Fakultät Informatik
Hochschule Reutlingen
Alteburgstraße 150
D-72762 Reutlingen

Masterthesis

Bestimmung der Dokumentenähnlichkeit basierend auf Bayessche Statistik für eine Big-Data Information Retrieval Lösung

Elisabeth Agnes Mpessa Enangue

Studiengang: Services Computing

Betreuer Hochschule: Prof Dr.-Ing Christian Decker
Betreuer Unternehmen: Steve Strauch
Abgabetermin: 31. Juli 2018

Abstract

In the last years Cloud computing has become popular among IT organizations aiming to reduce its operational costs. Applications can be designed to be run on the Cloud, and utilize its technologies, or can be partially or totally migrated to the Cloud. The application's architecture contains three layers: presentation, business logic, and data layer. The presentation layer provides a user friendly interface, and acts as intermediary between the user and the application logic. The business logic separates the business logic from the underlaying layers of the application. The Data Layer (DL) abstracts the underlaying database storage system from the business layer. It is responsible for storing the application's data. The DL is divided into two sublayers: Data Access Layer (DAL), and Database Layer (DBL). The former provides the abstraction to the business layer of the database operations, while the latter is responsible for the data persistency, and manipulation.

When migrating an application to the Cloud, it can be fully or partially migrated. Each application layer can be hosted using different Cloud deployment models. Possible Cloud deployment models are: Private Cloud, Public Cloud, Community Cloud, and Hybrid Cloud. In this diploma thesis we focus on the database layer, which is one of the most expensive layers to build and maintain in an IT infrastructure. Application data is typically moved to the Cloud because of , e. g. Cloud bursting, data analysis, or backup and archiving. Currently, there is little support and guidance how to enable appropriate data access to the Cloud.

In this diploma thesis the we extend an Open Source Enterprise Service Bus to provide support for enabling transparent data access in the Cloud. After a research in the different protocols used by the Cloud providers to manage and store data, we design and implement the needed components in the Enterprise Service Bus to provide the user transparent access to his data previously migrated to the Cloud.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

Cloud computing has changed in the last years the computing resources consumption and delivery model in the IT industry, leading to offer them as services which can be can be accessed over the network. The idea of virtualizing resources and provide them on demand to the users, like water, electricity, etc. under a metered service aims to deliver computing as a utility. Users are offered a single system view in a fully virtualized environment where computing resources seem unlimited, and can be accessed through Web interfaces and standardized protocols. Cloud providers target to maximize their benefits by maximizing the resources usage with a minimum management effort or human interaction, while the Cloud consumers can significantly reduce their capital expenditures in their IT infrastructure by outsourcing the demanded computational and storage resources to a Cloud environment.

In the following sections we discuss the problem statement and motivating scenario this thesis relies on.

1.1. Problem Statement

A multi-tenant aware architecture in a Cloud environment is one of the main keys for profiting in a Cloud infrastructure. Virtualization and simultaneously usage of resources by multiple users allow Cloud providers to maximize their resources utilization. However, a multi-tenant environment requires isolation between the different users at different levels: computation, storage, and communication [SAS+12]. Furthermore, the communication to and from the Cloud infrastructure must support different protocols.

Migration of an application to the Cloud can be divided into four different migration types: component replacement with Cloud offerings, partial migration of functionalities, migration of the whole software stack of the application, and cloudifying the application [ABLS13]. In this diploma thesis we focus on the needed support when the first migration type takes place. For example, due to an explosive data growth a tenant may decide at some point in time to migrate and host his local business data in a Cloud storage infrastructure, while maintaining his application's business logic on-premise. Bachmann provides a prototype which assists the tenant during the data migration process from a local storage system to a Cloud data store, and between Cloud data stores [Bac12]. However, as described before his work covers the migration process, but it does not provide data access or data modification after the migration.

An Enterprise Service Bus is a central piece in a Platform-as-a-Service (PaaS) environment for providing flexible and loosely coupled integration of services as well as multi-tenant aware and multi-protocol communication between services. In this diploma thesis we extend the multi-tenant aware prototype of an Enterprise Service Bus (ESB) produced in [Muh12],

[Ess11], and [Sá12]. The final prototype must provide multi-tenant and multi-protocol communication support, and transparent Cloud data access to tenants who migrate their application data partially or completely to the Cloud.

The use of an intermediate component in data transfer may have a negative impact on the overall data processing in an application. For this reason, we provide an evaluation using example data from an existing TPC benchmark in order to investigate the impact on the performance and to propose future optimizations [Tra01].

1.2. Motivating Scenario

IT industries aim to reduce their budget in expensive hardware investment and maintenance, e.g. Database Management Systems, while maintaining data access and persistency over time. In order to fulfill a budget reduction while maintaining their data and database system requirements, the data can be migrated to different Cloud storage providers available nowadays in the market. Considering the application's migrations types discussed in [ABLS13], migrating local data to the Cloud, and then accessing it from the application's hosted on-premise, can be considered as a *partial or complete replacement of components with Cloud offerings*. Such migration requires a reconfiguration, rewiring, and adaptation activities on both the migrated and non-migrated components.

The *Cloud Data Migration Application* assists the user in the migrating decision and process of local data to a Cloud storage provider, or from two different Cloud storage providers [Bac12]. It contains a registry of the different available Cloud data stores, e.g. Amazon Dynamo DB [Amaa], Amazon RDS [Amac], Google Cloud SQL [Goob], and detects possible incompatibilities between the source and target data sources prior to the migration process.

Migration of data can be either seen as the migration of only the Data Layer, or as a part of the migration of the whole application [ABLS13]. The approach we consider as the start point in this diploma thesis is the migration of the Data Layer, which contains two sublayers: the *Database Layer (DBL)* and the *Data Access Layer (DAL)*. The DBL contains the database information, e.g. location, access credentials, etc., and gives the DAL a direct communication support to the database server. The DAL provides simplified access support to the upper layers of the data stored in a backend database. However, migrating the Data Layer to the Cloud implies adaptations and rewiring of the original application. One of our main goals in this diploma thesis is to minimize the needed adaptations in the non-migrated and migrated components by providing a transparent access to the user's data migrated to a Cloud datastore.

As the Enterprise Service Bus is an established integration middleware for services in Service-Oriented Architectures (SOA), and due to its multi-protocol support and reliable internal messaging routing, we use it as a central piece for providing a multi-tenant aware, transparent and reliable communication support between the on-premise and the off-premise layers of the application.

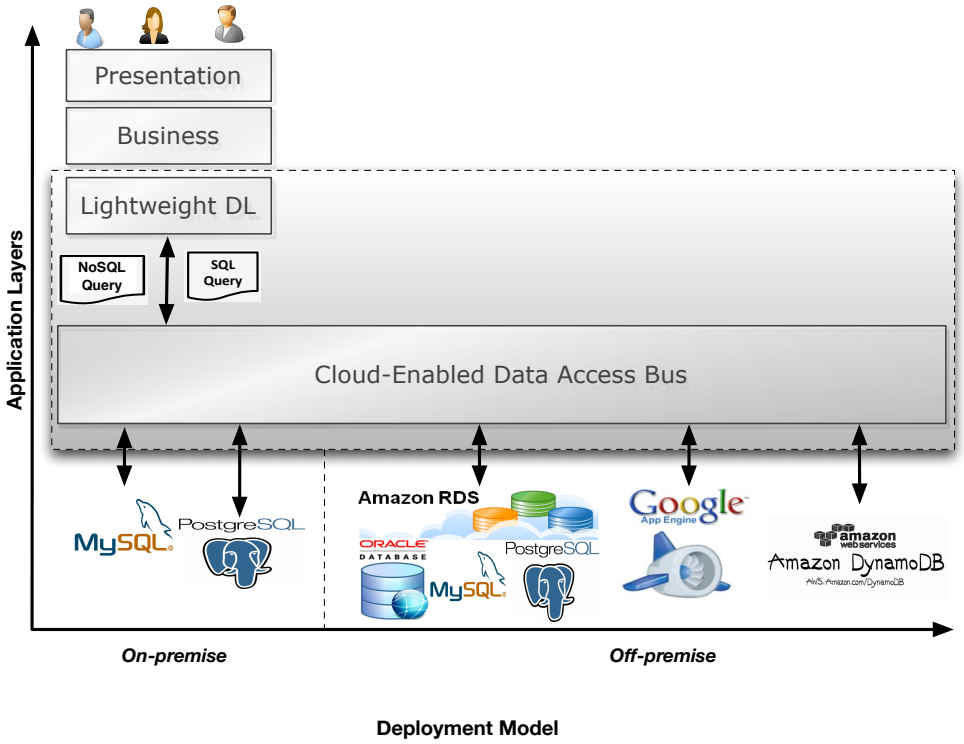

Figure 1.1.: Migration Scenario to be filled

As shown in Figure 1.1, the Cloud-Enabled Data Access bus provides access support between the hosted on-premise, and off-premise application's layers. Its main goal is to provide communication isolation between different applications and users, and maintain the transparency that the DL provided before the migration to the upper layers of the application's architecture. Support must be provided for two different databases types: MySQL and NoSQL databases, and between different providers. A tenant who migrates its data, e.g. to the Google SQL Datastore in Google App Engine, as shown in Figure 1.1, must be able to access his data with minimum adaptations of the components. Furthermore, storing or retrieving data whose storage is divided into multiple datasources requires a dynamic routing between backend data stores. Compatibility between different SQL and NoSQL databases must be also ensured. However, query and data transformation between different data sources types is out of the scope of this diploma thesis.

1.3. Definitions and Conventions

In the following section we list the definitions and the abbreviations used in this diploma thesis for understanding the description of the work.

Definitions

List of Abbreviations

The following list contains abbreviations which are used in this document.

API	Application Programming Interface
BC	Binding Component
BLOB	Binary Large Object
CLI	Command-line Interface
DBaaS	Database-as-a-Service
DBMS	Database Management System
DBS	Database System
EAI	Enterprise Application Integration
EIP	Enterprise Integration Patterns
ESB	Enterprise Service Bus
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure-as-a-Service
JBI	Java Business Integration
JDBC	Java Database Connectivity
JMS	Java Message Service
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MDBS	Multidatabase System
MEP	Message Exchange Patterns
NIST	National Institute of Standards and Technology
NM	Normalized Message
NMF	Normalized Message Format
NMR	Normalized Message Router
NoSQL	Not only Structured Query Language

OSGi	Open Services Gateway initiative *(deprecated)*
ORDBMS	Object-relational Database Management System
PaaS	Platform-as-a-Service
POJO	Plain Old Java Object
RDBMS	Relational Database Management System
SA	Service Assembly
SaaS	Software-as-a-Service
SE	Service Engine
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol *(deprecated)*
SQL	Structured Query Language
STaaS	Storage-as-a-Service
SU	Service Unit
TCP	Transmission Control Protocol
URI	Uniform Resource Identifier
UUID	Universally Unique Identifier
WSDL	Web Services Description Language
XML	eXtensible Markup Language

1.4. Outline

The remainder of this document is structured as follows:

- **Fundamentals, Chapter 2:** provides the necessary background on the different concepts, technologies, and prototypes used in this diploma thesis.
- **Related Works, Chapter 3:** discusses relevant State of the Art and positions our work towards it.
- **Concept and Specification, Chapter 4:** functional and non-functional requirements are discussed in this section.
- **Design, Chapter 5:** gives a detailed overview of the differerent component's architecture, and the needed extensions to the already existing ones.

- **Implementation, Chapter 6:** the implemented components, as well as the necessary extensions or changes are detailed in this section from the point of view of coding and configuration.
- **Validation and Evaluation, Chapter 7:** in this chapter we test the final prototype based on the scenario described in this document.
- **Outcome and Future Work, Chapter 8:** we provide a conclusion of the developed work and investigate some ideas in which this diploma thesis can be extended.

2. Fundamentals

In this chapter we give an explanation about the technologies and concepts this diploma thesis relies on. We start describing the fundamental concepts and introduce the components and prototypes that form the basis of our work.

2.1. Cloud Computing

In the last decades our world has become more and more interconnected. This interconnection added to the increase of the available bandwidth and the change in business models have forced IT Systems to fulfill its demands, leading to its reorganization into a public utility which offers public services, like water, electricity, etc. 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 user consumes the Cloud provider's computing capabilities automatically without the need of human interaction.
2. Broad network access: computing capabilities are available via 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.

Although the three service models described above provide both data computation and storage capabilities for the consumer, they do not provide to the customer the possibility to directly and uniquely purchase access of storage services. In this diploma thesis we concentrate in two concrete models: *Database-as-a-Service (DBaaS)* and *Storage-as-a-Service (STaaS)*. Cloud storage providers target a selected number of consumers, who process their data on-premise, but do not want to cover the expenses of a local database system, or a backup system, among others. The Cloud storage model alleviates the need in organizations to invest in database hardware and software, to deal with software upgrades, and to maintain a professional team for its support and maintenance [HHM02]. DBaaS and STaaS can be considered quite similar, except for one of their main distinction characteristics: their access interface. The former is the most robust data solution offered as a service, as it offers a full-blown database functionality. It can be accessed via the most common database protocols, such us MySQL, Oracle, etc, or by REST interfaces supporting Structured Query Language (SQL). Examples of this model are Amazon RDS [Amac] and Oracle Cloud [Orab]. On the other hand, the latter provides REST, SOAP over Hypertext Transfer Protocol (HTTP), or Web-based interfaces in order to perform the operations over the stored data [WPG+10]. Examples of this model are Amazon Dynamo [Amaa] , Google App Engine Datastore [Gooa], and Dropbox [Dro].

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 Service-Oriented Architecture (SOA) are related styles 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 through the network. Therefore, we can deduce that the SOA model can be applied in the Cloud computing approach. As the ESB is the central piece of SOA, the

need of the ESB in a Cloud computing infrastructure as an integration middleware for the Cloud services is essential.

2.2. Service-Oriented Architecture

Weerawarana et al. define SOA as an specific architectural style that is concerned with loose coupling and dynamic binding between services [WCL+05].

In the last years communication between external components whose functionalities are exposed as services has been a hard task when there was not previous agreement on message protocols, data types and encoding, and used middleware technology. Due to the economic and technological growth needed, enterprises had to adapt the SOA paradigm in their existing IT Infrastructure. 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's functionality is exposed, but not its internal logic.
- 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.

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 provides support for service registration and binding.

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].

2.2.1. Enterprise Service Bus

The flow of data and information is a key for driving business decisions in IT organizations [Cha04]. Furthermore, the interaction between loosely coupled components within an organization or with third party organizations requires distributed systems mechanisms which provide communication support for different protocols, and reliability. SOA has fulfilled this main requirement by providing an integration environment with minimal (or any) integration efforts.

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 an 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]. A 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. Furthermore, the ESB can form the core of a pervasive grid [Cha04]. Services supported by an organization can be organized between the ESBs conforming the grid, as well as its access between the organizational departments, and services provided to third party organizations.

When receiving a service description (Web Services Description Language (WSDL)) and data from the service requester, the ESB 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 enable the communication between the parts.

As the ESB is the central component in SOA, and established as integration middleware for services, in this diploma thesis we focus on the required modifications and extensions in the open-source ESB Apache ServiceMix 4.3 to provide transparent communication support between the applications and its data located in on-premise databases, or migrated to off-premise data stores.

2.3. Multi-tenancy

One of the main decision variables for utilizing a Cloud computing environment are capital expenditures. The main goal of a Cloud consumer is to minimize its business costs when migrating to the Cloud. According to Chong, a SaaS solution benefits a Cloud customer with the following advantages [CC06]:

- The Cloud consumer does not directly purchase a software license, but a subscription to the software offered as a service by 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 on 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.

Multi-tenancy in a SaaS environment allows the Cloud providers to lower the cost per customer by optimizing the resources usage in the Cloud infrastructure. The software serves multiple tenants concurrently, who share the same code base and data storage systems. Chong and Carraro [CC06] define 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*. Business softwares are becoming more complex and tend to demand an individual customer support and an increase of the computing and storage resources in the infrastructure. This fact leads to an increase in the infrastructure investment and maintenance costs. 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.

The reasons discussed above are also applicable in the DBaaS and STaaS models. Storage and retrieval of data involve high maintenance and management costs. The data management cost is estimated to be between 5 to 10 times higher than the data gain cost [AP11]. Furthermore, storing data on-premise does not only require storing and retrieving data, but also requires dealing with disaster recovery, Database Management System (DBMS), capacity planning, etc. Most of the organizations prefer to lead their investments to their local business applications rather than becoming a data management company [AP11]. Cloud storage providers offer a pay-per-use storage model, e.g. based on storage capacity or based on number of connections to the storage system, and ensure that the stored data will persist over time and its access

through the network. However, security and confidentiality are the main constraints when moving private data to a shared public infrastructure.

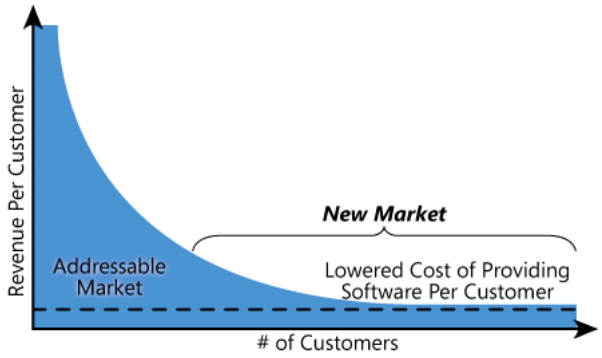

Figure 2.1.: New market opened by lower cost of SaaS [CC06]

In the Figure 2.1 the economics of scaling up to a high number of customers while reducing the software price is analyzed. Cloud providers have reached a new market formed by small or medium businesses without enough budget for building an on-premise IT infrastructure.

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]. Andrikopoulos et al. identify two fundamental aspects in multi-tenant awareness: communication, and administration and management [ABLS13]. The former involves isolated message exchanges between tenants and the latter allows tenants individual configuration and management of their communication endpoints. Utilizing an ESB as the central piece of communication middleware between applications in a PaaS environment forces it to ensure multi-tenancy at both communication, and administration and management, as mentioned before. The multi-tenancy support modifications made in the open-source ServiceMix 4.3 are the results of [Ess11], [Muh12], and [Sá12]. In this diploma thesis we reuse and extend those results in oder to provide multi-tenant transparent Cloud data access in the Cloud through the ESB, when the application's data is migrated and accessed through the ESB in a Cloud infrastructure.

The migration of an application's stack to the Cloud can be done at different levels of the application's stack: Presentation Layer, Business Layer, and Data Access Layer. The Replacements of Components with Cloud offerings migration type is the least invasive type of migration [ABLS13]. In this diploma thesis we focus on this type of migration, concretely when the used Cloud offering is the database system. Migration of the data can be either seen as the migration of the Data Layer (Data Access Layer and Database Layer) or of the whole application [ABLS13]. Migration of the Data Layer to the Cloud means migrating the both data management and data access to the Cloud, while maintaining its transparency to the application's Business Layer.

In a Cloud infrastructure where Cloud storage is offered, Feresten identifies four main tenant requirements: security, performance, data protection and availability, and data management [Fer10]. Multi-tenancy in a storage system can be achieved by aggregating tenant-aware

meta-data to the tenant's data (see Figure 2.2), or by physical storage partitioning, but this is not sufficient when fulfilling the data management, and the flexibility requirement. Tenants must have independent access and management, as if they accessed their own data storage systems. For this purpose, storage vendors have introduced the concept of *virtual storage container*, a tenant-aware management domain which grants all (or most of) the database management operations over the storage container, as described in Figure 2.2.

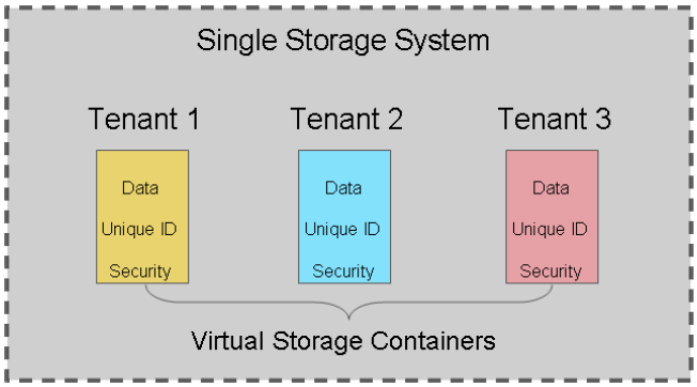

Figure 2.2.: Attributes of a Virtual Storage Container [Fer10]

In this diploma thesis we must take into account the different approaches that most of the Cloud storage vendors have taken into account, in order to provide the tenant transparent access through the ESB to his virtual storage container in one or more Cloud infrastructures.

2.4. Java Business Integration

The interaction between enterprises' applications has suffered in the past from lack of standardized technologies, leading each of the enterprises to develop their own or acquiring vendor-specific integration technology. Java Business Integration (JBI) is defined by the Java Community as an integration technology which maximizes the decoupling between components and defines an interoperation semantic founded on standards-based messaging. This allows different vendor-specific components to interoperate in a multivendor "echosystem" [JBI05].

The key which leads to the integration of different components relies on a unique message format in the JBI environment which different plugged-in components use to communicate within the environment. External components are not directly connected, but through a mediator. 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 and provide different services:

- A Service Engine (SE) provides transformation and composition services to other components.
- A Binding Component (BC) provides the connectivity between the external services and the JBI environment. They support many different 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.

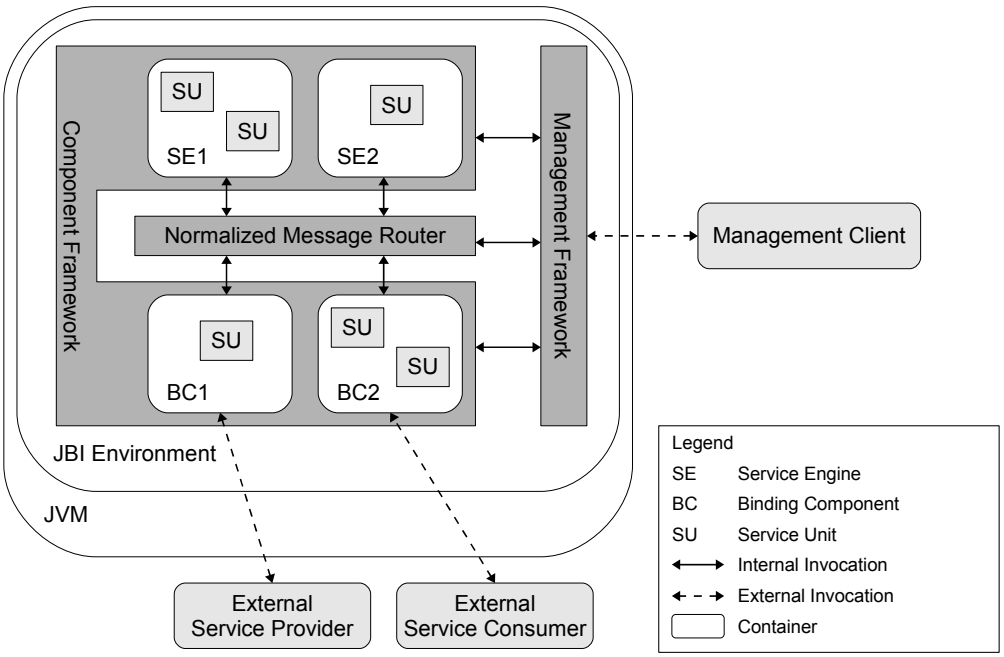

Figure 2.3.: Overview of JBI Architecture. Figure 4 in JBI specification document [JBI05].

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 framework provides loose coupling between modules in a Java environment. It provides a strong support for module versioning and third party modules referencing. The OSGi defines a framework for deployment support in a Java Virtual Machine (JVM) of downloaded or extended resources 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, which can be exposed as bundle services or just run internal processes. A valid OSGi bundle can be installed in any valid OSGi container due to the standardized packaging and bundle management.

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.

Apache ServiceMix 4.3.0 is built on and OSGi-based runtime kernel, which provides a lightweight container that enables the deployment of various bundles [RD09]. Its architecture and functionalities are described in the following section.

2.6. Apache ServiceMix

In this diploma thesis we extend a multi-tenant aware version of Apache ServiceMix 4.3.0, referred to it in this document as ServiceMix. Essl evaluates different available ESB solutions in the market, and as output of his work provides a selection decision for extending ServiceMix in order to support multi-tenancy [Ess11]. As mentioned in Section 2.3, a multi-tenant ESB solution in a PaaS environment must support tenant-aware communication, and tenant-aware administration and management. The support is provided by Muhler and Gomez in their

corresponding works in [Muh12], [Sá12], leading to a multi-tenant ServiceMix prototype supporting different communication protocols. We will refer to it as ServiceMix-mt.

As a main difference with previous versions' architectures, 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 a 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.

The main advantage in the ServiceMix 4.3.0 it is its full compliance with the JBI specification. Its JBI container has as its main component the NMR (See Figure 2.4). In the JBI container users are provided with JBI deployment support and management. The communication between the JBI and OSGi container, e.g. from one OSGi service to a JBI Binding Component can be achieved through the NMR using its API wrapped as an already deployed OSGi bundle. This fact eases the integration process of components between different ServiceMix's versions, and take advantage of it in this diploma thesis. Furthermore, JBI components or endpoint configurations packed as SE or SA and deployed in ServiceMix are internally packed into OSGi bundles.

ServiceMix is shipped with different JBI components already deployed as OSGi bundles in its OSGi container. In this thesis we will concentrate on the following ones: HTTP and Apache Camel. Apache Camel is a powerful open source integration framework based on Enterprise Application Integration (EAI) [APA11a]. Furthermore, it provides a widespread list of components which support different communication protocols. 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]. Apache Camel provides a set of maven archetypes which already contain the structure for developing custom camel components.

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 an external service. When a message arrives to a consumer endpoint of a JBI Binding 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.

The ServiceMix-mt prototype we extend in this diploma thesis already provides multi-tenant support for different communication protocols: HTTP, JMS, and E-mail. However, the data retrieval and storage in an application's architecture relies on one specific layer: Data Access Layer, and its main used communication protocols for data transfer are in most of the cases vendor specific. Communication with SQL databases, such as MySQL, Oracle,

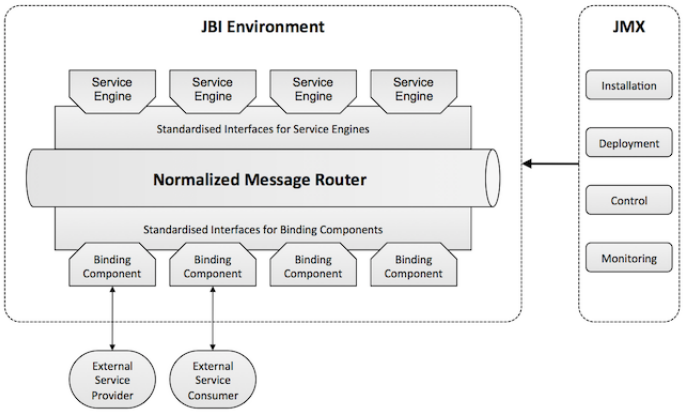

Figure 2.4.: Architecture of Apache ServiceMix [ASM]

and PostgreSQL are managed by its vendor-specific native driver, which implements the communication protocol with the database server in the DBMS. Such protocols are not supported in the multi-tenant prototype ServiceMix-mt, and must be taken into account in the extension of the prototype in this diploma thesis. On the other hand, communication with Not only Structured Query Language (NoSQL) databases can be also considered vendor-specific, because most of the Cloud storage providers facilitate its own API to the users to manipulate data in their data containers, but almost all of them provide either REST or SOAP over HTTP interfaces. This fact permits us to reuse and extend the multi-tenant HTTP BC in ServiceMix-mt. The components we create or extend in this diploma thesis are identified by CDASMix (Cloud Data Access Support in ServiceMix-mt).

2.7. Binding Components

In this section we describe the JBI BCs shipped in the ServiceMix-mt prototype this diploma thesis focuses on, and the transport protocols they support.

2.7.1. Multi-tenant HTTP Binding Component

ServiceMix provides HTTP communication support in its HTTP JBI BC. Its HTTP consumer and provider endpoints are built on the HTTP Jetty 6 server and Jakarta Commons HTTP Client respectively, providing support for both REST and SOAP over HTTP 1.1 and 1.2 requests.

The original HTTP BC is extended in the ServiceMix-mt prototype to provide multi-tenant support in [Muh12] and [Sá12]. Muhler provides an internal dynamic creation of tenant-aware endpoints in the BC, by injecting tenant context in the JBI endpoint's URLs [Muh12]. Gomez provides a NMF with tenant context information in its properties for routing in the NMR [Sá12]. However, in this diploma thesis we must not only provide tenant isolation at

the tenant level, but also isolation at the user level. We discuss this requirement in detail in Chapters 4 and 5.

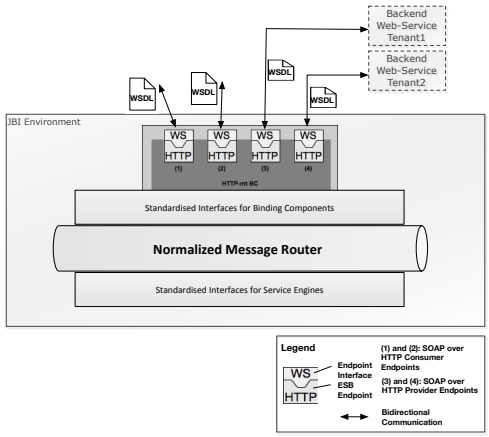

Figure 2.5.: Multi-tenant HTTP Binding Component [Sá12].

As seen in Figure 2.5, the multi-tenant HTTP BC is mainly used in ServiceMix-mt to support the SOAP over HTTP communication protocol by exposing a Web service in the tenant-aware consumer endpoint and consuming an external Web service in the provider endpoint. 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 headers and one body. The header may contain processing or authentication data for the ultimate receiver 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 loosely 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 Message Exchange Patterns (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.

Most of the Cloud storage providers provide an HTTP interface to the tenants for data management, retrieval, and storage. In this diploma thesis we extend this JBI BC in order to provide the tenant a transparent access to his NoSQL Cloud data stores.

2.8. Service Engine

A SE can provide different kinds of services, e.g. business logic, routing, and message transformation. In this diploma thesis we will mainly concentrate on one: Apache Camel [APA11a], which is wrapped in a ServiceMix-camel JBI SE in ServiceMix, and in a ServiceMix-camel-mt JBI SE in ServiceMix-mt for multi-tenancy awareness..

2.8.1. Apache Camel

Apache Camel is an open-source integration framework based on known Enterprise Integration Patterns (EIP) which supports the creation of routes 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]. In ServiceMix, Apache Camel is shipped in a JBI SE. 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. Route configurations deployed in ServiceMix must follow the JBI compliance: files describing the route configuration must be packed in SU, and the latter in a SA. Apache Camel provides Maven archetypes which generate the needed route configuration files (in XML or POJO formats) where the developer can program the route between the different supported endpoints [AMV]. 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. Furthermore, the routing patterns supported by Apache Camel are the point-to-point routing and the publish/subscribe model.

The endpoints representation in Apache Camel is based on Uniform Resource Identifier (URI). This allows this SEs to integrate with any messaging transport protocol supported in the ESB, e.g. HTTP, Java Message Service (JMS) via ActiveMQ, E-Mail, CXF, etc. The ServiceMix-camel JBI SE provides integration support between camel and JBI endpoints. Muhler extends this component and allows dynamic internal creation of tenant-aware endpoints in the ServiceMix-camel-mt JBI SE [Muh12]. The main goal of this extension is to provide an integrated environment between JBI and camel supported endpoints. However, multi-tenancy is supported at the tenant level only in the JBI endpoints. In this diploma thesis we aim to enable multi-tenancy not only at the tenant level, but also at the user level, as discussed in Chapters 4 and 5.

For enabling data access support with SQL DBMS in ServiceMix-mt we extend a well-known camel component: Camel-jdbc. The Camel-jdbc component enables Java Database Connectivity (JDBC) access to SQL databases, using the standard JDBC API, where queries and operations are sent in the message body [Thec]. This component requires the developer to statically denote the data source configuration (user, password, database name, etc.) in both the endpoint URI and route configuration file. As discussed in Chapters 4 and 5, this requirement is opposite to our approach, due to the dynamism we need in creating connections to the different DBaaS providers. We extend and produce a custom camel component: Camel-jdbccdasmix (*cdasmix* stands for Cloud Data Access Support in ServiceMix-mt).

2.9. Structured Query Language Databases

The SQL stands nowadays as the standard computer database language in SQL Database System (DBS). SQL is a tool for organizing, managing, and retrieving data stored by a computer database [GW99]. The SQL language is nowadays one of well known languages in the IT sector. Thus, we introduce in the following sections in the SQL DBS and their specific communication protocol we use in this thesis, rather than on the language they support.

The final prototype of this diploma thesis aims to provide support for most of the DBS communication protocols available in the market. However, we crashed into vendor-specific communication protocol implementations along the available DBMS in the market, rather than a common standardized communication protocol. For this reason, we provide support for incoming connections which comply the MySQL DBS communication protocol and give the hints for supporting the PostgreSQL DBS communication protocol. However, for outgoing connections we have not found such problem, due to the management of the different vendor's native drivers provided by JDBC, which is introduced at the end of this section.

2.9.1. MySQL Database System

MySQL is nowadays the most popular Open Source SQL Relational Database Management System (RDBMS) [Ora13]. Data is stored following a relational storage model, where data is represented as tuples, and grouped into relations. The main storage structure managed in this type of database are tables, which can be linked together by establishing relationships governed by rules, e.g. one-to-one, one-to-many, many-to-many, etc.

The MySQL server is one of the main components in the DBMS. It is a client/server system which consists of a multi-threaded SQL server which supports different backends, several different client programs and libraries, administrative tools, and a wide range of application programming interfaces (APIs) [Ora13]. Its main functionality we discuss in this diploma thesis is the protocol it supports for I/O operations between the client and the server. The MySQL communication protocol has changed over time and over the DBMS version upgrades, leading to different new user authentication methods, new data types, etc. In this diploma thesis we cover the MySQL versions 5.x support. Due to the compatibility of the native JDBC drivers along the different versions, the supported protocol in our prototype is full compatible with the last released JDBC MySQL native driver.

The MySQL communication protocol is used between the MySQL client and server. Implementations of the protocol can be found in the MySQL server, the MySQL native driver Connector/J (Java implemented) and in the MySQL proxy. As it is described in Figure 2.6, the whole communication process between a MySQL client and a MySQL server is divided into three phases: connection phase, authentication phase, and command phase, and their main transferred information unit are MySQL packets. The MySQL packet configuration and the supported data types are described in Chapter 6.

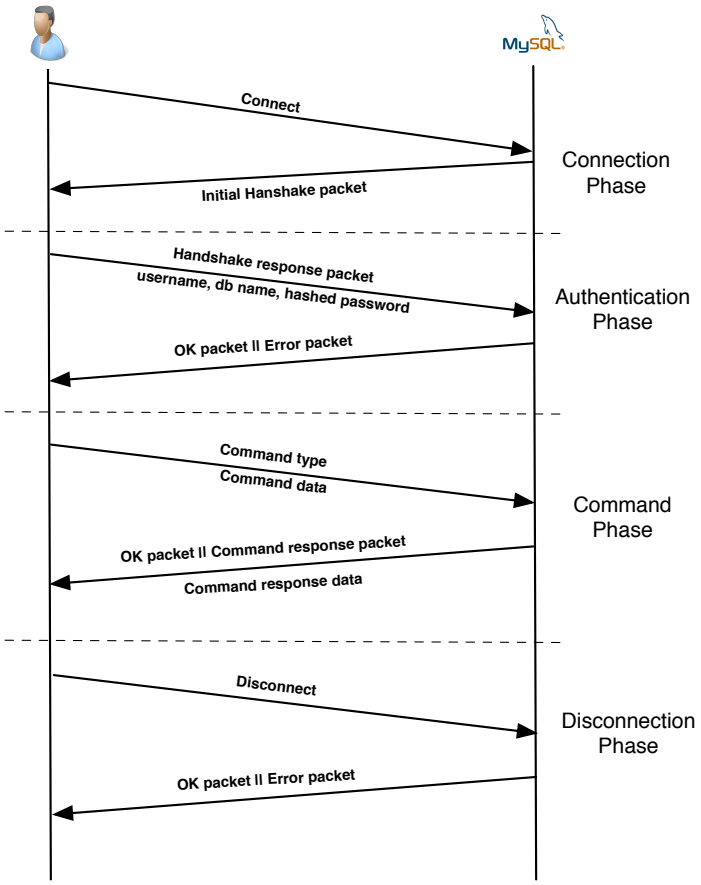

Figure 2.6.: MySQL communication protocol in the four communication phases [Ora13]

During the connection phase, the client connects via Transmission Control Protocol (TCP) to the port where the main MySQL server thread listens on (commonly used port 3306). In the connection and authentication phases, the MySQL server sends to the client an initial handshake packet, containing information about the server, server flags, and a password challenge. The client responds with his access credentials and communication configuration flags. When the authentication succeeds, the command phase is initiated. This phase is actually where the operations on the database or on the server take place, e.g. server configuration, querying, statements execution, etc. The connection between the client and the server must be always ended in the client side, except for internal errors in the server where the communication is interrupt and lost.

In this diploma thesis we extend a Java implementation of the MySQL protocol, which is described in more detail in Chapter 3, and adapt it for its integration and communication in ServiceMix-mt.

2.9.2. PostgreSQL Database System

PostgreSQL is known as an Object-relational Database Management System (ORDBMS). An ORDBMS is quite similar to the a RDBMS model explained in the last section, but its main difference is that it also supports the object-oriented database model, where objects are stored in database schemas can be accessed using the SQL.

The PostgreSQL DBMS also implements a client/server model for its I/O operations in the database. In contrast to the MySQL server, the PostgreSQL server defines the following cycles depending on the state of the connection: start-up, query, function call, copy, and termination [The96]. During the start-up phase, the client opens a connection and directly provides its user name and the database he wants to connect. This information identifies the particular protocol version to be used. The server responds with an authentication challenge which the client must fulfill.

The MySQL's SQL command phase is in this server denoted as a query cycle. A query cycle is initiated with the reception of an SQL command, and terminated with the response of the query execution.

The function call cycle allows the client with execute permissions to request a direct call of an existing function in the system's catalog. The copy cycle switches the connection into a distinct sub-protocol, in order to provide a high-speed data transfer between the client and the server.

The termination of a successful or failed client/server communication is handled in the termination cycle, which involves the transfer of a termination packet from the client to the server in the successful case, and from the server to the client when the termination is due to a failure.

2.9.3. MySQL Proxy

The MySQL Proxy is an application which supports the MySQL communication protocol between one or more MySQL clients and MySQL servers [Ora12]. In a distributed storage system where different clients connect to different servers a proxy which acts as a communication intermediary may significantly increase the overall performance. The MySQL proxy supports communication management between users, communication monitoring, load balancing, transparent query alteration, etc. Oracle releases a MySQL proxy which supports MySQL 5.x or later, and implemented in the C programming language.

Integrating a MySQL server into a ESB collisions with the main concept of an ESB as an intermediary technology between services. For this reason, in this diploma thesis we integrate and extend a Java version of a MySQL proxy developed by Continuent Inc.: Tungsten Connector [Con].

2.9.4. Java Database Connectivity

JDBC is widely used in the connection to databases in the Java programing language. JDBC technology allows programers to use the Java programming language to exploit "Write Once, Run Anywhere" capabilities for applications that require access to enterprise data.[Oraa]. Its management of different vendor-specific native drivers allows businesses not to be locked in any proprietary architecture, but to be able to connect to different databases simply by specifying the driver's name and the connection properties in the JDBC URL.

The JDBC Driver Manager or DataSource Object implements the selection of the appropriate vendor's native driver specified in the JDBC URL. However, the vendor's native driver must be installed prior to execution.

In this diploma thesis we take advantage of this technology in order to enable our final prototype to support a multi-protocol database outgoing communication (from the prototype to external DBS).

2.10. NoSQL Databases

RDBMSs ensure data persistency over time and provide a wide set of features. However, the functionalities supported require a complexity, which is sometimes not needed for some applications, and harms important requirements in Web applications or in SOA based applications, e.g. throughput. NoSQL data stores aim to improve the efficiency of large amount of data storage while reducing its management cost [Lai09]. NoSQL databases are designed to support horizontal scalability without relying on the highly available hardware [Str]. In a Cloud storage environment where the user sees the available computing and storage resources as unlimited, a NoSQL support in a Cloud storage environment might be adequate.

NoSQL DBS operate as a schema-less storage system, allowing the user to access, modify or freely insert his data without having to make first changes in the data structure [SF12]. Cloud providers provide the users with an Application Programming Interface (API) for accessing, modifying, and inserting data into his isolated container. For example, a user's Amazon Dynamo DB table and item can be accessed by its RESTful API, or by installing at the user's side application the Amazon Web Services SDK [Amaa]. Furthermore, it provides the users through its Web-based management console the available management operations.

Due to the growth of the NoSQL support along different Cloud vendors, in this diploma thesis we provide a multi-tenant and transparent communication support for NoSQL backend data stores in different Cloud providers. In the following sections we introduce the categorization of the different NoSQL databases we aim to support in this diploma thesis, mentioning and giving examples of Cloud data stores available nowadays in the market.

2.10.1. Key-value Databases

In a key-value datastore elements are uniquely identified by an id, which the data store does not take into account its type, and are simply stored as a Binary Large Object (BLOB) . A user can get the value for the key, put a value for the key, or delete a key from the data store [SF12]. Its storage model can be compared to a map/dictionary [Str]. Products offering this data storage model in a Cloud infrastructure are Amazon DynamoDB [Amaa], Google Cloud Storage [Gooc], Amazon SimpleDB [Amae] , Amazon S3 [Amad], etc. In this diploma thesis we mainly focus on the following key-value data stores: DynamoDB, and Google Cloud Storage.

Amazon DynamoDB's data model includes the following concepts: tables, items, and attributes [Amaa]. The attributes are a key-value, where the value is binary data. Attributes are stored in items, and these are stored in tables. Items stored in a table can be retrieved by referencing its unique id. The number of attributes is not limited by Amazon, but each item must have a maximum size of 64 KB. Accessing stored data in this data store can be mainly done in two ways: using the functionalities provided by the AWS SDK, or using the Cloud storage RESTful API.

Google Cloud Storage's data model includes the following concepts: buckets and objects [Gooc]. Buckets contain on or more objects. The objects are identified within a bucket with its unique id. Users can perform I/O operations on both buckets and objects. For this purpose, Google Cloud storage provides RESTful API.

In this diploma thesis we use an ESB for accessing transparently tenant's databases migrated to the Cloud. Servicemix-mt provides multi-tenant HTTP support [Sá12]. Therefore, we reuse and extend the multi-tenant HTTP BC in order to provide dynamic routing between the different data stores.

2.10.2. Document Databases

Document databases can be considered as a next step in improving the key-value storage model. In this storage model, documents are stored in the value part of the key-value store, making the value content examinable [SF12]. Documents with different schemas are supported in the same collection, and can be referenced by the collection's key or by the document's attributes. One of the main difference in the attributes specification regarding RDBMS is that in document stores document's attributes cannot be null. When there is an attribute without value, the attribute does not exist in the document's schema. Products implementing this data storage model are Apache CouchDB, MongoDB, etc. [Thea] [10G].

Mongo DB defines two storage structures: collections and documents [10G]. A specific database contains one or more collections identified by its unique id. A specific collection stores one or more documents. Collections and documents stored in a database can be accessed, inserted and modified using the RESTful API supported by the database system.

Apache CouchDB defines two storage structures: databases and documents. Data stored in CouchDB are JavaScript Object Notation (JSON) documents. The main difference between this two described databases is that MongoDB implements a two step access to the documents: database, collection, and document. Apache CouchDB provides a RESTful API for I/O operations.

This databases are not offered by Cloud providers like Amazon or Google, but as a software which can be deployed in user instances, e.g. Amazon EC2 AMI [Amab].

2.10.3. Column-family Stores

One of the most known Column-family data stores is Cassandra. Column-family data stores store data in column families (groups of related columns which are often accessed together) as rows that have many columns associated with a row key [SF12]. This approach allows to store and process data by column instead of by row, providing a higher performance when accessing large amount of data, e.g. allowing the application to access common accessed information in less time.

Cassandra has as its smallest unit of storage the column, which consists of a timestamp and a name-value pair where the name acts as a key [SF12]. As in the relational model, a set of columns form up a row, which is identified by a key. A column family is a collection of similar rows. The main difference with the relational model is that each of the rows must not have the same columns, allowing the designer and the application consuming large amounts of data to customize the columns in each row, and the rows in each column family.

Cassandra is not shipped with a RESTful API for I/O operations. However, there are several open-source services layers for Cassandra, e.g. Virgil [Thed].

2.11. JBIMulti2

A multi-tenant management 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, endpoint and routing configurations files are packed in SUs, and the latter in SAs for deployment. However, there is a lack of user-specific data during deployment. Muhler solves this problem in JBIMulti2 by injecting tenant context in the SA packages, making them tenant-aware [Muh12].

The architecture of the JBIMulti2 system is represented in Figure 2.7. 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.

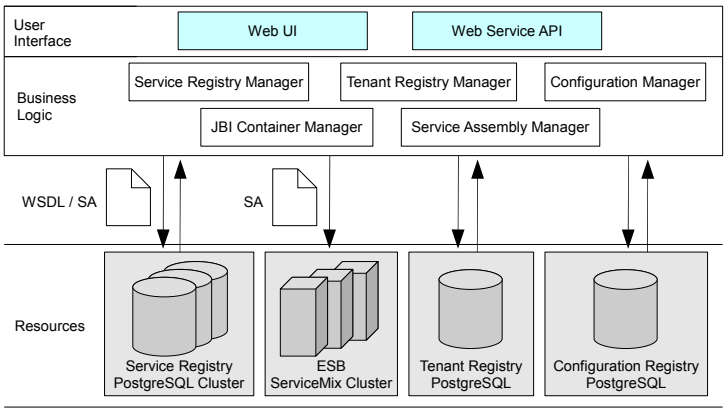

Figure 2.7.: JBIMulti2 System Overview [Muh12]

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 and user id value. In this thesis we need tenant information during runtime. We reuse and extend the databases schemas produced by Muhler, specifically the Service Registry.

The system provides a 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 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, PostgreSQL, etc. The initialization of the application is described in both Chapter 7 and in the JBIMulti2 setup document [Muh].

2.12. Cloud Data Migration Application

The Cloud Data Migration Application provides support to the user before and during the data migration process to the Cloud. It contains a registry of different Cloud data hosting solutions and its properties, which are used during the decision process. The decision process

consists in selecting the Cloud provider which best fits the user's operational and economical interests, and in detecting incompatibilities with the selected target data store. The different steps of the migration process are shown in Figure 2.8.

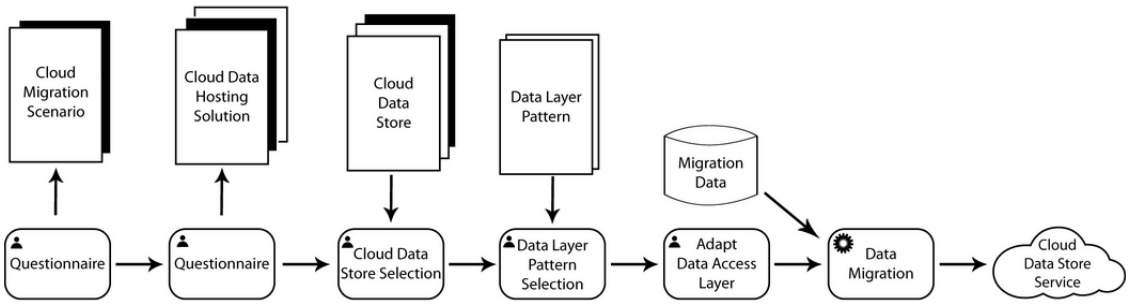

Figure 2.8.: Cloud Data Migration Process [Bac12]

In the *data layer pattern selection* and *adapt data access layer steps*, the user must specify how to connect to the data store his data is migrated to, and provide the necessary information to establish the connection. The extension of ServiceMix-mt for enabling Cloud data access support allows the user to select this prototype for transparently access the migrated data.

2.13. Apache JMeter

Apache JMeter is a Java-based application which provides support for load testing and performance measurement [Theb]. It provides support for different communication protocols, e.g. HTTP, SOAP, database via JDBC, etc. A multi-protocol support enables the application to be used for testing different layers of an application, e.g. presentation layer, and database layer. Furthermore, the user can configure in JMeter different load parameters, e.g. number of threads, iterations, etc., in order to build a heavy load simulation to run on the backend server. Simulation results are presented in structured formats for posterior analysis.

In this diploma thesis we provide an evaluation of the behavior of the final prototype. Due to the JDBC functionality supported by JMeter, we use it to generate the load test cases which are run on ServiceMix-mt.

3. Related Works

In this chapter we provide a general overview on the different approaches that are taken into account in order to provide a reliable, secure, and transparent communication between on-premise application's layers and off-premise Cloud data stores. Furthermore, we discuss about the needed adaptations different authors specify that the on-premise application's layers must address when migrating their underlying layers to a Cloud infrastructure. We compare it to the ones we transparently support in our approach, and the ones the user should consider. We finally mention the improvements we need to perform to the original prototype ServiceMix-mt, and continue our discussion dividing it into the two main DBMS available nowadays in the market: SQL and NoSQL databases.

A migration process of the Database Layer of an application to the Cloud may pop up several incompatibilities with the new hosting environment, which need to be addressed prior to the migration decision. Strauch et al. aim to address such incompatibilities by defining a set of *Cloud Data Patterns*, which target finding a solution for a challenge related to the data layer of an application in the Cloud for a specific context [SAB+12]. Incompatibilities a user may find when migrating the application's data layer can be on the level of the schema representation, supported set of queries or query language, communication protocol, security, etc. Strauch et al. focus mainly in two non-functional aspects: enabling data store scalability and ensuring data confidentiality [SAB+12].

The former deals with maintaining the quality of service levels when the workload increases, for both write and read operations. There are two scalability mechanisms when dealing with data: vertical and horizontal scalability. A vertical scalable system can be obtained by introducing more powerful hardware, or moving to a more powerful database system, while a horizontal scalable system deals with splitting data into groups which are stored in different partitions or different database systems, also known as *sharding*. Due to the absence of support for accessing a *sharded database* between different database systems, the concepts of a database proxy and sharding-based router are introduced. In this first approach, a proxy component is locally added below each data access layer [SAB+12]. A proxy below each data access layer instead of a common proxy on top of the database layer dismisses a common point of failure when accessing the data. In the second approach, a local sharding-based router is added below each of the data access layer. A sharded-based router contains the needed knowledge about the location of the *sharded databases*. In our approach, we don't only partially follow both of the concepts, but integrate them in a single component. We consider a sharded-based router as a proxy with routing capabilities. Therefore, as it is discussed in Chapter 5, enhancing an ESB with the required *sharded databases* knowledge and with standardized communication protocols, it allows us to utilize it as a sharded-based router, and as a proxy. Furthermore, the single point of failure avoidance can be ensured by increasing the number of ESB instances and balancing the load between them. As discussed before, we do not fully comply with this approach. The development of a proxy or sharded-based router component below each data

access layer forces each application to deploy at least one proxy or sharded-router instance in their system. In our approach we propose the utilization of our prototype as a shared transparent Cloud data access layer by connecting to a data access endpoint which supports a specific DBMS multi-tenant aware communication protocol (e.g. MySQL or PostgreSQL). For this purpose, we propose the concept of a lightweight Data Layer, where the adaptations to its sublayers are minimized, e.g. modification of data access host, port, etc. The data access endpoint acts as a database protocol-aware proxy, forwarding the requests to the NMR of the ESB. We enhance the Myosotis Tungsten Connector and provide access control, caching functionality, and full integration in the ESB OSGi container, and with the NMR [Con].

Ensuring data confidentiality is presented in [SBK+12]. Their work deals with critical data management between on-premise and off-premise data stores, and categorizes data into different confidentiality levels to prevent data disclosures. The former is achieved by aggregating information which categorizes data into different categories and confidentiality levels. The latter deals with keeping confidential data on-premise. With data filtering, pseudonymization, and anonymization, data is either prevented from being externally routed, or secured when routed to a public Cloud [SBK+12]. The pseudonymization technique provides to the exterior a masked version of the data while maintaining its relation with the original data, and the anonymization provides to the exterior a reduced version of the data. In this diploma thesis' approach, we assume that the application's owner has decided on which data should be and cannot be migrated, and that the business layer is hosted on-premise. Therefore, there is no data processing in a public Cloud environment. Our final prototype provides confidentiality between different tenants of the system by injecting tenant information in our messages and providing tenant-aware routing, and different multi-tenant aware endpoints. We do not need to provide support for pseudonymization or anonymization techniques, in contrast to [SBK+12].

Replacement of components which build an application with Cloud offerings leads the developers to face an application's adaptation process. For example, migrating a local database to a private Cloud or to a public Cloud, or sharding a database between on-premise and off-premise data stores forming a single data store system, can not be accessible without adapting the non-migrated application's layers to the new storage system. Andrikopoulos et al. identify the needed adaptations actions when migrating a data layer to the Cloud [ABLS13]: address reconfiguration, patterns realization, incompatibilities resolution, query transformation, and interaction with data store allowance. Our main goal in our final prototype is to minimize the number of adaptations the user must perform when migrating application's data to a Cloud data store. The adaptations of the ESB must encompass the described adaptations in a transparent way to the user, in order to internally support in our prototype compatibility between the application and the different data stores, and lower the adaptation operations number at the application's side, e.g. only address reconfiguration.

Federated Database Systems are a type of Multidatabase System (MDBS) that allow accessing and storing data which is stored in different and noncontiguous databases through a single interface. Sheth and Larson define them as a collection of cooperating but autonomous component database systems, which can be integrated to various degrees, and can be accessed by a software which controls and coordinates the manipulation of the database systems which

conforming the federated database system. This distributed database model allows users to access and store data among different database systems, which can be located in different continents, without dealing with multiple connections to the different database systems, query and data transformation, address adaptation, etc. MDBS are accessed through a single endpoint which provides a single logical view of the MDBS, and users can access the different DBMS which form the MDBS (see Figure 3.1).

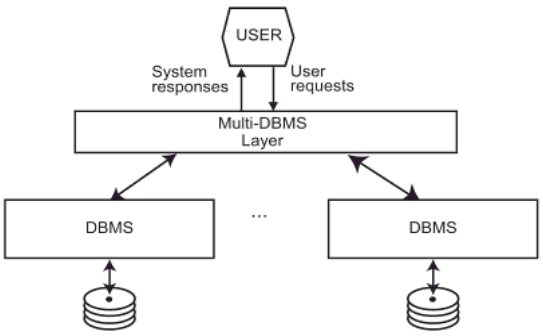

Figure 3.1.: Components in a multidatabase system [TP11]

A popular implementation architecture for a MDBS is the mediator/wrapper approach [TP11]. Mediators exploits knowledge to create information for upper layers, while wrappers provide mapping between different views, e.g. relational vs. object-oriented views. We can consider our approach as a MDBS with some modifications and less functionalities. In the first place, using the ESB as a single entrance point to the data system while managing different backend autonomous Cloud or traditional data stores comply with the main concept of a MDBS. Furthermore, Cloud data store providers may implement the same distributed database model, whereby we could find two logical levels for accessing the physical data. However, we do not accurately follow the mediator/wrapper approach. In our approach we exploit data provided by the tenant during the migration decision and process, by providing an interface to register the backend data store/s information in our system, for future routing purposes. Furthermore, compatibility information is registered in order to apply the needed query or data transformation between data stores. However, the transformation is out of the scope of this diploma thesis, and the support of table joins between databases located in different Cloud data stores are out of scope as well.

As described in the previous chapter, multi-tenancy is one of the main requirements in a Cloud environment. Muhler, Essl, and Gomez provide an extended version of ServiceMix 4.3, which supports multi-tenancy at two different levels: communication, and administration and management [Muh12], [Ess11], [Sá12]. However, their prototype supports tenant isolation at the level of tenants. A DBMS, e.g. MySQL, by default provides access to one default user and supports multiple users creation [Ora13]. Therefore, in our approach we must not only consider isolation at the tenant level, but also at the user level. We assume that the tenant is the default user which migrates his data store to a Cloud environment, but the migrated data store may contain one or more users. In our prototype we ensure tenant and user isolation at both communication, and administration and management levels.

Over the past decades, caching has become the key technology in bridging the performance gap across memory hierarchies via temporal or spatial localities; in particular, the effect is prominent in disk storage systems [HLS+12]. Han et al. investigate how cost efficiency in a Cloud environment can be achieved, specially in applications which require a high I/O activities number, and present a CaaS (cache-as-a-service) model. Cloud providers offering data storage solutions present pricing models based on the storage size, usage per time, or number of requests. Amazon RDS costs $0.025 per hour for a Micro DB Instance usage [Amac], while Amazon DynamoDB $0.01 per hour for every 50 units of read capacity [Amaa], and Google Cloud Storage $0.01 per 1000 PUT, POST, GET requests per month [Gooc]. An I/O-intensive application whose database is hosted in the Cloud may produce a significant economic cost. The cost of continuously retrieving data from the Cloud data store, when existing temporal proximity between the data accessed, can be considered unnecessary, and reducible. Furthermore, the application's overall performance can be reduced due to the network latency and, in the scope of this work, the use of an ESB to access the Cloud data store. In this diploma thesis we do not provide cashing as a service, but include cashing support to the sharded-based router pattern described in [SAB+12]. Uralov enhaces ServiceMix-mt with cashing support for dynamic discovery and selection of Cloud data hosting solutions [Ura12]. However, we must adapt and extend it due to the lack of support of functionalities we require and the lack of full OSGi compliance.

3.1. SQL Approaches

In this section we discuss about the different approaches ESB vendors take into account when accessing or storing data in SQL database systems, and compare it to the support we provide in this work. Most of the ESB vendors provide support for data persistency in RDBMS. However, it is restricted to output connections to DBMS. Fuse ESB provides JDBC data source support, enabling users to connect to a database and make SQL based queries and updates [Fus11]. This connection can be established during routing, when receiving a JMS or HTTP message in a consumer endpoint, etc. However, they do not provide support for native database communication protocol incoming messages, e.g. MySQL or PostgreSQL communication protocol. Thus, data consumer endpoints supporting native database protocols are not deployable. The same limited support is provided in the integration framework Camel, in its component Camel-jdbc [Thec].

JBoss presents its Enterprise Data Service Platform containing data services with SOA support, and an ESB [Red11]. Any organization currently using an ESB can interoperate their Data Services Platform through open standards [Red11]. Accessing the database layer using methods which implement the SOA, e.g. Web services, requires the application to support such methods. One of the main goals in this diploma thesis is to minimize the adaptations in the DAL when migrating the data to the Cloud. Furthermore, as we can see in Figure 3.2, connection utilizing native database protocol is established directly from the business application logic to the data service. In our approach we propose a native database connection through the ESB to the data service.

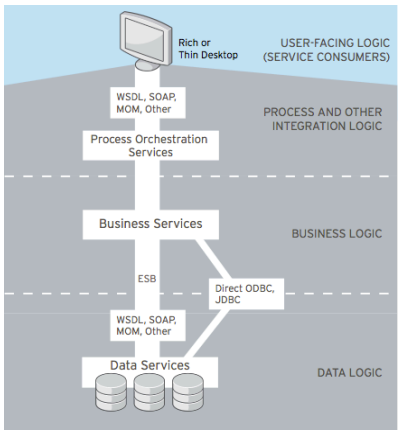

Figure 3.2.: JBoss Enterprise Data Services Platform [Red11]

3.2. NoSQL Approaches

In this section we discuss the different supports for accessing NoSQL databases of the main Cloud data stores vendors in the market, emphasizing on the communication protocols supported, the interfaces provided, and the Cloud storage standard CDMI.

Schilling provides an overview on the different supported communication protocols, API, Command-line Interface (CLI), and message payload formats in Cloud providers offering NoSQL storage solutions. Amazon NoSQL provides two main supports for accessing and modifying data in their data stores: AWS SDK for Java, or a REST API. The former requires the installation and the adaptation of the DAL to the set of functionalities provided by their JAVA SDK [Amaa]. Therefore, due to our main goal of minimizing changes in the on-premise application, we discard this approach, and do not provide further information. The latter consists in a set of functions accessible through HTTP messages and supporting a JSON payload format. Google Cloud Storage provides a REST API where XML payload is supported as default, and JSON in an experimental phase [Gooc]. MongoDB provides both a Java API and a REST API for accessing the databases [10G].

In the first place, we note that most of the NoSQL database providers offer their own API developed in different languages for accessing their data stores. However, as discussed before, the installation of an external API forces to make a considerable number of modifications. Most of the Cloud data store providers support for REST operations. The CDMI standard defines a Cloud data access standardized protocol JSON over HTTP and provides a set of operations on the stored data [Sto12]. Therefore, we consider this as the standardized access method and provide support for it in our prototype.

In the second place, we notice that one or more Cloud data store providers offering a NoSQL database categorized into a specific NoSQL family name their data storage structures in different ways. For example, Amazon DynamoDB stores tables and items, Google Cloud Storage buckets and objects, Amazon SympleDB domains and items, and Amazon S3 bucket and Object. However, we find that the different vendor's storage structures can be grouped

into two main groups: main information structure, and secondary information structure. This grouping solution for this difference is discussed in detail in Chapter 5.

4. Concept and Specification

In this chapter we first provide an overview of the system and its components which provide support for accessing on-premise and off-premise Cloud data stores after migrating the data to the Cloud. In the second part of this chapter we specify the functional and non-functional requirement the system must fulfill, and provide a list of the use cases, which extend the use cases description provided in [Ura12] and [Muh12].

4.1. System Overview

To provide transparent access support for migrated data, we present in this section an overview of the system, and its components. As we can see in Figure 4.1, we divide the system into two main parts: the *Cloud Data Migration Application*, and the Cloud data access subsystem, which we name in this diploma thesis CDASMix (Cloud Data Access Support in ServiceMix-mt). However, in this diploma thesis we do not focus on the *Cloud Data Migration Application*, but include it in the system's overview in order to explain the role of CDASMix in the context of the migration of the DL to the Cloud. We consider the different tenant's applications hosted in their environment not as part of our system, but as consumers of the services provided by it. We must specify that the system overview described in Figures 4.1 and 4.2 shows the state after the data migration, when the data is already hosted in the backend Cloud provider. However, we include the migration process explanation in this section.

In the first part of our system, the *Cloud Data Migration Application* provides support for the data migration process, from an traditional to a Cloud data store, or between Cloud data stores [Bac12]. After the tenant provides the required source and target data store configuration, the application calculates possible incompatibilities between data sources, and presents them to the tenant. If they exist, the tenant must resolve the incompatibilities before migrating the data. In the end phase of the migration process data can be easily migrated to the Cloud by providing the application with the DBMS access credentials.

From the point in time where the data migration process is terminated, either the application or the tenant must choose if he directly connects to his data source in the Cloud, or if he prefers to transparently access his data in the Cloud utilizing our Cloud-enabled data bus. If the latter is chosen, either the application or the tenant must register which communication protocol is required and register access and configuration data in our registry, e.g. database type, database URL, access credentials, etc. For this purpose, we enhance the administration and management system's (JBIMulti2) Web service API with Cloud data access registering capabilities, as described in the following sections.

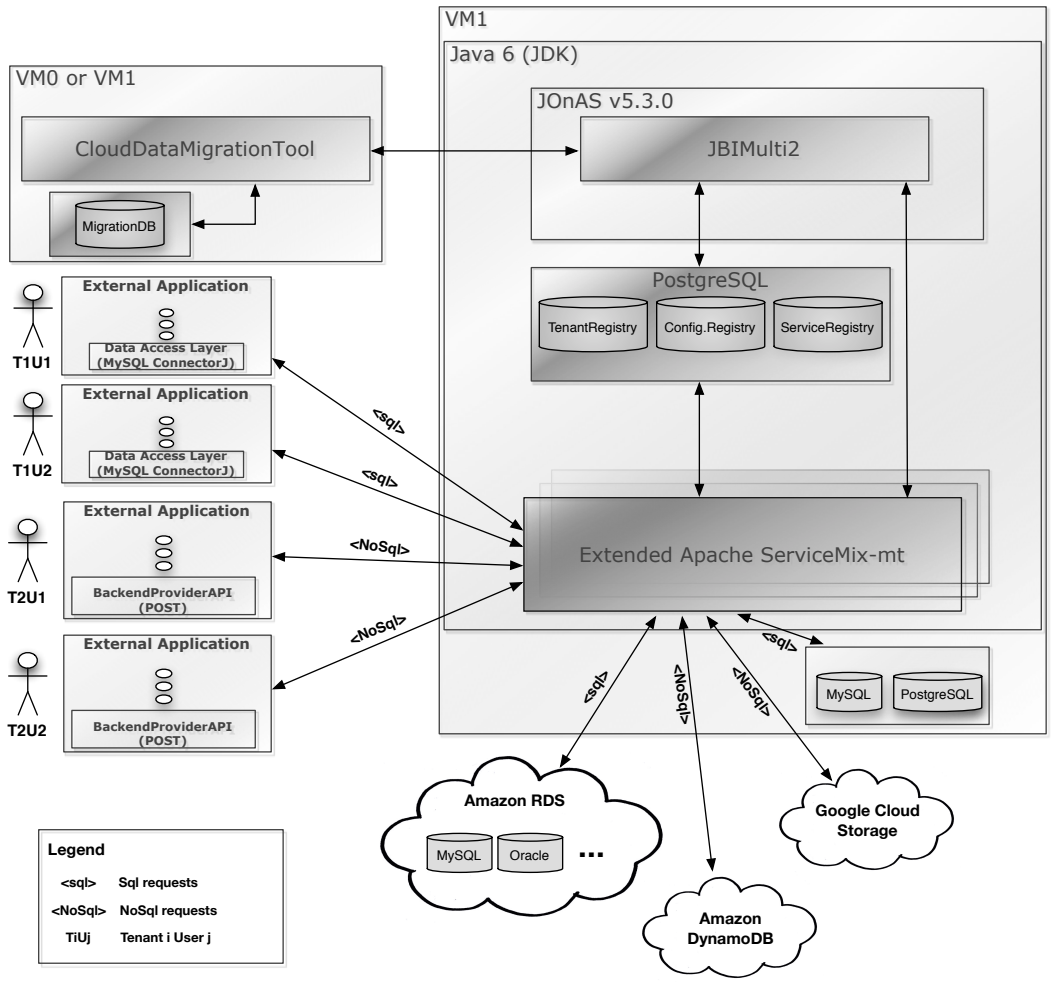

Figure 4.1.: Transparent Cloud data access system overview, including the *Cloud Data Migration Application* [Bac12]. Note: represents the system in the post-migration phase

The transparent Cloud data access access support is achieved by the interaction of three main components: JBIMulti2, registries containing tenant-aware information, and an extended version of ServiceMix-mt (see Figure 4.1). JBIMulti2 deploys in ServiceMix-mt the SAs containing the endpoint and routing configurations selected by the tenant, which support two different communication protocols: MySQL and HTTP. From this point the DAL of the tenant's application can retrieve and modify data in his data container in the Cloud through the ESB connecting to a single logical endpoint which connects to multiple physical backend data stores. In our approach we provide also the possibility, either to configure a connection to the traditional database, e.g. when a hybrid model is pursued, or to utilize a DBMS provided in our system, which is described in the following subsection.

4.1.1. Components

In Figure 4.2 we specify the main components which build the subsystem mentioned in Section 4.1. We highlight the components which require an extension, and the new components which are implemented and included in the system.

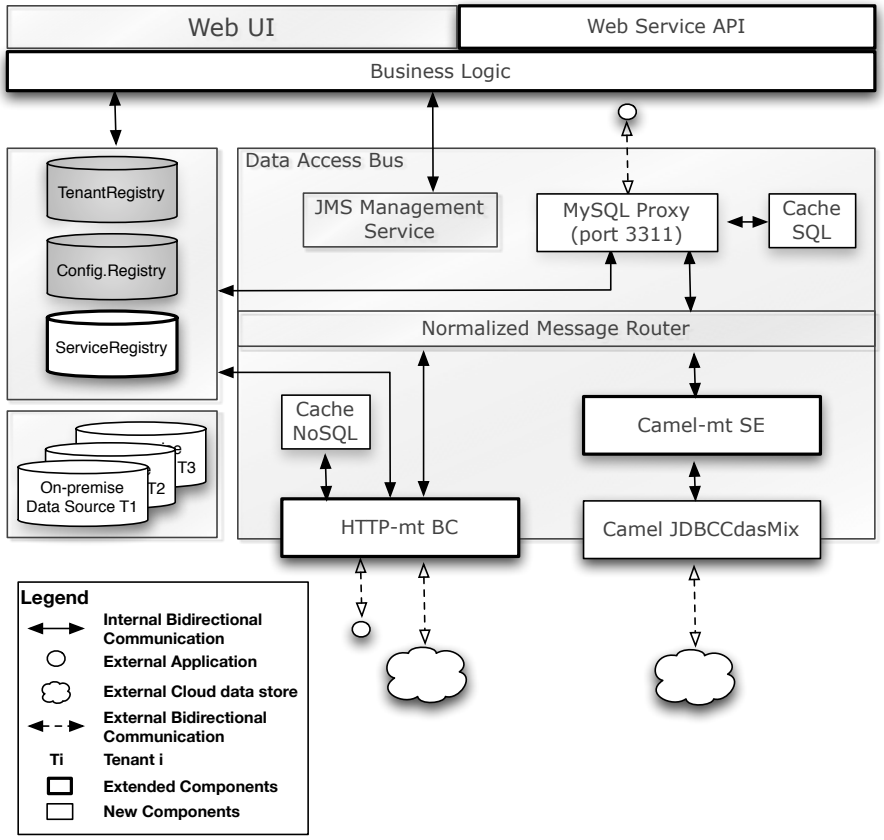

Figure 4.2.: Transparent Cloud data access components overview.

As described in Section 4.1, we extend the JBIMulti2 Web service API and its associated business logic. The operations we include perform access and modifications to one particular registry: the Service Registry. This registry persists information about services, its policies, and SAs deployed by one tenant. The last kind of information is the one we mainly focus on, due to the information which is contained in it: the tenant-aware endpoint configuration in ServiceMix-mt. Therefore, we extend this registry to persist the configuration about the data stores. We make a differentiation between data stores and name it source and target data sources, to be able to relate the one that the tenant physically accesses and the one which the tenant logically accesses, which is the one that the system physically accesses. To support transparent access to two different database types, we divide our architecture and implementation into the the communication protocols they support: MySQL for MySQL databases, and HTTP for NoSQL databases (see Figure 4.2).

For the first one, the single access physical endpoint is provided through a TCP port, which

then forwards the message to the appropriate tenant's endpoint in the multi-tenant Servicemix Camel component, and this to the component which physically connects to the backend SQL DBMS. For the second one, we extend the Servicemix-http-mt in order to physically connect to the backend NoSQL data stores.

The possibility of migrating a database to the VM instance where the ESB runs is also supported. However, we do not provide a multi-tenant DBMS where a database or table is shared between multiple tenants, since it is not a requirement in this diploma thesis. The ensured isolation in this case is the one provided by a DBMS between different databases. Furthermore, backup, restoration, administration, and horizontal scalability services are not supported. We provide in the VM instance where CDASMix runs a MySQL database system. The number and types of databases systems supported in the VM instance relies on the administrator, and the PostgreSQL database system where the system registries are stored must be independent from the PostgreSQL instance which hosts the migrated tenant's databases.

As represented in Figure 4.2, we enhance ServiceMix-mt with cashing support, in particular for the two different types of databases we support. The cashing mechanism supports storage of access control data, as well as retrieved data from the backend data store, e.g. a bucket, or a set of rows. The reasons for using a divided cashing system instead of a single one is explained in Chapter 5.

4.2. Multi-tenancy

In this section we detail the multi-tenant requirements the system must fulfill in order to ensure tenant-isolation at two levels: communication and storage. The final prototype must ensure a multi-tenant aware transparent access to data hosted in the Cloud. Although we provide storage support in our system for hosting migrated tenant's data (see Section 4.1), it does not implement a multi-tenant storage model, because this is not a goal in our design. Therefore, we rely on the multi-tenant storage models different Cloud providers implement.

4.2.1. Communication Requirements

The final prototype must not only support a multi-tenant, but also a multi-protocol communication between endpoints. ServiceMix-mt is shipped with the following multi-tenant aware BCs: HTTP, JMS, and E-mail [Sá12]. However, the existing communication protocols for data transfer purposes leads us to discard the JMS and E-mail. RDBMS, e.g. MySQL and PostgreSQL, implement their own protocol in their client/server model, at the TCP level of the network stack. At the client side the protocol is supported by the native drivers provided to the developers, e.g. MySQL Connector/J, and at the server side by the different components which build the database server [Ora13]. This fact forces us to provide a vendor-oriented communication protocol environment, by providing support for the different protocols in independent components, rather than in a single standardized component.

Communication in, and from the ESB must be multi-tenant aware. We divide the required isolation between tenants into the following sub-requirements:

- **Tenant-aware messaging**: messages received in the ESB and routed between the tenant-aware endpoints should be enriched with tenant and user information.
- **Tenant-aware endpoints**: in ServiceMix-mt tenants pack a common endpoint configuration packed in a SU, which is then deployed as a SA in ServiceMix-mt's JBI container [Sá12]. The multi-tenant aware BC dynamically modify the endpoint's URL by injecting tenant context in it. In a database system in our scenario we do not have only tenants as the main actors, but also the different users which can access a tenant's database. Therefore, the tenant-aware endpoints should be dynamically created by injecting tenant and user information in the endpoint's URL. Furthermore, we must ensure tenant and user authentication in the system.
- **Tenant-aware routing and context**: the deployment of tenant-aware endpoints should be followed by the creation of a tenant-aware context. Resources involved in a routing operation from one consumer endpoint to one provider endpoint can be shared between different tenants, but they must manage the routing operations in different tenant-aware contexts. The routing operations between two endpoints must identity the tenant and user who initiated the routing.
- **Tenant configuration isolation**: configuration data persisted in our system should be isolated between tenants. A tenant's endpoint configuration data contains sensible information which identifies and allows access to the tenant's backend data stores.
- **Tenant-aware correlation**: in a request-response operations, the response obtained from the backend data store must be correlated with the tenant's request to the system, and ensure that one tenant does not receive responses from another tenant's request.

4.2.2. Storage Requirements

Due to the fact that our system does not primarily requires multi-tenant aware storage support, but we rely on multi-tenant aware storage systems in the Cloud, we summarize the main requirements for isolating data between tenants in database systems.

Curino et al. identify as a primary requirement for a Cloud provider offering a DBaaS model security and privacy [CJZ+10]. A system running multiple database servers and each server multiple database instances must contain the necessary meta-data to provide tenant-aware routing in the system, and ensure that one tenant can only access the information in his database instance. Furthermore, privacy of stored data between tenants can be ensure by encrypting all tuples [CJP+11]. Curino et al. introduce *CryptDB*, a subsystem of a relational Cloud which provide data encryption and unencryption functionalities for persisting data, and for accessing data via SQL queries which are not aware of the encrypted storage mechanism in the system. However, it is known that the key challenge in managing encrypted data in the Cloud is doing it efficiently.

4.3. Dynamic Transparent Routing

Data access and modification from different back-end data stores must be supported in a transparent way between the tenants. We divide this requirement into two sub-requirements we consider our system must fulfill: Transparency, and Dynamic Routing.

4.3.1. Transparency

Giving a single logical view on a distributed database system abstracts the tenant from knowing the physical location of his data. The system must provide a single access endpoint for accessing and modifying data previously migrated to the Cloud. We must perform internally the necessary search operations in the registry containing the back-end data stores information, and the necessary mapping operations with the meta-data included in the tenants' requests. After those operations, our system must forward the requests to the back-end data stores and forward the responses to the tenants' requests.

4.3.2. Dynamic Routing

Providing transparency by exposing a single endpoint to retrieve data from different back-end data stores requires the system to support dynamism in its routing mechanisms. One tenant may migrate one database to the Cloud, or *shard* a database between different databases in the Cloud, or different Cloud providers. This fact forces us to support connections to the back-end data stores dynamically rather than statically. Furthermore, when query and data transformation is required due to version or schema direct incompatibility between the tenant's requests and the back-end data store support, transformation mechanisms must ensure transformation of queries and data in order to provide a full transparent access. When transformations are not needed, the message should not be routed through a transformer component. In the prototype developed in this diploma thesis query and data transformations are out of scope.

When *sharding* a database, the tenant can split the data between two or more databases, or database systems. In order to minimize the changes in the application's DAL, we must support a single physical frontend endpoint while processing the tenant's request through one or more physical back-end endpoints. Special cases such as queries containing JOIN operations in between tables stored in different back-end SQL databases are not supported. However, the system should support the execution of multiple SQL queries in one request, which is known as *multiple-queries*.

4.4. Cache

Cashing mechanisms in applications with high number of I/O operations benefits the application's performance, as described in Chapter 3. The prototype developed in this diploma

thesis provides support for application's I/O operations. Therefore, cashing is one of the main components which can drive to a better system performance.

In our system the cache must be shared between the tenants using it. Hence, data cashed must be previously enriched with tenant and user context information. Operations in our system which require data from local or remote databases, e.g. authentication operations or data retrieval operations, should utilize the cashing mechanism to reduce the response time of the system.

Tenant operations which perform changes in their data stored in the Cloud may lead to inconsistencies between the data persisted in the cache and the updated data in the Cloud. Freshness of data in the cashing system must be ensured.

4.5. Integration Requirements

Apache ServiceMix 4.x versions are built on an OSGi-based runtime kernel, Apache Karaf [ASM], [APA11b]. However, they provide JBI support for users migrating to newer versions. For new users it is recommended to consider JBI deprecated, and build the components and expose the services in OSGi *bundles*. We consider that developing our components as OSGi *bundles* eases the compliance with newer versions in ServiceMix, and enables loose coupling between components. When a component is modified, the components in the OSGi container which used the services of the modified component must not be redeployed. However, we find that the ServiceMix-mt component are JBI compliant. We must then provide integration support between the JBI components and the OSGi *bundles*. The NMR in ServiceMix eases the integration of the JBI and OSGi providing an API for accessing the NMR and creating message exchanges. However, the messages routed in ServiceMix-mt are in a NMF, which is a completely different format with the message formats supported in the MySQL, or JSON over HTTP communication protocols. The system must ensure the appropriate conversion and mapping operations for marshaling and demarshaling incoming and outgoing messages.

As described in previous sections, tenant's requests can be routed in ServiceMix-mt directly between one consumer and one provider tenant-aware endpoint when no query or data transformation is required. When transformation is required, we must design our components to be easily integrated with a future transformer component as an intermediary in the route between endpoints.

JBIMulti2 is the application built on top on ServiceMix-mt to enable administration and managements operations on the ESB. Therefore, the deployment of tenant-aware endpoint configurations can be only done through JBIMulti2. The *Cloud Data Migration Application* lacks of connection and access to the registries where the tenants' configuration data is stored. In order to avoid a database connection from the migration application, which may be hosted in an external server, to the registries which contain the tenant meta-data, we must provide an interface to allow either the tenant or the migration application to register the data store meta-data.

4.6. Use Cases

In this section we extend the tenant operator use cases which are described in Muhler's and Uralov's approach [Muh12], [Ura12]. The tenant operators are the users with less permissions in JBIMulti2, who perform service registration and endpoint configuration operations in ServiceMix-mt. An overview of the set of use cases for the tenant operators is presented in Figure 4.3. The set of use cases we add to the previous version are highlighted, and described in detail.

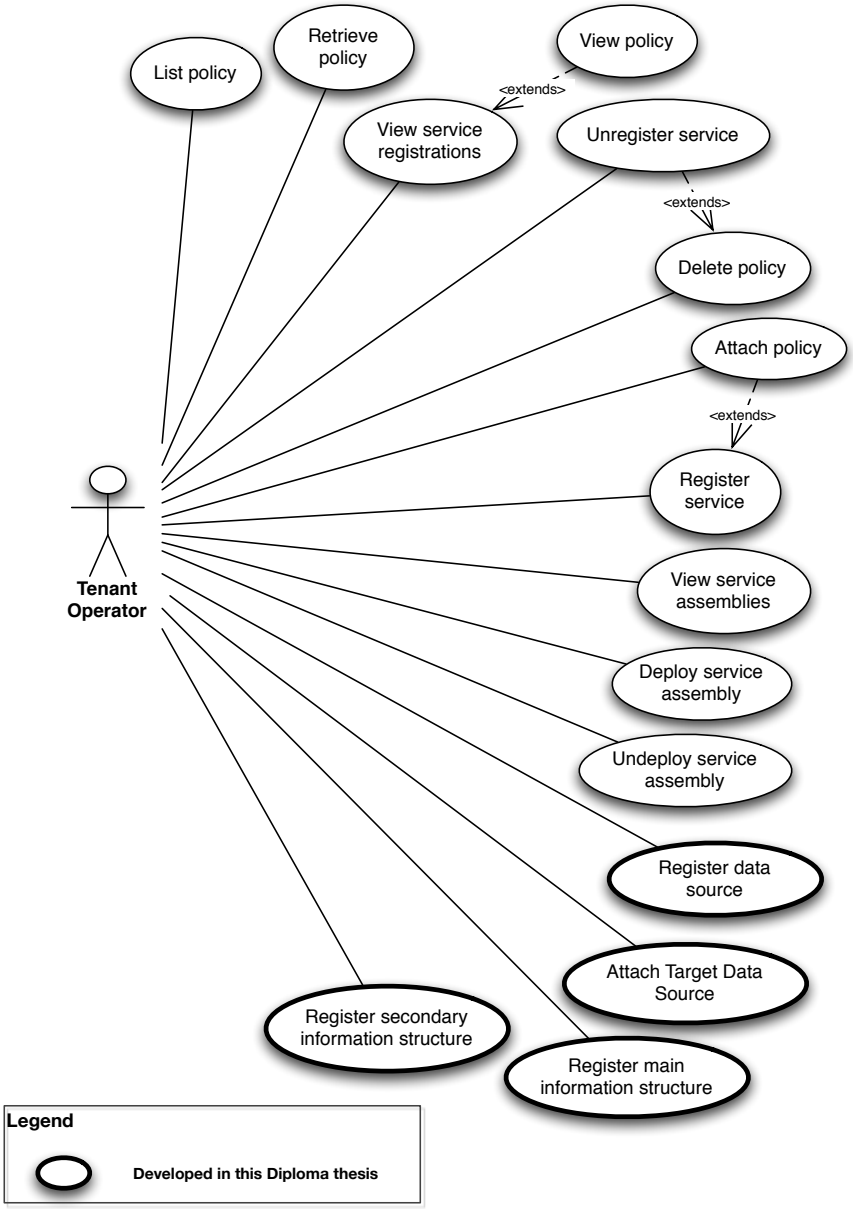

Figure 4.3.: Use case diagram for tenant operator.

Name	**Register Data Source**
Goal	The tenant operator wants to register a source and backend data source configuration data and associate it to an existing service assembly.
Actor	Tenant Operator
Pre-Condition	The tenant operator has the permissions in the service unit contingent and has registered a service assembly with no datasources associated to it.
Post-Condition	Source and backend data source configuration data is stored successfully, and associated with a tenant's operator service assembly.
Post-Condition in Special Case	Source and backend data source configuration data are not stored, and not associated with the specified service assembly.
Normal Case	1. The tenant operator selects the service assembly to which the data sources are associated. 2. The source and target data sources configuration data are registered in the system, and associated to the specified service assembly.
Special Cases	1a. The service assembly does not exist. a) The system shows an error message and aborts. 2a. The service assembly is already associated with a source data source. a) The system shows an error message and aborts. 2b. The source data source configuration data already exists. a) The system shows an error message and aborts. 2c. The target data source configuration data already exists. a) The system shows an error message and aborts.

Table 4.1.: Description of Use Case *Register Data Source*.

Name	**Attach Target Data Source**
Goal	The tenant operator wants to attach one backend data source configuration data and associate it to an existing service assembly and source data source.
Actor	Tenant Operator
Pre-Condition	The tenant operator has the permissions in the service unit contingent and has registered a source data source.
Post-Condition	The target data source is attached successfully to the specified source data source, and associated with the specified service assembly.
Post-Condition in Special Case	Target data source configuration data is not attached, and not associated with the specified service assembly.
Normal Case	1. The tenant operator selects the service assembly to which the source data source is associated. 2. The tenant operator selects the source data source to which the target data source must be associated. 3. The target data source configuration data is attached to the specified source data source, and associated to the specified service assembly.
Special Cases	1a. The service assembly does not exist. a) The system shows an error message and aborts. 2a. The source data source does not exist. a) The system shows an error message and aborts. 3a. The target data source already exists. a) The system shows an error message and aborts.

Table 4.2.: Description of Use Case *Attach Target Data Source*.

Name	**Register Main Information Structure**
Goal	The tenant operator registeres a main information structure and associate it with the specified source and target data source.
Actor	Tenant Operator
Pre-Condition	The tenant operator has the permissions in the service unit contingent and has registered the specified source and target data source.
Post-Condition	The source and target main information structure meta-data is registered successfully, and associated with the specified source and target data source.
Post-Condition in Special Case	Source and target main information structure are not registered, and are not associated with the specified source and target data source.
Normal Case	1. The tenant operator selects the service assembly to which the source data source is associated. 2. The tenant operator selects the source and target data source to which the source and target main information structure is associated. 3. The source and target main information structure are registered and associated with their source and target data source.
Special Cases	1a. The service assembly does not exist. a) The system shows an error message and aborts. 2a. The source or target data source does not exist. a) The system shows an error message and aborts. 3a. The main source or target information structure already exists. a) The system shows an error message and aborts.

Table 4.3.: Description of Use Case *Register Main Information Structure*.

Name	**Register Secondary Information Structure**
Goal	The tenant operator registeres a secondary information structure and associates it with the specified source and target main information structure.
Actor	Tenant Operator
Pre-Condition	The tenant operator has the permissions in the service unit contingent, has registered a source and target data source, has registered a source and target main information structure, and the source and target datasources are NoSQL databases.
Post-Condition	The source and target secondary information structure meta-data are registered successfully, and associated with the specified source and target main information structure.
Post-Condition in Special Case	Source and target secondary information structure are not registered, and are not associated with the specified source and target main information structure.
Normal Case	1. The tenant operator selects the service assembly to which the source data source is already associated. 2. The tenant operator selects the source and target data source to which the main information structure are associated. 3. The tenant operator selects the source and target main information structure to which the secondary information structure is associated. 4. The source and target secondary information structure are registered and associated with their source and target main information structure.
Special Cases	1a. The service assembly does not exist. a) The system shows an error message and aborts. 2a. The source or target data source does not exist. a) The system shows an error message and aborts. 2b. The source and target data source is a MySQL database. a) The system shows an error message and aborts. 3a. The main source or target information structure does not exist. a) The system shows an error message and aborts. 4a. The secondary source or target information structure already exists. a) The system shows an error message and aborts.

Table 4.4.: Description of Use Case *Register Secondary Information Structure*.

4.7. Web Service Interface

The system must provide an interface with a set of operations to allow both the tenants and the *Cloud Data Migration Application* to interact with the system and configure the connections between the single physical consumer endpoint and the multiple provider endpoints.

4.8. Non-functional Requirements

In this section we list and describe the non-functional requirements our system must fulfill. The non-functional requirements described in this thesis are independent from the ones satisfied by the Cloud data store providers, which are specified in the Service Level Agreement between the Cloud provider and the user.

4.8.1. Security

Securing the tenant context information in our system is one of the main requirements we must fulfill. Tenant context in our system does not only contains tenant configuration data, but also contains the necessary meta-data from the backend data sources, which include the databases schemas and its access credentials. In order to ensure confidentiality and integrity of the data migrated to the Cloud, tenant configuration data must be visible only to the system, and not transferred to third parties through the Web or Web service interface.

4.8.2. Backward Compatibility

In this diploma thesis we must face to two architectural tendencies in ServiceMix-mt: OSGi-based components and JBI-based components. Backward compatibility with the multi-tenant JBI components developed in [Sá12], [Muh12], and [Ess11] must be ensured. At the same time we must build the new components following the OSGi tendency.

Compatibility with non multi-tenant aware endpoint configurations must be ensured in the system. In ServiceMix-mt we extend existing components, e.g. ServiceMix-http-mt, ServiceMix-camel-mt, JDBCCdasmix, etc., and deploy them as custom components. By deploying the extended components as separate custom components, we avoid conflicts with the non multi-tenant aware components ServiceMix is shipped with, e.g. ServiceMix-http [ASM], ServiceMix-camel [ASM], Camel-jdbc [Thec], etc. Configuration of non multi-tenant aware endpoints is still supported in ServiceMix-mt.

4.8.3. Performance

As discussed in previous sections, different performance indexes usually drive down in a system which relies on a high I/O operations number. Therefore, we must integrate additional mechanisms, e.g. cashing, in order to alleviate the performance fall. We mention in Chapter 3 not only performance benefits in terms of system efficiency when cashing, but also an economic efficiency when access to data involves an economic cost.

4.8.4. Scalability and Extensibility

The system should offer clustering functionality and scale appropriately in a Cloud infrastructure. JBIMulti2 enables administration and management of more than one instance of ServiceMix [Muh12]. The horizontal scalability is out of the scope of this diploma 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.

4.8.5. Maintainability and Documentation

The source code provided in this diploma thesis should be well commented and documented. The needed documentation should be shipped in the system's package in order to provide the necessary steps and tips to lead to a running system and possible future extensions. Furthermore, a test suite containing the main operations and data needed to run on the Web service interface to configure the system must be provided for future automation purposes.

5. Design

In this chapter we present the architectural solution taken into account to build the system which fulfills the requirements specified in Chapter 4. Due to the required communication support for SQL and NoSQL databases, we separate the architectural approaches and provide them separately. JBIMulti2 and ServiceMix-mt are the subsystems we must reengineer in order to aggregate transparent and dynamic routing functionalities. Therefore, we also provide in this chapter the needed extensions in the components conforming the system, e.g. service registry in JBIMulti2, and NMF in ServiceMix-mt.

5.1. Service Registry

The service registry developed in Muhler's approach [Muh12], and extended in Uralov's work [Ura12], contains information related to the services, and policies which can be dynamically retrieved by the tenants. The services contains a WSDL file and implements an interface for its access. Furthermore, policies are attached to it in order to provide a dynamic discovery of services based on a set of rules. In this diploma thesis, we do not focus on service registering or service discovery, but on the endpoint configurations which are deployed by the tenants. These are stored in the service assembly entity, where the deployed SAs are persisted with tenant context information.

A SA contains the SUs where the endpoint configuration is described. Therefore, we consider the service assembly entity as the start point of our extension in the service registry schema. We use the data sources name to refer to the user's data sources, and categorize them as source or target data sources in the *dsLocationId* attribute. The data source name in source data source registrations must equal to the database name the user includes in his request to CDASMix. Source data sources are the ones the tenant physically accesses in our system, while the target data sources are the ones that the tenant logically accesses, but the system physically accesses. Physical location of the target data sources are described in the *dsEndpointURL*, and define the database endpoint's host, port, and database name (an example is provided in Listing 7.1). A SA contains the routing information configuration for routing operations between endpoints. Hence, one service assembly identifies the tenant's user source data sources. The source datasource can be connected to one or more target data sources, as shown in Figure 5.1. The self-referencing relation in the TenantBackendDS entity allows to reference one source data source with n target data sources.

As discussed in Chapter 3, providing support for two different databases models, and for its different families (e.g. in NoSQL databases) forces us to dissect their storage structures, and provide a generic and extensible model in order to store its meta-data. The SQL data stores have two main storage structures: database and table. Inside the table we find a

second storage structure, rows and columns. The NoSQL databases are divided into different families, and for each family we find between the different vendors different namings which refer to the same storage structure. However, they all have in common the support of a main, and secondary storage structures.

The data source entity stores the necessary information for providing access in the system, and for accessing the target databases in the target data sources. The configuration information is provided either by the tenant, or by the *Cloud Data Migration Application*, during and/or after the data migration process to the Cloud. We provide isolation in the registration of sensible information, e.g. access credentials, by indexing the table with the tenant and user Universally Unique Identifier (UUID). Whether transformation operations are required is detected by analyzing the value contained in the *dsType* attribute. The format of its value must comply the following standard: family-mainInfoStructure-secInfoStructure-version, e.g. sql-database-table-mysql5.1.2 for a MySQL database system, or nosqlKeyValue-bucket-object-1.0 for a bucket in Google Cloud Storage [Ora13], [Gooc].

The main information structure entity persists the data related to the database's main storage structure, e.g. table in MySQL databases, and table, bucket, or domain in NoSQL databases. These storage structures are identified by its name in the backend data store. The locationid attribute in this entity categorizes one main information structure as source or target main information structure. A composite key formed by the name, locationid, and tenantid, is not possible in this entity. One tenant id may have the same name to identify two main information structure which are of a different database type, e.g. one SQL table, and one NoSQL bucket with the same name. Therefore, the entries in this entity are identified by a incremental id.

The secondary information structure is used in our system only to persist meta-data for NoSQL databases. The knowledge level in the system of tenant's migrated SQL databases lowers to the table storage structure, and one database is uniquely associated with one tenant, user, and endpoint in the target database system, e.g in Amazon RDS one database is accessed by the database instance identifier, and the host name [Amac]. However, different main storage structures can be accessed in NoSQL databases through the same user's endpoint. Therefore, we need to persist one or more main storage structures per database system endpoint, and provide a second level, the secondary information structure, to register a tenant's NoSQL database meta-data. For example, one tenant can create one or more buckets in the Google Cloud Storage system, and access them through the same endpoint [Gooc]. Buckets store one or more objects, which are identified by their name. We must then associate a tenant's NoSQL data store with one or more buckets, and these with one or more objects. A composite key formed by the name, tenantid, and locationid, is not possible, for the same reason discussed previously. One tenant id may have the same name to identify two secondary information structure which are of a different NoSQL database type, e.g. one document, and one object with the same name. Therefore, the entries in this entity are identified by a incremental id. As the relationships between secondary and main information structure, main information structure and data source, and between data sources (source and target) are a one to one relation, we avoid with the cascade option deleting information which might be

related with multiple entities, e.g. one target data source which is associated with two source data sources.

The entities related to the upper entities, e.g. a second information structure related to a main information structure, and this to a data source, and to a service assembly, are set with the Java persistence property *Cascade*. The service assembly is the upper entity which defines the connection between one source, and multiple target data sources. The undeployment of a service assembly involves the deletion of the data sources which are related to the service assembly, and the main and secondary information structures.

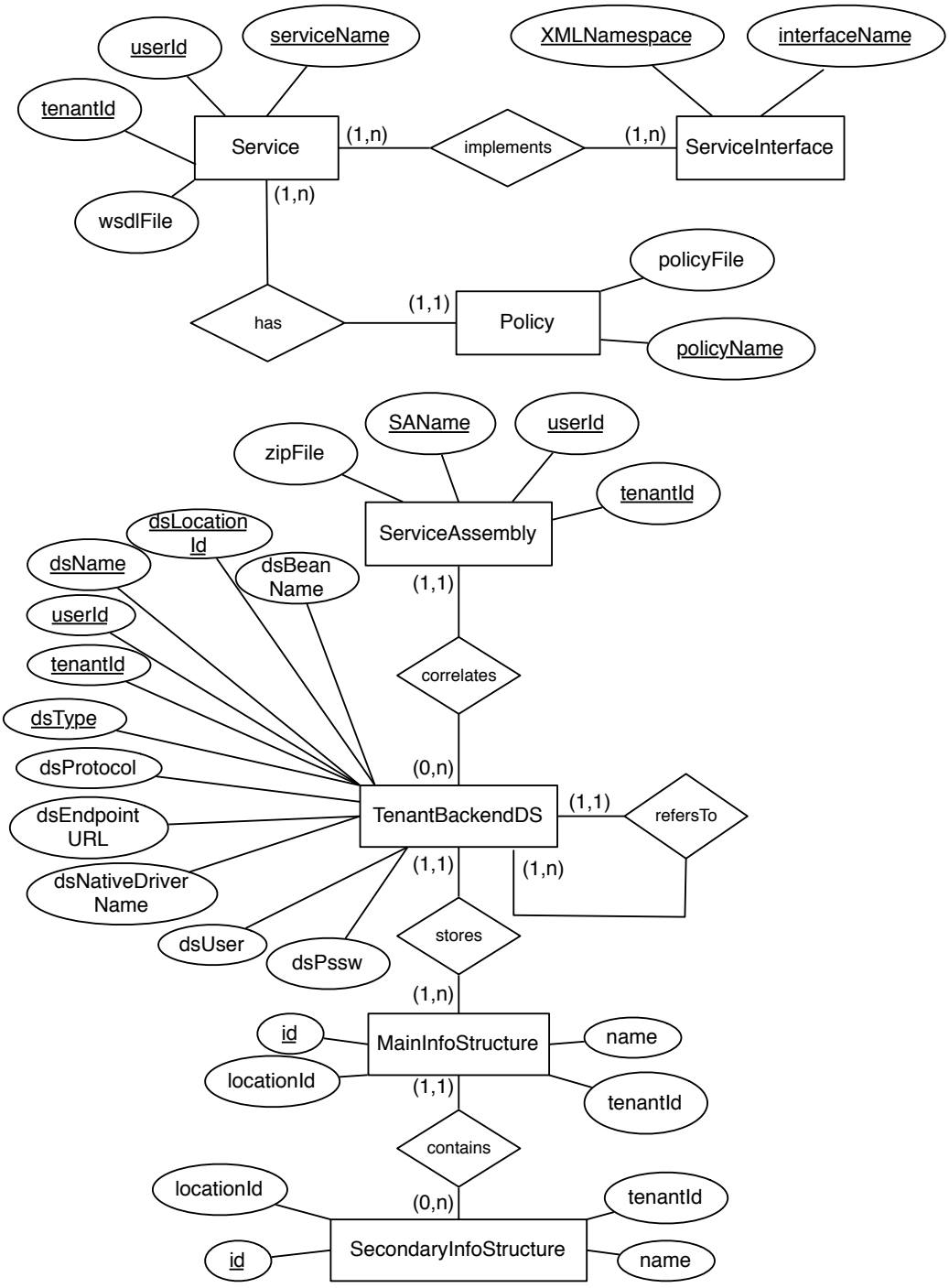

Figure 5.1.: Extended service registry ER Diagram using (Min, Max) notation. Note: extended from outputs in [Muh12] and [Ura12]

5.2. Normalized Message Format

In this diploma thesis we modify the NMF design in [Sá12], due to the different data types which are transferred in it. Gomez presents a NMF where the data is transferred in its body in XML format, which suites for the communication protocols his prototype supports (SOAP, JMS, and E-mail), and the text data type contained in the requests. Furthermore, the multi-tenant aware routing supported in his approach is static between endpoints (e.g. one incoming request is routed from one tenant's consumer endpoint to one tenant's provider endpoint). The complexity of the data types transferred in the system and the dynamic routing between backend Cloud data stores leads us to modify the meta-data transported in the NMF properties section, and to send the data in the attachment section. We are also forced to design two different data structures for the NMF: one for request messages, and another for response messages. The data and meta-data contained in the MySQL requests and responses messages require the utilization of different data structures to transmit them in the NMF (see Listing B.1). MySQL requests contain user, authentication, etc. meta-data, while the MySQL response contain meta-data which describes the data transferred.

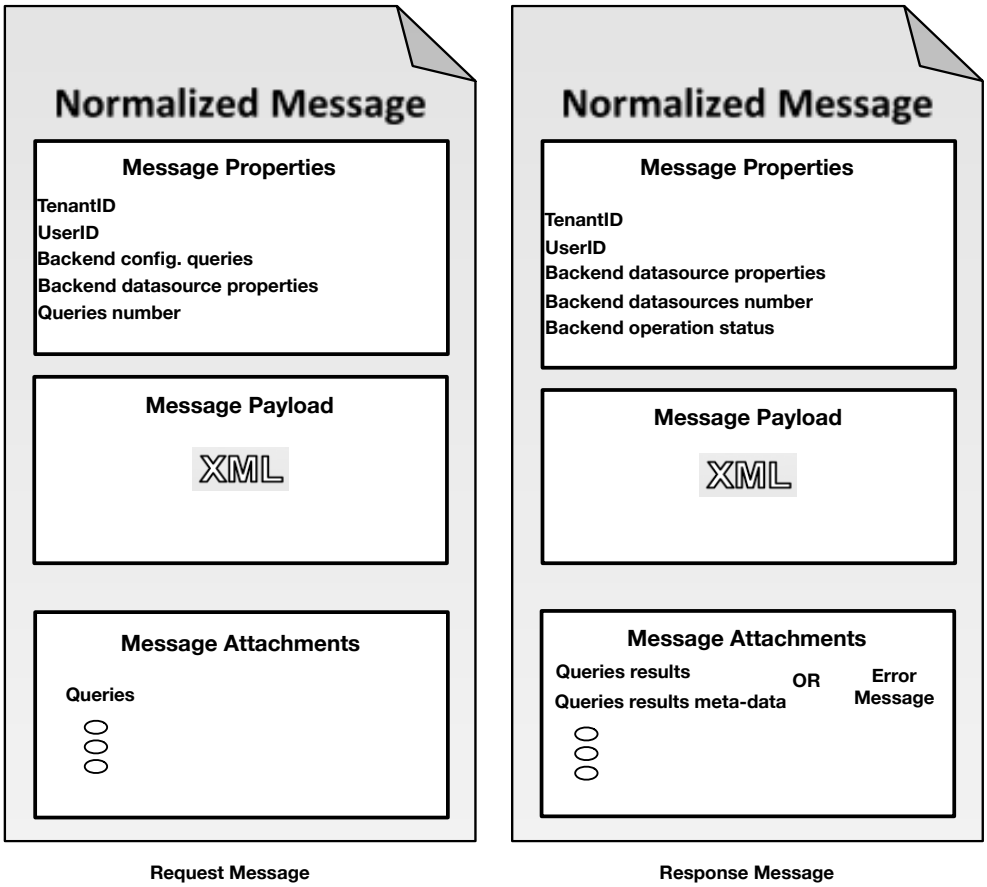

Figure 5.2.: Design of the Normalized Message Format used in the system.

Data access or modification requests to backend data stores through ServiceMix-mt are

received in the vendor's communication protocol, e.g. MySQL communication protocol for MySQL database systems, or JSON over HTTP for NoSQL databases, must be marshaled to the NMF structure. The NMF sections we use for incoming requests are the message properties, and the message attachment (see Figure 5.2). In the message properties we attach tenant context information (tenant and user UUID), and the request's meta-data, which is divided into:

- Backend configuration queries: server configuration queries sent prior to the request query, e.g. setting the predefined language, character encoding, etc. These must be executed in the backend database system, and are sent to the backend database system prior to the request query.
- Backend data source properties: in this property structure the set of backend data sources meta-data are stored, e.g. access credentials, main (and secondary) information structures names, data source type, etc.
- Queries number: the number of queries which are sent as attachment. The default value is one, but in multi-querying this value increases.

The NMF body supports text data format. The data transferred to or retrieved from a backend Cloud data store is represented as different data types, e.g. string, integer, float, binary, etc., and must be transferred as binary data. Therefore, the queries contained in the user's requests are sent in the attachment section of the NMF, which supports objects serialization. The queries are stored in a vector structure which can contain multiple queries. The NMF request must be demarshaled to the appropriate database system's communication protocol before forwarding the user's request to the backend database system.

Responses from the backend data stores are correlated with the user's request, marshaled, and sent back in the response NMF (see Figure 5.2). Data retrieved from a data store, e.g. in a data retrieval or update operation, contains both data and meta-data. Both informations are stored in vector structures in the attachment section of the response NMF. In case of error, the error message is stored in the response NMF, and forwarded back to the user. In the message properties section the tenant context information (tenant and user UUID) is stored, and the response meta-data, which contains the following information:

- Backend data source properties: in this property structure the set of backend data sources meta-data are stored, e.g. main (and secondary) information structures names, data source type, etc.
- Backend data source number: the number of targeted data stores.
- Backend operation status: the overall operation status. We consider operations over one or more target data stores as atomic. If one or more backend data stores returns an error, the operation status is set to error, and an error is forwarded back to the user.

The message body may be used for sending structured data in XML format. However, we do not send information in this NMF section in this diploma thesis, due to the need to transfer the requests and response data as binary serializable objects (serializable vectors) in order to

avoid errors in the data content if the data is transformed to string, and structured in XML format.

5.3. SQL Support Architectural Overview

In this section we provide an overview of a preliminary, and final architectural approaches designed in this diploma thesis, in order to support a transparent data access to backend SQL databases. We first expose the integration approaches we should consider, and the main problems found when implementing the first approach which led us to design a second architectural approach.

5.3.1. Integration

As described in Chapter 4, we build the new components in ServiceMix-mt following the OSGi compliance. However, these must interact with components which follow the JBI specification. The integration between components built for different containers in ServiceMix-mt must be done at two levels: messaging, and resources sharing. The ServiceMix-mt NMR API OSGi bundle exposes a set of operations for sending messages through the NMR to a specified target endpoint. Hereby we can perform message exchanges between endpoints configured on OSGi bundles and endpoints configured on JBI components, and provide communication support between components hosted in the two containers.

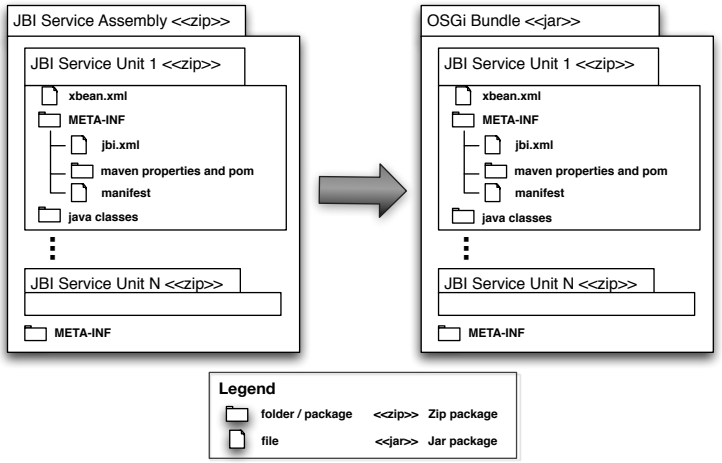

Figure 5.3.: ServiceMix 4.x repackaging mechanism for deploying JBI components in OSGi container.

The resources sharing integration level refers to the deployment of JBI components in the OSGi container, and the utilization of packages exposed by OSGi bundles in the OSGi container in a loosely coupled manner. The former is part of the integration between containers provided in ServiceMix 4.x versions and described in Figure 5.3. The deployment mechanism of a

JBI SA into an OSGi container is simple: repackaging of the SA as a JAR. However, this cannot be consider a full integration in the OSGi container. OSGi bundles contain in their *META-INF* folder one fundamental file for the OSGi kernel: the *manifest* file. This contains a description of the bundle, the packages it imports, and exports. Imported packages can be either statically stored in the bundle or imported from third party bundles, and exported packages are the ones which exposed to third party bundles. These can be imported with an internal class loading mechanisms developed in the OSGi container. The repackaging of the SA into a JAR, as it is shown in Figure 5.3, contains the *META-INF* folder, and the *manifest* file. However, the latter only contains information about the author, date of creation, but it does not contain information related to the exported packages, and the needed packages to be imported. This *manifest* file describes the SA, and not the SUs. Java classes and package importing description are contained in the SU package, and not in the SA package. Therefore, the OSGi container cannot register in its registry the packages it exports as a service, and cannot load the imported packages to the bundle context. This fact forces us to statically include in the SA, and the SU the packages which are referenced in each SU, and leads to scalability constraints with the tenant-aware deployment process of the system.

ServiceMix 4.x versions are shipped with different JBI BCs packed as an OSGi bundle. However, they are deployed as OSGi bundles, and not as SAs, in order to enable loose coupling and package sharing between components in the OSGi container. ServiceMix-mt allows the deployment of the JBI BCs, but deployment of JBI BCs as OSGi bundles is not supported in the JBIMulti2 application. This lack of support forces us to design a second architectural approach, as described in the following sections.

5.3.2. Approach 1

SQL database systems provide access to their databases through an endpoint, which is represented as an URL. The native driver used in the data access layer of an application connects to the endpoint, authenticates, sends the query, and reads the response. Therefore, we must support in our system the same operational steps. As discussed in this diploma thesis, the communication protocol varies between different vendors. In this diploma thesis we provide support for incoming MySQL messages. Tenants must access our system through a single physical endpoint. This endpoint is provided by a MySQL proxy which is enriched with authentication, cashing, marshaling, and demarshaling operations. As shown in Figure 5.4, the MySQL Proxy Bundle implements the MySQL server operations which are related with the client/server communication protocol. In Figure 5.4 we specify a server running on port 3306. However, this value can be configured before the deployment of the OSGi component. This component is built as an OSGi bundle and its packages are exported as services in the OSGi container. It interacts with three different components in the system: the NMR API, the Cache, and the Service Registry.

Cashing mechanisms are implemented in both the Registry-Cache, and the MySQL Proxy Bundle. The former provides an API for creating cache instances, and for persisting and retrieving data. We create a separate cache instance for the SQL support due to the need of a custom key creation mechanism which may not coexist in a shared cache between different

bundles, as well as the needed isolation of sensible tenant configuration information from third party bundles. The system provides a set of operations which ease the creation of multi-tenant aware keys for persisting frontend authentication data, and queries results form the backend database systems.

The NMR API is shipped in ServiceMix as an OSGi bundle which exports its API as an OSGi service. The set of operations included in its API allows OSGi bundles to create, send, and receive message exchanges to JBI endpoints (see Figure 5.4). Before creating the message exchange, the MySQL Proxy Bundle must build dynamically the target endpoint's URL by injecting the tenant context information, and service and endpoint name.

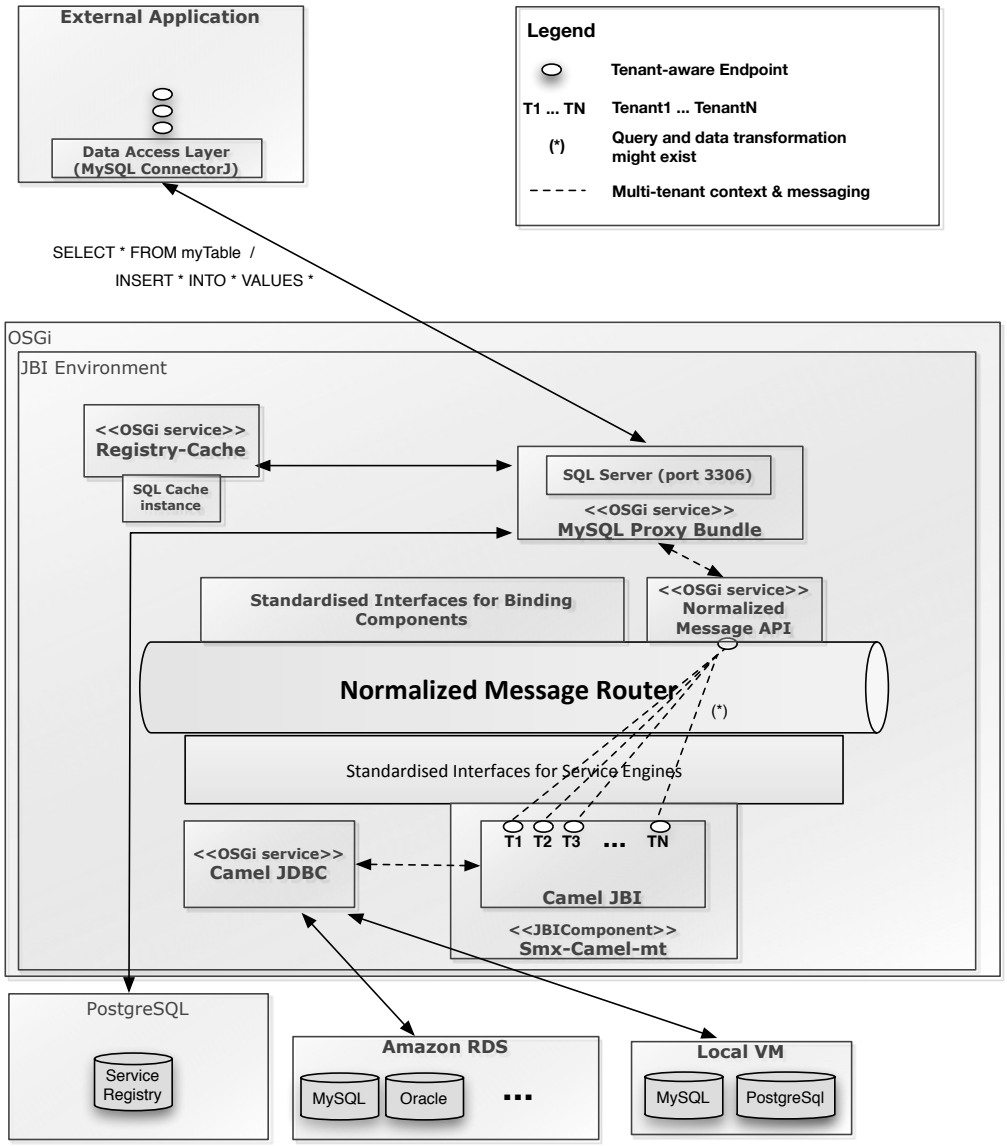

Figure 5.4.: Architectural overview of the design approach one to support the MySQL communication protocol and routing to backend SQL databases.

Apache Camel provides a set of components which integrate most of the communication technologies available in the market. Moreover, it provides an archetype for creating custom components we use in this diploma thesis. ServiceMix provides a Camel JBI BC which integrates the JBI container with the camel router. Muhler extends this component in ServiceMix-mt and enriches it with multi-tenancy awareness. However, the supported multi-tenancy is at the level of tenants, and not at the level of tenant's users. We extend this component and provide user and tenant isolation between endpoints, by injecting the tenant and user UUID in the endpoint's URI (see Listing 5.1).

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

Listing 5.1: Extended Tenant-aware endpoint URI in extended Backus-Naur Form (EBNF) [Muh12].

With multi-tenancy at the tenant and user level, each user can deploy one JBI tenant-aware endpoint in the ServiceMix-Camel-mt SE. The routes deployed from each tenant-aware endpoint are performed under a different context, and an instance of the targeted component in the route is created. Therefore, with this approach we provide multi-tenancy at the messaging, endpoint, and routing and component context levels.

Due to the lack of JDBC support in ServiceMix for creating provider endpoints, we develop a custom camel component, and enrich it with JDBC support for three database systems: MySQL, Oracle, and PostgreSQL. This component is extensible to more database systems when including its native driver, and is build as an OSGi bundle and its packages are exported as an OSGi service. Messages received from the NMR are demarshaled to the backend database system communication protocol, and the response marshaled, correlated, and sent back to the MySQL Proxy Bundle. The demarshalers in this bundle provide the necessary support for transforming the NMF response to a MySQL message.

As described in the previous section, ServiceMix-mt provides JBI and OSGi support and integration, but with some constraints. JBI components cannot import packages from OSGi bundles exporting its packages. The Servicemix-Camel-mt SE provides integration with the camel router for a set of camel components. The camel manual specifies the need for adding statically the custom component packages in the JBI SU which contains the route definition. Therefore, this leads us to scalability problems in each of the SA deployed by the tenants. The SA size increases with the new supported database systems, and forces to redeploy all the SUs containing the custom camel component when it is modified. This leads to management, storage capacity, and network capacity inconveniences. Hence, we provide a second, and final approach which is very similar to this one, but utilizing the ServiceMix-camel component deployed as OSGi bundle.

5.3.3. Approach 2

In this second architectural design approach we address the scalability problems caused by the JBI package dependencies in the SUs described in the previous section. This approach is similar to the first one presented, and its main difference relies on the routing from the tenant-aware JBI endpoints to the custom camel component *cdasmixjdbc* (see Figures 5.5 and 5.4). The functionalities and operations in the MySQL proxy bundle do not differ with the previous approach.

The message is routed from the MySQL proxy bundle to the tenant-aware JBI endpoint deployed in ServiceMix-camel-mt. The multi-tenant message processor instance in ServiceMix-camel-mt routes the message to the *JBItoCamelJdbc* endpoint.

As discussed before, the JBI BCs deployed in ServiceMix are OSGi friendly. This means that the OSGi contains a valid manifest file where the description of the bundle, import packages, and export packages are specified. SUs deployed on this component are able to reference other OSGi packages exposed as a service in the OSGi service registry. We provide a single endpoint deployed in the ServiceMix-Camel component where the requests are sent to: the *JBItoCamelJdbc* endpoint. When the *JBItoCamelJdbc* endpoint is deployed on the ServiceMix-Camel OSGi bundle, this searchs in the OSGi container for the *CdasmixJDBC* component, and creates an instance of the component. Messages routed to the *JBItoCamelJdbc* endpoint are then forwarded to the *CdasmixJDBC* component, which selects the appropriate JDBC native driver, creates a connection, demarshals the request, and forwards the request to the backend Cloud data store server. The connection is established after creating an instance of a *DataSource*, which is saved in the Java Naming and Directory Interface (JNDI) registry for future connections, in order to avoid the creation of more than one *DataSource* instance per user per backend Cloud data store.

Responses retrieved from the backend Cloud data store are correlated with the initial request and routed back to the MySQL proxy bundle, which demarshals the retrieved data and sends it as a binary TCP stream.

In this approach a new instance of the *cdasmixjdbc* is not created per tenant endpoint, but is shared between the tenants. Therefore, we cannot ensure an independent component context at the provider endpoint. However, messages contain the tenant information, and the *cdasmixjdbc* component interacts with the backend database system establishing separate JDBC connections per request. Full multi-tenancy, at the levels of component creation, and endpoint level is not ensured, but it is ensured at the messaging, and context levels. Although full multi-tenancy is not supported, we avoid the deployment of SUs which contain the *CdasmixJDBC* component in it, and whose size increase may lead to scalability problems in the system. Furthermore, we prevent the deployment of the same component n times, for the n multi-tenant aware endpoints, and prevent future management problems when modifying or upgrading the *CdasmixJDBC* component.

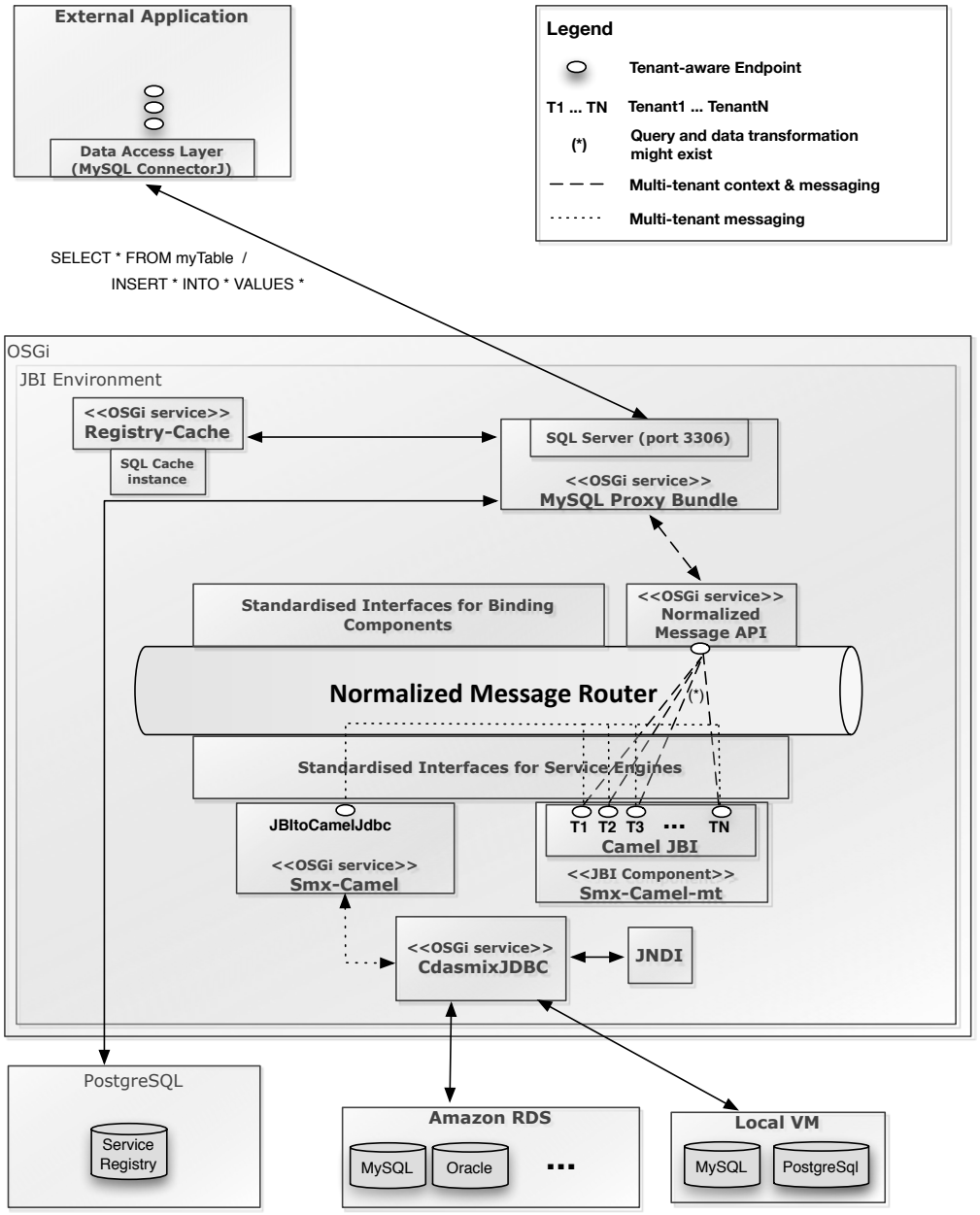

Figure 5.5.: Architectural overview of the design approach two to support the MySQL communication protocol and routing to backend SQL databases.

5.4. NoSQL Support Architectural Overview

Transparent access to different NoSQL family databases and providers, e.g. Amazon DynamoDB, Google Cloud Storage, MongoDB, must be supported in the system. In this section we provide an architectural design to support the communication between the tenant and backend NoSQL databases. We first discuss the integration mechanisms with the components implemented in [Muh12] and [Sá12], and then provide the required adaptations.

5.4.1. Integration

ServiceMix-mt is shipped with an HTTP BC which supports the SOAP over HTTP communication protocol. As discussed in Chapter 3, most of the Cloud data store providers offer a REST interface for accessing, and modifying data. The payload format supported by most of the vendors, and specified in the CDMI specification is JSON [Sto12]. Therefore, we find suitable the reuse of this component, and its enrichment with the required operations which are described in the following subsection.

The packaging required to deploy the component as a SA forces us to statically include the imported packages in the BC SA. Unlike in the SQL architectural approach one, the deployed HTTP endpoint configuration in this case references an endpoint which is addressed in the BC the configuration refers to. Therefore, we can include the needed packages in the BC and not in the SAs containing the tenant-aware endpoint configurations.

5.4.2. Architectural Overview

The multi-tenant aware HTTP BC provides tenant isolation at the tenant level. As in the Servicemix-camel-mt, we extend the component and provide isolation at the tenant and user level.

The architectural design presented in Figure 5.6 is similar to the one presented in [Sá12]. The main reason of this similitude is that we must extend the functionalities of the BC rather than making a substantial architectural modification.

In this approach we consider that the tenant in using the driver provided by the Cloud data store provider to access the consumer endpoint in ServiceMix-mt. The different Java SDKs which are offered by the providers support a set operations and data conversions, build the HTTP request, and forward it to the data store endpoint. To support the different storage structures naming, authentication mechanisms, and data types, we extend the HTTP BC at the marshaling level, routing level, and enhance it with I/O operations with the service registry. Furthermore, to avoid adversely performance effects, it must support cashing mechanism. HTTP provider endpoints are modified in order to able data retrieval from different backend data stores, as long as the source and target database's family and version are the same.

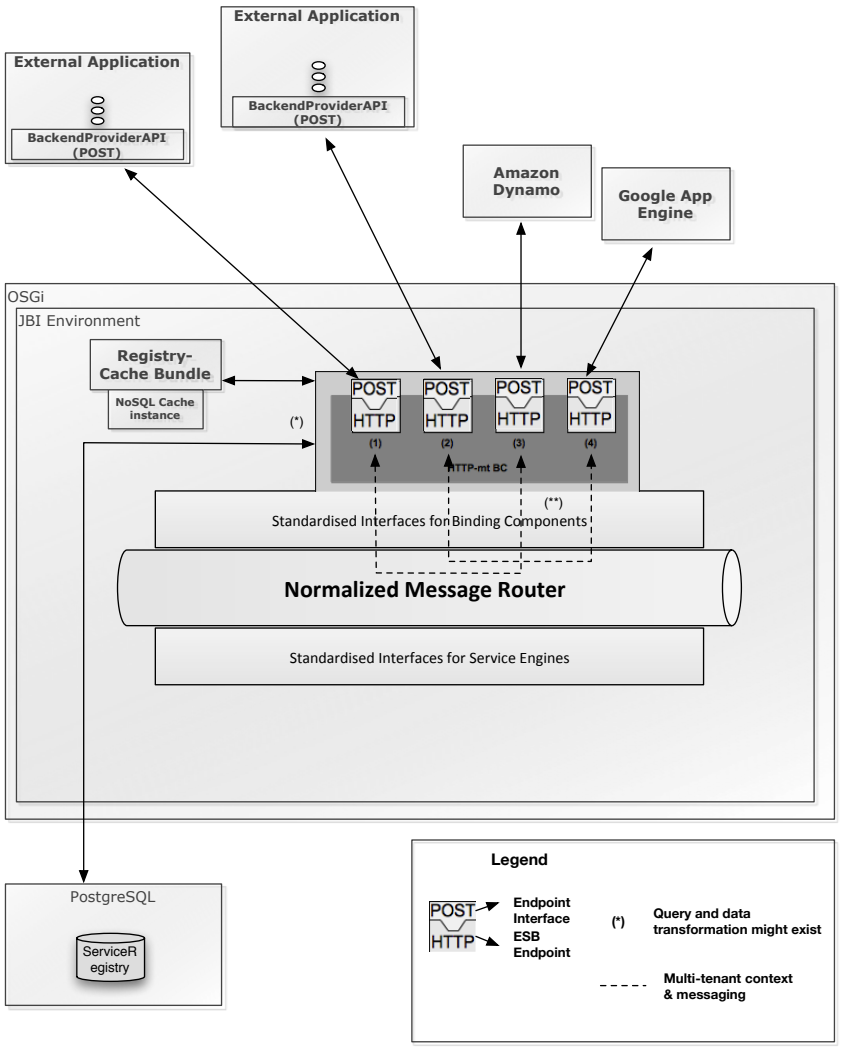

Figure 5.6.: Architectural overview of the design approach to support JSON over HTTP communication protocol and routing to NoSQL data stores.

When query or data transformation is needed, the message is forwarded from the HTTP consumer to the transformer component's endpoint, which must then forward the message to the HTTP provider endpoint.

6. Implementation

In this chapter we describe the challenges and problems during the implementation phase to fulfill the requirements specified in Chapter 4 and the design presented in Chapter 5 of the system. Furthermore, we discuss the incompatibilities found with components we must extend. We divide, as in the previous chapters, the implementation phase into the SQL and NoSQL databases support, and provide a separate section for the extensions made to JBIMulti2 and the Cache.

6.1. SQL Support Implementation

The implementation of the different components described in the SQL architecture approaches in Chapter 5 are explained in this section. We describe the implementation of the three main components which provide support for MySQL incoming requests, and a multi-database protocol for outgoing requests. NMF routing in ServiceMix-mt is supported by the NMR and a set of endpoints which are deployed on the ServiceMix-camel, and ServiceMix-camel-mt.

6.1.1. CDASMix MySQL Proxy

The CDASMix MySQL Proxy provides support for MySQL connections. It implements the functionalities of a MySQL proxy and the needed routing operations and transformation to integrate it in ServiceMix-mt. As discussed in Chapter 3, the original MySQL proxy is implemented in the C programming language. Therefore, we cannot integrate it with ServiceMix-mt, as this runs on a Java platform. The Myosotis Tungsten Connector is part of a data replication engine developed by Continuent, Inc [Con]. One of the components which build the application is a MySQL proxy written in the Java programming language. This component suites the proxying requirements in this diploma thesis, and it runs on the Java platform. However, it is not OSGi compliant, it is not integrated with the JBI environment, and the backend databases' meta-data must be explicitly defined in its configuration file. Furthermore, it provides a direct connection from the component to the database. We utilize it as a base for implementing the CDASMix MySQL Proxy, and aggregate the required functionalities of this diploma thesis. We present the class diagram of the component in Section A.1, but omit the attributes and operations listings due to the high amount of operations, and attributes we manage in the component.

ServiceMix-mt is built as an OSGi container with JBI integration functionalities. Furthermore, ServiceMix 4.x and later version consider the JBI framework deprecated in their systems. Therefore, we build the CDASMix MySQL Proxy component as an OSGi bundle. During the bundle activation process, the OSGi container creates one instance of the class which

implements the *BundleActivator* interface, and executes its operations. The class *OSGIHandler* of the MySQL Proxy OSGi bundle implements those operations, and allows us to instantiate the classes which are used in the component, its properties, and the references to third party OSGi bundles. Furthermore, it loads the server properties file from the bundle package. The instantiation of Java classes packed in third party bundles can be done by looking up the associated bundle service in the OSGi registry. The lookup can be done either by accessing with Java code the *BundleContext* operations (see Listing 6.1), or using Spring functionalities. Spring enables the properties setting and the external bundle reference in XML format (see Listing 6.1), in the *beans.xml* file. We utilize this feature to reference to the following OSGi bundles: the NMR API shipped with ServiceMix, and the JBI ServiceMix Registry developed by Uralov [Ura12].

```
-- Spring --

<beans xmlns="http://www.springframework.org/schema/beans"
...

  <bean id="nmr" class="iaas.unistuttgart.de.mysqlproxy.osgi.OSGIHandler">
          <property name="nmr">
              <osgi:reference interface="org.apache.servicemix.nmr.api.NMR" />
          </property>
  </bean>
</beans>

-- Java --

...

ServiceReference sr = context.getServiceReference(SERVICE_REGISTRY_BUNDLE_NAME);

this.serviceRegistry = (ServiceRegistry) context.getService(sr);

...
```

Listing 6.1: Reference to external OSGi bundle JBI Service Registry Cache and NMR API services utilizing Java libraries and Spring XML beans.

The *mysqlproxy.server* package contains the main classes of this component. The *Configuration* class loads the configuration file, reads it, and sets the properties of the server which accepts the incoming MySQL requests. The *ServerThread* starts the server and listens in the port defined in the server configuration file. This class interacts with the Java library *java.util.concurrent.ExecutorService*, in order to provide a pool of threads for processing the incoming requests. The maximum number of concurrent connections the server can accept is defined in the configuration file, and its default value is 100. Due to possible delays in the backend Cloud data stores, the threads termination time can be set in the configuration file, and its default value is 10 seconds. When restarting the OSGi bundle, or shutting down ServiceMix-mt, the component shut down process is delayed due to the termination of the threads.

For each connection accepted in the server thread, an instance of a protocol handler is created. We highly recommend to utilize the MySQL client native driver MySQL Connector/J *mysql-connector-java-5.1.22-bin.jar*, as we follow the MySQL protocol it implements in this component. The *ConnectionHandler* is implemented as a Java interface, in order to allow future extensibility for different database server protocols. In this diploma thesis we provide support for the MySQL communication protocol. However, we do not recommend to implement a multiple protocol support in the component. The server thread listens to one specific port, and this fact may lead with conflicts between the protocols or the classes which implement the protocol support. Furthermore, a change or upgrade of a specific protocol would involve the redeployment of the component, and a service outage of all the protocols supported. The system resources consumption, e.g. Java heap memory or CPU, is limited for each bundle in the OSGi container, and may lead to scalability problems with the maximum number of concurrent threads in the pool. We recommend to implement the different database server communication protocols in independent OSGi components, but profiting from the resources sharing support for common resources, e.g. cache, in the OSGi container. Each *ConnectionHandler* instance creates a *ProtocolHandler*. The *ProtocolHandler* class groups the support for the MySQL communication protocol phases: connection, authentication, command, and termination (see Section 2.9).

The internal server states are defined as waiting, sent server greeting, and command processing. During the waiting phase, the server waits for the client's MySQL packet to arrive. In the send server greeting phase, the MySQL server greeting is sent to the client, and the server waits for the authentication challenge response. When the authentication is successful, the server is in the command phase. The command phase groups the interpretation of the request, verification of the tenant and user information, message transfer to the *CDASMixjdbc* component, and the transmission of the MySQL response via the TCP socket created by the client.

The logic of the query processing is located in the method *processQuery*. This method receives the SQL query contained in the MySQL packet, and processes it. The processing of the query is divided into the following activities: classification, meta-data retrieval, meta-data construction, and response transfer. The classification of queries is fundamental in a system which provides cashing functionalities. We differentiate the received queries as server configuration queries, data retrieval queries, and database modification queries. In the first group we classify the queries which are related to the specific server configuration that a MySQL client requires. This queries must be directly executed in the backend SQL Cloud data store. Therefore, they are forwarded directly to the *CDASMixjdbc* component. The data retrieval queries involve those that read the data stored in a database. The response to this queries is cached in order to read them directly from the cashing component in a close future. The database modification queries perform modification on the backend Cloud data stores, such as table creation, row update, or row insertion. These cannot be cached, but the entries in the cache which contain data related to the modified tables are invalidated. The meta-data of the user's request is retrieved from the *mysqlproxy.registry* package, which includes information about the target database, access credentials, etc. Multi-querying processing is supported in the prototype, allowing users to include in one execution statement more than one query. For

each of the queries a set of meta-data for the request is created, as one query may contain different meta data from another one (e.g. tables which are located in different backend Cloud data stores).

The operations on the MySQL packages are included in the *commons.mysql* package. This contains the *MySQLPacket* class, which provides a set of I/O operations for reading and writing on TCP streams, and specifically adapted to the MySQL packet format, e.g. read byte, read integer, read long, read string, write byte, write string, write long, etc. The *MySQLPacket* class provides a Java based representation of the MySQL packet. The processing of binary content in the SQL queries requires a special transformation in the scope of the system. For integration purposes with querying transformation component, we must send the query as a string which can be parsed prior to the transformation. Therefore, we encode the binary data contained in the MySQL request into base64, and inject it in the string representation of the query. Binary data in a received MySQL request is identified by the prefix *binary*. In the backend Cloud data store the binary data is stored in base64 format, but this transformation must be transparent to the user. Hence, the class provides decoding support when the component receives from the backend data store a previously stored base64 format, and sends it to the client as binary data. The *MySQLConstants* class included in this packet contains the protocol constants.

The integration with the JBI framework of ServiceMix-mt is provided by the *mysql.jbi* package, where marshaling and demarshaling of the requests and responses to and from the NMF occurs. The detailed content of the marshalled incoming requests and responses is described in Appendix B.1. As the MySQL requests and responses differ, in Chapter 5 we design two different contents. The data retrieval operation responses to the MySQL client must be sent as *java.sql.ResultSet* objects, which are contained in the *java.sql.Statement*. This fact forces us to implement such objects, and its methods, as the response from the *CDASMixjdbc* component stores the retrieved data in a *java.util.ArrayList*. The demarshaler in this package creates the *java.sql.ResultSet* and the *java.sql.Statement*, and populates it with the retrieved data in the *java.util.ArrayList*. Furthermore, the marshaler creates the *java.sql.ResultSetMetadata* with the meta-data retrieved from the backend database server. The meta-data contained in this object does not refer to connection or tenant-aware meta-data, but the meta-data of the table, rows, and columns of the backend database where the query is executed. The *NMConstants* class contains the standardized naming we define of the properties and attachments in the NMF. The *MessageExchangeHandler* class implements the operations which directly invoke the routing operations in the NMR API. The target JBI endpoint URI is dynamically created from the tenant and user UUID retrieved from the cache, or the service registry, and inserted in the *MessageExchange* target endpoint URI. After the creation of the *MessageExchange*, the synchronous *sendsync* operation of the NMR API is invoked, the message is routed, and the thread waits until the reception of the message from the JBI endpoint.

As described before, the CDASMix MySQL proxy component interacts with two main components to retrieve tenant's information: Registry Cache, and the Service Registry. The former is provided by the OSGi bundle *jbi.servicemix.registry*, while the latter is connected through a set of methods provided in the *mysqlproxy.registry* package in the CDASMix MySQL proxy component. The cashing interface provided by the *jbi.servicemix.registry* implement the put

and get operations, where the cashing key, and the data to store are provided. The *mysql-proxy.registry* accesses the cache through this interface, but dynamically creates the cashing key for the object it stores, following the format described in Listing 6.2. Three different types of information identified by a unique tag are stored in the cache: tenant authentication, tenant backend data source properties, and query results.

```
1 CACHE_TENANT_INFO_ID_TAG    + "#" + tenantId + "#" + userId + "#" + srcdb + "#" + srcTable;
2 CACHE_TENANT_AUTHENTICATION_TAG + "#" + tenantId + "#" + userId + "#" + srcdb;
3 CACHE_QUERY_RESULT_TAG      + "#" + tenantId + "#" + userId + "#" + targetdb + "#" + query;
```

Listing 6.2: Dynamic creation of cache keys to store the different types of data and metadata.

When the information is not located in the cache, the service registry is queried, and its result cashed. The CDASMix MySQL proxy establishes a database connection with the PostgreSQL database server where the service registry is hosted when the component is deployed. A dedicated connection for this component reduces the latency of creating a database connection per received MySQL request.

6.1.2. Multi-tenant ServiceMix Camel Service Engine

The ServiceMix-camel-mt JBI SE provides integration support between the Camel router, and the JBI environment. This SE includes in its package the *camel-core* libraries, in order to interact with the camel endpoints which are by default included in it, e.g. camel-email, camel-jms, camel-http, camel-ftp, etc. Muhler extends in his work this SE in order to include multi-tenancy awareness [Muh12]. In this subsection we describe the modifications he makes, and the extensions we develop to make it compatible with the requirements of our system.

The Camel router provides support for describing its routes in two styles: POJO, and Spring. The former allow programmers to describe the camel routes in a Java class which extends camel's *RouteBuilder* class, while the latter allows the route description in XML format. Files which describe the routes are packed in SUs, and these in a SA. The *CamelSpringDeployer* class in ServiceMix-camel-mt implement the necessary methods for extracting the routes for deployment. Muhler extends this class in *CamelSpringDeployerMT*, and includes tenant-aware operations for injecting tenant-aware information in the service and endpoint URIs. However, the tenant-aware information is not enough in our system, due to requirements specified in Chapter 4. Therefore, we extend his work by adding the tenant's user information in the multi-tenant information injected in the URIs.

A Camel JBI endpoint is identified by the prefix *jbi:*, and the endpoints are classified into two types: consumer, and provider. Consumer endpoints receive message exchanges from endpoints exposed in the NMR, while providers forward the message exchanges to JBI endpoints exposed in the NMR, or to external services. The *CamelConsumerEndpoint* and *CamelProviderEndpoint* classes include the endpoint configuration, and the necessary methods for creating and accepting message exchanges. Hence, we modified them and included

multi-tenancy awareness operations which listen to and send messages to multi-tenant aware JBI endpoints.

In the design approach two for SQL support described in Chapter 5 we define two Camel JBI endpoints for routing requests to the *CDASMixjdbc* component: the tentants' JBI endpoints, and the *JBIToCamelJdbc* endpoint. The former is deployed on ServiceMix-camel-mt, and the latter on ServiceMix-camel. Both routes definitions are packed and deployed in different SUs, and SAs, due to their deployment on different components. In the URI endpoint definition we set the Camel JBI option *check serialization= nocheck*, which deactivates the serialization checking of objects contained in the property section of the NMF. The objects the system sends in the NMF properties section implement the interface serializable. Therefore, there is no need of overloading the system with checking operations. The *JBIToCamelJdbc* endpoint forwards the messages exchanges to the *CDASMixjdbc* camel component (see Listing 6.3).

6.1.3. Camel CDASMix JDBC Component

The Camel CDASMix JDBC component provides support for creating and exchanging requests with external database systems via JDBC. We implement the communication support for the following three database systems: MySQL, PostgreSQL, and Oracle. The connections to the databases are established by utilizing its client native drivers via JDBC: *mysql-connector-java-5.1.22*, *postgresql-9.1-901.jdbc3*, and *ojdbc14-10.2.0*. These are included in the OSGi bundle of the *CDASMixjdbc* component, and *DataSource* objects are created for establishing the connection with the backend database systems. The *CdasmixDataSource* interface is implemented by the three protocol specific datasources: *CdasmixMySQLDS*, *CdasmixPostgresDS*, and *CdasmixOracleDS* (see Figure A.2). Therefore, the database specific *DataSource* creation is handled in the *DataSourceHandler* class, which evaluates the target datasource driver property contained in the NMF. This class interacts with the JNDI registry of the system for registering, and retrieving the *DataSource* instances.

```
1 ...
2 <camelContext id="camel"
3         xmlns="http://camel.apache.org/schema/spring">
4         <route>
5             <from uri="jbi:endpoint:http://jbimulti2.iaas.uni-stuttgart.de/
                  CamelJBIJdbcService/CamelJBIJdbcEndpoint?serialization=nocheck" />
6             <to uri="jdbccdasmix:myDataSource" />
7         </route>
8 </camelContext>
9 ...
```

Listing 6.3: Route definition for JBIToCamelJdbc JBI endpoint to JDBCCDASMix Camel Component.

This component is developed as a custom camel component, and in the Camel registry its endpoints are identified by the prefix *cdasmixjdbc* (see Listing 6.3). The OSGi container allows the redeployment of the component without the need of redeploying the external components

which use its libraries. During the activation process of this OSGi bundle, an instance of the component is created and registered in the Camel registry. Camel endpoints containing in its URI the *jdbccdasmix* component create an instance of the *JdbcCdasmixComponent* class. The original Camel-jdbc component requires the user to specify the data source name in the endpoint URI, and its properties in the bean configuration. However, we do not require the explicit description of the data source in the URI, as the component retrieves it from the NMF, and creates it dynamically. The *JdbcCdasmixEndpoint* class creates an instance of the producer endpoint, the *JdbcCdasmixProducer*. The *JdbcCdasmixProducer* includes the main logic of the component. When a NMF is received in the *jdbccdasmix* endpoint, its properties are extracted and analyzed, the connection to the backend database is established, the message is demarshaled, the request on the database system is performed, the response from the database system is marshaled into the NMF, and forwarded back to the endpoint which created the exchange. The response's meta-data is stored in the properties section of the NMF, while the data is stored in the attachment section of the NMF. The storage of attachments in the NMF through the Camel API requires the storage of the data in a class which implement the *javax.activation.DataSource* class. Threfore, we implement two different classes which implement it, and store different data: the data retrieved from the backend database system (when success), or the error messages (when an error occurs). Operations on different backend database systems in one message exchange are considered atomic. If one backend database system answers with an error, the message exchange status is set to error, and forwarded to the endpoint which created the message exchange.

The support of JDBC in a custom component and a unique endpoint URI prefix allows the developers to use the existing Camel-jdbc component for their non multi-tenant aware connections.

6.2. NoSQL Support Implementation

The prototype requirements described in Chapter 4 specify the need of support to NoSQL Cloud data stores. In Chapter 5 we present a design for accessing backend NoSQL Cloud data stores via a standardized protocol which is supported in most of the data stores interfaces: JSON over HTTP. Gomez [Sá12] and Muhler [Muh12] extend the ServiceMix-http JBI BC in order to provide multi-tenancy awareness at the communication level. We reuse and extend the BC to support the communication protocol required in this diploma thesis.

6.2.1. Multi-tenant ServiceMix HTTP Binding Component

As discussed in previous chapters, the ServiceMix-mt JBI BC implements multi-tenancy awareness for the HTTP communication protocol. However, the multi-tenancy awareness supported ensures endpoint isolation at the tenant level, but not at the user level. Ensuring tenant isolation in a system which connects to multiple tenant's databases, and is accessed by multiple tenant's users, is not enough. Therefore, we modify the operations implemented in Muhler's approach [Muh12], in order to ensure tenant and user isolation.

The HTTP endpoint configuration is described in the *xbean.xml* file of the SU, and its XML format complies the Spring specification. The *XBeanDeployerMT* class extends the original *BaseXBeanDeployer* class in ServiceMix, which performs the loading of the endpoint properties, and the creation and configuration of the deployed endpoint. The *XBeanDeployerMT* created in Muhler's work provides the operations which inject tenant information in the endpoint's URI during the deployment process [Muh12]. We modify such operations in order to inject tenant and user information in the endpoint's URI during the endpoint creation. The HTTP consumer and provider endpoints are described in the *HttpConsumerEndpoint* and *HttpProviderEndpoint* respectively. The former implements the necessary operations for accepting incoming HTTP requests to ServiceMix-mt, while the latter implements the operations to communicate with an external HTTP server. Both endpoints are created on a Jetty server, which provides an interface for sending HTTP requests, and listening on a specific port for incoming HTTP requests. Therefore, the endpoints process incoming, and outgoing requests. Request messages are marshaled and demarshaled into or from the NMF, a message exchange is created, and the response message is correlated with the initial request.

As described in Chapter 3, the different NoSQL Cloud data store provider products available nowadays belong to a specific NoSQL family, but the naming the Cloud providers use to identify the storage structures is different. We address this issue in Chapter 5, but we find further issues in its communication protocol. Although most of the NoSQL Cloud data store providers, e.g. Amazon [Amaa], Google Cloud Storage [Gooc], provide a REST interface for accessing their stores, the HTTP content and header values are not standardized between the providers. For this reason, they provide their users their Java SDKs for accessing the Cloud services. Operations on the driver's API create the JSON over HTTP requests. Furthermore, MongoDB [10G] does not provide a REST interface, but there are several external projects which create an Jetty HTTP server, and transform the HTTP requests into parameters of the Mongo Java SDK. For these reason we are not able to provide a standardized marshaling interface in the HTTP JBI BC for all the Cloud data store providers, but we concentrate on the three which we mention before.

```
1  POST / HTTP/1.1
2  host: dynamodb.us-east-1.amazonaws.com
3  x-amz-date: Mon, 16 Jan 2012 17:49:52 GMT
4  x-amz-target: DynamoDB_20111205.CreateTable
5  Authorization: AWS4-HMAC-SHA256 Credential=AccessKeyID/20120116/us-east-1/dynamodb/
       aws4_request,SignedHeaders=host;x-amz-date;x-amz-target,Signature=145
       b1567ab3c50d929412f28f52c45dbf1e63ec5c66023d232a539a4afd11fd9
6  content-type: application/x-amz-json-1.0
7  content-length: 23
8  connection: Keep-Alive
9
10
11 {"RequestItems":
12     {"highscores":
13         {"Keys":
14             [{"HashKeyElement":{"S":"Dave"}},
15             {"HashKeyElement":{"S":"John"}},
16             {"HashKeyElement":{"S":"Jane"}}],
```

```
        "AttributesToGet":
            ["score"]
        }
    }
}
```

Listing 6.4: Amazon Dynamo DB JSON over HTTP sample request [Amaa].

Marshaling operations in the HTTP JBI BC are provided in the classes *DefaultHttpConsumerMarshaler*, and *DefaultHttpProviderMarshaler*. Hence, we start the extension of the marshalers at this point. On the other hand, the marshalers do not support marshaling of JSON content, as the NMF stores XML format. Therefore, we include JSON marshaling operations which are provided by the Jackson v1.8.9 [Cod]. We select this version, as is the one provided in Amazon's AWS SDK [Amag]. We provide a marshaler handler interface, which is implemented for the different Cloud data store providers. The handler is chosen based on the HTTP headers or target HTTP URL content (see Listings 6.4 and 6.5). As in the MySQL approach, marshaling operations include filling the NMF properties with the tenant's source and backend data source configuration data retrieved from the service registry, and storage of the HTTP request content as an attachment. Properties and attachments identifiers are listed in the *NMConstants* class.

```
- Request -
POST /upload/storage/v1beta1/b/myBucket/o?uploadType=media&name=myObject HTTP/1.1
Host: www.googleapis.com
Content-Type: image/jpeg
Content-Length: number_of_bytes_in_JPEG_file
Authorization: your_auth_token

{JPEG data}


- Response -
HTTP/1.1 200
Content-Type: application/json

{
  "name": "myObject"
}
```

Listing 6.5: Google Coud Storage JSON over HTTP sample request [Gooc].

The same method described above applies in the HTTP provider endpoints. Demarshaling operations in the multi-tenant aware provider endpoint create the HTTP requests for each Cloud provider (see Listings 6.4 and 6.5) in a separate handler, based on the target data store properties which are stored in the NMF.

Authentication mechanisms in the HTTP consumer endpoint processor rely on the tenant and user UUIDs, and follow Gomez's approach [Sá12]. However, we do not retrieve the UUIDs from the SOAP header, but from the HTTP header where the authentication data is

stored. Both authentication and data retrieval are susceptible of being cashed in the system. The cashing mechanisms we use for this BC is the same as in the CDASMix MySQL Proxy component. The difference relies on a separate cache instance for NoSQL requests, and the need of including the JBI ServiceMix Registry library in the SA of this BC, due to the problems found when loading external OSGi bundles from components packed as SAs.

6.3. Extensions

Apart from the components which we implement in this diploma thesis, we extend existing components developed in the works from Muhler [Muh12], and Uralov [Ura12]. We extend the JBIMulti2 application developed by Muhler, and adapt the JBI ServiceMix Registry performed by Uralov.

6.3.1. JBIMulti2

The multi-tenant aware administration and management application JBIMulti2 offers a list of operations through its Web service interface in order to allow tenants to deploy multi-tenant aware endpoints in ServiceMix-mt. However, it does not provide support for persisting the tenant's Cloud data stores communication meta-data. In Chapter 5 we describe the necessary modifications on the service registry, and the need of extending the application's Web service interface to avoid a connection from the *Cloud Data Migration Application* to the database system which stores the tenant configuration data.

The Service Registry database schema is created using the Java Persistence API, and annotating the Java classes with meta-data. We extend the Java class *ServiceAssembly*, which stores the SAs deployed by the tenant. The *ServiceAssembly* class contains the schema representation, and the data storage and retrieval operations.

The Web service interface developed by Muhler [Muh12], and extended by Uralov [Ura12], contains several management operations, e.g. deploy service assembly, create tenant, create user, deploy JBI BC, etc., which allows the system administrator and tenants to perform management operations on ServiceMix-mt. The *Cloud Data Migration Application* described in Chapter 2 retrieves from the user the backend database system meta-data, such as access credentials, database name, etc. The application may be hosted on a separate server as JBIMulti2. In order to avoid a direct connection from the *Cloud Data Migration Application* to the Service Registry, which contains sensible data, we extend the JBIMulti2 Web service interface, and provide the operations for registering the tenant's backend Cloud data store meta-data.

6.3.2. Cache

The cashing support in ServiceMix-mt is provided in the OSGi component JBI ServiceMix Registry. Uralov provides a set of operations to set up a cache instance, store elements, and

retrieve stored elements [Ura12]. However, the JBI ServiceMix Registry OSGi bundle libraries are not registered as an OSGi service. Therefore, this component is not accessible from third party OSGi bundles in the OSGi container. We modify its original *BundleActivator* class and include the OSGi service registration to allow its usage from external OSGi bundles.

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation
    ="ehcache.xsd" dynamicConfig="true">
<diskStore path="java.io.tmpdir"/>
  <cache name="sql.proxy.cache"
      maxEntriesLocalHeap="10000"
      eternal="false"
      timeToIdleSeconds="120"
      timeToLiveSeconds="120"
      maxEntriesLocalDisk="10000000"
      diskExpiryThreadIntervalSeconds="120"
      memoryStoreEvictionPolicy="LRU"
      copyOnRead="true" logging="true"
      overflowToDisk="false" statistics="true">

        <persistence strategy="localTempSwap"/>
    </cache>
</ehcache>
```

Listing 6.6: Eh cache configuration for SQL support.

The cache instances are created on the Ehcache 2.6.0 component [Ter], which provides support to store serializable objects indexed by a key. A cache instance configuration in the Ehcache component must be specified in an XML file, which is described in Listing 6.6. However, the cashing support is not multi-tenant aware. We implement a dynamic creation of multi-tenant aware cache keys in order to ensure isolation between the cashed tenant's information, and data (see Listing 6.2).

7. Validation and Evaluation

In this chapter we provide the validation, and evaluation of the system. We must ensure that the requirements specified in Chapter 4 are fulfilled in the design and implementation phases. In Section 7.1 we describe the steps which should be followed to initialize the system, and the testing scenarios. After the initialization we execute the test cases in Section 7.2, and monitor the incoming requests to ServiceMix-mt, and the outgoing requests to the backend Cloud data store. Due to the extensions implemented on the ESB, we evaluate in Section 7.3 its behavior, and the impact that our modifications have on the original ServiceMix-mt.

7.1. Deployment and Initialization

The validation and evaluation of the prototype must close to the motivating scenario. Therefore, we must perform the testing of the prototype in a Cloud environment. We are provided with a VM image in the FlexiScale Cloud infrastructure [Fle], which runs the operative system Ubuntu 10.04 64 bits. The following components are deployed in the VM:

- ServiceMix-mt 4.3.0: the multi-tenant aware ServiceMix 4.3.0. In addition to the OSGi bundles, JBI SAs, and JBIMulti2 shared library [Muh], in its deploy folder we store the JBI ServiceMix Registry, CDASMix MySQL Proxy, and the CamelJDBCCdasmix OSGi bundles for deployment.
- JOnAS 5.2.2: the Java application server which hosts the JBIMulti2 application.
- MySQL database 5.1: MySQL database system for performing evaluation and validation of the prototype with a local database instance.
- PostgreSQL 9.1.1: PostgreSQL database system which stores the tenant-aware configuration data in the Service Registry, Configuration Registry, and Tenant Registry.

For more information about the deployment, and initialization of the ServiceMix-mt and JBIMulti2 please refer to the document "Manual for the JBIMulti2 Implementation" [Muh]. On the other hand, we utilize de following off-premise instances:

- Amazon RDS db.m1.micro instance: MySQL 5.5 database system hosted in the Amazon RDS Cloud infrastructure.
- Amazon DynamoDB table: NoSQL key-value database for storing objects in the created tables.
- Google Cloud Storage: NoSQL key-value database for storing buckets and objects.

We must maximize the approximation of the testing and evaluation scenarios with the motivating scenario. Therefore, we perform the testing cases from a local machine in the University of Stuttgart network infrastructure, in order to simulate the access to the data from an on-premise data access layer. Communication with the JBIMulti2 application for tenant configuration, and deployment of SAs is established using soapUI 3.6. Communication with the MySQL endpoint in ServiceMix-mt is established using the MySQL Connector/J 5.1.22, while for the different tenant-aware HTTP endpoints we use the Java libraries provided by the Cloud data store providers.

7.2. Validation

In this section we provide an overview of the messages, and programs used for the validation of the prototype. The transmission of the following message samples requires the system to be started, and the operations until the deployment of the tenant operator's SA to be executed. These messages are provided in a soapUI test suite shipped with the prototype. From the moment when the tenant deploys the SA, the configuration of the frontend and backend data stores can be associated to it. The data store configuration is done through the execution of operations which are accessible through a Web service interface. Therefore, either the *Cloud Data Migration Tool*, or the tenant can configure their database connections through the ESB to the backend Cloud data store where his data is migrated to.

```
1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsdl="
       http://jbimulti2.iaas.uni-stuttgart.de/wsdl">
2     <soapenv:Header>
3        <ctxjmu2:TenantContext xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.de/tenant-
             context">
4           <ctxjmu2:TenantId>${tenantProperties#taxiCompany-UUID}</ctxjmu2:TenantId>
5           <ctxjmu2:UserId>${tenantUserProperties#tenantOperatorA}</ctxjmu2:UserId>
6           <ctxjmu2:OptionalEntry>
7               <ctxjmu2:Key>password</ctxjmu2:Key>
8               <ctxjmu2:Value>tenantOperatorApw</ctxjmu2:Value>
9           </ctxjmu2:OptionalEntry>
10       </ctxjmu2:TenantContext>
11    </soapenv:Header>
12    <soapenv:Body>
13       <wsdl:attachDataSource>
14          <wsdl:serviceAssemblyName>servicemix.cdasmix.jbi.camel-SA</wsdl:serviceAssemblyName
                >
15          <wsdl:sourceDataSourceName>tenant1tpchaws</wsdl:sourceDataSourceName>
16       <wsdl:sourceDataSourceType>mysql-database-table-5.1.3</wsdl:sourceDataSourceType>
17          <wsdl:sourceDataSourceProtocol>mysql</wsdl:sourceDataSourceProtocol>
18          <wsdl:sourceDataSourceURL>localhost:3311</wsdl:sourceDataSourceURL>
19          <wsdl:targetDataSourceName>tenant1tpchaws</wsdl:targetDataSourceName>
20       <wsdl:targetDataSourceType>mysql-database-table-5.1.3</wsdl:targetDataSourceType>
21          <wsdl:targetDataSourceProtocol>mysql</wsdl:targetDataSourceProtocol>
22          <wsdl:targetDataSourceURL>tenant1tpchaws.c0giusfeitsw.us-east-1.rds.amazonaws.
                com:3306/tenant1tpchaws</wsdl:targetDataSourceURL>
```

```
23          <wsdl:targetDataSourceNativeDriverName>com.mysql.jdbc.Driver</
                wsdl:targetDataSourceNativeDriverName>
24          <wsdl:targetDataSourceUser>tenant1user1tpch</wsdl:targetDataSourceUser>
25          <wsdl:targetDataSourcePassword>passwordtenant1user1</wsdl:targetDataSourcePassword>
26       </wsdl:attachDataSource>
27    </soapenv:Body>
28 </soapenv:Envelope>
```

Listing 7.1: Add Source and Target SQL Data Source SOAP over HTTP sample request.

The message described in Listing 7.1 contains the needed information for registering and providing access to the tenant operator A in the MySQL CDASMix Proxy, and connecting with an SQL database system in a backend Cloud data store. The tenant operator must authenticate with JBIMulti2 in order to register the connection meta-data, e.g. protocol, URL, database type, access credentials, etc. In Listing 7.1 the tenant operator A, which is a user of the tenant Taxi Company, registers for the endpoint configuration described in the *servicemix.cdasmix.jbi.camelSA* a connection with a MySQL 5.1.3 database system instance in the Amazon RDS infrastructure. When more than one target database system wants to be attached to an existing source data source registered in the system, this provides the *AttachTargetDataSource* operation, which has as pre-condition the existence of the specified source data source.

```
1  <soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsdl="
       http://jbimulti2.iaas.uni-stuttgart.de/wsdl">
2     <soapenv:Header>
3        <ctxjmu2:TenantContext xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.de/tenant-
             context">
4           <ctxjmu2:TenantId>${tenantProperties#taxiCompany-UUID}</ctxjmu2:TenantId>
5           <ctxjmu2:UserId>${tenantUserProperties#tenantOperatorA}</ctxjmu2:UserId>
6           <ctxjmu2:OptionalEntry>
7              <ctxjmu2:Key>password</ctxjmu2:Key>
8              <ctxjmu2:Value>tenantOperatorApw</ctxjmu2:Value>
9           </ctxjmu2:OptionalEntry>
10       </ctxjmu2:TenantContext>
11    </soapenv:Header>
12    <soapenv:Body>
13       <wsdl:attachDataSourceMainInformationStructure>
14          <wsdl:serviceAssemblyName>servicemix.cdasmix.jbi.camel-SA</wsdl:serviceAssemblyName
                >
15       <wsdl:sourceDataSourceName>tenant1tpchaws</wsdl:sourceDataSourceName>
16       <wsdl:targetDataSourceName>tenant1tpchaws</wsdl:targetDataSourceName>
17       <wsdl:sourceDataSourceType>mysql-database-table-5.1.3</wsdl:sourceDataSourceType>
18       <wsdl:targetDataSourceType>mysql-database-table-5.1.3</wsdl:targetDataSourceType>
19       <wsdl:mainInformationStructureName>customer</wsdl:mainInformationStructureName>
20        </wsdl:attachDataSourceMainInformationStructure>
21    </soapenv:Body>
22 </soapenv:Envelope>
```

Listing 7.2: Add Source and Target SQL Main Information Structure SOAP over HTTP sample request.

When the source and target data source configuration are registered, the tenant operator (or through the *Cloud Data Migration Tool*) can register the storage structure meta-data of his database. For example, for a MySQL database system instance in Amazon RDS, the tenant operator must provide the tables' name (see Listing 7.2), or for a bucket stored in a backend Google Cloud Storage container the tenant must provide the bucket identifier.

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:wsdl="
    http://jbimulti2.iaas.uni-stuttgart.de/wsdl">
   <soapenv:Header>
      <ctxjmu2:TenantContext xmlns:ctxjmu2="http://jbimulti2.iaas.uni-stuttgart.de/tenant-
          context">
         <ctxjmu2:TenantId>${tenantProperties#taxiCompany-UUID}</ctxjmu2:TenantId>
         <ctxjmu2:UserId>${tenantUserProperties#tenantOperatorA}</ctxjmu2:UserId>
         <ctxjmu2:OptionalEntry>
            <ctxjmu2:Key>password</ctxjmu2:Key>
            <ctxjmu2:Value>tenantOperatorApw</ctxjmu2:Value>
         </ctxjmu2:OptionalEntry>
      </ctxjmu2:TenantContext>
   </soapenv:Header>
   <soapenv:Body>
      <wsdl:attachDataSourceSecInformationStructure>
         <wsdl:serviceAssemblyName>servicemix.cdasmix.jbi.http-SA</wsdl:serviceAssemblyName>
         <wsdl:dataSourceName>dataSourceTest1</wsdl:dataSourceName>
         <wsdl:dataSourceType>nosql-bucket-object-5.1.3</wsdl:dataSourceType>
         <wsdl:sourceDataSourceName>dynamodb</wsdl:sourceDataSourceName>
     <wsdl:targetDataSourceName>dynamodb#datasourcej</wsdl:targetDataSourceName>
     <wsdl:sourceDataSourceType>nosqlKeyValue-table-item-1.0</wsdl:sourceDataSourceType>
     <wsdl:targetDataSourceType>nosqlKeyValue-table-item-1.0</wsdl:targetDataSourceType>
     <wsdl:mainInformationStructureName>customers</wsdl:mainInformationStructureName>
         <wsdl:secondaryInformationStructureName>customer1</
             wsdl:secondaryInformationStructureName>
      </wsdl:attachDataSourceSecInformationStructure>
   </soapenv:Body>
</soapenv:Envelope>
```

Listing 7.3: Add Source and Target NoSQL Secondary Information Structure SOAP over HTTP sample request.

Registration of secondary information structure meta-data applies only for NoSQL databases. The message described in Listing 7.3 contains the registration meta-data for an item stored in a table in the Amazon DynamoDB infrastructure.

After the migrated data's meta-data is registered in the system, the tenant operator can access transparently the system using the standardized database access protocols, e.g. MySQL for the MySQL database system, and HTTP for NoSQL databases.

```
...

connect = DriverManager
          .getConnection(
```

```
5               "jdbc:mysql://" + "IPADDRESS" + ":3311/" + DB +"?allowMultiQueries=true;
                    cacheServerConfiguration=true&" +
6           "jdbcCompliantTruncation=false&zeroDateTimeBehavior=convertToNull&tinyInt1isBit=
                false&allowMultiQueries=true&yearIsDateType=false",
7           TENANT_ID + "#" + USER_ID,"tenant1password");
8
9   ...
10
11
12  resultSet = executeQuery(connect, "select␣ID␣from␣customer;␣select␣*␣from␣mainInfoTest4;" +
13      "select␣*␣from␣mainInfoTest4␣as␣t1,␣mainInfoTest1␣as␣t2␣where␣t1.ID␣=␣t2.ID;");
14  while (resultSet.next()) {
15    System.out.println("IDs␣" + resultSet.getInt("C_CUSTKEY"));
16  }
17  if (statement.getMoreResults()) {
18    ResultSet nextRs = statement.getResultSet();
19    while (nextRs.next()) {
20      System.out.println("ID_" + nextRs.getInt("ID"));
21      System.out.println("Name_" + nextRs.getString("Name"));
22      System.out.println("SecondName_" + nextRs.getString("SecondName"));
23    }
24  }
25  if (statement.getMoreResults()) {
26    ResultSet nextRs = statement.getResultSet();
27    while (nextRs.next()) {
28      System.out.println("ColumnCount:_" + nextRs.getMetaData().getColumnCount());
29      System.out.println("ID1_" + nextRs.getInt(1));
30      System.out.println("name_" + nextRs.getString(2));
31      System.out.println("SecondName_" + nextRs.getString(3));
32      System.out.println("ID2_" + nextRs.getInt(4));
33      System.out.println("name_" + nextRs.getString(5));
34    }
35  }
36
37  ...
```

Listing 7.4: Retrieve data via JDBC and MySQL protocol migrated to a backend Cloud data store, or local database system.

In Listing 7.4 we provide part of a Java program which connects via the MySQL Connector/J native driver with the MySQL CDASMix Proxy component on the port 3311. The options attached to the connection URL must be set as in the provided code in order to increase the performance of the communication, e.g. multi-querying for multiple queries in one statement, cache server configuration in order not to request the server configuration data per request, etc. In the MySQL request in Listing 7.4, the tenant operator authenticates with the tenant and user UUID, and its password. Password is read in the system, but it is not compared in the authentication process, as the CDASMix MySQL proxy does not implement the hashing mechanisms of a MySQL server. After successful authentication, it executes a query which contains multiple queries in the same request. Each query is processed independently in the system when retrieving information from the different backend database systems, e.g. the *customer* table is stored in Amazon RDS, while mainInfoTest4 and mainInfoTest1 are stored

in the local MySQL database system. The response is received as a single statement which contains n result sets, one for each query in the multi-query.

7.3. Evaluation

Extensions implemented in ServiceMix-mt may affect on its performance, and system's resources consumption. Therefore, in this section we focus on evaluating, in the first place, the operative system's resources the ESB consumes. In the second place, we evaluate the difference between connecting directly to the backend Cloud or local data store (without transparency to the user), and the utilization of a transparent connection through the Cloud-Enabled Data Access Bus. For evaluation purposes we utilize the TPC-H benchmark for generating the data, and Apache JMeter for performing the load tests.

7.3.1. TPC-H Benchmark

The TPC-H benchmark is a set of libraries written in the C language which provide support for generating large volumes of data for populating the database system which wants to be evaluated [Tra01]. Furthermore, it generates queries with a high degree of complexity to be executed on the databases where the generated data is stored. In this diploma thesis we utilize the TPC-H benchmark to generate the data which is stored in the local MySQL database system, and in the backend MySQL database instances hosted in Amazon RDS. The data generated by the benchmark varies in size, and is stored in a database which follow the schema described in Figure 7.1.

The TPC-H data and queries generator operations are mainly managed in two executables, which are generated by building the library with the *make* command: *dbgen*, and *qgen*. The former has as input option the size, which we set in this diploma thesis to 1GB, while the latter generates a set of queries for the generated data. After the data and queries generation, the user must manually import the data into the database system to evaluate. For more information about the processes of generating data and queries we attach in the prototype a short tutorial which is adjusted to the operative system we use in the FlexiScale's VM: Ubuntu 10.04.

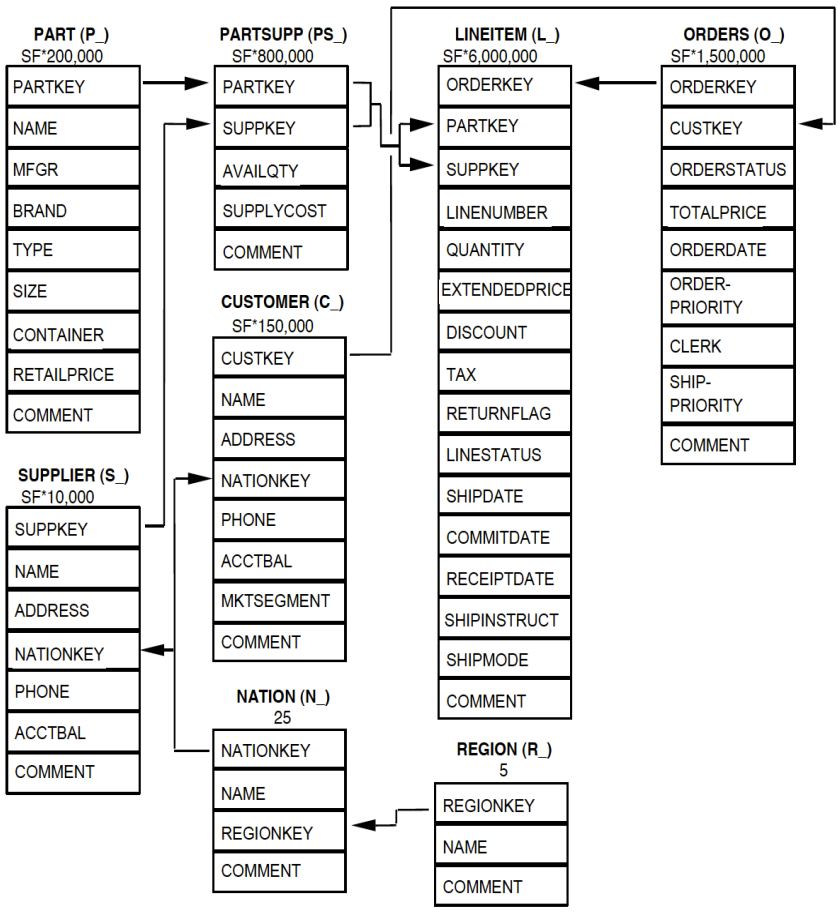

Figure 7.1.: TPC-H database schema generated [Tra01].

7.3.2. Evaluation Overview

The system we set up for the evaluation is built up of three subsystems which are hosted in different infrastructures. The first subsystem resides in a private network provided in the University of Stuttgart, and identified by the hostname *dyn139.iaas.uni-stuttgart.de*. A local machine with Apache JMeter 2.9 installed is connected to the network. In the evaluation we aim to approach as much as possible to the motivating scenario: database layer migrated to the Cloud, and accessing the remote database system from the on-premise application's data access layer. Therefore, we decide to perform the queries in a load generator program from a private network (see Figure 7.2).

We classify in Figure 7.2 as the second subsystem the Amazon RDS database systems which are host in the Amazon Cloud [Amac]. Amazon provides the user with the option to deploy his database instances in different regions. We select the N. Virginia region, create the tenant 2 database on a db.m2.xlarge database instance, and transfer the table schemas and data generated by the TPC-H benchmark. For billing purposes we utilize the *EDUStudentGrantsSpring-Summer2012* [Amaf]

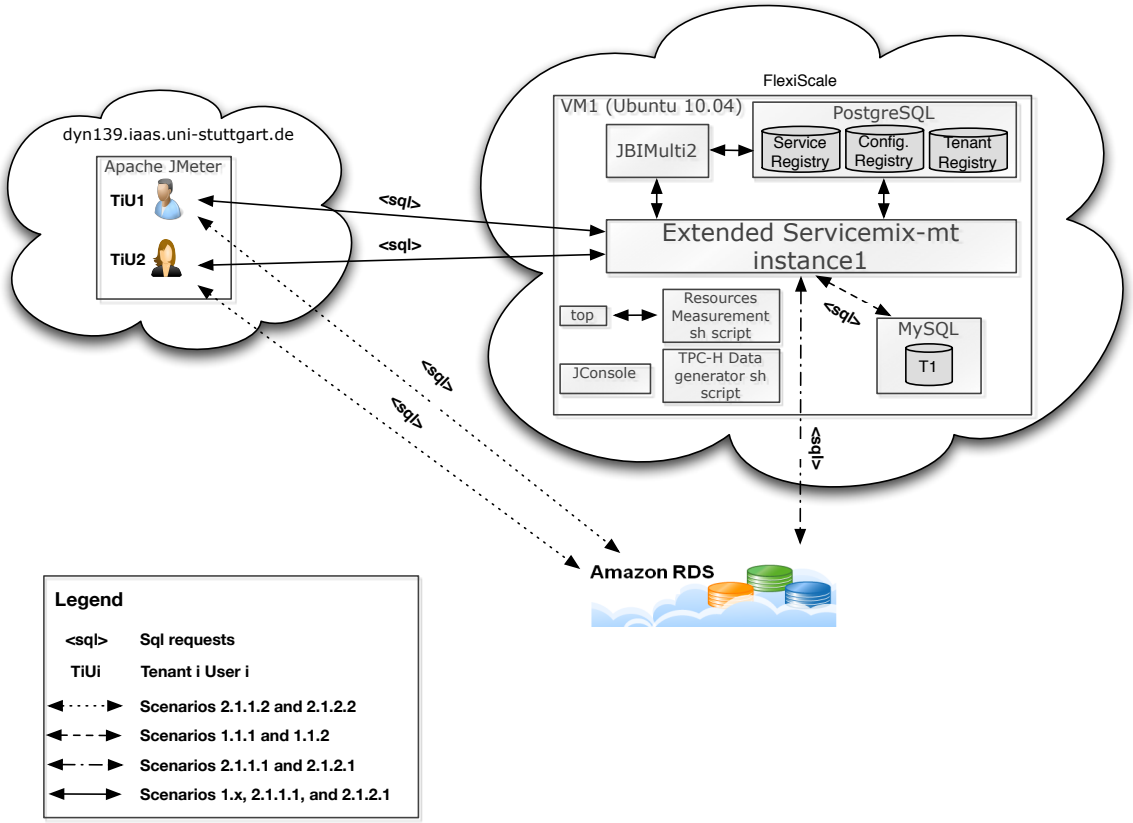

Figure 7.2.: Evaluation architecture overview for one tenant, two users, and local and remote SQL Cloud data store.

The third subsystem is hosted in a VM image in the FlexiScale Cloud infrastructure [Fle]. In an Ubuntu 10.04 64 bits operative system with a Java 6 VM we install the following components (see Figure 7.2):

- JOnAS 5.2.2.
- PostgreSQL 9.1.1.
- Extended version of JBIMulti2
- A MySQL 5.1 database system which hosts the tenant 1 database.
- Extended version of ServiceMix-mt: we modify its minimum and maximum heap consumption allowance, and set it to minimum 256 MB, and maximum 1 GB.
- TPC-H data and query generator.
- Resource measurements component

The resource measurements component measures the CPU utilization of the ServiceMix-mt Java process. Its memory consumption is measured using the JConsole program provided by the JVM (Java Virtual Machine). In this evaluation we are interested in measuring the heap

consumption, rather than the memory which is consumed by the JVM. The TPC-H is not used for measurement purposes, but only for data and query generation purposes.

The evaluation scenarios are defined in Table 7.1. We define 8 different scenarios, which follow the following criteria:

- Direct connection to the backend database system vs. connection through ServiceMix-mt.
- Number of Users, number of concurrent requests per user, and number of requests per user.
- Data stored in a local MySQL database (in the same instance as ServiceMix-mt) vs. data stored in an Amazon RDS MySQL database instance.

```
1 select l_returnflag, l_linestatus,sum(l_quantity) as sum_qty,sum(l_extendedprice) as
      sum_base_price,sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,sum(
      l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,avg(l_quantity) as
      avg_qty,avg(l_extendedprice) as avg_price,avg(l_discount) as avg_disc,count(*) as
      count_order from lineitem where l_shipdate <= DATE_SUB(date('1998-12-01'), interval 107
      day) group by l_returnflag, l_linestatus order by l_returnflag, l_linestatus;
```

Listing 7.5: Query included in the MySQL requests of the load generator.

The evaluation includes the measurements for the following performance units: throughput (requests per second), data transfer speed (KB/sec), CPU utilization (%), and Memory utilization (MB). Furthermore, the scenarios are run utilizing the same data retrieval query, which is detailed in Listing 7.5.

Id	User num.	Threads / user	Req. / thread	Backend database	Through ESB
1.1.1.1	2	20	100	Local MySQL	Yes
1.1.1.2	2	20	100	Local MySQL	No
1.1.2.1	2	50	200	Local MySQL	Yes
1.1.2.2	2	50	200	Local MySQL	No
2.1.1.1	2	20	100	MySQL Amazon RDS	Yes
2.1.1.2	2	20	100	MySQL Amazon RDS	No
2.1.2.1	2	50	200	MySQL Amazon RDS	Yes
2.1.2.2	2	50	200	MySQL Amazon RDS	No

Table 7.1.: Specification of the different scenarios to be evaluated. **Note:** Evaluated the performance for connection through CDASMix and direct to Amazon RDS [Amac].

7.3.3. Evaluation Analysis

In this subsection we discuss and present the evaluation results obtained from the execution of the scenarios described in Table 7.1. Before getting into the discussion, we point out a

problem in the evaluation, which leads us to discard the scenario 2.1.2.2 (see Table 7.1). The throughput obtained in the scenario 2.1.1.2 (see Table 7.1) is in average 3,3 requests per minute, and lasts approximately 10 hours. This fact makes us delete the last scenario execution, due to the low performance obtained, which we assume that it is related with the Quality of Service assigned to the student grants credits profile in Amazon RDS.

As it can be seen in Figure 7.3, utilizing ServiceMix-mt as the database layer component for communicating with a MySQL database system locally deployed lowers the throughput in a 32,74 % for 2 users, 20 concurrent threads per user, and 100 requests per thread. However, when the thread and request number are increased, the throughput exponentially decreases, as we reach the limit of concurrent connections supported in the MySQL CDASMix Proxy. When we compare it with a direct connection to a locally deployed MySQL database system, we can see that the concurrent requests are better handled by the MySQL server. In case of avoiding cashing support in ServiceMix-mt, the throughput would considerably decrease when increasing the load. When the distance between the different layers of the application increases, e.g. accessing a database layer deployed in a Cloud infrastructure A, and the database system is located in a Cloud infrastructure B, the number of requests per second decreases (see scenario 2.1.1.1 in Figure 7.3). However, the difference we obtain from hosting the database layer on premise, but accessing a database system off-premise is 98,64 % worse, if we compare if with the access through ServiceMix-mt. We must denote that in these scenarios there is a high temporal proximity of equal requests. Therefore, the cashing mechanism in the system increases considerably the throughput when accessing data hosted off-premise through ServiceMix-mt.

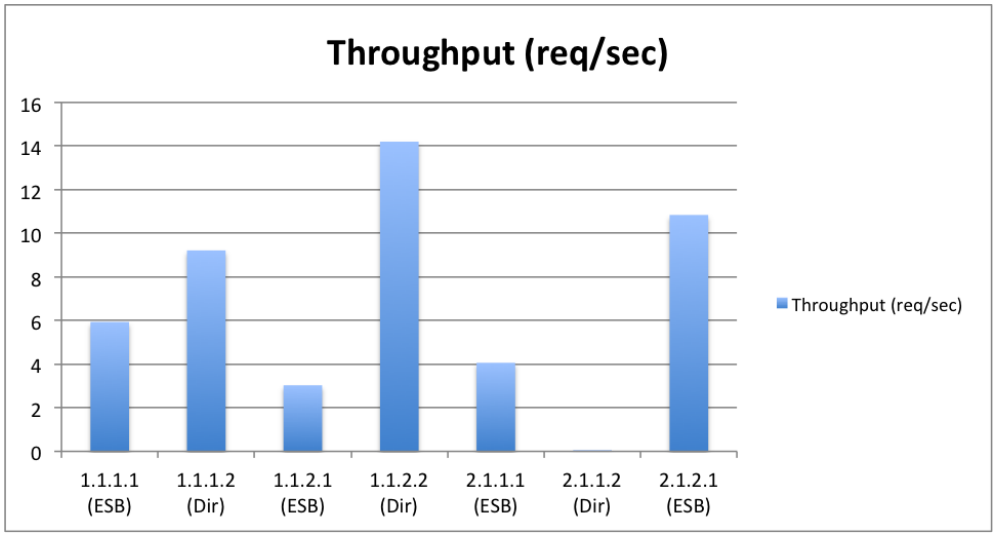

Figure 7.3.: Throughput (requests per second) for the different scenarios described in Table 7.1.

The amount of KB per second transmitted in the different scenarios correlates with the tendency which can be seen in throughput (see Figures 7.3 and 7.4).

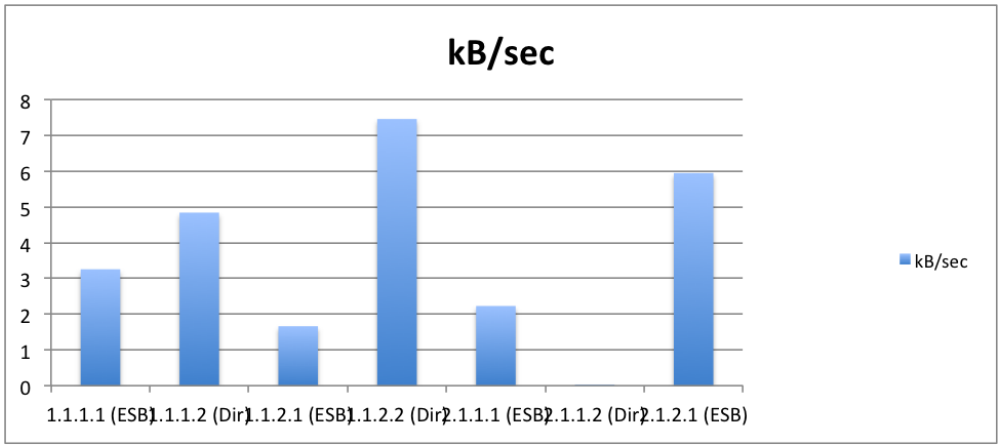

Figure 7.4.: Transmission speed (KB per second) for the different scenarios described in Table 7.1.

Memory utilization maintains stable along the different scenarios. We observe that in non of the scenarios the maximum heap size (1 GB) is reached in maximum or average values (see Figure 7.5). We obtain a lower memory utilization for the scenarios where data retrieval from a backend Cloud data store is involved. This difference relies on the network latency of having the data hosted in a database system which is not in the same network (in our evaluation, the database instance hosted in Amazon RDS), and the low network latency of having the database system in the same machine as the database layer. A greater number of threads handling the routing requests are blocked due to a higher response time when accessing a remote database system.

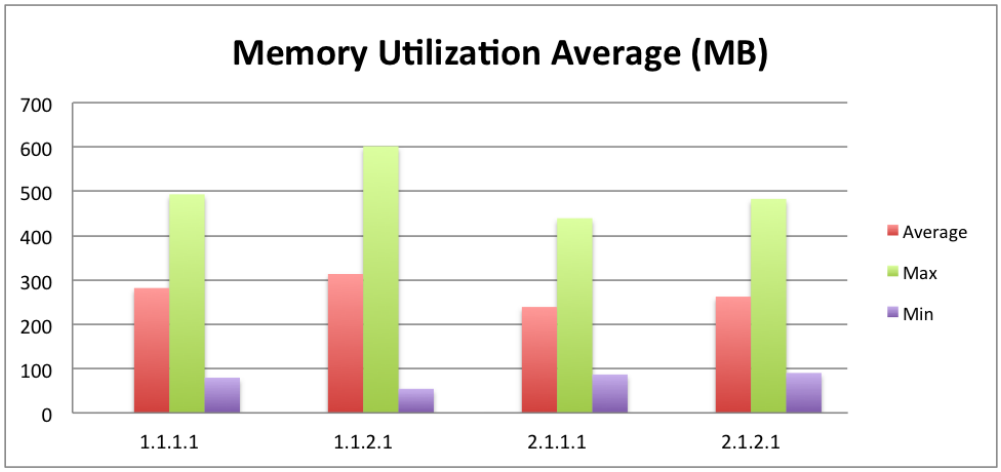

Figure 7.5.: Memory utilization (MB) for the different scenarios described in Table 7.1 where ServiceMix-mt is involved.

The same difference seen in the memory utilization can be observed in the CPU consumption (see Figure 7.6). When the requests are executed on a local database system, the response time per request is highly lower than a response time from a remote database. Therefore, the CPU utilization averages and maximum values are closer to each other. However, when

accessing the MySQL database instance in Amazon RDS, the maximum values correlate with the scenarios 1.1.1.1 and 1.1.2.1, but in average the CPU utilization is lower due to a higher number of blocked threads.

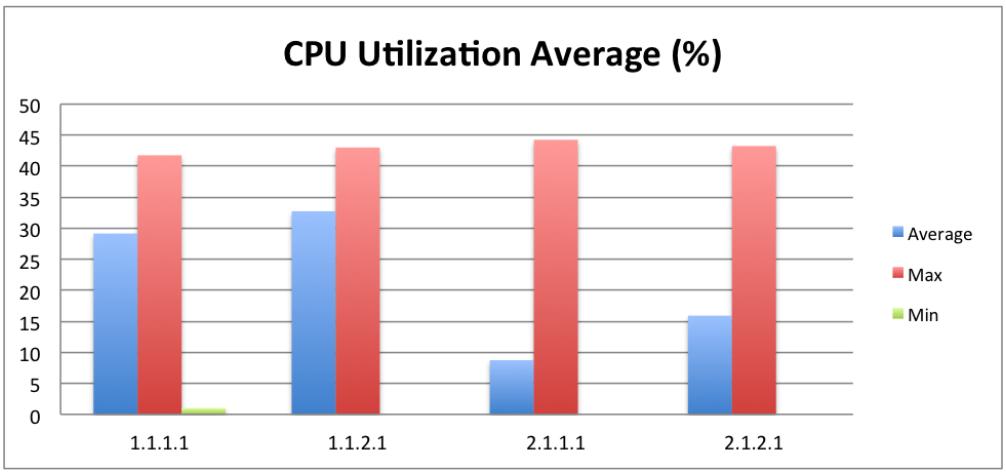

Figure 7.6.: CPU utilization (%) for the different scenarios described in Table 7.1 where ServiceMix-mt is involved.

8. Outcome and Future Work

Migration of one or more application layers to the Cloud aims to reduce the cost in the required IT infrastructure to host and maintain such layer within an organization. Adaptations on both migrated and non migrated layers are a must when part of an application is migrated to a Cloud infrastructure. In this diploma thesis we start from the basis of a partial migration of the application, particularly the database layer. The database layer includes the operations which provide data storage and retrieval support. Furthermore, the software, hardware and maintenance required to host and maintain this layer require an economical budget substantially greater than the needed for the business, or presentation layers of an application. Migrating the application's data to the Cloud requires rewiring the database connections to the backend Cloud data store, and adaptating the upper layers to match the operations, and data representations supported. Providing transparent communication support for accessing and storing data on application's databases in the Cloud demands a multi-protocol, and multi-tenant component. In this diploma thesis we extend a multi-tenant aware ESB in order to utilize it as the database layer of the application, and access multiple Cloud data stores providing SQL and NoSQL databases.

In Chapter 2 we present the necessary background about the technologies we use, and the components we reuse in this diploma thesis, e.g. JBIMulti2 [Muh12], JBI and OSGi frameworks, etc. Furthermore, we categorize the databases systems which are supported in the prototype, and subcategorize them based on their storage model and communication protocols.

After researching on the SQL and NoSQL database systems properties in Chapters 2 and 3, we find that most of database communication protocols are not standardized, and differ along the different database vendors. Therefore, we are forced in Chapter 5 to develop components which adjust to specific communication protocols: HTTP, and MySQL. The research described in Chapter 3 leads us to find a lack of standardization in the communication at the TCP level of the SQL database vendors, but the existence of components which support other communication protocols for incoming requests, e.g. HTTP, and utilize via JDBC the different database vendors' native driver to forward them to the backend database system [Red11]. However, approaches in this direction forces the developer to adjust their data access layer, whose adaptations we aim to minimize when utilizing our system. NoSQL Cloud data store providers show a lack of standardized naming, and provide the database users with their drivers for I/O operations. In order to address the lack of a standardized naming and access in the NoSQL providers, we categorize in the system's registry the user's databases meta-data into different categories and subcategories, and access the backend data stores via the HTTP protocol supported in ServiceMix-http-mt [Sá12].

The functional and non-functional requirements the system must fulfill are described in Chapter 4. After analyzing the requirements, providing an overview of the system, and

specifying the necessary use cases, we move to the design of the prototype in Chapter 5. We divide the design into the design of common components, and database specific components for the following databases types: SQL, and NoSQL databases. Apache ServiceMix-mt is used as the main component in the system for enabling a multi-protocol and multi-tenant communication support between backend databases. We design two different NMFs' content for requests for enabling a dynamic routing of requests between the backend database systems. We extend the JBIMulti2 registries schemas to support the storage of tenant's migrated database configuration data. Components which implement the different databases systems communication protocols, e.g. MySQL and HTTP, and components enabling routing between the multi-tenant aware consumer and provider endpoints are presented in Chapter 5. However, the SQL database support is limited in the system to one specific database system: MySQL. Separate components can be implemented to provide support for more SQL database vendors, e.g. PostgreSQL or Oracle. Furthermore, the NoSQL databases support is limited to the backend databases which support the HTTP communication protocol.

The implementation, validation, and evaluation of the system which complies with the requirements and the design of Chapters 4 and 5 respectively, is explained in Chapters 6 and 7. We first validate the system by creating backend databases which contain custom data, and data generated by the TPC-H benchmark [Tra01]. After configuring the communication configuration in CDASMix through the JBIMulti2 Web service interface, the tenant can communicate with the backend Cloud data store. Communication configuration should be done in the future through a user friendly Web interface, by integrating the *Cloud Data Migration Tool* and JBIMulti2 Web interfaces.

For evaluating the advantages and disadvantages of utilizing CDASMix as the communication component in a database layer, we create an evaluation baseline for a backend MySQL database, run the different evaluation scenarios, and discuss its results. Applications with a high number of I/O operations often suffer a high performance decrease [HLS+12]. Therefore, we enhance our prototype with a temporal multi-tenant cashing mechanism. Results describing the behavior of the system with a high load of data requests is described in Chapter 7, and demonstrate the advantages of cashing when accessing databases in the Cloud through CDASMix. An evaluation of the system's communication performance between NoSQL databases is recommended in future works.

Further future works involve a secure authentication mechanism in CDASMix, as well as horizontal scalability of the system, and query transformation. The former involves implementing in the CDASMix MySQL Proxy the authentication mechanisms supported in the MySQL database server, and including password verification in the authentication phase in the multi-tenant HTTP BC. Horizontal scalability can be obtained between multiple instances of ServiceMix-mt building the system. Future versions of CDASMix can provide separate but connected ServiceMix-mt instances for routing requests for SQL and NoSQL database systems, or implementing a load balancer between the multiple instances building the data access layer. The developed version of CDASMix does not provide query and data transformation between different database versions, or database vendors. However, the system's design and implementation is extensible. A transformation component can be inserted between the endpoints in ServiceMix-mt.

Appendix A.

Components

A.1. CDASMix MySQL Proxy

The MySQL proxy OSGi bundle is implemented on the Continuent Tungsten Connector [Con], which is a Java MySQL proxy which directly connects with the backend MySQL database system. We extend and adapt this proxy in order to integrate it with ServiceMix, aggregate transparency, multi-tenant awareness, cashing, and dynamic connection with the backend Cloud data sources.

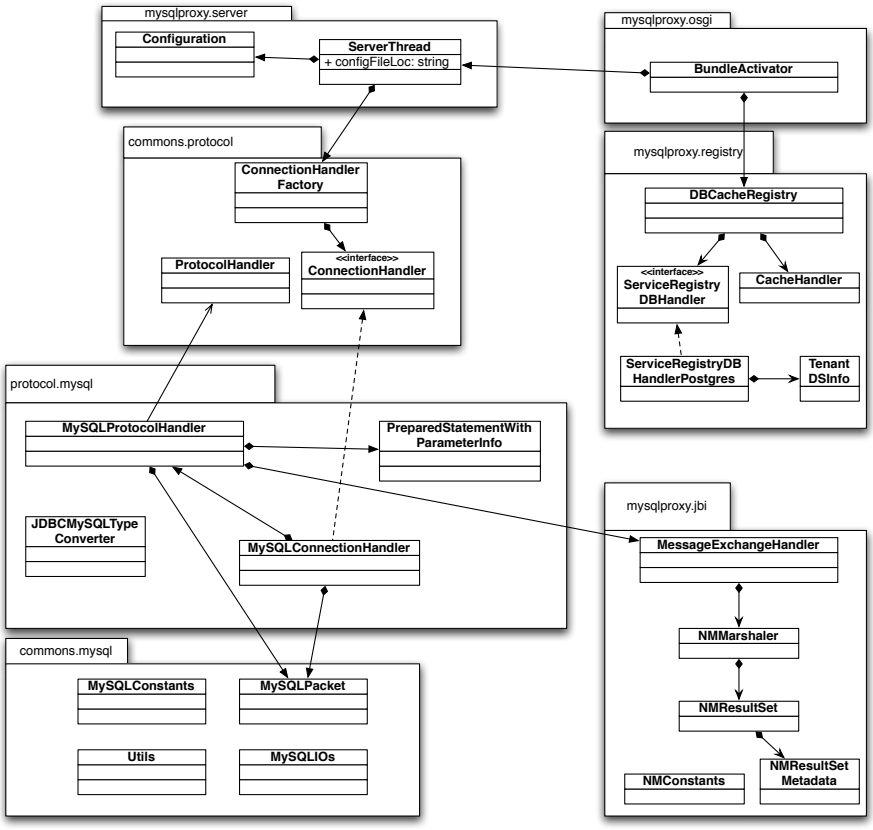

Figure A.1.: OSGi bundle providing MySQL support in ServiceMix-mt

A.2. CDASMix Camel JDBC

The *cdasmixjdbc* component is a custom component which is built and deployed as an OSGi bundle in ServiceMix-mt. It provides support for connections with backend SQL Cloud data stores, and message marshaling and demarshaling.

Figure A.2.: OSGi bundle and Camel component providing JDBC support in ServiceMix-mt

Appendix B.

Messages

In this chapter we provide an overview of the requests which are sent to, and received from the extended ServiceMix-mt. For MySQL requests we provide the TCP packets which are transferred between the application, ServiceMix-mt, and the backend MySQL database system. For NoSQL requests we present messages samples which are in JSON format, but its content varies among the different backend Cloud data store providers.

B.1. Normalized Message Format Content Description

In this section we provide an overview of the data structures which are sent in the NMF. The sections of the NMF where the data and meta-data are sent are the *properties*, and the *attachment*. In the Listing B.1 we detail the contents sent in each of the sections, and the data structures in which the data and meta-data are stored.

```
1  ########## Normalized Message Adaptation for CDASMix ############
2
3  Properties for the backend connections are stored in this format. At this time, only joins
       which involve tables in the same backend db are supported:
4
5  ////////////////////////////////////////////////////////////////////
6
7    - MySQL || NoSql -> NMF (Request Message)
8      // Main properties
9      - target_data_sources : number of target data sources. This number will set the length
           of the properties vector and will set the length of the vector queries
10     - tenantId : string (UUID)
11     - userId : string (UUID)
12     if (mysql)
13       - mysqlServerPropsQueries:vector<string>      // server configuration queries that the
             proxy has received from the jdbc driver, e.g. SET names, SET character_set_*
14
15     // Backend datasource dependent properties : stored as Properties in vector<properties>.
16     - Tenant configuration data
17       - source & target dataSource name : string
18       - source & target dataSource type : family-mainInfoStructure-secondaryInfoStructure-
             version
19       - source & target dataSource protocol : mysql | http.[xml|rest]
20       - target datasource endpoint url: string
21       - target datasource user and password : string, string
```

```
22        - source & target endpoint type: endpoint.type.jdbc | endpoint.type.http | endpoint.
              type.jms
23        - targetJBIEndpointURI: QName
24        - source and target information structure information:
25                if source is sql
26                  - src_main_info_structure_name : string
27                  - target_main_info_structure_name : string
28                if source is nosql
29                  - src_main_info_structure_name : string
30                  - target_main_info_structure_name : string
31                  - src_secondary_info_structure_name : string
32                  - target_secondary_info_structure_name : string
33
34        - if the target protocol == mysql
35          - target native driver name : native driver name, e.g. "com.mysql.jdbc.driver"
36          - bean DS name (if exists)
37          - escapeProcessing : true | false               //statement property of mysql
38          - fetchSize : int (statement property of mysql)
39          - returnGeneratedKeys : true | false            //statement property of mysql
40
41      // Attachment
42      if (SQL)
43        - Query/ies go/es in the NMF body as a vector<String>, with query data (for insert
              queries).
44      else
45        - NoSQL JSON payload
46
47
48  ////////////////////////////////////////////////////////////////////////
49
50    - NMF -> MySQL || NoSql (Response Message)
51      // Main properties
52      - target_data_sources : number of target data sources. This number will set the length
            of the properties vector and the length of the result
53      - tenantId : string (UUID)
54      - userId : string (UUID)
55      - target_op_status = ok | error
56
57      - if source protocol == mysql
58
59        - updateCount = int           // the current result as an update count
60        - resultSetMetadataNumber: int      // number of result set in the vector of response
              (index in the result set metadata vector)
61
62      // Backend datasource dependent properties : stored as Properties in the vector<
            properties>.
63      - source & target dataSource name : string
64      - source & target dataSource type : family-mainInfoStructure-secondaryInfoStructure-
            version
65      - source & target dataSource protocol : mysql | http.[xml|rest]
66      - source & target endpoint type: endpoint.type.jdbc | endpoint.type.http | endpoint.type
            .jms
67      - source and target information structure information:
68              if source is sql
69                - src_main_info_structure_name : string
```

```
70              - target_main_info_structure_name : string
71            if source is nosql
72              - src_main_info_structure_name : string
73              - target_main_info_structure_name : string
74              - src_secondary_info_structure_name : string
75              - target_secondary_info_structure_name : string
76
77    // Attachment
78    - if target_op_status == error
79      - target_op_error : vector<Hashmap<String,String>>            // map of error code and
            error message per backend db
80      else
81
82
83      - if source protocol == mysql
84
85        - vector<arraylist[HashMap<columnName,value>]]>           // result sets are
            inserted as arraylist in the body
86
87        - ResultSetMetadata: Vector<HashMap<String,Object>>
88          - columncount : int                       // number of columns in this ResultSet
              object.
89          - columntype : arraylist<int>                 // the designated column's SQL type.
90          - isnullable : arraylist<int>                 // if the column can contain null
              values
91          - isAutoIncrement : arraylist<int>                // if the column values
              increment when the rows do
92          - tableName : arraylist<string>                 // name of the table where a
              column is contained
93          - stringcolumnlabel : arraylist<string>             // label which sql proposes
              for printing reasons
94          - stringcolumnname : arraylist<string>              // column name
95          - ColumnDisplaySize : arraylist<int>              // Indicates the designated
              column's normal maximum width in characters.
96          - Scale : arraylist<int>                    // column's number of digits to right
              of the decimal point
97          - RowCount : int                        // number of rows
98          - ColumnMapper : Hashmap<int,String> (column index, name)   // mapping of the
              column index with the column name
99          - ColumnSigned : arraylist<int>                 // if the values of the columns
              are signed or unsigned
100       else
101       - versionControl
102       - nosql metadata
103       - nosql JSON payload
104
105 ////////////////////////////////////////////////////////////////////
106
107   Note: response to the user -> distributed atomic transaction mechanism used. If one of
          the backend datasources reports an error as a response, the final response to the
          tenant is an error response:
108
109     - Error response: string of the error + the datasource/s which gave the error.
110
111     - OK response: the full result set, this means, the vector<arraylist[HashMap<
```

```
        columnName,value>]]> created in the data transformation SE or jdbccdasmix
        component.
112
113
114 ////////////////////////////////////////////////////////////////////
```

Listing B.1: Detail of the content and data structures used to send the requests' data and meta-data.

B.2. MySQL TCP Stream

In this section we provide two TCP streams which are captured with the *ngrep* program for UNIX [Sou]. The first stream captures the TCP packets on port 3311, where the MySQL component in ServiceMix-mt listens for incoming connections (see Listing B.2). The second stream captures the TCP packets on port 3306, where the a locally deployed MySQL server listens for incoming connections (see Listing B.3).

```
interface: eth3 (109.231.70.232/255.255.255.248)
filter: (ip or ip6) and ( port 3311 )

T 109.231.70.234:3311 -> 129.69.214.249:52190 [AP]
  45 00 00 00 0a 35 2e 31    2e 31 2d 53 65 72 76 69    E....5.1.1-Servi
  63 65 4d 69 78 2d 34 2e    33 2e 30 00 2d 02 00 00    ceMix-4.3.0.-...
  53 28 72 4d 51 30 6c 61    00 00 0d a2 02 00 00 00    S(rMQ0la........
  00 00 00 00 00 00 00 00    00 00 00 32 2f 54 6f 44    ...........2/ToD
  4d 4a 39 2a 70 69 49 00    00                         MJ9*piI..

  ...

T 129.69.214.249:52190 -> 109.231.70.234:3311 [AP]
  0f 00 00 00 03 53 45 54    20 4e 41 4d 45 53 20 75    .....SET NAMES u
  74 66 38                                              tf8

T 109.231.70.234:3311 -> 129.69.214.249:52190 [AP]
  07 00 00 01 00 00 00 02    00 00 00                   ...........

  ...

T 129.69.214.249:52190 -> 109.231.70.234:3311 [AP]
  1d 00 00 00 03 73 65 6c    65 63 74 20 2a 20 66 72    .....select * fr
  6f 6d 20 6d 61 69 6e 49    6e 66 6f 54 65 73 74 31    om mainInfoTest1
  3b                                                    ;

T 109.231.70.234:3311 -> 129.69.214.249:52190 [AP]
  01 00 00 01 02 46 00 00    02 03 64 65 66 12 69 6e    .....F....def.in
  66 6f 72 6d 61 74 69 6f    6e 5f 73 63 68 65 6d 61    formation_schema
  0d 6d 61 69 6e 49 6e 66    6f 54 65 73 74 31 0d 6d    .mainInfoTest1.m
  61 69 6e 49 6e 66 6f 54    65 73 74 31 02 49 44 02    ainInfoTest1.ID.
  49 44 0c 21 00 0b 00 00    00 03 01 02 00 00 00 4a    ID.!...........J
  00 00 03 03 64 65 66 12    69 6e 66 6f 72 6d 61 74    ....def.informat
  69 6f 6e 5f 73 63 68 65    6d 61 0d 6d 61 69 6e 49    ion_schema.mainI
  6e 66 6f 54 65 73 74 31    0d 6d 61 69 6e 49 6e 66    nfoTest1.mainInf
  6f 54 65 73 74 31 04 6e    61 6d 65 04 6e 61 6d 65    oTest1.name.name
  0c 21 00 ff ff 00 00 fc    01 00 00 00 00 05 00 00    .!..............
  04 fe 00 00 02 00 13 00    00 05 01 33 10 54 68 69    ...........3.Thi
  73 69 73 41 4e 65 77 4e    61 6d 65 33 33 08 00 00    sisANewName33...

  ...

T 129.69.214.249:52190 -> 109.231.70.234:3311 [AP]
  01 00 00 00 01                                        .....

T 109.231.70.234:3311 -> 129.69.214.249:52190 [AP]
  07 00 00 01 00 00 00 02    00 00 00                   ...........

T 129.69.214.249:52190 -> 109.231.70.234:3311 [R]
  00 00 00 00 00 00                                     ......

T 129.69.214.249:52190 -> 109.231.70.234:3311 [R]
  00 00 00 00 00 00                                     ......
```

Listing B.2: TCP Stream for a MySQL communication captured on port 3311 with the program *ngrep* [Sou].

```
interface: lo (127.0.0.0/255.0.0.0)
filter: (ip or ip6) and ( port 3306 )

  ...

T 127.0.0.1:46409 -> 127.0.0.1:3306 [AP]
  0f 00 00 00 03 53 45 54    20 4e 41 4d 45 53 20 75    .....SET NAMES u
  74 66 38                                              tf8

T 127.0.0.1:3306 -> 127.0.0.1:46409 [AP]
  07 00 00 01 00 00 00 02    00 00 00                   ...........

  ...

T 127.0.0.1:46409 -> 127.0.0.1:3306 [AP]
  50 00 00 00 03 75 70 64    61 74 65 20 6d 61 69 6e    P....update main
  49 6e 66 6f 54 65 73 74    31 20 73 65 74 20 6e 61    InfoTest1 set na
  6d 65 3d 27 54 68 69 73    69 73 41 4e 65 77 4e 61    me='ThisisANewNa
  6d 65 33 33 27 20 77 68    65 72 65 20 6e 61 6d 65    me33' where name
  3d 27 54 68 69 73 69 73    41 4e 65 77 4e 61 6d 65    ='ThisisANewName
  32 32 27 3b                                           22';

T 127.0.0.1:3306 -> 127.0.0.1:46410 [AP]
  45 00 00 00 0a 35 2e 31    2e 36 37 2d 30 75 62 75    E....5.1.67-0ubu
  6e 74 75 30 2e 31 30 2e    30 34 2e 31 00 c0 02 00    ntu0.10.04.1....
  00 5f 67 6b 63 2f 5e 6a    7b 00 ff f7 08 02 00 00    ._gkc/^j{.......
  00 00 00 00 00 00 00 00    00 00 00 00 4e 53 35 4e    ............NS5N
  68 44 57 6a 2f 48 5e 21    00                         hDWj/H^!.

T 127.0.0.1:46410 -> 127.0.0.1:3306 [AP]
  4a 00 00 01 8f a2 02 00    ff ff ff 00 21 00 00 00    J...........!...
  00 00 00 00 00 00 00 00    00 00 00 00 00 00 00 00    ................
  00 00 00 00 72 6f 6f 74    00 14 f3 54 d0 fa f3 56    ....root...T...V
  e9 c0 43 2c 4c 78 18 88    ac de 4c 5e aa d1 64 61    ..C,Lx....L^..da
  74 61 53 6f 75 72 63 65    54 65 73 74 31 00          taSourceTest1.

T 127.0.0.1:3306 -> 127.0.0.1:46410 [AP]
  07 00 00 02 00 00 00 02    00 00 00                   ...........

  ...

T 127.0.0.1:46410 -> 127.0.0.1:3306 [AP]
  1d 00 00 00 03 73 65 6c    65 63 74 20 2a 20 66 72    .....select * fr
  6f 6d 20 6d 61 69 6e 49    6e 66 6f 54 65 73 74 31    om mainInfoTest1
  3b                                                    ;
```

Listing B.3: TCP Stream for a MySQL communication captured on port 3306 with the program *ngrep* [Sou].

Bibliography

[10G] 10Gen, Inc. Mongo DB. http://www.mongodb.org/.

[ABLS13] V. Andrikopoulos, T. Binz, F. Leymann, and S. Strauch. How to Adapt Applications for the Cloud Environment. Challenges and Solutions in Migrating Applications to the Cloud. In: Springer Computing (to appear), 2013.

[Amaa] Amazon Web Services, Inc. Amazon DynamoDB. http://aws.amazon.com/dynamodb/.

[Amab] Amazon Web Services, Inc. Amazon EC2. http://aws.amazon.com/ec2/.

[Amac] Amazon Web Services, Inc. Amazon Relational Database Service (Amazon RDS). http://aws.amazon.com/rds/.

[Amad] Amazon Web Services, Inc. Amazon S3. http://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html.

[Amae] Amazon Web Services, Inc. Amazon SimpleDB. http://docs.aws.amazon.com/AmazonSimpleDB/latest/DeveloperGuide/DataModel.html.

[Amaf] Amazon Web Services, Inc. Amazon Web Services Educational Grants. http://aws.amazon.com/grants/.

[Amag] Amazon Web Services, Inc. AWS SDK for Java. http://aws.amazon.com/sdkforjava/.

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

[AP11] M. A. ALzain and E. Pardede. Using Multi Shares for Ensuring Privacy in Database-as-a-Service, Proceedings of the 44th Hawaii International Conference on System Sciences - 2011, 2011.

[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/.

[Bac12] T. Bachmann. Entwicklung einer Methodik für die Migration der Datenbankschicht in die Cloud. Diploma thesis nº 3360, Institute für Architecture von Anwendungssystemen, University of Stuttgart, 2012.

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

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

[CJP+11] C. Curino, E. P. C. Jones, R. A. Popa, N. Malviya, E. Wu, S. Madden, H. Balakrishnan, and N. Zeldovich. Relational Cloud: A Database-as-a-Service for the Cloud, 2011.

[CJZ+10] C. Curino, E. Jones, Y. Zhang, E. Wu, and S. Madden. Relational Cloud: The Case for a Database Service, 2010.

[Cod] Codehous. Jackson Java JSON-processor. `http://jackson.codehaus.org/`.

[Con] Continuent, Inc. Continuent Tungsten Connector. `http://sourceforge.net/apps/mediawiki/tungsten/index.php?title=Introduction_to_the_Tungsten_Connector`.

[Dro] Dropbox, Inc. Dropbox. `www.dropbox.com`.

[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.

[Fer10] P. Feresten. Storage Multi-Tenancy for Cloud Computing, March 2010. SNIA Cloud Storage Iniciative.

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

[Fle] Flexiant, Ltd. Flexiscale Cloud Infrastucture. `http://www.flexiscale.com/`.

[Fus11] FuseSource. *Fuse ESB 4.4 – Product Introduction*, 2011. `http://fusesource.com/docs/esb/4.4/esb_prod_intro/`.

[Gooa] Google, Inc. Google App Engine Data Store Overview. `https://developers.google.com/appengine/docs/python/datastore/overview`.

[Goob] Google, Inc. Google Cloud SQL. `https://developers.google.com/cloud-sql/`.

[Gooc] Google, Inc. Google Coud Storage. `https://developers.google.com/storage/docs/getting-started`.

[GW99] J. R. Groff and P. N. Weinberg. *SQL, the Complete Reference*. Osborne, MacGraw-Hill, 1999.

[HHM02] B. I. Hakan Hacıgümüs and S. Mehrotra. Providing Database as a Service, Proceedings of the 18th International Conference on Data Engineering (ICDEí02), 2002.

[HLS+12] H. Han, Y. C. Lee, W. Shin, H. Jung, H. Y. Yeom, and A. Y. Zomaya. Cashing in on the Cache in the Cloud, 2012.

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

[Lai09] E. Lai. No to SQL? Anti-database movement gains steam. *Computer World*, 2009. `http://www.computerworld.com/s/article/9135086/No_to_SQL_Anti_database_movement_gains_steam_`.

[Muh] D. Muhler. Manual for the JBIMulti2 Implementation. Modified by Santiago Gomez, 2012.

[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.

[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`.

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

[Oraa] Oracle, Inc. The Java Database Connectivity. `http://www.oracle.com/technetwork/java/overview-141217.html`.

[Orab] Oracle, Inc. Oracle Cloud Database. `https://cloud.oracle.com/mycloud/f?p=service:database:0`.

[Ora12] Oracle, Inc. MySQL Proxy Guide, 2012.

[Ora13] Oracle, Inc. MySQL 5.6 Reference Manual, 2013.

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

[RD09] T. Rademakers and J. Dirksen. *Open Source ESBs in Action*. Manning Publications Co., 2009.

[Red11] Red Hat, Inc. Gap analysis: The case for data services, 2011.

[Sá12] S. G. Sáez. Integration of Different Aspects of Multi-tenancy in an Open Source Enterprise Servis Bus. Student thesis nº 2394, Institute of Architecture of Application Systems, University of Stuttgart, 2012.

[SAB+12] S. Strauch, V. Andrikopoulos, U. Breitenbücher, O. Kopp, and F. Leymann. Non-Functional Data Layer Patterns for Cloud Applications. Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science. CloudCom., December 2012.

[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.

[SBK+12] S. Strauch, U. Breitenbuecher, O. Kopp, F. Leymann, and T. Unger. Cloud Data Patterns for Confidentiality, Closer 2012 - 2nd International Conference on Cloud Computing and Services Science, 2012.

[SF12] P. J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley Professional, 2012.

[Sou] Sourceforge. NGrep. `http://ngrep.sourceforge.net/usage.html`.

[Sto12] Storage Networking Industry Association. Cloud Data Management Interface (CDMITM), June 2012.

[Str] C. Strauch. NoSQL Databases. `http://www.christof-strauch.de/nosqldbs.pdf`.

[Ter] Terracota, Inc. Ehcache. `http://ehcache.org/`.

[Thea] The Apache Software Foundation. Apache Couch DB. `http://couchdb.apache.org/`.

[Theb] The Apache Software Foundation. Apache JMeter 2.9. `http://jmeter.apache.org/`.

[Thec] The Apache Software Foundation. Camel JDBC. `http://camel.apache.org/jdbc.html`.

[Thed] The Apache Software Foundation. Virgil Cassandra RESTful API. `https://github.com/hmsonline/virgil`.

[The96] The PostgreSQL Global Development Group. PostgreSQL 9.1 Reference manual, 1996.

[TP11] O. M. Tamer and V. Patrick. *Principles of Distributed Database Systems*. Springer, 2011.

[Tra01] Transaction Processing Performance Council. TPC Benchmark, 2001. `http://www.tpc.org/information/benchmarks.asp`.

[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 nº 3347, Institute of Architecture of Application Systems, University of Stuttgart., 2012.

[W3C04] W3C. 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.

[WPG+10] J. Wu, L. PING, X. GE, Y. Wang, and J. FU. Cloud Storage as the Infrastructure of Cloud Computing. 2010 International Conference on Intelligent Computing and Cognitive Informatics, 2010.

All links were last followed on March 21, 2013.

Acknowledgement

I am heartily thankful to my supervisor Steve Strauch from the University of Stuttgart for his encouragement, guidance and support in all the phases of this diploma thesis. I am also grateful to Dr. Vasilios Andrikopoulos for his advices and useful tips. Special thanks to my family, friends and girlfriend for their moral support.

Santiago Gómez Sáez

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, 22nd March 2013 _____

(Santiago Gómez Sáez)