh12]. Essl has implemented the communication approach by extending the marshalling and demarshalling in different JBI BCs to include multi-tenancy awareness in the NM. In this thesis we will extend the JMS and E-mail BCs to make them multi-tenant aware and modify approaches taken into account in the marshaling and demarshaling of incoming and outgoing requests. After the extension we will evaluate the impact of our implementation in ServiceMix's performance by extending and using a

configurable load benchmark provided by AndroitLogic [Ltd12].

2.7 Binding Components

In this Section we describe the JBI BCs this student thesis focus on and the transport protocols they support. As specified before, ServiceMix ships with a list of installed BCs. The user can use maven archetypes to configure and create JBI endpoints in the installed JBI BCs. The endpoint configuration files are packed in SUs and in one SA one or more SUs can be packed for deployment.

2.7.1 SOAP over HTTP

ServiceMix ships with a JBI BC named servicemix-http, which supports SOAP over HTTP. Its communication channel is managed by a HTTP server based on Jetty 6 for incoming requests and by a Jakarta Commons HTTP Client for outgoing requests. It provides support for SOAP versions 1.1 and 1.2, and for different Message Exchange Patterns (MEP), as well as several WS* properties.

SOAP defines an XML message format which is sent over the network and a set of rules for processing the SOAP message in the different SOAP nodes which build the message path between two endpoints [WCL+05]. A SOAP message is a composition of three main elements: a SOAP envelope, header and body. A SOAP envelope may contain zero or more header and one body. The header may contain processing or authentication data for the final SOAP node or for the intermediate nodes through the message is routed. The message payload or business data is included in the SOAP body. SOAP is used as a message framework for accessing Web services in loose coupled infrastructures [WCL+05]. The Web service consumer specifies the functionality to invoke in the SOAP body. If the Web service functionality has a request-response MEP, a SOAP message is used to send the response data when the corresponding operation has been executed successfully or the error data in case an error occurred during execution.

SOAP messages can be sent using different network protocols. In this student thesis we use the standardized HTTP protocol.

2.7.2 JMS

The usage of enterprise messaging products allows the integration of loosely coupled components and their combination to build a reliable and flexible system. The JMS specification defines interfaces and semantics which allow clients to access an enterprise messaging product and to reduce the complexity a developer must learn when using them [HBS+02].

Hapner et al. define a JMS provider as an entity which implements JMS for a messaging product [HBS+02]. JMS defines two main roles and two domains in a message exchange

operation, which supports both synchronous and asynchronous communication. In a pont-to-point approach, the message provider deposits a message in a queue. The queue is persistent, this means that it stores the message until the message consumer reads successfully the message from it. We can mostly compare the publish/subscribe approach with the multi-cast protocol in a network, where one provider sends one message to multiple receivers. In this approach one provider publishes a message in a topic to which several consumers are subscribed. When the message is published, it is stored in the topic until all of the receivers have received it. The message format separates the message in two main parts: header and body. Users can set custom headers and the the message supports SOAP 1.1 and 1.2 in its body, despite we send XML format in it.

In this student thesis we use the ActiveMQ 5.4.2 JMS provider which is integrated with ServiceMix [ASM]. The Apache Karaf console provides several commands for administration, management and view on the queues managed by the JMS broker [ASM]. In Chapter 6 we explain in detail the JMS configuration patterns in ServiceMix.

2.7.3 E-Mail

The internet electronic mail transport is a widely well known and used specification. The Internet Message Format structures an e-mail message in two main parts: header and body [Gro08]. The header carries information about the user and receiver (e-mail addresses and signature). The user can also set custom headers with string values. The body carries the message the user wants to send, which is a US-ASCII text represented format. In this student thesis we use XML format in both custom headers and message body.

ServiceMix ships with a non multi-tenant aware mail BC, which supports the following email protocols: POP and IMAP for incoming e-mails, and SMTP for outgoing e-mails. It defines two endpoint types: mail poller and mail sender. The mail poller is the consumer endpoint. It polls every time interval (usually configured by the user) for incoming e-mails in the provided e-mail account and transforms the e-mail into a NMF. The mail BC demarshals the NM into an e-mail and sends it to the provided e-mail addresses.

2.8 Service Engine

As described previously, SE is a JBI component that consumes or supplies services within a JBI container. They provide different kinds of services, e.g. business logic, routing, and message transformation. In ServiceMix we will mainly concentrate on one: Apache Camel [APA11a], which will be described in the following point.

2.8.1 Apache Camel

Apache Camel lets you create the Enterprise Integration Patterns to implement routing and mediation rules in either a Java based Domain Specific Language (or Fluent API), via Spring

based XML Configuration files or via the Scala DSL [APA11a]. The Camel SE packed in a SU is packed as a SA and is shipped with ServiceMix. The routing or mediation rules between two or more endpoints can be specified in an Spring Configuration file or in a Plain Old Java Object (POJO) file whose's class extends the Apache Camel *RouteBuilder* class. The routing patterns supported by Apache Camel are the point-to-point routing and the publish/subscribe model. The configuration in a XML file reduces the configuration complexity to a minimum effort of the developer. However, a configuration in a POJO class increases the developing complexity but allows the developer to provide logic, filtering, dynamic routing, etc. In the *RouteBuilder* class a developer can access, for example, the header of a NM and select the target endpoint dynamically depending on the implemented logic.

The endpoints representation in Apache Camel is Uniform Resource Identifier (URI) based. This allows this SEs to integrate with any messaging transport protocol supported in the ESB, e.g. HTTP, JMS via ActiveMQ, E-Mail, CXF, etc. In this student thesis we implement a multi-tenant aware approach for JMS support based on dynamic routing between tenants. In the Chapters 5 and 6 this approach will be specified.

2.9 JBIMulti2

As mentioned in previous sections, Muhler has implemented a tenant-aware administration and management approach for enabling multi-tenancy awareness in ServiceMix. A multi-tenant system must fulfill several requirements, such as data and performance isolation between tenants and users, authentication, specification of different user roles, resources usage monitoring, etc. In a JBI environment, users deploy SAs (set of SUs containing BCs and/or SE configurations) to a ServiceMix instance in order to create endpoints which are exposed as services. In a multi-tenant JBI environment, we must ensure that the endpoints configured by one tenant by deploying SA are accessible only by the tenant proprietary of those endpoints. Furthermore, control over the deployed artifacts, as well as the different users configured by one tenant, is needed.

The architecture of the JBIMulti2 system is represented in Figure 2.5. We can distinguish two main parts in the system: business logic and resources. JBIMulti2 uses three registries for storing configuration and management data. When a tenant (or a tenant user) is registered, an unique identification number is given to them and stored in the Tenant Registry. Both Tenant Registry and Service Registry are designed for storing data of more than one deployed application. The former for storing tenant information and the latter for providing a dynamic service discovery functionality between the different applications accessed through the ESB. The Configuration Registry is the key of the tenant isolation requirement of the system. Each of the stored tables are indexed by the tenant id value.

The system provides an user interface for accessing the application's business logic. Through the business logic, the management of tenants can be done by the system administrator or the management of tenant's users can be done by the tenants. Furthermore, when deploying the different tenant's endpoint configurations packed in SAs, the system first makes modifications in the zip file for adding tenant context information and then communicates with the Apache

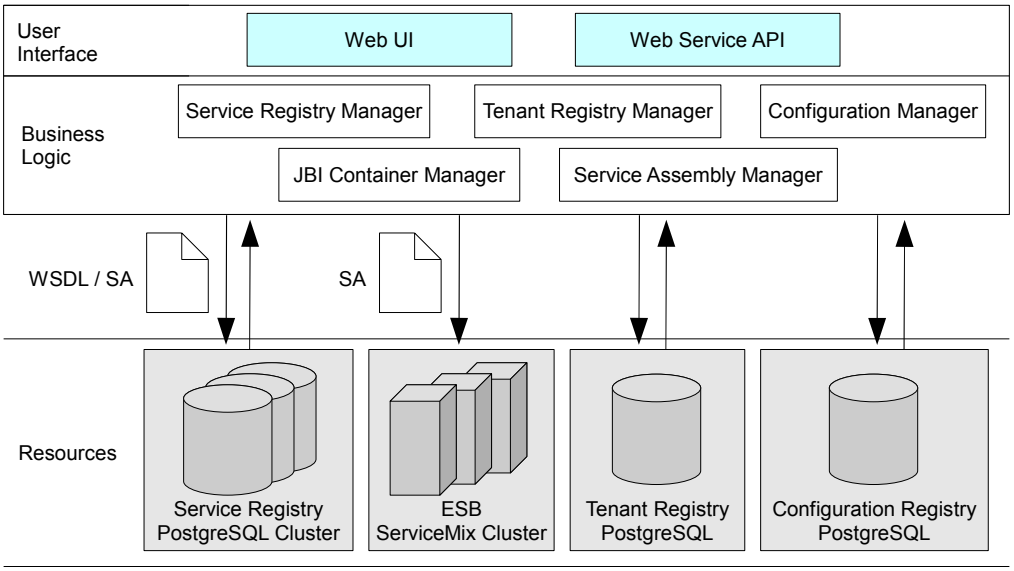

Figure 2.5: JBIMulti2 System Overview [Muh12]

ServiceMix instance by using a JMS Topic to which all the ServiceMix instances are subscribed to. The JMS management service in ServiceMix deploys the received SA injected in the received JMS message using the administration functionalities provided in ServiceMix. The communication between the business layer and the ServiceMix instance is unidirectional. When successful deployment, the endpoint is reachable by the tenant. When an error occur during deployment, an unprocessed management message is posted in a dead letter queue.

JBIMulti2 requires the previous installation of components, e.g. JOnAS server, Apache Tomcat (in version 1.0 of the Taxi Scenario, in version 2.0 Tomcat is depreceated), Postgresql, etc. The initialization of the application is described in both Chapter 7 and in the JBIMulti2 setup document [Muh].

3 Related Works

In this Chapter we discuss which approaches for multi-tenancy awareness have been already implemented in different middleware solutions, as well as ESB evaluation benchmarks implemented and shared by different vendors, and how they differ from the approaches taken into account in this student thesis.

Several multi-tenant approaches have been discussed and implemented for the SaaS model. Chong and Carraro provide a four-level maturity model for complying with scalability, configurability, and multi-tenant efficiency in a SaaS delivery model [CC06]. Furthermore, they present three models for managing multi-tenant data in a SaaS environment [FC06]. Gao et al. perform an analysis identifying the accessible resources which can be possibly shared between tenants. Moreover, they evaluate the isolation patterns and possible customization approaches based on different multi-tenant scenarios [GGW+09]. By implementing multi-tenancy approaches in a SaaS solution, SaaS providers can sell applications as configurable Web services while optimizing their resources costs and usage and *catching the long tail*.

Walraven et al. identify two main disadvantages when designing and implementing multi-tenancy in the application level [WTJ11]. In a multi-tenant SaaS solution the application developer has to ensure data and configuration isolation between the tenants, as well as isolating them from the application provider's management data in the infrastructure. This features add extra complexity when designing and implementing a SaaS solution respect to single-tenant applications which are deployed per individual tenant in a PaaS model. In addition, two multi-tenancy approaches are analyzed: multiple application instances and one single application instance shared between multiple tenants. The former is adapted for supporting up to dozens of tenants with hundreds of users per tenant, while the latter can support a larger number of small tenants [GGW+09]. Each of the tenants demands individual customization requirements which can't be addressed by configuring the application, but having to modify software in it [WTJ11]. However, in a PaaS solution this problem is not reached when each of the tenants deploy their implemented or acquired applications individually.

However, there are few Cloud providers which offer PaaS solutions with multi-tenant awareness for deploying multi-tenant aware applications on top of it [SASL12]. Clients are focusing on the costs savings and vendor alternatives for running their applications in a multi-tenant shared container PaaS. This fact enables running middleware as a service, for example ESB-as-a-service, across multiple tenants [WSO12]. Previous surveys and evaluations have been done on different ESB solutions to analyze its multi-tenancy awareness. Some of the analyzed solutions have been: Apache ServiceMix, Microsoft BizTalk Server, RedHat JBossESB, Mule ESB, OW2 Petals ESB, IBM WebSphere ESB, WSO2 ESB. From the analysis performed, only two ESB solutions provide multi-tenancy awareness: IBM WebSphere ESB and WSO2 ESB

[Ess11]. WSO2 Carbon is an OSGi based platform which is shipped with several OSGi components. The multi-tenancy pattern is reached by developing a SOA middleware on top of the WSO2 Carbon which allows tenants to deploy components whose data, code, and requests are isolated between tenants. The individual configuration and communication of each tenant is managed by the Apache eXtensible Interaction System v. 2 (Axis2) engine, where the individual configuration of each tenant is stored in a structure which cannot be accessed by other tenants. Stored tenant data is not accessed directly by the tenants, but accessed through a multi-tenant aware layer which includes tenant information to the data access operations. IBM WebSphere ESB is shipped with multi-tenancy awareness by ensuring tenant isolation in terms of tenant based identity and access management with the utilization of their proprietary component IBM Tivoli Access Manager [PLW+07]. Both mechanisms applied in the ESB solutions described above are not mechanisms which we can use and extend for other ESB solutions. WSO2 ESB relies the tenant communication and administration on Axis2 and the IBM WebSphere ESB multi-tenancy approach is based on proprietary technologies [SAS+12].

A similar approach to what we have taken for enabling multi-tenancy in ServiceMix is described by Walraven et al. [WTJ11]. They propose a middleware support for tenant-specific configuration for deploying component-based multi-tier applications. Our work differs from the mentioned in the usage of tenant context information for allowing multi-tenant aware management and administration, data and communication isolation. Moreover, our implementation can be applied to any ESB solution which complies with the JBI specification, while Walraven et al. approach implements a multi-tenancy support layer on top of Google App Engine (GAE) which has built-in support for tenant data isolation and is compatible with the used Guice dependency injection framework [Goo].

Several ESB performance evaluations have been done by different vendors and the analytical results made public. WSO2 has evaluated the performance of ESB solutions from different vendors in their Performance Testing - Round 3 [WSO08]. They include an evaluation for ServiceMix v3.2.1 for different scenarios. However, we have discarded the utility of their implemented benchmark because of two main reasons: the benchmark evaluates the performance based on one tenant and multiple users requesting the same endpoint and the ServiceMix version utilized for their evaluation is previous to the one we have extended. Furthermore, they have detected an anomaly in the ESB which didn't forward the incoming SOAPAction to the backend service. AndroitLogic has presented their last ESB performance evaluation over eight different ESB solutions in their Round 6, including ServiceMix 4.3.0, the version we have worked on [Ltd12]. In this they define different scenarios which they use for their performance analysis, e.g. Direct Proxy, the scenario we use for our evaluation. Because of not being able to complete successfully the scenario involving XSL Transformations with ServiceMix 4.3.0, the authors have not published the results for the different scenarios for ServiceMix.

The benchmark used for the AndroitLogic Round 6 is not multi-tenant aware [Ltd12]. It is used for testing different endpoints sequentially, not concurrently. However, it includes the feature of configuring concurrent users invoking the same endpoint. In this thesis we reuse, but adapt and extend, the AndroitLogic Driver in order to make it multi-tenant aware between

different endpoints while maintaining the non multi-tenant awareness for comparing the impact of the modifications we have performed in the ESB solution. In Chapter 8 we discuss the needed extensions and modifications for performing the evaluation of our multi-tenant aware ESB.

4 Concept and Specification

In this chapter we describe the requirements which the integrated prototype should meet in terms of administration and management, and communication of a multi-tenant aware ESB in a PaaS Cloud environment. Some of the requirements have been already met in the implemented approaches in both diploma and master thesis and some are reengineered for optimizing its performance in ServiceMix [Muh12], [Ess11]. After specifying them, we provide an overview of the integrated prototype which we use for testing the taxi scenario [4Ca].

4.1 Requirements

In this section we identify the functional and non-functional requirements that the outcome of this thesis should comply to. The two approaches this thesis integrates identify several functional and non-functional requirements in administration and management, and communication in a multi-tenant aware ESB solution [Muh12], [Ess11]. We fully adhere the former ones and we change, for performance and usability improvement, the latter ones. The outcome of this student thesis should be tested using the taxi scenario [4Ca] described in Section 1.2, and its performance should be evaluated using different scenarios based on the Direct Proxy Service scenario from the AndroitLogic ESB Performance Round 6 [Ltd12]. Both of the scenarios which are used in this thesis require two different integration requirements, which are described in the Sections 4.1.2 and 8.1.1.

4.1.1 Communication Requirements

As described before, ServiceMix is shipped with several JBI BCs which support different communication protocols, but are non multi-tenant aware. In a PaaS model, an ESB solution as its main middleware should fulfill the functional and non-functional requirements which are described in this section. We identify the requirements which are not in the scope of this student thesis and reference the diploma or master thesis where these are fulfilled.

Functional requirements

The extension of ServiceMix for multi-tenancy awareness we implement in this student thesis should conform the following functional requirements:

- **Multi-protocol communication:** ServiceMix is shipped with a set of JBI BCs and SEs which conform a multi-protocol communication platform. The BCs which should support a multi-tenant aware communication must support the following communication protocols: JMS, SOAP over HTTP and E-mail. In addition to the mentioned before, a non multi-tenant aware communication has to be supported in the set of communication protocols which the primitive prototype of ServiceMix supports, e.g. HTTP, JMS, E-mail, Short Message Peer-to-Peer (SMPP), Simple Network Management Protocols (SNMP), etc.

- **Transparent tenant aware endpoint configuration:** a configured JBI endpoint in ServiceMix is reached by two main properties: service name and endpoint name, which are both preceded by the user specified URI namespace. In a multi-tenant environment where the tenants provide the same BCs or SEs, several endpoint properties should be internally changed. This change has to be transparent to the user and the exposed JBI endpoint should be accessible through a service represented by a standardized URI format, which is standardized for the different multi-tenant aware BCs. As described in Section 2.4, the configuration of a BC or a SE is described in a SU, which is packed in a SA. The SA deployed by the different tenants should be isolated in two levels of our system architecture: JBIMulti2 and ServiceMix. In JBIMulti2 this is achieved by indexing the tables, where this packages are stored, with tenant information [Muh12]. In ServiceMix the isolation between the tenants' endpoints configuration must be ensured.

- **Transparent tenant-aware creation and management of resources:** the BCs we extend for multi-tenancy support the following transport protocols: SOAP over HTTP, JMS and E-mail. The services provided in the cited protocols consume different types of resources that a JBI consumer endpoint depends on : a SOAP over HTTP consumer endpoint has to be exposed as a Web service with WSDL description, while a JMS consumer endpoint consumes messages from a created tenant topic or queue and the E-mail consumes resources from an external mail server. The resources created when a tenant endpoint is configured have to ensure isolation between tenants and have to be accessible only by the tenant proprietary of the endpoint.

- **Tenant-aware messaging:** messages sent to and from the multi-access ESB consumer and provider endpoints should be enriched with structured tenant context information. The communication protocols provide different mechanisms for enriching their messages. We analyze in the following chapters the tenant context information and how they can be included in each of the protocols. Furthermore, the NMs exchanged between endpoints belonging to one tenant should also contain tenant information.

- **Tenant-aware routing:** the routing mechanism between two or more tenant's endpoints has to be able to identify the tenant who make the request. The message has to be routed only if the tenant invoking the consumer endpoint authenticates the communication. With this requirement we ensure that one tenant can't get a request routed through another tenant's endpoints, as well preventing the system from malicious attacks.

- **Tenant aware correlation:** In the previous requirement we specified the preconditions which are set for routing a request in message exchange, but we didn't specified the

requirements for the responses. The multi-tenant aware ESB has to be able to correlate both requests and the responses for each of the tenants and ensure that a synchronous or asynchronous response is routed back to the appropriate tenant endpoint.

Non-functional requirements

The extension of ServiceMix for multi-tenancy awareness we implement in this student thesis should conform to the following non-functional requirements:

- **Security:** the extended multi-tenant aware BCs should implement security mechanisms. The tenant context information should be only visible to the tenant and the system, to avoid possible system attacks. The multi-tenant aware BC should be capable of unencrypting the tenants' incoming messages and encrypting the routed outgoing messages from the system to the backend service. Encryption and unencryption are out of the scope of this student thesis. However, we provide each tenant consumer endpoint with a tenant authentication mechanism before creating the NM and the message exchange.
- **Backward Compatibility:** Servicemix is shipped with non multi-tenant aware binding components. Therefore, we need our extended prototype to provide backward compatibility with the original BC configurations and non multi-tenant aware communications.
- **Performance:** the negative impact ServiceMix's performance due to the extension for enabling multi-tenancy awareness in the system should be minimized. Essl proposes the use of tenant context information which requires the retrieval of extra tenant information from a Tenant Registry before creating the message exchange [Ess11]. This implies an independent retrieval of data per request received in each tenant's consumer endpoint. The system should minimize the retrieval of external data to ServiceMix in order to minimize the performance penalty due to implementation of the multi-tenant communication approach.
- **Scalability and Extensibility:** the integrated prototype should offer clustering functionality and scale appropriately in a Cloud infrastructure. JBIMulti2 complies administration and management between more than one instance of ServiceMix [Muh12]. However, the communication to the system composed of two or more instances of ServiceMix should be managed in order to route the messages to a tenant's consumer endpoint located in one specific (or replicated in more than one) instance of ServiceMix. The Horizontal Scalability is out of the scope of this student thesis. This feature is contained in the diploma thesis "Extending an Open Source Enterprise Service Bus for Horizontal Scalability Support" [Fes12]. The integrated prototype should be upgradable and for this goal the decoupling of components have to facilitate changes in functionality.
- **Dynamic Service Discovery:** a multi-tenant aware ESB in a Cloud environment must provide dynamic discovery of the services the tenants provide. For this purpose, the service broker can search the services which best fits for the consumer policy

requirements. This functionality is out of the scope of this student thesis, but being implemented in the master thesis "Extending an Open Source Enterprise Service Bus for Dynamic Discovery and Selection of Cloud Data Hosting Solutions based on WS-Policy" [Ura12].

- **Maintainability and Documentation:** the source code provided in this student thesis should be well commented and documented. Moreover, the provided documentation should be user friendly and should lead to a ease setting up and extending the system in the future.

4.1.2 Integration Requirements

The outcome of this student thesis must be integrated with the taxi scenario originated from the 4CaaSt project to build the taxi application [4Ca]. For this purpose, and after analyzing the different components forming the taxi application, we need to specify additional requirements the system should fulfill for integration purposes. In the version two of the taxi application the point-to-point connections between the Business Process Execution Language 2.0 (BPEL) processes installed in OW2 Orchestra and the Web services these consume have to be replaced with a communication through ServiceMix endpoints (Section 6.1). The BPEL processes installed in Orchestra communicate using SOAP over HTTP protocol. Furthermore, some of the components of the taxi application have to be multi-tenant aware to communicate with the tenant aware JBI endpoints of the ESB (e.g. TaxiCompany and TaxiTransmitter), and some components are non multi-tenant aware and have to communicate through non multi-tenant aware JBI endpoints in ServiceMix (e.g. Processes installed in Orchestra, CMF and GoogleServices). As last requirement to fulfill, the taxi request is mainly Orchestrated by several BPEL processes which communicate with different services to retrieve information and book a taxi. The overall process time is variable but synchronous. The multi-tenant endpoints correlating one tenant's taxi booking response have to be synchronized with the process replying the taxi request.

5 Design

In this chapter we present the architectural and technological solution taken into account to integrate two approaches for enabling multi-tenancy in a ESB solution [Ess11], [Muh12]. Furthermore, it fulfills the requirements described in the Chapter 4 and provides a detailed design for easing the implementation cycle described in Chapter 6. We start defining the architecture of the prototype we should implement in this thesis and we continue by giving more details on specific components that need to be extended or modified. As discussed in the previous chapters, some communication approaches taken into account in the master's thesis [Ess11] need to be improved. When describing them, we specify the main differences and the main advantages of the design approach we take.

5.1 Integration Architecture

The tenant-aware architectural design of this student thesis is composed of two main parts: administration and management, and communication through the ESB components. Muhler proposes an administration and management system, JBIMulti2, which is described in Chapter 2 [Muh12]. In this system he extends components in ServiceMix and develops new ones to fulfill tenant-aware management requirements. The communication between JBIMulti2 and the multiple instances of ServiceMix is established by a unidirectional connection to a JMS topic each of the ESB instances subscribe to. In each of the ServiceMix instances Muhler deploys the JMSManagement OSGi bundle which consumes the topic's management messages, which contain lifecycle instructions, e.g. install and uninstall SA, and the JBI BCs to deploy, as message payload. When unsuccessful deployment, the instruction message is stored in a queue of unprocessed messages. When successful deployment, the tenant-aware endpoint configuration packed in a SA is installed in ServiceMix isolated from other tenants by modifying the SA to reference it to a specific tenant. This reference is done by inserting tenant context in an XML file saved in each of the SAs, which contains the tenant user and tenant Universally Unique Identifier (UUID), and the tenant's URL. We use this information for authenticating the incoming tenant's requests to the ESB.

In a Cloud environment the communication between tenant-aware endpoints in an ESB must be isolated. ServiceMix does not provide this functionality. For this purpose, Muhler extends the HTTP BC and the Apache Camel SE for enabling a transparent configuration of tenant-aware endpoints (see Section 5.4). However, we consider that an ESB should support tenant-aware communication in more than one communication protocol, and the communication within tenant-aware endpoints has to be enriched with tenant information. Hereby, we extend the JMS and the E-mail BCs to provide a multi-protocol tenant-aware

communication environment (see Figure 5.1), and we include tenant information in the NM for future message processing, e.g. tenant-aware QoS mechanisms in the ESB.

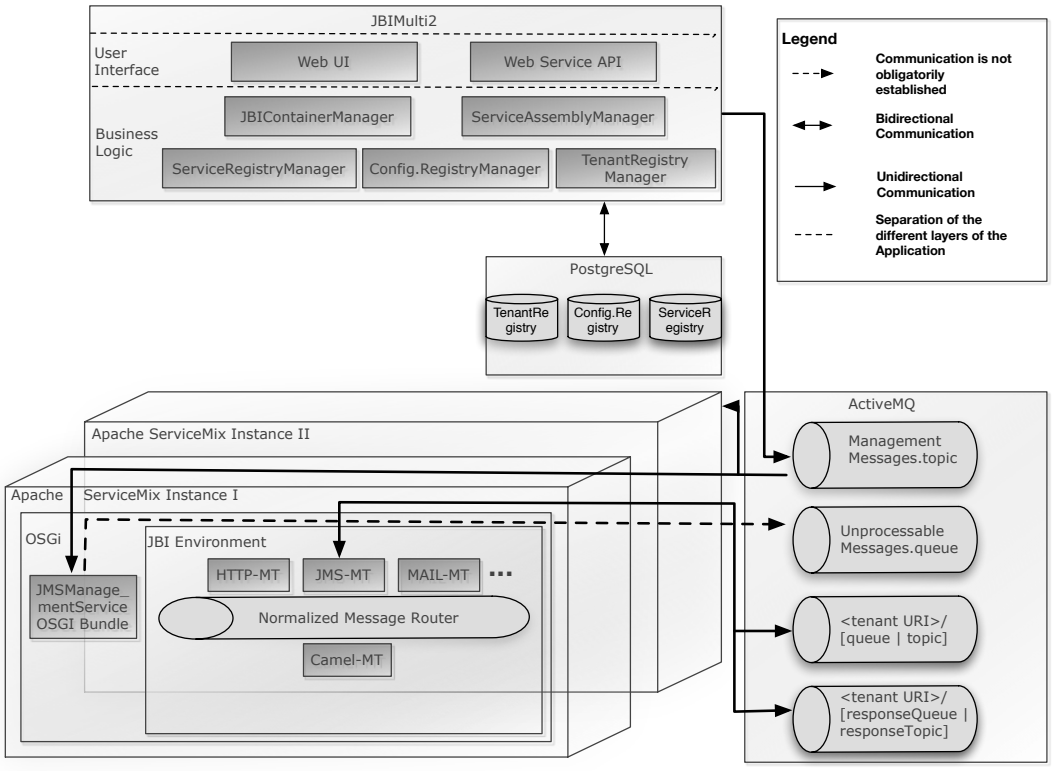

Figure 5.1: Architectural overview of the integration of the administration and management, and communication approaches.

Backward compatibility is one of the most important non-functional requirements this thesis should fulfill. We consider that a tenant-aware communication can be established between two tenant's endpoints if the incoming message has tenant information. To provide backward compatibility, we don't modify the original BCs and redeploy them, but create new ones with our approaches implemented in them, e.g. servicemix-http-mt, servicemix-jms-mt, servicemix-mail-mt. The namespace utilized in these BCs is modified in order to be able to have both tenant and non tenant-aware BCs in the ESB.

5.2 Tenant Context

In a multi-tenant environment both incoming and outgoing requests to and from the ESB should carry information which uniquely identifies the tenant. Each of the ESB BCs or SEs involved in a communication between two external endpoints need to handle this information in order to accept or reject message exchanges in the ESB.

Muhler and Essl represent the required tenant information data in an XML schema [Muh12], [Ess11]. In this they obtain the uniqueness of a tenant identifier by utilizing the UUID format which is generated by the java class java.util.UUID in JBIMulti2 [Gro05], [Ess11]. Furthermore, the tenant information representation distinguishes two main contexts: full tenant context and simple tenant context. In the former the tenant information contains the tenant and user identifiers, and optional key-value pairs entries, while in the latter the tenant information is represented by a tenant context key. Essl proposes the use of a tenant context key in transport protocols which are not extensible for supporting the tenant information representation as structured data [Ess11], and the retrieval of the full tenant context information from the system's Tenant Registry. However, in this student thesis we modify this approach in order to fulfill two main requirements: tenant authentication and performance optimization.

Bray et al. define XML documents as an entity container of parsed data made up of characters [BPSM+08]. The tenant context information can be sent in the transport protocols we extend as character chains complying the XML format representation, and embedded as properties or headers values which are parsed in the ESB. We propose the usage of the full tenant context in the transport protocols where text is supported either as a property or as a header of the message sent to the ESB. In this, the tenant and user identification are mandatory in a tenant-aware communication, while the optional key-value pairs are voluntary (see Listing 5.1).

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns="http://jbimulti2.iaas.
       uni-stuttgart.de/tenant-context"
4      targetNamespace="http://jbimulti2.iaas.uni-stuttgart.de/tenant-context">
5
6      <xsd:simpleType name="uuidType">
7          <xsd:restriction base="xsd:string">
8              <xsd:pattern value="[a-f0-9]{8}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{4}-[a-f0-9]{12}
                   " />
9          </xsd:restriction>
10     </xsd:simpleType>
11
12     <xsd:group name="tenantUserId">
13         <xsd:sequence>
14             <xsd:element name="TenantId" type="tns:uuidType" />
15             <xsd:element name="UserId" type="tns:uuidType" />
16         </xsd:sequence>
17     </xsd:group>
18
19     <xsd:element name="TenantContext">
20         <xsd:complexType>
21             <xsd:sequence>
22                 <xsd:group ref="tns:tenantUserId" />
23                 <xsd:element name="OptionalEntry" minOccurs="0" maxOccurs="unbounded">
24                     <xsd:complexType>
25                         <xsd:sequence>
26                             <xsd:element name="Key" type="xsd:string" />
27                             <xsd:element name="Value" type="xsd:anyType" />
```

```
28                          </xsd:sequence>
29                        </xsd:complexType>
30                      </xsd:element>
31                    </xsd:sequence>
32                  </xsd:complexType>
33              </xsd:element>
34  </xsd:schema>
```

Listing 5.1: Tenant context XSD of the incoming and outgoing messages to and from the ESB. Modified XSD version based on Essl and Muhler [Ess11], [Muh12].

With this approach, we eliminate the latency caused by the single tenant context, where the full tenant context data must be queried in two out of the three transport protocols we extend: JMS and E-Mail. Furthermore, we develop an authentication mechanism which reuses the tenant context information stored in each of the tenant-aware SAs. This feature adds parsing load in the BCs but eliminates the latency produced by either a cache or database connection and information querying per received request in the ESB. Furthermore, as the tenant and user UUID identifies uniquely one tenant and user respectively, this information must be encrypted before sending the request and unencrypted for authenticating the tenant in the ESB. However, the confidentiality is out of the scope of this student thesis.

5.3 Tenant-aware Normalized Message Format

As described in Section 5.1, the communication between two tenant aware endpoints has to be transparent to the tenant. The protocol neutral message format used in ServiceMix in a message exchange is the NM. For tenant-awareness in the ESB we don't only need to configure isolated endpoints for each of the tenants, but also to provide tenant-aware information in the message exchanged between the tenant endpoints in a transparent way to the user. Essl proposes the enrichment of the ESB NM with the tenant information contained in the request message or retrieved from the tenant registry, in the properties field of the NM [Ess11].

Two or more properties are included in his message representation: tenant id, user id and optional values. However, we consider that the structured tenant information in the incoming requests should be represented in a structured format in the NM properties. In this approach we propose two property maps: *jbimulti2.tenantcontext.mandatory* and *jbimulti2.tenantcontext.optional* (see Figure 5.2). The former stores the tenant and user id, while the latter stores the optional key-value pairs. The input message payload is transformed to XML format and embedded as the message payload of the NM. The attachments contained in the request are inserted as attachment in the NM.

Figure 5.2: Tenant-aware NM created after tenant successful authentication.

5.4 Tenant-aware Binding Components

The multi-tenant approach implemented in the extension of the tenant-aware BCs should manage the tenant endpoint configuration in a transparent way to the user, as well as it must ensure that a message exchange involves only endpoints configured in a tenant's SU. ServiceMix defines an user endpoint by the user's defined namespace followed by the service name and the endpoint name. To ensure data isolation between tenants, Muhler proposes the modification of the endpoint's URI during the configuration of the endpoint in the deployment phase [Muh12]. As described in the Listing 5.2, the tenant id will be the key for isolating endpoints between tenants. The BC is able of creating as much services as tenant-aware endpoints configurations are deployed in their SUs. It creates a common service prefix which is followed by the tenant UUID, the service and endpoint name declared in the XBean configuration file in the SU.

During deployment, ServiceMix configures the endpoints described in the bean definitions files contained in the SUs. The BaseServiceUnitDeployer is used as the JBI SU manager in ServiceMix. Its *XBeanDeployer*, which sets configuration data in each of the components, is extended in out tenant-aware BC. Our extension injects data read from the tenant context file in the SU and configures the endpoint with tenant-aware data. This mechanism allows two

```
1 /*
2   input: tenantId, serviceLocalPart, endpointName, configuredServiceNamespacePrefix
3   example: {jbimulti2:tenant-endpoints/<tenant id>}TaxiServiceConsumer:
        TaxiServiceConsumerEndpoint
4  */
5 serviceEndpoint        ::= serviceName ":" endpointName
6 serviceName            ::= "{" serviceNamespacePrefix tenantId "}" serviceLocalPart
7 serviceNamespacePrefix ::= "jbimulti2:tenant-endpoints/" | configuredServiceNamespacePrefix
```

Listing 5.2: Tenant-aware endpoint URI in extended Backus-Naur Form (EBNF) [Muh12].

tenants to deploy the same SU files and to get a different endpoint configuration after the deployment.

In addition to a tenant-aware endpoint configuration, we need to manage the resources consumed by each of the tenant-aware endpoints associated to each of the transport protocols we support in the ESB. In the following sections we will describe how the resources are created, distributed and destroyed in our multi-tenant environment.

5.4.1 SOAP over HTTP

In a HTTP consumer endpoint configuration during the deployment phase, the SU deployer builds an URI for accessing the created service's endpoint by attaching the service and endpoint names to the default URI, e.g. *http://localhost:8192/*. In the multi-tenant approach, Muhler creates dynamic URI by injecting tenant information read from the tenant context file in the SU where the endpoint is configured [Muh12]. The dynamic URI is represented in the Listing 5.3, where we can see that two tenants' HTTP endpoints URI differentiate each other in the tenants' ids or tenants' URLs.

```
1 /*
2   input: tenantId, tenantUri, serviceLocalPart, endpointName, configuredLocationUriPrefix
3   example: http://localhost:8193/tenant-services/<tenantid | tenantURI>/TaxiServiceConsumer/
        TaxiServiceConsumerEndpoint
4  */
5 locationUri       ::= locationUriPrefix ( tenantId | tenantUri ) serviceEndpoint
6 locationUriPrefix ::= "http://localhost:8193/tenant-services/" | configuredLocationUriPrefix
7 serviceEndpoint   ::= "/" serviceLocalPart "/" endpointName
```

Listing 5.3: Tenant-aware service URL [Muh12].

In a Web service consuming environment, we have discussed in previous chapters that the SOAP over HTTP protocol is used and the consumed Web service description is provided in the WSDL file. In the tenant-aware ServiceMix each of the SOAP over HTTP JBI consumer endpoints is exposed to the exterior as a Web service (see Figure 5.3). When this type of endpoint is deployed in our multi-tenant approach, the Web service consumer can access the Web service description by appending *main.wsdl* to the described URI in Listing 5.3.

The WSDL description is dynamically modified during the endpoint configuration with the tenant-aware endpoint's URI.

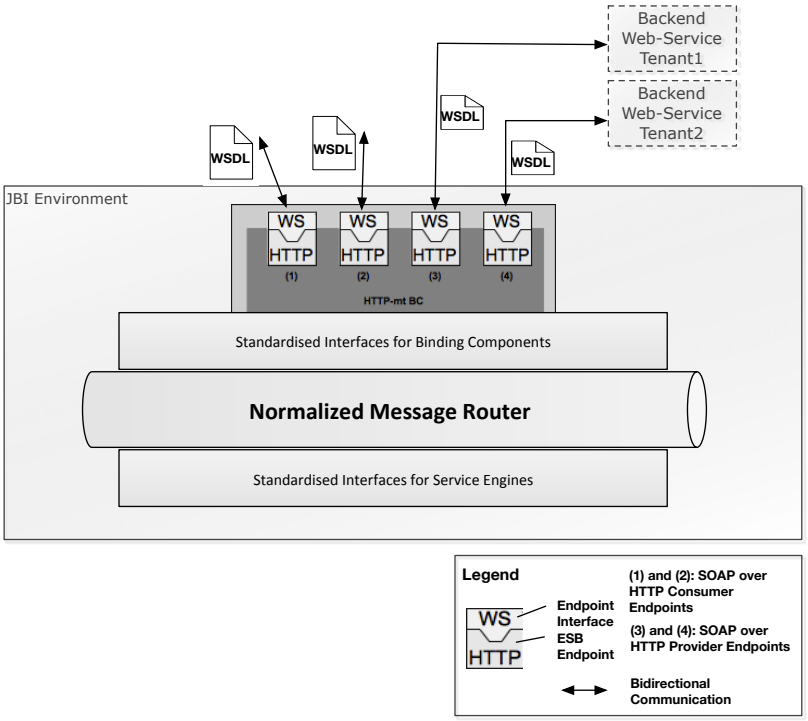

Figure 5.3: Design overview of the multi-tenant HTTP JBI BC based on two tenants.

When a SOAP request reaches a tenant-aware consumer endpoint, we must ensure authentication before creating the NM. This functionality can be added in the SOAP to NM marshaler. The marshaler transforms the incoming request into the internal ESB message format. If the tenant context information contained in the SOAP header leads to a successful authentication, the marshaller must insert the tenant context information and the optional key-values pairs into the NM. Otherwise, an error should be sent to the requester and the administrator should be notified. In our approach we use a SOAP error response for unsuccessful authentication and we log the authentication attemps for system monitoring.

On the other side, in the tenant-aware HTTP provider endpoints, the BC must support sending tenant-aware HTTP requests to external services. Each of the providers endpoints are configured to reach a provided external service by de-marshaling the NM to an HTTP (or SOAP over HTTP) request. The NM properties must be embedded as HTTP or SOAP headers. The backend service must be manually configured by the tenant. The taxi scenario converges all tenants' request into a BPEL process which finds the available taxi. However, if two or more tenants deploy SU which invoke different external backend Web services, the WSDL or the URL must be provided (see Figure 5.3). However, the inclusion of this feature is out of the scope of this thesis.

5.4.2 JMS

ServiceMix is shipped with an ActiveMQ instance out-of-the-box which can be access through console commands in Apache Karaf [ASM]. In this section we provide two approaches for supporting tenant-aware routing between tenant-aware JMS JBI endpoints in the ESB. In Section 5.2 we introduce the XML structured tenant context information the incoming request must include. The JMS message format supports optional properties by defining the name and the value of them [HBS+02]. The tenant context information is included as a message property and the body contains the tenant data. Both data are required to be in XML format.

In the following sections we provide an overview on the two possible approaches which can be taken into account. Both of the approaches have been implemented and tested, and our design decision is discussed in the Chapter 6.

Design Approach 1

Tenant data isolation is a must in multi-tenant architectures. Hereby, we must ensure isolation between the messages of different tenants. In this first approach, one of our main priorities is to minimize the resource consumption, in this case the JMS queue and topic number, while ensuring message isolation. In this approach, we propose a shared queue and topic between the tenants' JMS consumer endpoints. In a JMS configuration we can configure a message selector based on a filter for consuming messages. This filter can be adjusted to each of the tenant context information in the tenant endpoints. However, for future compatibility, the JMS broker does not provide unencryption mechanisms before performing the filtering in the endpoints, reason why we couldn't continue with this approach.

However, Apache Camel provides a dynamic routing mechanism between target endpoints called *Recipient List* [APA11a]. With this feature, we first unencrypt the message, authenticate the tenant, and finally route it based on the tenant information contained in the message. In the JMS queue mode, the unencryption of the message is done by a shared JMS consumer endpoint which consumes from one shared queue, verifies if the communication is tenant-aware, and statically routes the message to the dynamic router. We propose in this approach a different authentication mechanism. The dynamic router is responsible for authenticating the tenant before routing the request. For this purpose, we need a connection with the tenant registry and configuration registry, but we need to minimize the latency produced by a database connection. We propose a component which connects the ESB with the tenant registry and includes a cache mechanism for recently tenants authenticated (see Figure 5.4). When the tenant is successfully authenticated, the message exchange between the camel router and the JMS provider endpoint is created, and the provider endpoint stores the tenant's message in the tenant's queue.

In the publish/subscribe mode, the tenant JMS endpoints subscribe to a shared topic, but only one endpoint is capable of unencrypting the received JMS message and creating the NM. In the provider endpoint, the message is published in a tenant's topic.

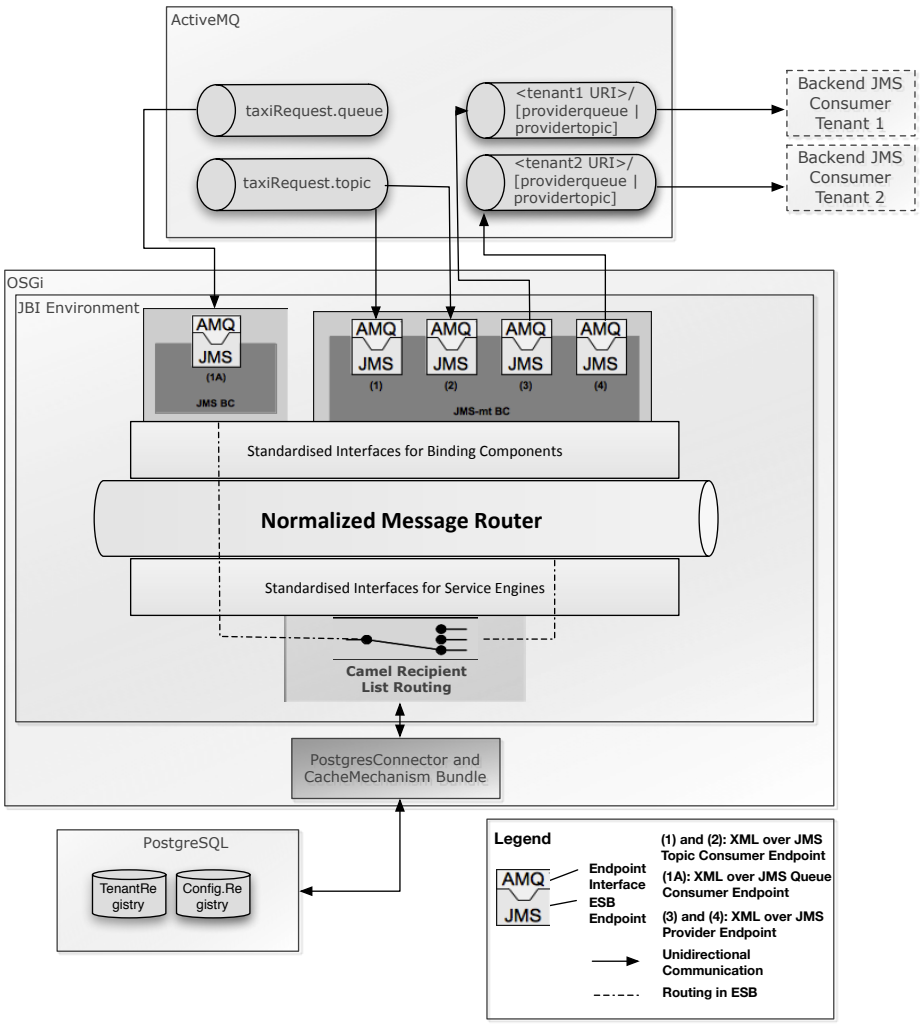

Figure 5.4: Design overview of a first approach of the multi-tenant JMS JBI BC based on two tenants.

In both queue or topic cases, the tenant can provide their queue or topic name and connection URL, or the tenant-aware BC will transparently create this resource for the tenant in the out-of-the-box ActiveMQ. The names for the resources are shown in the Figure 5.4. In the first case, the tenant has to provide the resource details before the deployment. However, this feature is out of the scope of this student thesis. In our approaches we consider the creation of resources which will be consumed by the tenants in our system.

Design Approach 2

In this approach we focus on the performance rather than the resources consumption. For both consumer and provider's tenant-aware endpoints, one queue or one topic is created. This can be thought as not scalable, but ActiveMQ can be configured to provide massive scalability by setting a network of brokers [AMQ]. With this feature, the acceptable number of queues and topics is linearly increased to the number of brokers conforming a network.

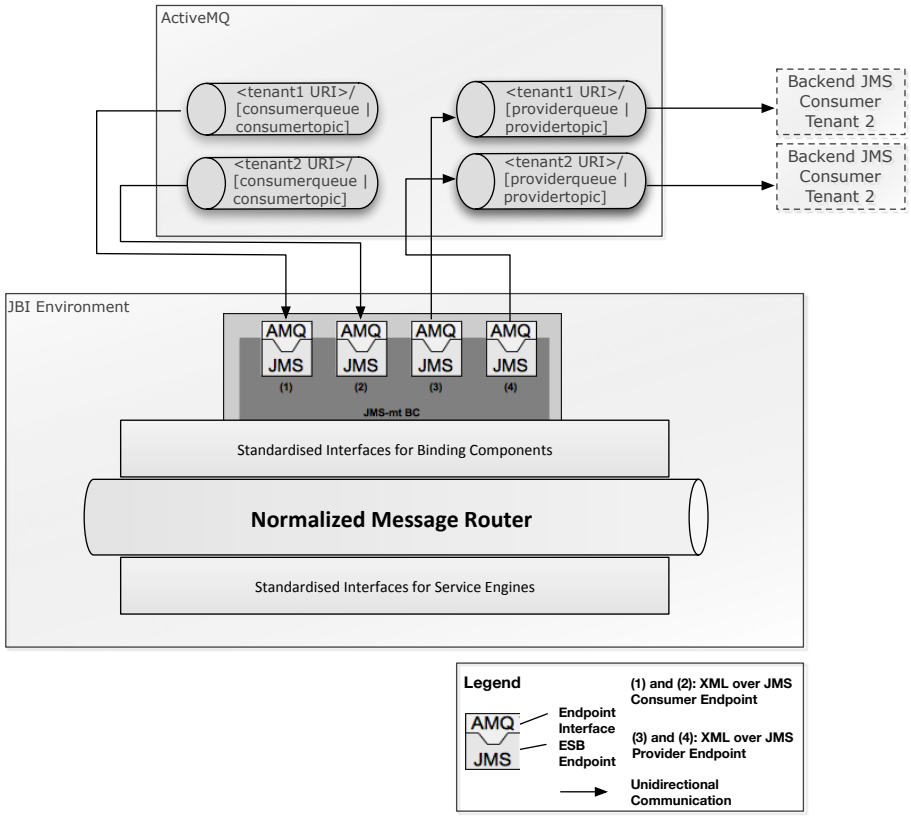

Figure 5.5: Design overview of a second approach of the mutlti-tenant JMS JBI BC based on two tenants.

With one queue and one topic per endpoint, at the consumer side we ensure that another tenant's SU component, e.g. marshaler, does not receive the message, and we discard the need of the connection to the tenant registry during authentication. We can integrate the same authentication method used in the extended HTTP and E-mail BCs marshalers, leading to a better communication performance and to a bottleneck avoidance. This approach considers the creation of one queue or topic per tenant-aware consumer endpoint in the ActiveMQ shipped with ServiceMix and with a standardized name shown in Figure 5.5. The same mechanism is adopted for a tenant-aware provider endpoint, which requires to consume one queue or topic to send or publish a message. As in the first approach, this can be provided by the tenant or can be created in our system.

5.4.3 E-Mail

As the last BC this student thesis must extent for multi-tenancy awareness, we introduce a design approach for the multi-tenant E-email BC. As described in Chapter 2, the E-mail BC supports the following mail protocols: SMTP, IMAP, and POP3. In the tenant-aware consumer endpoints we utilize the IMAP protocol and in the provider endpoints the SMTP (see Figure 5.6). As pre-condition for deployment, the tenant must provide his or her E-mail address and access credentials for polling incoming messages and for sending the message.

When a message arrives to the tenant's E-mail inbox folder, it is consumed by the tenant's endpoint. The E-mail message format should comply the following requirements: the request is sent as a message body in XML format and the tenant context information in XML format in an E-mail customized header. The authentication mechanism is the same as the one used in the multi-tenant HTTP BC. In the tenant's provider endpoint, the destination E-mail address is taken, if exists, from the optional key-value pairs included in the request.

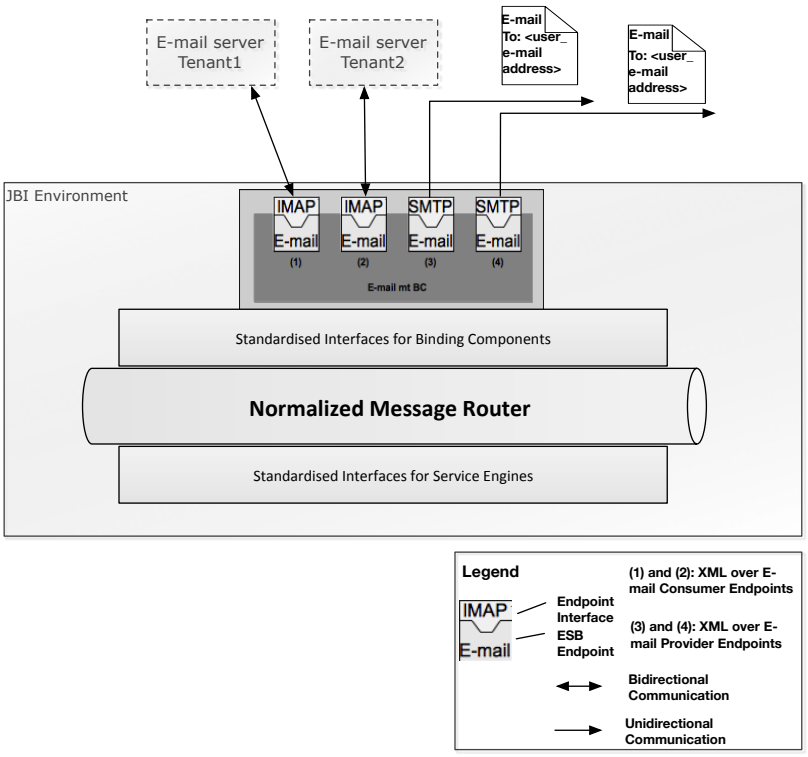

Figure 5.6: Design overview of the multi-tenant E-mail JBI BC based on two tenants.

6 Implementation

As described in the Chapter 1, the outcome of this student thesis must be integrated with the taxi scenario of the 4CaaSt project [4Ca]. In the Section 6.2 we describe the adaptations we need to make in our initial multi-tenant approaches in order to support both tenant-aware and non tenant-aware communications between the components. Furthermore, we specify which components of the taxi application support multi-tenancy and which components support only a non multi-tenant communication. In Section 6.3 we detail the implementation approaches and challenges when developing the multi-tenant SOAP over HTTP, JMS, and Email BCs.

The implementation of the different components in this student thesis is organized as a hierarchical Maven project [AMV]. Apache Maven allows programmers to build different Java-based projects by specifying the tasks and plugins in the Project Object Model (POM) file which are executed during the building process of the project. Programmers can then build using Maven different types of packages for deployment, e.g. Web Application Archive (WAR), Enterprise Archive (EAR), SAs, and OSGi bundles. Furthermore, it provides support for specifying the dependencies between modules stored in a local or remote maven repository [AMV]. In this student thesis we have adapted and extended the maven project from Muhler in order to provide a single project containing the necessary components for building and deploying the taxi application [Muh12]. Maven provides several ServiceMix components endpoints configuration archetypes, which are templates we use for configuring the endpoints, and building the SU and SA for deployment. The original BCs implementation provided by Apache and after extended for multi-tenant awareness are included in a Maven project, built as a BC package, and referenced to by a specified namespace. For the tenant-specific endpoint configuration we first create the SU package content using the maven archetype for a specific endpoint in the BC, then set the configuration fields in the configuration file, and reference the multi-tenant BC this SU configures. Finally, the SUs should be packaged in a SA for deployment. The Java source code provided in this student thesis is compiled with the Java Development Kit (JDK) 6.

6.1 System Overview

The taxi application is built up of different components running on 4 different servers with a Java 6 jdk. The JOnAS v 5.3.0 hosts the management and logic of the application. Each of the tenants deploy their TaxiCompany Web interface for taxi requestors' usage and the TaxiTransmitter for the taxi drivers' usage. Both of the interfaces have to be tenant aware in order to communicate through the tenant aware endpoints. As exposed before, a tenant's

request is routed between two tenant aware endpoints if a successful tenant authentication occurs. The communication to and from the main BPEL processes (TaxiServiceProvider and ContextIntegration, see Figure 6.1) is done through the ESB. We must indicate that the BPEL processes are non multi-tenant aware and they receive and reply SOAP over HTTP requests and responses respectively. Hereby the ESB must handle both tenant and non tenant aware routing between this components, as well as between the main BPEL processes, CMF, and GoogleServices adapters), by being able to create both tenant aware and non tenant aware endpoints.

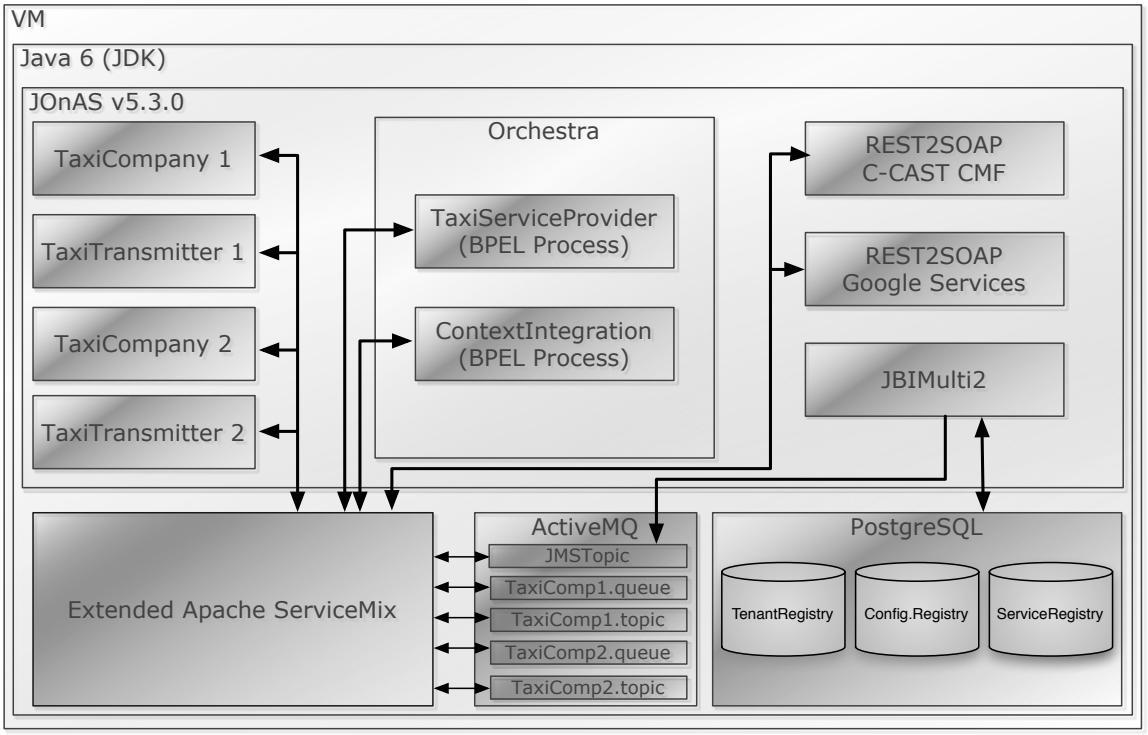

Figure 6.1: System overview of the version two of the taxi application [4Ca]. **Note:** the queues and topics created for the different tenants, in this case TaxiCompany1 and TaxiCompany2 are created only when the tenant deploys an JMS endpoint configuration

The JBIMulti2 utilizes three main components in the system: registries created in a PostgreSQL database, resources in ActiveMQ and ServiceMix. When deploying a tenant's SA the application initiates an unidirectional communication to a management queue from which a management OSGi bundle consumes the management messages from JBIMulti2 and performs managements operations in ServiceMix, such as deploy and undeploy. The deployed multi-tenant SAs configure the tenants' endpoints for BCs or SEs. When deployed a multi-tenant SA which packages a JMS endpoint configuration, resources are transparently created in the out-of-the-box ActiveMQ instance which is shipped with ServiceMix. However, it is out of the scope of this student thesis an interface for connecting the different tenants with the created JMS resources.

6.2 Taxi Scenario Integration

In this section we describe the implementation approaches for integrating the multi-tenant ESB with the already developed components, in order to build the taxi application v2.0. In the next section we divide the system described in Figure 6.1 (see Section 6.1) in two main subsystems, based on the tenant-awareness of its components.

In the taxi application v2.0 we distinguish two main parts, one multi-tenant aware and one non multi-tenant aware. The developed binding components in the ESB permit the deployment of multi-tenant and non multi-tenant BCs. This feature enables the integration of the components conforming the taxi application. The taxi application v2.0 utilizes the JOnAS server for hosting the upper components shown in Figure 6.2 and uses SOAP over HTTP communication protocol between its components.

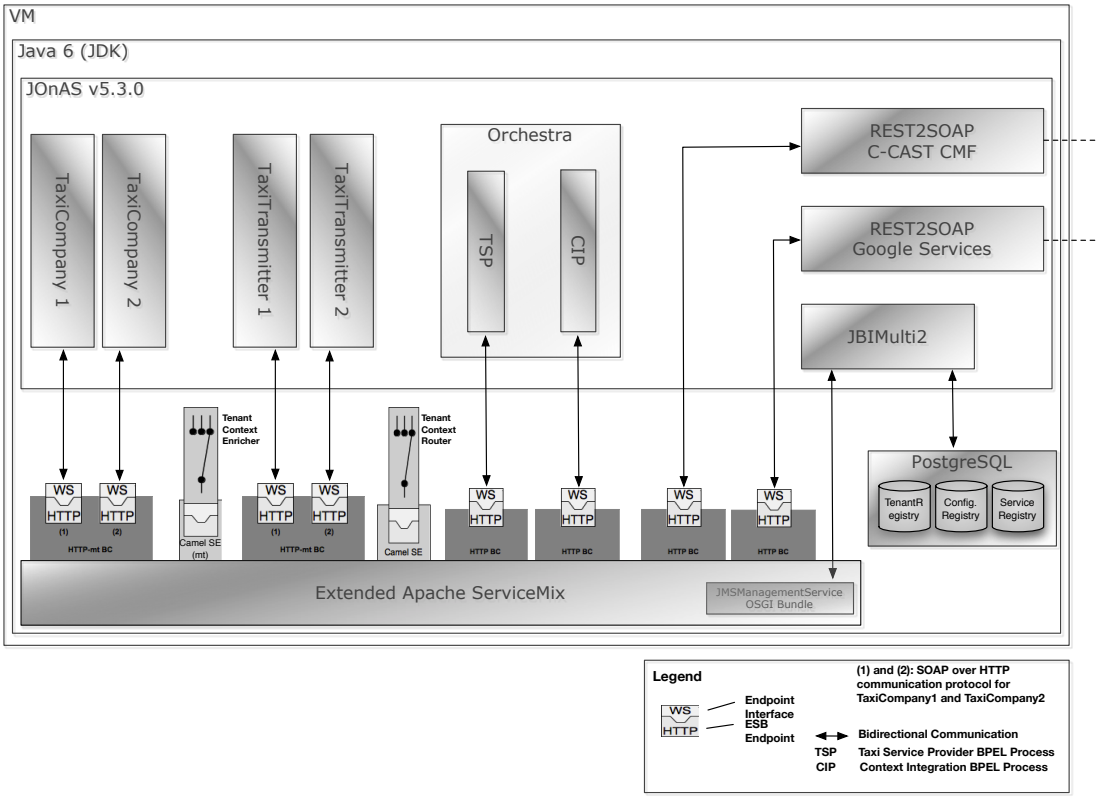

Figure 6.2: Implemented and Configured BC for integration with Taxi Scenario v2.0 [4Ca], [Muh12].

In this prototype we present two taxi companies, whose front-end components are their corresponding taxi company and taxi transmitter Web interfaces, both tenant specific. They communicate with the taxi service provider BPEL process through a single non multi-tenant provider endpoint configured in the ESB. We follow the single endpoint approach for invoking the taxi service provider process due to the lack of multi-tenant support it provides. An extension of the OW2 Orchestra engine for multi-tenancy awareness would enable a

multi-tenant deployment of the taxi service provider process [OWO]. If more than one instance of the processes can be deployed, e.g. one process per tenant, individual endpoints for invoking the different instances would be needed. The HTTP configuration in the SU packed in the SA can be deployed by more than one tenant due to the namespace modification the multi-tenant HTTP BC performs, e.g. the namespace used for the taxi application *http://www.taxiserviceprovider.eu/ transmitter/definitions* is replaced with the namespace *jbi:endpoint:jbimulti2:tenant-endpoints/<tenantUUID>* [Muh12]. However, the taxi application's BPEL processes deployed in Orchestra don't implement multi-tenant awareness at the level of the SOAP header. For this reason, the NM content is enriched with tenant-aware context information retrieved from the ServiceName in a tenant-aware Camel SE and the message sent to the BPEL processes contains the tenant id as additional element of the SOAP body [Muh12]. The authentication of both taxi companies is made when the taxi booking request is sent through the ESB. This authentication procedure is detailed in Section 6.3.

In this prototype we maximize the use of the ESB in the application. For this reason, we propose an implementation which wires the communication between the different components in the application through the ESB. We modify the direct connection configuration between the BPEL processes deployed in Orchestra, and the CMF and GoogleServices components (see Figure 6.2, and configure non multi-tenant endpoints in the ESB through which the messages between the components are routed.

The taxi service provider BPEL process, after calculating the nearest available taxi by requesting location and context information from both CMF and GoogleServices components, sends a transport request to the taxi driver [Hag11]. In our prototype we present two sets of taxi drivers, each one belonging to one specific company. For tenant-awarness purposes, we must ensure that the transport request is routed to the appropriate taxi company. The routing based on the tenant context is performed by the tenant context router, which builds the tenant-aware route from the tenant id included in the transport request. With this approach we ensure that one taxi company driver does not receive a transport request from another taxi company. However, we should allude the lack of multi-tenant awareness in the CaaS component the CMF retrieves the context data from [4Ca], [CCA]. This absence requires the context integration provider BPEL processes to calculate the minimum distance to the customer's location for all the sets of taxis registered in the CaaS system [CCA]. Anyhow, the multi-tenant awareness implementation in the CaaS system is out of the scope of this student thesis.

6.3 Multi-tenant Binding Components

The main requirement in this student thesis is to enable multi-tenancy in ServiceMix by integrating multi-tenant administration and management, and communication approaches. The BCs supporting the communication protocols we focus on must be modified to ensure isolation between endpoints while enabling the tenants to deploy a shared endpoint configuration template. During deployment the internal endpoint deployer must make the necessary configuration modifications to provide independent endpoints to the tenants.

The tenant-aware endpoints in the ESB are configured conforming the JBI specification. Endpoint properties are set in a SU, and one or more SUs must be packed in a SA for deployment. Maven provides several templates for configuring endpoints in the supported BCs in ServiceMix, which we utilize for generating the SAs the tenants deploy. Maven archetypes generate the structure of the SU and the necessary files for setting the endpoint properties. The file we focus on in this thesis is the *xbean.xml* file, where the endpoints' properties are set in XML format, and the POM file, where the maven project properties are described and the dependencies on other modules are listed. In the following sections we detail which archetypes we use and which endpoint parameters we set.

6.3.1 SOAP over HTTP

The HTTP BC shipped with ServiceMix embeds the configuration of abstract endpoints which support the HTTP communication protocol and the protocols build on top of it, such as SOAP for Web services communication. Muhler enables multi-tenancy in this BC by injecting tenant context information in the endpoint configuration namespace [Muh12]. In a SOAP over HTTP communication protocol where a Web service operation is invoked through the ESB, the ESB consumer endpoint should publish the WSDL the Web service consumer retrieves, and the exposed service should be addressed by an URI. For both requirements, Muhler injects tenant context information in the URI for accessing both the Web service endpoint and the WSDL information (see Listing 6.1). With this approach two or more tenants can deploy the same endpoint configuration, which is then modified with the tenant context information during deployment. However, if two or more tenants invoke different Web services, in this implementation their WSDL files should be included in the SUs.

```
1 /*
2   input: tenantId, tenantUri, serviceLocalPart, endpointName, configuredLocationUriPrefix
3   example: http://localhost:8193/tenant-services/54ed4755-5965-4b47-a121-d25907e29c04/
        TaxiRequestService/TaxiRequestServiceEndpoint
4  */
5 locationUri       ::= locationUriPrefix ( tenantId | tenantUri ) serviceEndpoint
6 locationUriPrefix ::= "http://localhost:8193/tenant-services/" | configuredLocationUriPrefix
7 serviceEndpoint   ::= "/" serviceLocalPart "/" endpointName
8 WSDL Location   ::= locationUri/main.wsdl
```

Listing 6.1: Tenant Endpoint URI Example [Muh12].

Nevertheless, multi-tenancy communication is not supported in Muhler's approach. Tenant endpoints are isolated from each other, but the internal NM routed in the ESB contains no tenant information for processing it in the ESB, e.g. tenant based routing. For fulfilling this requirement, we integrate and modify Essl's approach for enabling multi-tenancy support in communication [Ess11]. Essl proposes the extension of the consumer endpoint marshaler and the injection of tenant information which is either contained in the incoming SOAP request or retrieved from the tenant registry [Ess11]. Furthermore, he proposes backward compatibility in multi-tenant endpoints without taking into account tenant authentication in

the communication. This leads to the routing of messages which are not sent by the endpoint's tenant proprietary.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:typ="http://www.taxiserviceprovider.eu/types">
   <soapenv:Header>
  <typ:tenantId>dc9cdf31-0cf1-4151-99c4-39c1a8d769a6</typ:tenantId>
  <typ:userId>09539020-26af-43d2-970c-2464d710e3ac</typ:userId>
  <typ:optionalEntry>
        <typ:key>tenantName</typ:key>
        <typ:value>typiCompany</typ:value>
    </typ:optionalEntry>
    <typ:optionalEntry>
        <typ:key>userName</typ:key>
        <typ:value>Mr. John Doe</typ:value>
    </typ:optionalEntry>
    <typ:optionalEntry>
        <typ:key>mailUserContact</typ:key>
        <typ:value>typicustomer1@googlemail.com</typ:value>
    </typ:optionalEntry>
   </soapenv:Header>
   <soapenv:Body>
      <typ:taxiBookingRequest>
         <typ:customerID>110092</typ:customerID>
         <typ:originLocation>Kronenstrasse 2, stuttgart</typ:originLocation>
         <typ:destinationLocation>flughafenstrasse 1, stuttgart</typ:destinationLocation>
         <typ:customerInfo>
            <typ:phone>3333</typ:phone>
            <typ:email>sss@fff</typ:email>
         </typ:customerInfo>
      </typ:taxiBookingRequest>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 6.2: Tenant-aware SOAP over HTTP message example. Namespaces are imported from the taxi scenario [4Ca].

In this student thesis we modify the communication approach and integrate it with Muhler's multi-tenant administration and management approach. In the first place we assume that one tenant endpoint only accepts communications from the tenant propietary of that endpoint. Therefore, each tenant must authenticate with the ESB with their tenant context information (see Listing 6.2). We implement the tenant authentication in the consumer endpoint marshaler by retrieving the tenant context information stored in the tenant context file in the deployed SU, and comparing it with the tenant context information contained in the SOAP request. With this approach each SU contains a Java class which extends the HTTP BC's *http.endpoints.HttpSoapConsumerMarshaler* class, which is referenced from the endpoint's bean configuration. The tenant information contained in the header field of the SOAP request is inserted by message interceptors as a NM map property named *javax.jbi.messaging.protocol.headers* which contains as keys QNames and as values DocumentFragment structures. When successful authentication, the NM is created with the data

contained in the SOAP over HTTP request and the tenant context information is added as property maps, with QNames values as keys and Objects as values, in the NM. In the provider endpoint we extend the http.endpoints.HttpSoapProviderMarshaler for injecting the tenant context information in the SOAP header field of the output message .

6.3.2 JMS

The JMS BC shipped with ServiceMix supports both SOAP over JMS and plain text over JMS communication protocol. In this thesis we extend the second approach for enabling multi-tenancy communication awareness. The transparent configuration of tenant-aware endpoint in our extended version must be integrated with the JBIMulti2 application [Muh12]. Therefore, we must provide the same deployment operations as the multi-tenant HTTP BC or multi-tenant Camel SE, in order facilitate the integration with the JMSManagement Service component of JBIMulti2. We extend the ServiceMix *servicemix.common.xbean.BaseXBeanDeployer* in the multi-tenant JMS BC in order to perform the necessary changes in the endpoint configuration during deployment. The *BaseXBeanDeployer* class provides management methods for deploying endpoints whose configurations are described in the bean configuration file of the SU.

During deployment operation, we modify the endpoint configuration and inject tenant information retrieved from the tenant context file stored in the SU. As in the multi-tenant HTTP BC, the endpoint namespace is modified with the tenant context information. Such information is modified in the classes *org.apache.servicemix.jms.endpoints.JmsConsumerEndpoint* and *org.apache.servicemix.jms .endpoints.JmsProviderEndpoint*. In Chapter 5 we specify two design approaches for enabling multi-tenancy in the BC supporting the JMS protocol: one shared queue or topic vs. one queue and one topic per tenant.

```
JMS Header
-------------------------------------------------------------------------------
JMS.propertyName = tenantContext
JMS.propertyValue (tenantContext) =
  <?xml version="1.0" encoding="UTF-8"?>
  <taxiScenario xmlns:typ="http://www.taxiserviceprovider.eu/types">
          <typ:tenantId>892732f8-58ff-464d-883f-1099c36b8544</typ:tenantId>
          <typ:userId>2a736c63-de32-410c-b9ce-0d00247a78db</typ:userId>
          <typ:optionalEntry>
            <typ:key>tenantName</typ:key>
            <typ:value>TaxiCompany</typ:value>
          </typ:optionalEntry>
          <typ:optionalEntry>
            <typ:key>userName</typ:key>
            <typ:value>Mr. John Doe</typ:value>
          </typ:optionalEntry>
          <typ:optionalEntry>
            <typ:key>mailUserContact</typ:key>
            <typ:value>taxicustomer1@googlemail.com</typ:value>
          </typ:optionalEntry>
  </taxiScenario>
```

```
22 ------------------------------------------------------------------------------
23 JMS Body
24 ------------------------------------------------------------------------------
25 JMS.body =
26   <?xml version="1.0" encoding="UTF-8"?>
27   <taxiScenario xmlns:typ="http://www.taxiserviceprovider.eu/types">
28     <typ:taxiRequest>
29       <typ:from>Universitaet strasse 38, Stuttgart</typ:from>
30       <typ:to>Koenigstrasse 12, Stuttgart</typ:to>
31       ...
32     </typ:taxiRequest>
33   </taxiScenario>
34 ------------------------------------------------------------------------------
```

Listing 6.3: Tenant-aware XML over JMS message example. Namespaces are imported from the taxi scenario [4Ca].

In the first approach, we configure a JMS consumer endpoint which consumes messages from a shared queue between tenants for incoming messages to the ESB. The NM messages are enriched with the tenant context information included in the JMS message property *tenantContext* (see Listing 6.3), and routed to a camel router, which is extended to support tenant authentication. We deploy an OSGi bundle which provides the tenant authentication operations and includes a Least Recently Used (LRU) cache mechanism which reduces the number of queries to the database system when a tenant has authenticated in a recent past. When successful authentication, the endpoint namespace is built with the tenant context information and the message is routed to the tenant's provider endpoint. For the dynamic routing we use the *@RecipientList* annotation in a Java class which is referenced from the route configuration in the camel context configuration. In the multi-tenant JMS provider endpoint for both approaches we create for each tenant transparently the necessary resources for both queue and topic options. The creation of resources is made during deployment. The queue name is built in the *JmsProviderEndpoint* class from the tenant URI specified in the tenant context file, e.g. for *taxicompany* the provided queue name is *www.taxicompany.com/<ProviderQueue | ProviderTopic>*. In the tenant provider endpoint the marshaler builds the JMS message with the tenant context information and sends it to the *ProviderQueue* or *ProviderTopic*. For the topic model, we configure a shared topic in the consumer side. The consumer marshalers in the tenant-aware endpoints perform the authentication. The marshaling operations for tenant authentication, and the tenant-aware NM creation are described later in this section. However, we describe in Chapter 7 the disadvantages of this approach for both topic and queue models in the system's performance.

In the second approach, we propose individual resources for the tenants in the JMS queue and topic model. After injecting tenant context information in the endpoint configuration, the queue or topic name is built with tenant information and a request and reply queue or topic are created in the classes *JmsConsumerEndpoint* and *JmsProviderEndpoint*. The former can be identified with *ConsumerQueue | ConsumerTopic* in the consumer endpoint and *ProviderQueue | ProviderTopic* in the provider endpoint, and the latter with *responseQueue | responseTopic* in both consumer and provider endpoints, as described in Listing 6.4. In this implementation

we use an external ActiveMQ broker for management of queues and topics of the provider endpoints.

```
Consumer Endpoint
  www.taxicompany.com/<ConsumerQueue | ConsumerTopic>
  www.taxicompany.com/<ResponseQueue | ResponseTopic>

Provider Endpoint
  www.taxicompany.com/<ProviderQueue | ProviderTopic>
  www.taxicompany.com/<responseQueue | responseTopic>
```

Listing 6.4: Queue and topic names examples in multi-tenant JMS BC.

The tenant context information is included in the *tenantContext* JMS custom header. It as a string that complies the XML specification (see Listing 6.3) . This facilitates the marshaling of the tenant context information for both tenant authentication, and enrichment of the NM. For the former, we extend the class *org.apache.servicemix.jms.endpoints.DefaultConsumerMarshaler* in the SU and reference it from the endpoint bean configuration. The tenant context information in the message is compared with the tenant context information in the tenant context file, and the NM is created when successful authentication. For the latter, we include in the *org.apache.servicemix.jms.endpoints.AbstractJmsMarshaler* BC class marshaling support for parsing the tenant context information and the JMS message content, and inserting the tenant context in the NM. We follow the same procedure in the JMS provider endpoint and extend the class *org.apache.servicemix.jms.endpoints.DefaultProviderMarshaler*. The tenant context information set as a NM property is converted into XML format and included as a JMS message property.

6.3.3 Email

The mail BC shipped with ServiceMix supports the E-mail specifications [Gro08]. In this thesis we extend this BC to enable multi-tenant aware communication. The transparent configuration of tenant-aware endpoints in our extended version must be integrated with the JBIMulti2 application [Muh12]. Therefore, we must provide the same deployment operations as the multi-tenant HTTP BC or multi-tenant Camel SE, in order to integrate it with the JMSManagement Service component of JBIMulti2. We extend the ServiceMix *servicemix.common.xbean.BaseXBeanDeployer* in the multi-tenant mail BC in order to perform the necessary changes in the endpoint configuration during deployment. The *BaseXBeanDeployer* class provides management methods for deploying endpoints whose configurations are described in the bean configuration file.

During the deployment operation, we modify the endpoint configuration by injecting tenant information retrieved from the tenant context file stored in the SU. As in the multi-tenant HTTP BC, the endpoint namespace is modified and tenant context information is included in the endpoint's namespace and configuration. Such information is modified in the classes *org.apache.servicemix.mail.MailPollerEndpoint* and *org.apache.servicemix.mail.MailSenderEndpoint*.

The SMTP and POP3 mail servers used in this student thesis for receiving and sending E-mails must be provided by the tenant.

```
Email Custom Header
-------------------------------------------------------------------------
Email.headerName = tenantContext
Email.headerValue (tenantContext) =
  <?xml version="1.0" encoding="UTF-8"?>
  <taxiScenario xmlns:typ="http://www.taxiserviceprovider.eu/types">
          <typ:tenantId>892732f8-58ff-464d-883f-1099c36b8544</typ:tenantId>
          <typ:userId>2a736c63-de32-410c-b9ce-0d00247a78db</typ:userId>
          <typ:optionalEntry>
            <typ:key>tenantName</typ:key>
            <typ:value>TaxiCompany</typ:value>
          </typ:optionalEntry>
          <typ:optionalEntry>
            <typ:key>userName</typ:key>
            <typ:value>Mr. John Doe</typ:value>
          </typ:optionalEntry>
          <typ:optionalEntry>
            <typ:key>mailUserContact</typ:key>
            <typ:value>taxicustomer1@googlemail.com</typ:value>
          </typ:optionalEntry>
  </taxiScenario>
-------------------------------------------------------------------------
Email Body
-------------------------------------------------------------------------
Email.body =
  <?xml version="1.0" encoding="UTF-8"?>
  <taxiScenario xmlns:typ="http://www.taxiserviceprovider.eu/types">
    <typ:taxiRequest>
      <typ:from>Universitaet strasse 38, Stuttgart</typ:from>
      <typ:to>Koenigstrasse 12, Stuttgart</typ:to>
      ...
    </typ:taxiRequest>
  </taxiScenario>
-------------------------------------------------------------------------
```

Listing 6.5: Tenant-aware XML over E-mail message example. Namespaces are imported from the taxi scenario [4Ca].

We configure and modify both mail poller and mail sender endpoints in ServiceMix. In the mail poller, we must ensure tenant authentication and NM enrichment with tenant context information. The tenant authentication is implemented in the SU configuring the poller endpoint. The *org.apache.servicemix.mail.marshaler.DefaultMailMarshaler* class can be extended and then referenced from the endpoint bean configuration. We authenticate the tenant by retrieving the tenant context data from both *tenantContext* header in the E-mail message and the tenant context file in the SU (see Listing 6.5). Therefore, the tenant XML data contained in the incoming E-mail data must be parsed before the creation of the NM. We modify the *org.apache.servicemix.mail.utils.MailUtils* to parse and convert the tenant context xml data into NM properties, and to set the destination E-mail addresses from the tenant context

information. The *DefaultMailMarshaler* sets a NM property which contains the received XML E-mail body (see Listing 6.5). For this reason, we should set the NM content with the data received as E-mail body in the consumer endpoint marshaler. In the provider marshaler we extend the *org.apache.servicemix.mail.marshaler.DefaultMailMarshaler* in order to set the list of destination E-mail addresses in the *To* NM header.

6.4 Multi-tenant Service Engine

The integration of the multi-tenant ESB with the taxi scenario described in Figure 6.2 contains a multi-tenant Camel SE which routes messages between tenant-aware endpoints. We describe in this section the approaches that Muhler has implemented for the multi-tenant Camel SE [Muh12].

6.4.1 Apache Camel

The Apache Camel router shipped with ServiceMix is non multi-tenant aware [APA11a]. The internal routing in Camel is done between Camel URIs, which are transformed to JBI service endpoint names in order to provide compatibility with JBI endpoints. The transformation is performed in the Java class *org.apache.servicemix.camel.CamelConsumerEndpoint*. Muhler has modified the implementation to inject tenant information during the transformation operation, and provider and consumer endpoint creation operations [Muh12]. In the taxi application we configure two tenant-aware Camel endpoints in the multi-tenant Camel SE. The incoming message is enriched with tenant information and forwarded to the Web service provider endpoint.

7 Test

In order to ensure that we fulfill the requirements listed in Chapter 4, in this chapter we provide a testing of the main components we implemented and we describe how we initialize the different testing scenarios. In Section 7.2 we provide three different testing scenarios, one per extended BC, and we monitor both incoming and outgoing messages to and from the ESB.

7.1 Deployment and Initialization

The components conforming the taxi application are deployed in a single virtual machine. The tests performed in Section 7.2 test each of the multi-tenant BC separately. We could not integrate the JMS and E-mail communication protocols between the taxi application components due to the support of only SOAP over HTTP messages between the components. Therefore, we can divide the testing scenarios in two: the taxi application v2.0 prototype and the individual multi-tenant BCs.

The taxi application v2.0 integrates various components which are deployed in a JOnAS 5.3 and JOnAS 5.2 server. It differs from the first version in the use of one server for the deployment of the taxi companies and taxi transmitter Web interfaces, JBIMulti2 application, Orchestra processes, and CMF and GoogleDirections components, instead of the usage of two servers: JOnAS 5.2 and Apache Tomcat 7.0.23 [OWJ], [ATC]. A JOnAS 5.2 instance hosts the JBIMulti2 EAR package and configures the database connections to the PostgreSQL 9.1.1 middleware [Muh12], [PSQ]. A JOnAS 5.3 instance hosts the two taxi companies and taxi transmitters Web interfaces, Orchestra processes, and CMF and GoogleDirections components. These could not be deployed in the same server instance as the JBIMulti2 application due to a class loading failure Muhler also denotes. Therefore, in the instance hosting these components we must deactivate the JAXWS service which loads the classes that conflict with the ones included in the deployed packages. We have followed Muhler's guide for installing the JBIMulti2 application under JOnAS 5.2, and its connection to the Apache ServiceMix 4.3.0 through the JMSManagementService OSGi bundle and the Apache ActiveMQ 5.3.1 shipped with the ServiceMix package [Muh12], [ASM], [AMQ]. For the BPEL processes we follow Hagin's installation guide in Orchestra [Hag11]. The endpoint configuration for the routes between the processes, and the CMF and GoogleDirections connectors are packed in two SAs and deployed in the ServiceMix *deploy* directory. The deployment of the SOAP over HTTP consumer endpoints must be done through the JBIMulti2 application. For this reason, we must use its Web service interface for tenant and endpoint administration. We use SoapUI 3.6 [SOA] for tenant and user creation, JBI BCs deployment, quota and contingent assignment,

deployment of tenant-aware endpoints packed in a SAs, etc. These operations are included in a SOAP test suite Muhler has created and we extend in this thesis.

However, due to the lack of a multi-protocol support in most of the components of the taxi application, we were enforced to test the multi-tenant BCs separately. For these tests we have adjusted a minimum of two tenants per multi-tenant BC. Tenant administration and SA deployment is done using the JBIMulti2 application. We install a second instance of Apache ActiveMQ the provider endpoints are connected to, as the ActiveMQ broker provided with ServiceMix has no administration Web interface. We have also deployed a backend echo Web service in Tomcat 7.0.23 for testing the multi-tenant HTTP BC using SoapUI 3.6 [ATC], [SOA]. The mail BC requires the creation of one E-mail account per tenant. These are created under the GMail domain [Goo12].

7.2 Multi-tenant Binding Components

In this section we explain the individual tests which were run on each multi-tenant JBI BC. The deployment of the JBI BCs and the tenants' endpoint configuration first requires tenant administration operations provided by the JBIMulti2 application, and included in the test suit we have extended. Although we include screenshots with information of two tenants, we run the tests for a minimum of 2 tenants in the application. For all the tenants, we deploy the same endpoint configuration packed in a SA and sent to the JBIMulti2 application as a base64 string.

7.2.1 SOAP over HTTP

This testing scenario requires a backend Web service. As described in this Chapter, we implemented and deployed in Tomcat 7.0.23 two simple echo Web services: one implements an In-Only MEP and one implements an In-Out MEP. Their WSDL must be included as a file in both HTTP endpoint configurations in the SUs before deployment. For monitoring purposes, we install Wireshark 1.2.7 to listen to the HTTP POST messages to and from the Web service [Fou]. The SOAP messages are sent using test suites in SoapUI 3.6 to the tenant-aware endpoint in the ESB [SOA]. In the test shown in Figure 7.1 we test the In-Out MEP Web service for the tenant *TaxiCompany*. As SOAP headers we include the tenant context information (tenantId and userId) for authentication, and as SOAP body the taxi transport details. The Web service responses with the same information received in the SOAP body.

7.2.2 JMS

The test scenarios for the XML over JMS communication protocol have been done for both approaches described in Chapters 5 and 6. We have installed locally the HermesJMS application and configured a remote connection to the ActiveMQ 5.3.1 broker in ServiceMix [her], [AMQ]. For the first approach, we connect to a shared *TaxiRequest.queue* or *TaxiRequest.topic*. The

former is consumed by a non multi-tenant JMS endpoint and routed to the *tenant-aware router*. We have noticed while testing that the database connection slows down the performance of the Camel Recipient List Router, leading to the existence of a bottleneck. ServiceMix has four status for a JMS message exchange: *delivered*, *pending*, *dispatched*, and *inflight* [ASM]. Many of the messages were set with the *inflight* state, which means that the message exchange is still pending, and this state blocked further exchanges until its completion. The testing of the second approach gave us much better performance results. In this second approach, as shown in Figure 7.2, one consumer and provider queue or topic is created per tenant. The ActiveMQ 5.3.1 shipped with ServiceMix does not have the administration Web interface. For monitoring purposes, we have installed a separate instance of ActiveMQ 5.2.0 in order to be able to easily monitor the messages stored in the provider queue. The tenant context information stored as a JMS property is parsed at the consumer side and is set at the provider side as a JMS property.

7.2.3 E-Mail

A tenant-aware E-mail endpoint consumes or sends E-mail messages by connecting to the E-mail server of the tenant. The tenant's E-mail address must be provided in endpoint configuration file packed in the SA. In the test shown in Figure 7.3 we have connected via IMAP and SMTP the *TaxiCompanyB* E-mail endpoint with a mail box in Gmail [Goo12]. To set a custom header in the E-mail we use Mozilla Thunderbird v13.0.1 mail application [Moz]. The E-mail message sent to the tenant-aware provider mail endpoint is sent to the E-mail addresses contained in the tenant context header of the incoming E-mail message.

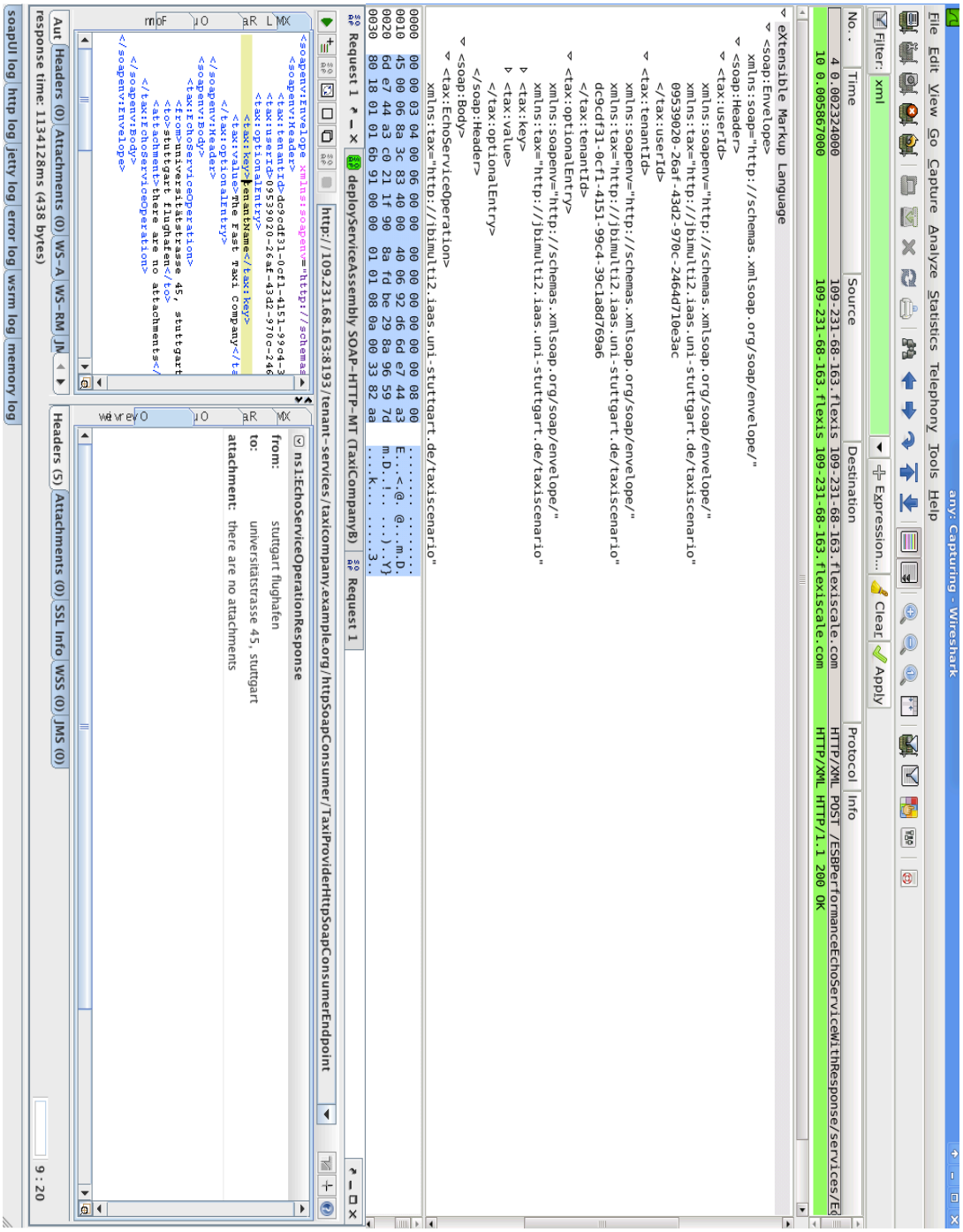

Figure 7.1: Multi-tenant SOAP over HTTP test overview for tenant TaxiCompany. Wireshark v1.2.7 used for listening of incoming messages to the echo Web service and SoapPUI 3.6 for invokation [Fou], [SOA].

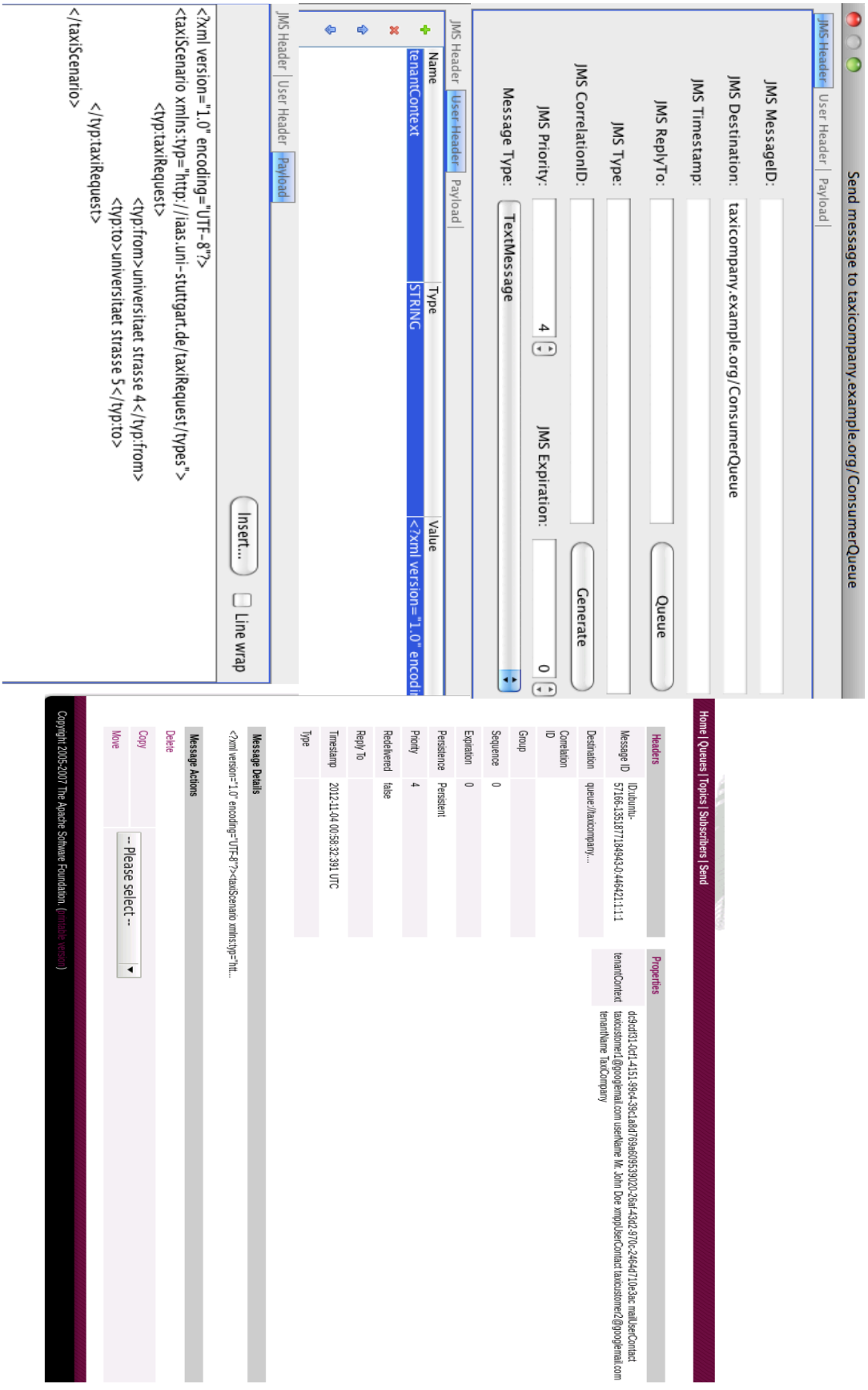

Figure 7.2: Multi-tenant XML over JMS test overview for tenant TaxiCompany. Provider queue created in separate instance of ActiveMQ 5.2.0 and visualized with its Web admin interface [AMQ].

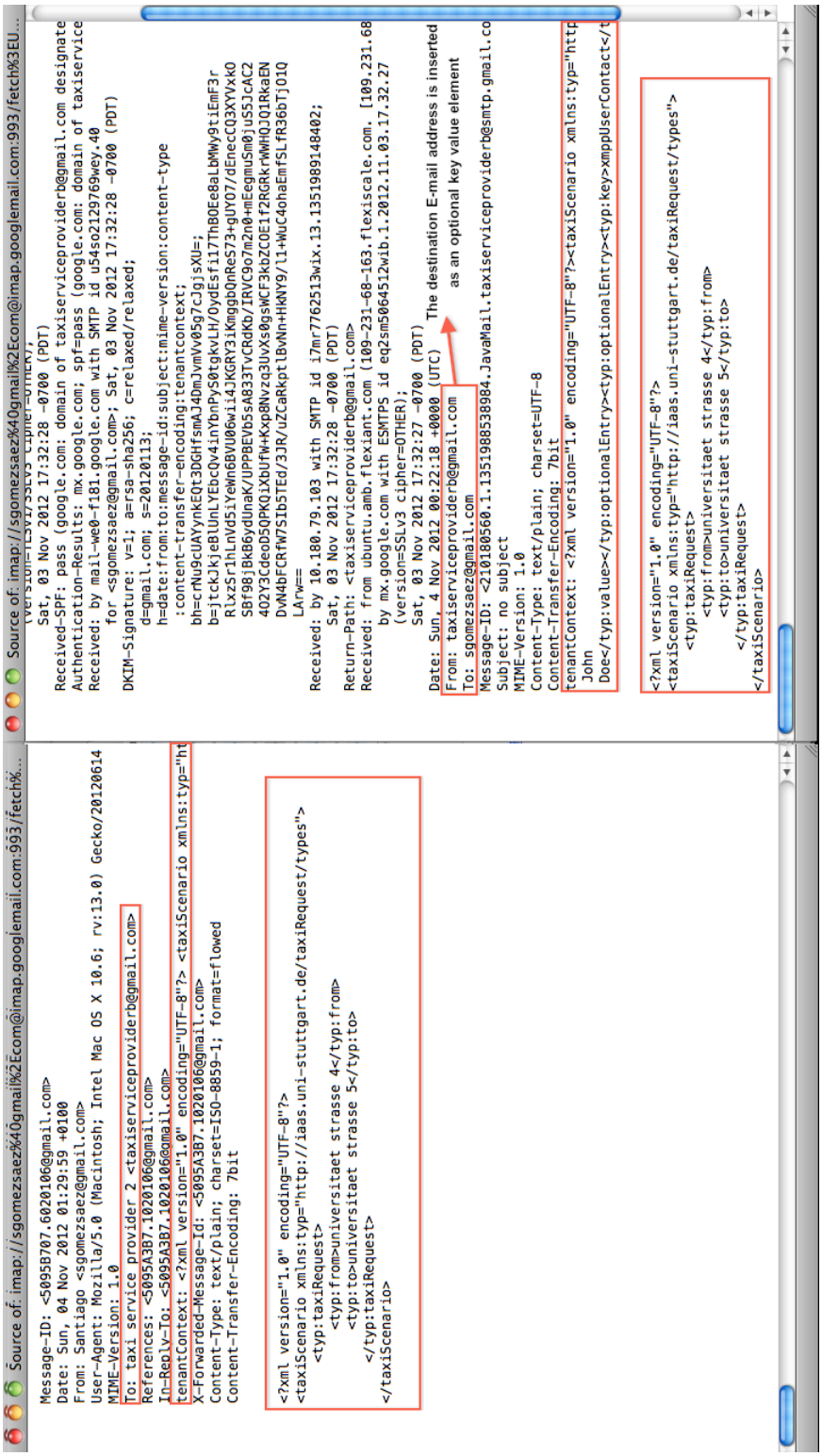

Figure 7.3: Multi-tenant XML over E-mail test overview for tenant TaxiCompanyB. Message is routed to the E-mail addresses specified as key-value pairs in the E-mail tenant context header.

8 Performance Evaluation

In the following sections we provide the requirements which should fulfill the performance evaluation of the ESB solution we extend for multi-tenancy awareness. For this purpose, we extend a load generator developed by AndroitLogic to enable it with tenant-aware messaging [Ltd12]. Finally, we provide an overview of the components and systems involved in the different scenarios of the evaluation, and the design approach and implementation of the extension.

8.1 Specification

8.1.1 Evaluation Requirements

In this student thesis we provide a performance analysis on the integration of the multi-tenant aware approaches in ServiceMix and measure the impact on the performance of the extended prototype. For this purpose, we need to fix which measurements we use for the evaluation. The driver should perform the following measurements: response time (measured in milliseconds) and throughput (measured in number of messages sent per second) respect to a backend service, and CPU and memory usage of the system hosting the instance of ServiceMix. The evaluation has to be done in different scenarios, each of them sending different messages number and sizes, for different multi-tenant and non multi-tenant aware endpoint configurations, as described in Table 8.1 [SASL12].

Number of Endpoints	Messages Size	ServiceMix Instances	Multi-tenancy awareness
1	0.5 / 1 KB	1	mt and non-mt
		2	non-mt
2	0.5 / 1 KB	1	mt and non-mt
		2	non-mt
4	0.5 / 1 KB	1	mt and non-mt
		2	non-mt
10	0.5 / 1 KB	1	mt and non-mt
		2	non-mt

Table 8.1: Specification of the different scenarios to be evaluated. In both multi-tenant and non multi-tenant aware evaluations, one user per endpoint / tenant is configured.
Legend: mt (multi-tenant aware), non-mt (non multi-tenant aware)

AndroitLogic has developed a performance analysis driver which fulfills most of the above requirements in different scenarios [Ltd12]. In our evaluation, we extend the Direct Proxy scenario from the AndroitLogic ESB Performance benchmark [Ltd12]. However, it doesn't achieve one of the main requirements of this student thesis: multi-tenant aware messaging and concurrent invocation between endpoints. Those two requirements should be included in an extended version of the primitive driver. Furthermore, the extension should be utilized with different ESB solutions and must be user-friendly configurable for the different scenarios. The output of the driver measurements should be analyzed, therefore the output data must be in structured format.

8.1.2 Evaluation Overview

In the Section 4.1 we have described the requirements that the evaluation should fulfill and the needed modifications in the utilized benchmark. As exposed in Figure 8.1, the evaluation is conformed by three main independent systems. We must ensure, for analyzable purposes, that we approximate as much as possible to a Web service standard real scenario: service requester invokes a backend service and both request and response are routed through the network. In our evaluation we must utilize the ESB as a mediator between the service requester and provider. In the first system (VM0), both service requestor and provider are deployed. The communication measurements are taken in two different components: throughput and response time in the AndroitLogic driver, while the number of incoming and outgoing requests, as well as the visualization of the messages, have to be monitored in an independent monitoring component.

In the second and third systems (VM1 and VM2 respectively) one instance of ServiceMix is deployed for routing the messages between the AndroitLogic driver and the backend service. A monitor component must perform the counting of the incomming and outgoing requests to and from the ESB, and a system monitor component should measure the ESB's resources consumption. The connection between the components in VM0 and VM2 is represented with a dashed line, because the VM2 is only use for non multi-tenant aware scenarios. Similarly, we have connected the components in VM0 with the components in VM1 with a continuos line, because this connection is used in both multi-tenant and non multi-tenant aware scenarios.

The JBIMulti2 application is used for deployment of the SAs which pack the endpoint configurations in the SUs. However, we do not include the JBIMulti2 application in our overview because we do not evaluate the JBIMulti2 performance, but the multi-tenant and non multi-tenant ServiceMix independently from the JBIMulti2 application.

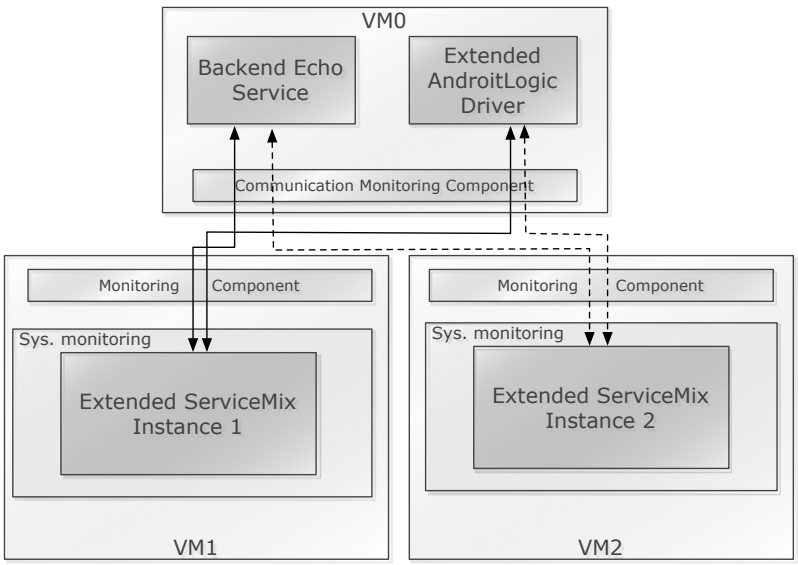

Figure 8.1: Overview of the components used for the ESB performance evaluation. **Note:** In the evaluation two different monitors are used. For communication the monitoring requires the counting and visualization of the incoming and outgoing requests. For system monitoring, the CPU and Memory usage should be measured.

8.2 ESB Performance Evaluation Architecture

AndroitLogic has developed in their ESB Performance Evaluation Round 3 a load generator for different scenarios. After analyzing its main features, we found it suitable for our work, but only if we can include tenant-awareness in the execution. We evaluate the SOAP over HTTP communication protocol in both native ServiceMix HTTP BC and in the multi-tenant HTTP BC. With this we want to evaluate not only the performance of the ESB solution we are using in our Cloud infrastructure, but also the penalty caused by the multi-tenant awareness implementation. The SOAP over HTTP protocol is well known for its usage in Web services. In this evaluation we use as a backend Web service an Echo Service which logs the received requests. For this purpose, we must push the scenarios as close as possible to a real Web service consumption. Therefore, we divide the evaluation system in two virtual machines connected by a network (see Figure 8.2).

The virtual machine one hosts the front and backends components: performance benchmark and the Web service. The Web service is deployed in an Apache Tomcat server. The extended performance benchmark is built of the following components: AndroitLogic driver, shell scripts and data converters. The AndroitLogic driver support concurrent users invoking the same endpoint, but not concurrent users between two or more endpoints. Furthermore, it does not support message modification for including tenant information. For this purpose, we have designed the shell scripts which can give support on those two requirements (see Figure 8.2). In the first place, the shell script modifies or does not modify the message which will be sent by the driver. In the second place, we perform concurrent invocations between

endpoints by creating several Unix background tasks of the driver. Each of the tasks results can be dumped in a shared file between the driver instances. However, the results come in non structured format for analysis. Therefore, we convert the data using a converter provided by AndroitLogic [Ltd12]. For monitoring the packet lost rate, we will listen on the server's port where the Web service listens with a well known monitoring tool, Wireshark [Fou].

We use the virtual machines two and three for hosting the ServiceMix instances. The two instances are used only in non multi-tenant scenarios. For both multi-tenant and non multi-tenant scenarios we must increase the number of concurrent calls to the endpoints. In the requirement we specify scenarios of one, two, four, and ten endpoints. The system performance measurement can be done by system commands. We provide a component which take CPU and Memory measurements and converts its output to structured data for analysis. However, the system memory usage measurements do not give variable percentages over time. The percentage shown is the one associated with the memory consumption of the JVM the ESB runs on, which is previously reserved and fixed over time. To get more representative data, we measure the heap consumption of ServiceMix in the JVM using Java Console, which give us a better representation of the variability between the different scenarios (See Figure 8.2). For monitoring the communication, an instance of Wireshark can also be used, but in our evaluation it is optional.

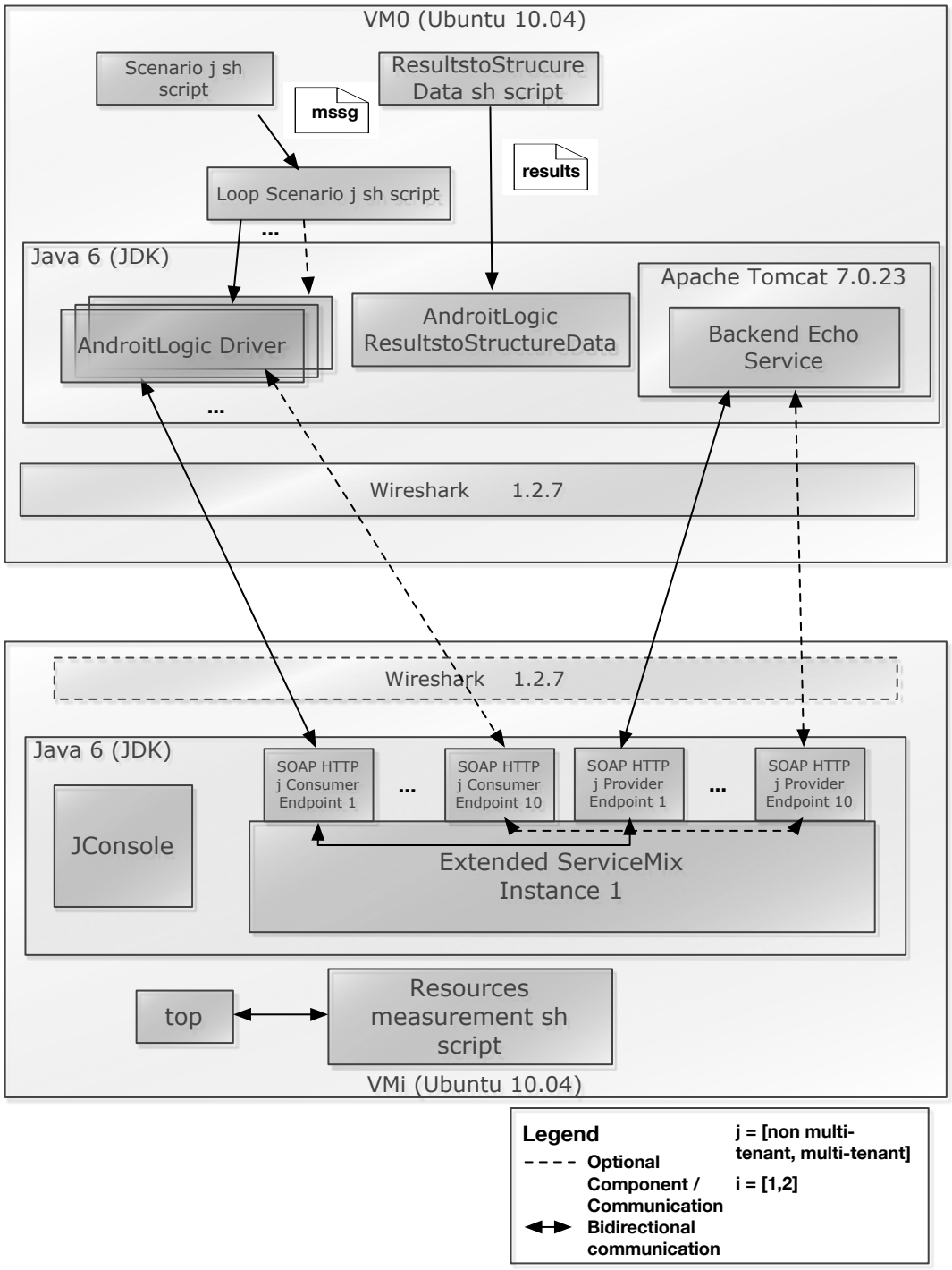

Figure 8.2: Architectural overview of the components used for the evaluation of the ESB performance. **Note:** We evaluate only ServiceMix, not the integrated version of ServiceMix with the JBIMulti2 application, in order to be able to perform a direct comparison between the multi-tenant and the non multi-tenant ServiceMix.

8.3 Evaluation

In this section we first describe the components we implemented to be able to reuse the existing load performance driver from AndroitLogic ESB Performance Round 6 and adapt it to the required multi-tenant scenarios [Ltd12]. In Section 8.3.2 we shortly describe the deployment and initialization in the virtual machines under the Flexiscale network [Ltd] and we evaluate the results obtained from the different scenarios.

8.3.1 ESB Performance Evaluation Benchmark

AndoitLogic has developed an ESB Performance Benchmark in their Round 6 [Ltd12]. Their benchmark evaluates the ServiceMix behavior under different scenarios, which vary in the request message size, number of requests per endpoint, and number of concurrent users per endpoint. Their load generator is included in the *org.adroitlogic.toolbox.javabench* package in their Management Toolbox application [Ltd12]. They provide an HttpBenchmark and a *data to csv format* converter. The former makes HTTP POST requests to the specified endpoint URL, and generates performance statistics, e.g. response time, throughput. The latter converts the driver's results to structured data for analysis. We reuse both components, but develop components on top which create several instances of the *javabench* to invoke multiple endpoints concurrently. Furthermore, we need to add tenant context information before invoking the driver. The components are implemented in UNIX shell scripts.

In Figure 8.3 we describe the overall structure of our loader package. We define three scenarios: scenario 1 is non multi-tenant aware and with one instance of ServiceMix, scenario 2 is non multi-tenant aware with two instances of ServiceMix, and scenario 3 is multi-tenant aware with 1 instance of ServiceMix [SASL12], [SAS+12]. The SOAP messages and the endpoints' URLs are read from XML files stored in the *scenariox* folder. The messages vary from 500B to 100K. However, in this student thesis we perform the analysis for the 500B and 1K messages' sizes. The main and secondary scripts used in this analysis are located under the main scripts folders and the results generated are stored in the *ResultsScenariox* folder. In addition, for the multi-tenant scenarios, the endpoint URL file must also contain the tenant context information, e.g. tenantId and userId.

For the evaluation scenarios we create using the ServiceMix HTTP maven archetype 11 SUs. Ten SUs contain non multi-tenant endpoint configurations of 10 consumer and 10 providers which support the SOAP over HTTP communication protocol, while one SU is configured to be deployed on the multi-tenant HTTP BC by 10 tenants, generating 10 tenant-aware endpoints (10 tenant-aware consumer and provider endpoints). The deployment phase is described in Section 8.3.2.

```
1 Load test scenario 1 and 2 shell script (non multi-tenant awareness):
2   Usage : loadtest_scenario1 scenario1|2 number_endpoints size_message(0.5K,1K) output_file
3 Load test scenario 3 shell script (multi-tenant awareness):
4   Usage : loadtest_scenario2 scenario3 number_endpoints size_message(0.5K,1K) output_file
```

Listing 8.1: Invocation parameters in the main shell scripts of the benchmark.

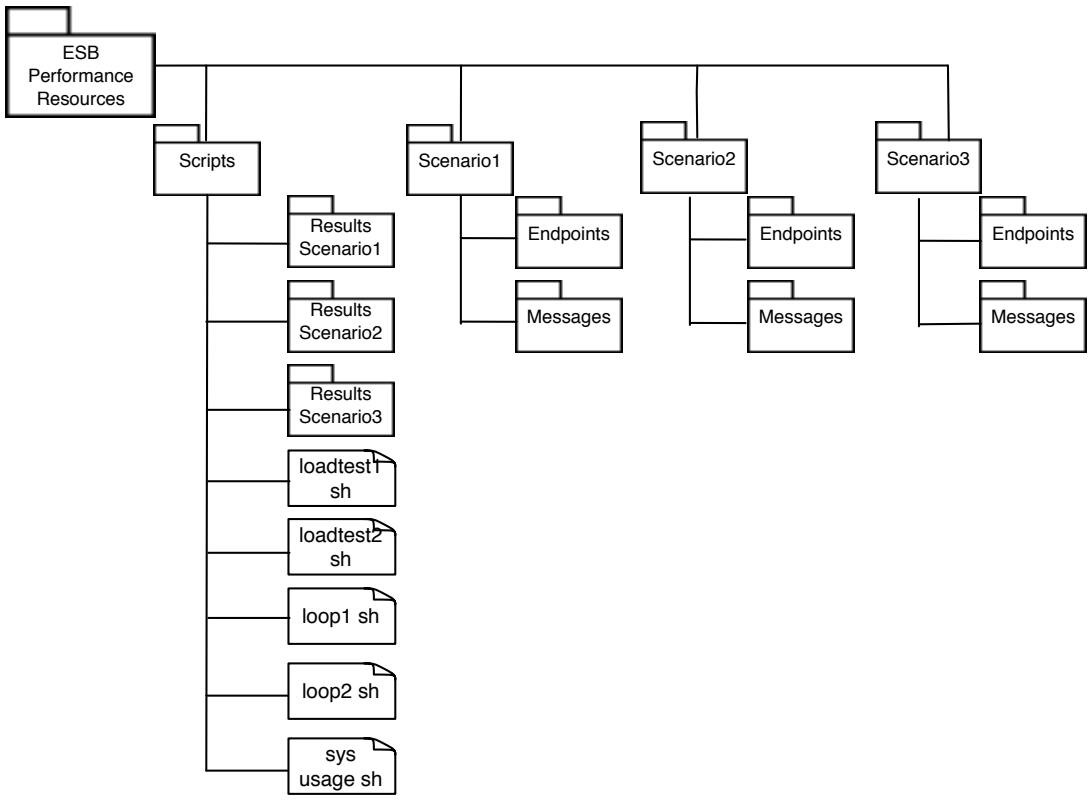

Figure 8.3: Overview of the package used for the evaluation of the ESB performance

For the non multi-tenant scenarios, we specify the needed invocation parameters in Listings 8.1, 8.2, and 8.3. The non multi-tenant aware shell script takes as one of the input parameters the file path where the SOAP message is stored and creates the result scenario file name based on the message size and the running scenario. Furthermore, the endpoints' URLs are retrieved from the endpoint configuration file and the script reads as many URLs as tenants we specify. For supporting concurrent calls between endpoints, we create as many instances of the *javabench* as endpoints by running a secondary script, the loop script (see Figure 8.3), which is executed as a Linux background task. The scenario is divided in two phases: warm-up and performance test. In the warm-up phase an amount of requests are concurrently sent to the endpoints in order to warm-up the ESB and get consistent results, while in the performance test phase we perform the measurements which can be analyzed. In the warmup phase the initial request number is set to 10240 requests. This value is then increased exponentially, and divided by the number of endpoints, as described in the following explanation. The loop script increments exponentially the number of requests and divides it by the number of endpoints in the ESB, e.g. during the performance test phase if an initial request number is set to 20, then it sends to each endpoint $20 * 2^i, i = [1, 2, 3, 5, 6]$ divided by the number of invoked endpoints in the scenario. The same sequence is applied for the multi-tenant scenario.

```
1 Loop scenario 1 and 2 (non multi-tenant aware):
2 Usage : loop_scenario1 iterations_loop  message_file  number_initial_requests
```

```
    number_concurrent_tenants  soap_action_header  url_endpoint
3
4   [iterations_loop] :            number of iterations the loop will be run.
5   [message_file] :               path to the file containing the SOAP message.
6   [number_initial_requests] :    initial number of requests.
7   [number_concurrent_tenants] :  number of concurrent endpoints to send requests.
8   [soap_action_header] :         SOAP action header.
9   [url_endpoint] :               endpoint's URL.
```

Listing 8.2: Invocation parameters in the non multi-tenant shell script of the benchmark.

The approaches taken into account for the non multi-tenant scenarios are similar to the multi-tenant scenario, as it is shown in Listing 8.2. The main difference relies on the need of tenant context information injection before the SOAP message is read and sent over HTTP concurrently to the tenant-aware endpoints. The concurrent reading of messages leads us to the need of using temporal files to store the messages containing tenant context information. The temporal files are created in the same directory as the requests files, and we modify the content by reading the original request in the loop2 script shown in Listing 8.3 and using the Linux *sed* command to inject the tenant context information read from the endpoints' URL definition file.

```
1  Loop scenario 3 (multi-tenant aware):
2  Usage: loop_scenario2 scenario phase 1 message_file number_initial_request
       number_concurrent_tenants soap_action_header url_endpoint tenantid userid size_message
3
4    [scenario]:                  scenario number. In this evaluation, scenario3.
5    [phase] :                    warmup or testESB.
6    [message_file] :             path to the file containing the SOAP message.
7    [number_initial_request] :   initial number of requests.
8    [number_concurrent_tenants] : number of concurrent endpoints to send requests.
9    [soap_action_header] :       SOAP action header.
10   [url_endpoint] :             endpoint's URL.
11   [tenantid] :                 tenantId.
12   [userid] :                   userId.
13   [size_message] :             message size.
```

Listing 8.3: Invocation parameters in the multi-tenant shell script of the benchmark.

With the above approaches we fulfill the requirement of getting the response time and the throughput when invoking and external Web service through an ESB. We have also implemented scripts for monitoring the system resources with the *top* Linux command and with the Java Console [Ora]. The former provides data at the system level, e.g. JVM memory consumption, while the latter provides data about the exact amount of JVM resources the ServiceMix process consumes.

8.3.2 ESB Performance Evaluation Analysis

In this student thesis we extend and add extra functionality to ServiceMix. Therefore we need to measure its behavior and compare it with the baseline, which is the non multi-tenant version of Apache ServiceMix 4.3.0 [ASM]. In the following sections we describe the systems

we use for evaluating the performance of the ESB, and we discuss some of the results we obtained.

Deployment and Initialization

The different scenarios we run in this evaluation must approximate as much as possible to real scenarios. Thus we utilize three Ubuntu 10.04 virtual machines in this evaluation connected by the network in the Flexiscale infrastructure [Ltd]. One virtual machine hosts the evaluation package and an echo Web service which implements the In-Only MEP and is deployed in Tomcat 7.0.23. Wireshark 1.2.7 is installed for monitoring incoming and outgoing requests to and from the echo Web service. In a second and third virtual machine we install one instance of ServiceMix 4.3.0 and the system resources measurement scripts. We have listed in Section 8.1 a set of multi-tenant and non multi-tenant scenarios. For the non multi-tenant scenarios we have deployed in the ServiceMix *deploy* directory a SA containing the configuration of 10 consumer and 10 provider endpoints. For the multi-tenant scenarios we performed the operations described in Section 7.2 and deploy through JBIMulti2 10 tenant-aware consumer and 10 tenant-aware provider endpoints. Both tenant-aware and non tenant-aware endpoints must be specified in the endpoint file the extended driver reads for sending the SOAP requests.

Evaluation Analysis

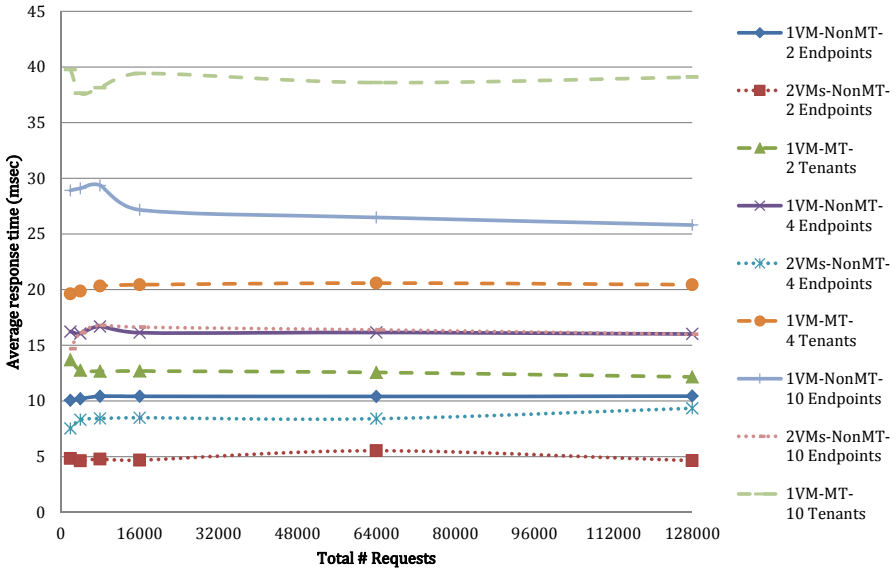

Figure 8.4: Response time of the different scenarios for 1KB SOAP over HTTP requests [SASL12].

The numerical values obtained from the communication driver are distributed per endpoint. Therefore, we make the average between the endpoints for each set of sent requests. The

latency when increasing the number of endpoints in the system increases between 25% and 45% for concurrent requests sent to 2, 4 and 10 endpoints (see Figure 8.4). We can see that our extended version declines the original ServiceMix's performance approximately 30% [SASL12]. Furthermore, we can observe that the response time for the different multi-tenant scenarios does not increase proportional to the number of tenants, but shows a better performance per endpoint. When distributing the requests between two ServiceMix instances we have observed that the response time is substantially reduced when increasing the number of endpoints.

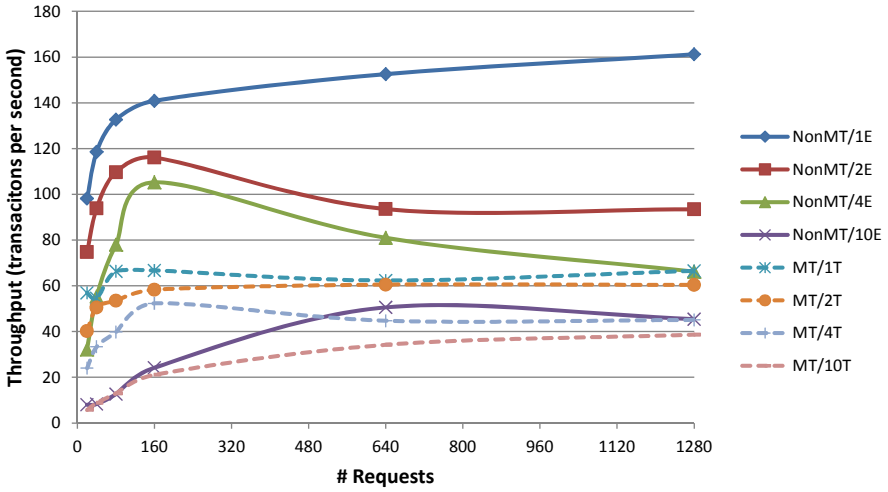

Figure 8.5: Throughput of the different scenarios for 1KB SOAP over HTTP requests [SASL12].

When measuring the throughput, the analysis took us to the same deduction we discussed above. When the number of endpoints decreases, we can observe a bigger gap between the native and the extended version of ServiceMix. As we increase the number of endpoints, e.g. 10 endpoints, we can see that the gap between approaches is exponentially narrowed (see Figure 8.5).

Finally, we have analyzed the CPU consumption of the process running the ServiceMix instance. We can observe in the average CPU consumption that there is a gap between both multi-tenant and non multi-tenant scenarios which increases when the number of endpoints increases. However, when increasing the number of endpoints the average CPU consumption stabilizes and maintains close to a constant value. The maximum CPU consumption values decreases in the non multi-tenant scenarios when increasing the number of endpoints, while for the multi-tenant scenario it linearly increases from two endpoints. In the 4 or 10 endpoints scenarios, the maximum CPU consumption difference between the native and the extended approaches are between 250% and 320% approximately (see Figure 8.6).

The tenant authentication procedure demands a greater number of resources when parsing the SOAP headers and the XML data from the tenant context file. However, this decline caused by the implemented authentication procedure and the marshaling of tenant context information within the process is much lower that the decline caused by establishing a connection with

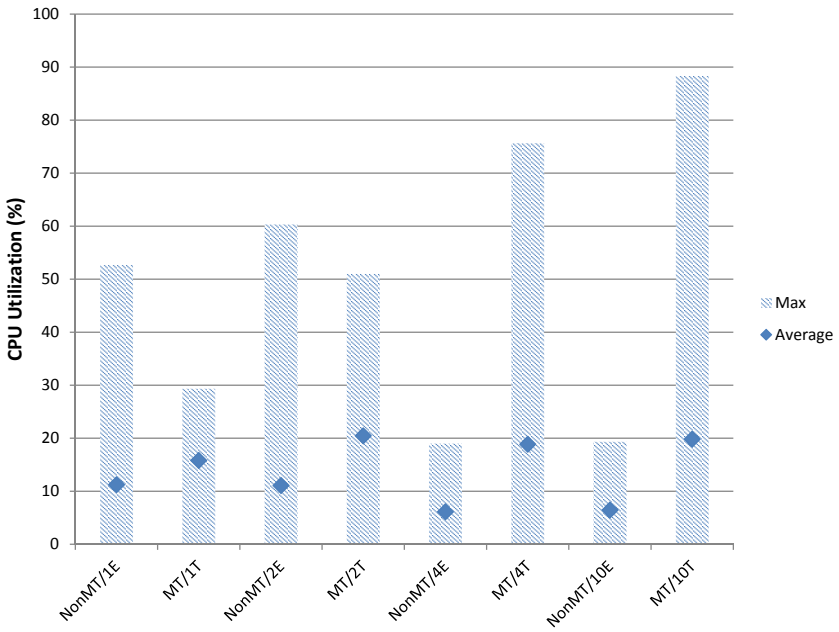

Figure 8.6: Overview of the CPU consumption over the different scenarios [SASL12].

an external database system and retrieving the tenant context information from an external source to the process.

9 Outcome and Future Work

The utilization of an ESB as the main piece of middleware for SOA in a Cloud environment forces multi-tenancy awareness to be a must in its requirements. This student thesis integrates the two main approaches for enabling multi-tenancy in an open source ESB: multi-tenant aware messaging and multi-tenant aware administration and management, as well as analyzes and compares the performance of the native and extended ESB solution in different scenarios, and produces as its main outcome an integrated version of the taxi application [4Ca].

In Chapter 2 we first provide the needed background on the technologies, communication protocols, and the main components this student thesis work with: ServiceMix and JBIMulti2 [ASM], [Muh12]. After acquiring the main knowledge of the solutions, we investigate in Chapter 3 different solutions which support multi-tenancy, and analyze approaches which have been already taken into account. Furthermore, we discuss the supported functionalities of the AndoitLogic load generator driver, and the possibility of its reuse in our performance analysis. The identification of requirements and the system overview presented in Chapter 4 guide us to perform the design of the different components of this student thesis in Chapter 5. The design leads to a multi-tenant and multi-protocol aware version of ServiceMix, supporting three communication protocols: SOAP over HTTP, JMS, and E-mail. Furthermore, we provide an integration design for the taxi application v2.0 prototype and we reengineer most of the implemented communication approaches in order to improve the system's performance and to add new functionalities, e.g. tenant context data structure modification, tenant authentication, and tenant-aware isolated endpoints in the ESB. One of the main requirements in a Cloud infrastructure is security. We implement tenant authentication but not tenant data integrity and confidentiality. The tenant context information sent to and from the ESB must be encrypted in future versions of the ESB.

Chapter 6 describes the challenges and approaches we faced for both the integration with the taxi scenario and the extension of the different JBI BCs. As discussed in Chapter 6, one of the main goals we have in the improved version of the taxi application is to maximize the ESB usage between components and to integrate a multi-tenant aware ESB with non multi-tenant aware components which build part of the taxi application, e.g. BPEL processes under Orchestra, CMF and GoogleDirections components, etc. This contrast forced us to perform changes in some of the taxi application components in order to adapt multi-tenancy at the communication level. Future versions of the taxi application should support multi-tenancy awareness in its components, and we consider that a bidirectional connection between JBIMulti2 and the taxi companies Web interfaces should be set in order to retrieve the tenant context information. Furthermore, the only communication protocol which actually supports the taxi application is the SOAP over HTTP. We have extended both JMS and Mail JBI BCs

for supporting a multi-protocol communication between customers and taxi drivers in a future version of the taxi application. This aspect directed us to build two different testing environments, one for the taxi application and one for the individual testing of the extended BCs described in Chapter 7.

For analytical purposes after implementation, we perform an evaluation of the performance in Chapter 8 of both native and extended versions of ServiceMix. This student thesis reuses and extends an existing SOAP over HTTP ESB performance benchmark. We adapt the benchmark to support multi-tenancy and evaluate the obtained results from different scenarios. However, we could not perform this analysis on more that one communication protocol. In the future it would be interesting to run the same scenarios on the XML over JMS communication protocol. Those results can give the ESB administrator a better output for offering the communication protocols which best execute in our ServiceMix version. Furthermore, we perform the evaluation of one important scenario in a Cloud infrastructure: dividing the load between more than one ServiceMix instance by emulating a load balancer. The results showed that the performance is not significantly increased with the increase of the number of endpoints, and this approach can be not worth its expenditure. However, as we discussed, we emulate load balancing. For more than one instance of ESB a load balancer should be integrated to the extended system.

Finally, we have integrated a multi-tenant ESB which connect different endpoints via different protocols, e.g. external consumers with external providers. However, data is nowadays the most important asset of any business [FC06]. The offering of a data-as-a-service solution in a Cloud environment where data can accessed through SOA mechanisms, ables the use of the ESB as a data access layer. With this approach data can be accessed from everywhere just by communicating with the ESB and without worrying about the underlying architecture, e.g. database vendor, connection drivers.

Bibliography

[4Ca] 4CaaSt – EU Project. http://www.4caast.eu/.

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

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

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

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

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

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

[BPSM+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (Fifth Edition), November 2008. http://www.w3.org/TR/REC-xml/.

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

[CCA] EU ICT Project Context Casting (C-CAST). http://www.ict-ccast.eu/.

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

[Ess11] S. Essl. Extending an Open Source Enterprise Service Bus for Multi-Tenancy Support. Master's thesis 3166, Institute of Architecture of Application Systems, University of Stuttgart, 2011.

[FC06] R. W. Frederick Chong, Gianpaolo Carraro. Multi-Tenant Data Architecture, June 2006. MSDN, http://msdn.microsoft.com/en-us/library/aa479086.aspx.

[Fes12] F. Fest. Extending an Open Source Enterprise Service Bus for Horizontal Scalability Support. Diploma Thesis 3317, Institute of Architecture of Application Systems, University of Stuttgart, 2012.

[Fou] W. Foundation. Wireshark 1.2.7. http://www.wireshark.org/.

[GGW+09] B. Gao, C. J. Guo, Z. H. Wang, W. H. An, and W. Sun. Develop and Deploy Multi-Tenant Web-delivered Solutions using IBM middleware: Part 3, March 2009. IBM, http://www.ibm.com/developerworks/webservices/library/ws-multitenant/.

[GMA] Google Maps API Web Services. http://code.google.com/intl/en/apis/maps/documentation/webservices/.

[Goo] Google. Google Guice Dependency Injection Frame. http://code.google.com/p/google-guice/.

[Goo12] Google. GMail, 2012. www.googlemail.com.

[Gro05] N. W. Group. A Universally Unique IDentifier (UUID) URN Namespace, July 2005. http://tools.ietf.org/html/rfc4122.

[Gro08] N. W. Group. Internet Message Format, October 2008. http://tools.ietf.org/html/rfc5322.

[Hag11] R. Hagin. Enabling Integration and Aggregation of Context Information into WS-BPEL Processes. Master's thesis, Institute of Architecture of Application Systems, University of Stuttgart, 2011.

[HBS+02] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout. Java Message Service (JMS) 1.1 Final Release JSR-914, 2002. http://jcp.org/aboutJava/communityprocess/final/jsr914/.

[her] Hermes JMS 1.14. http://www.hermesjms.com/confluence/display/HJMS/Home.

[JBI05] Java Business Integration (JBI) 1.0, Final Release, 2005. JSR-208, http://jcp.org/aboutJava/communityprocess/final/jsr208/.

[Ltd] F. Ltd. Flexiscale Cloud Infrastucture. http://www.flexiscale.com/.

[Ltd12] A. P. Ltd. ESB Performance - Latest Execution - Round 6, August 2012. http://esbperformance.org/display/comparison/ESB+Performance.

[Moz] Mozilla. Mozilla Thunderbird v13.0.1. https://www.mozilla.org/en-US/thunderbird/.

[Muh] D. Muhler. Manual for the JBIMulti2 Implementation. Modified by Santiago Gomez.

[Muh12] 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.

[MUTL09] R. Mietzner, T. Unger, R. Titze, and F. Leymann. Combining Different Multi-tenancy Patterns in Service-Oriented Applications. In *Proc. IEEE Int. Enterprise Distributed Object Computing Conf. EDOC '09*, 2009.

[NIS11] National Institute of Standards and Technology. The NIST Definition of Cloud Computing, 2011. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[OPG06] The Open Group. The SOA Work Group: Definition of SOA, 2006. http://www.opengroup.org/soa/soa/def.htm.

[OPG11] The Open Group. IBM Cloud Computing Reference Architecture 2.0, 2011. https://www.opengroup.org/cloudcomputing/uploads/40/23840/CCRA.IBMSubmission.02282011.doc.

[Ora] Oracle. Java Console. http://docs.oracle.com/javase/6/docs/technotes/guides/management/jconsole.html.

[OSG11] OSGi Alliance. OSGi Service Platform: Core Specification Version 4.3, 2011. http://www.osgi.org/Download/Release4V43/.

[OWJ] OW2 Consortium. JOnAS: Java Open Application Server. http://wiki.jonas.ow2.org/.

[OWO] OW2 Consortium. Orchestra: Open Source BPEL / BPM Solution. http://orchestra.ow2.org/.

[PLW+07] I. Poddar, M. Li, Q. Wang, Y. C. Guo, and Z. Gan. Securing a composite business service delivered as a software-as-a-service: Part I, secure multi-tenancy with WebSphere Portal Server, September 2007. http://www.ibm.com/developerworks/tivoli/library/t-cbssas/index.html#2.2.

[PSQ] PostgreSQL. http://www.postgresql.org/.

[SAS+12] S. Strauch, V. Andrikopoulos, S. G. Sáez, F. Leymann, and D. Muhler. Enabling Tenant-Aware Administration and Management for JBI Environments. Institute of Architecture of Application Systems (IAAS), University of Stuttgart, Stuttgart, Germany. 2012 IEEE International Conference on Service-Oriented Computing and Applications, 2012. (to appear).

[SASL12] S. Strauch, V. Andrikopoulos, S. G. Sáez, and F. Leymann. Implementation and Evaluation of a Multi-Tenant Open-Source Enterprise Service Bus, 2012. Institute of Architecture of Application Systems University of Stuttgart (under review).

[SOA] SmartBear Software. soapUI. http://www.soapui.org.

[Ura12] M. Uralov. Extending an Open Source Enterprise Service Bus for Dynamic Discovery and Selection of Cloud Data Hosting Solutions based on WS-Policy. Master's thesis, Institute of Architecture of Application Systems, University of Stuttgart, 2012. Not yet published. Realization in parallel to this student thesis.

[w3c04] Web Services Architecture, 11 February 2004. http://www.w3.org/TR/ws-arch/.

[WCL+05] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Prentice Hall, 2005.

[WSO08] WSO2. WSO2 Enterprise Service Bus (ESB) Performance Testing - Round 3, June 2008. `http://wso2.org/library/3740`.

[WSO12] WSO2. Cloud Native Advantage: Multi-Tenant, Shared Container PaaS (White Paper), June 2012. WSO2.

[WTJ11] S. Walraven, E. Truyen, and W. Joosen. A Middleware Layer for Flexible and Cost-Efficient Multi-tenant Applications. Middleware 2011, 2011.

All links were last followed on November 12, 2012.

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, November 14, 2012 _____

(Santiago Gómez Sáez)