Institute of Architecture of Application Systems

ESB^{MT}: A Multi-tenant Aware Enterprise Service Bus

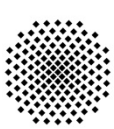
Steve Strauch, Vasilios Andrikopoulos, Santiago Gómez Sáez, Frank Leymann

Institute of Architecture of Application Systems,
University of Stuttgart, Germany
{strauch, andrikopoulos, gomez-saez, leymann}@iaas.uni-stuttgart.de

BibTeX:

```
@ARTICLE{,
author = {Strauch, Steve and Andrikopoulos, Vasilios and Gómez Sáez, Santiago
and Leymann, Frank},
title = {ESB^MT: A Multi-tenant Aware Enterprise Service Bus},
journal = {International Journal of Next-Generation Computing},
publisher = {Perpetual Innovation Media Pvt. Ltd},
volume = {4},
number = {3},
pages = {230--249},
url = {http://ijngc.perpetualinnovation.net},
year = {2013}
}
```

© 2013 Perpetual Innovation Media Pvt. Ltd.
The original publication is available at
International Journal of Next-Generation Computing



University of Stuttgart
Germany

ESB^{MT}: A Multi-tenant Aware Enterprise Service Bus

STEVE STRAUCH, VASILIOS ANDRIKOPOULOS, SANTIAGO GÓMEZ SÁEZ, and FRANK LEYMANN
IAAS, University of Stuttgart

Multi-tenancy, the sharing of the whole technological stack by different consumers at the same time, allows service providers to maximize resource utilization and reduce servicing costs per customer. Essential components of the contemporary enterprise environment like the Enterprise Service Bus (ESB) are therefore required to raise to the challenge of supporting and enabling multi-tenancy, becoming multi-tenant aware. Towards this goal, in this work we discuss the requirements for multi-tenant ESB solutions as fundamental building blocks in the Platform as a Service (PaaS) Cloud delivery model. Addressing these requirements, we propose a solution for dealing with multiple tenant contexts on the level of middleware, based on which we develop ESB^{MT}, an implementation-agnostic multi-tenant aware ESB architecture that we instantiate based on the Apache ServiceMix ESB open source solution. Evaluating the performance of our proposal required the extension of an ESB benchmark, the results of which for different deployment options we also present in this work.

Keywords: Enterprise Service Bus, Multi-tenancy, Cloud-enabled middleware

1. INTRODUCTION

Multi-tenancy and virtualization allow Cloud computing solutions to serve multiple customers from a single system instance sharing computational resources between them. Using these techniques, Cloud service providers maximize the utilization of their infrastructure, and therefore increase their return on infrastructure investment, while reducing the costs of servicing each customer. On the other hand, Cloud service consumers should be able to experience multi-tenancy in a transparent manner, without a loss in the performance of their applications due to the sharing of resources. While many industrial solutions exist for virtualization[1], multi-tenancy is an issue still under research and as such, it is the focus of this work.

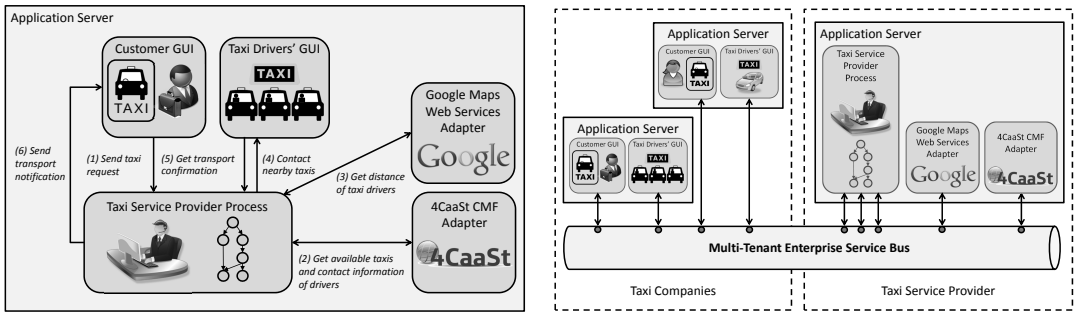
Multi-tenancy has been defined in different ways in the literature, see for example [Guo et al. 2007], [Mietzner et al. 2009], [Krebs et al. 2012]. Such definitions however do not address the whole technological stack behind the different Cloud delivery models [Mell and Grance 2011] (IaaS — Infrastructure as a Service, PaaS — Platform as a Service, SaaS — Software as a Service). For this purpose, in [Strauch et al. 2012] we defined multi-tenancy as *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.* The differentiation between *tenants* (organizational domains) and *users* (individual entities inside these groups) allows for different levels of granularity in the sharing of resources.

Realizing multi-tenancy requires that both the infrastructure and the applications depending on it become *multi-tenant aware*. Multi-tenancy awareness entails being able to differentiate between tenants, provide an appropriate level of data and performance isolation for each tenant,

[1]See for example: `http://www.xen.org`, `http://www.vmware.com/products/`, and `http://www.flexiant.com/`

The research leading to these results has received funding from projects 4CaaSt (grant agreement no. 258862) and Allow Ensembles (grant agreement no. 600792) part of the European Union's Seventh Framework Programme (FP7/2007-2013). The authors would like to thank Dominik Muhler for his valuable contribution to the development of ESB^{MT}.

Authors' address: IAAS, Universitätsstr. 38, D-70569 Stuttgart, Germany



(a) Communication of Web applications of Taxi Scenario without ESB

(b) Communication of Web applications of Taxi Scenario after introducing multi-tenant ESB

Figure 1: Overview of the Taxi Scenario

and allow tenants to be served by individually configured and managed services on demand. In this sense, multi-tenancy awareness of the application components and underlying infrastructure is the mechanism enabling multi-tenancy on the level of application. As such, it is the main focus of this work.

This definition of multi-tenancy awareness affects the different Cloud service models in different ways. In this work in particular we scope the discussion to the PaaS delivery model and investigate how multi-tenancy can be enabled for platforms offered as a service. In particular, we show how a critical middleware component of the PaaS model like the Enterprise Service Bus (ESB) [Chappell 2004] can be made multi-tenant aware, irrespective of the particular implementation technology used (i.e. which ESB solution is used). The concept of an ESB as the messaging hub between applications addresses the fundamental need for application integration and in the last years it has become ubiquitous in enterprise computing environments. ESBs control the message handling during service invocations and are at the core of each Service-Oriented Architecture (SOA) [Chappell 2004], [Josuttis 2007]. In order therefore to leverage the transition of enterprise environments to the Cloud paradigm, it is essential to make ESBs multi-tenant aware.

The contributions of this work can be summarized as follows:

(1) An identification of the requirements of enabling multi-tenancy for ESB solutions.

(2) A proposal for an implementation-agnostic ESB architecture called ESB^{MT} that fulfills these requirements.

(3) A prototype implementation of ESB^{MT}.

(4) A performance evaluation of the prototype.

The rest of this paper is organized as follows: Section 2 motivates this work by means of an informative scenario. Drawing from this scenario, Section 3 proceeds to identify and discuss the requirements for enabling multi-tenancy in ESB solutions as a part of the PaaS model. Section 4 investigates the impact of tenant awareness on the message processing within the ESB. The requirements identified in the previous two sections are addressed in Section 5, which presents ESB^{MT}, a generic, implementation independent ESB architecture. Section 6 discusses the realization of this architecture as a proof-of-concept implementation of our proposal. Section 7 presents the performance evaluation of our approach based on an existing ESB benchmark we extended for multi-tenancy support. Section 8 compares our proposal with existing works. Finally, Section 9 summarizes our findings and briefly presents future work.

2. MOTIVATING SCENARIO

The European Union research project 4CaaSt[2] aims to create a Cloud platform to design services and compositions based on Cloud-aware building blocks provided by the platform, offer them in a marketplace, and operate them at Internet-scale. The goal of the 4CaaSt platform is to lower the entry barrier for small and medium enterprises by offering an advanced environment, which reduces the effort to create innovative applications leveraging the benefits of Cloud computing. In the scope of 4CaaSt, the *Taxi Scenario* use case has been defined, where a service provider offers a taxi management software as a service to different taxi companies, i.e., *tenants*. Taxi company customers, who are the *users* of the tenant, submit their taxi transportation requests to the company that they are registered with. The taxi company uses the taxi management software to contact nearby taxi drivers. Once one of the contacted taxi drivers has confirmed the transportation request, the taxi management software sends a transport notification containing the estimated arrival time to the customer.

Figure 1a provides an overview of the realization of the taxi scenario without using an ESB. The taxi management software is implemented as a BPEL process [Alves et al. 2007]. The BPEL process leverages the 4CaaSt platform internal Context Management Framework (4CaaSt CMF), which provides context information about taxi cab locations and taxi driver contact details. Moreover, Google Maps Web Services[3] provide distance calculations between the location of a taxi cab and the pick up location. All components of the taxi booking service are Web applications deployed in Java Open Application Server (JOnAS)[4]. Additionally, the BPEL engine Orchestra[5] is deployed as a Web application, executing the taxi service provider process. As the service endpoints of the 4CaaSt CMF and the Google Maps Web Services are incompatible with the BPEL process, two adapter applications mediate between the BPEL process and these internal and external services. All applications communicate via point-to-point messaging connections.

Introducing an ESB as the messaging middleware (Figure 1b) enables loose coupling and provides a more flexible integration solution by avoiding hard-coded point-to-point connections. This makes the monitoring, management, and maintenance of the taxi application easier and more effective. Furthermore, enabling multi-tenancy at the ESB level allows *multiple* taxi companies to use the same taxi application offered as a service by a *single* provider (using customized GUIs for their customers and drivers if required) as shown in Figure 1b. Apart therefore from allowing taxi companies to outsource the development, deployment, operation, and management of such an application to a service provider, this solution also maximizes the benefits on the provider side.

3. REQUIREMENTS FOR MULTI-TENANT ESB

In the following, we discuss the requirements for multi-tenancy of ESB solutions as a key component of the PaaS model. For this purpose we first discuss how multi-tenancy affects the PaaS model in general, before refining the discussion further for ESBs.

3.1 Multi-tenancy in the PaaS Delivery Model

Discussing multi-tenancy requires that the views of all involved parties are considered, namely both the providers and the consumers of multi-tenant aware services and applications. From the providers' point of view, multi-tenancy allows the maximization of resource utilization and therefore enables maximization of profit. For service consumers, multi-tenancy has to be largely transparent, apart from providing access credentials when using the service or application. More importantly, consumers must have the impression that they are the only ones using the multi-tenant service or application, without suffering from side effects caused by other consumers

[2]The 4CaaSt project: `http://www.4caast.eu`
[3]Google Maps Web Services: `http://code.google.com/apis/maps/documentation/webservices/`
[4]Java Open Application Server (JOnAS): `http://jonas.ow2.org`
[5]Orchestra – Open Source BPEL / BPM Solution: `http://orchestra.ow2.org`

regarding, e.g., quality of services. Finally, consumers need to be provided with customization capabilities, such as taxi company-specific Web interfaces in the Taxi Scenario.

The three Cloud service models (I-,P- and SaaS) differ significantly in the granularity of the functionality provided to the consumer, and the required capability of the consumer to manage and control the underlying Cloud infrastructure [Mell and Grance 2011]. The responsibility of the provider and the effort of the consumer to enable multi-tenancy is therefore different, depending on the chosen Cloud service model. PaaS in particular, is the Cloud service model where the responsibility and effort of provider and consumer are nearly the same with respect to our definition of multi-tenancy. The exact effort of the consumer depends on the scenario and the application to be realized. The consumer is responsible to enable multi-tenancy of the application and the corresponding artifacts deployed on the platform; for example, the database schema used has to support multi-tenancy natively [Chong et al. 2006]. The provider has to enable multi-tenancy for the hardware resources and infrastructure, as well as the platform on which the various applications are deployed. Furthermore, for the sake of backward compatibility the deployment of non multi-tenant applications has also to be possible, otherwise the target community of the PaaS offering will be limited. Therefore, the service offered via PaaS by the provider has to support the deployment of both multi-tenant and non multi-tenant services and applications.

3.2 ESB Multi-tenancy Requirements

Following the discussion about multi-tenancy of PaaS components, in the context of project 4CaaSt we identified and categorized a set of *functional* and *non-functional* requirements for multi-tenant ESBs. Toward this goal we also refined the multi-tenancy characteristics (e.g. tenant awareness) identified in the literature, e.g. [Azeez et al. 2010], [Guo et al. 2007], [Mietzner et al. 2009], [Krebs et al. 2012].

Functional requirements. The following functionalities must be offered by any multi-tenant ESB:

- FR_1 *Tenant awareness*: An ESB must be able to manage and identify multiple tenants, i.e., tenant-based identification and hierarchical access control for tenants and their users must be supported.
- FR_2 *Tenant-based deployment and configuration*: The deployment and configuration of the ESB and the services available for a certain tenant should be managed in a transparent manner by the ESB.
- FR_3 *Tenant-specific interfaces*: A set of customizable interfaces must be provided, enabling administration and management of tenants and users, including both GUIs and Web services interfaces.
- FR_4 *Shared registries*: As the ESB solution will be embedded in a PaaS platform with other applications demanding similar information, the approach must come with a shared for other PaaS components registry of tenants/users and a shared registry of services.
- FR_5 *Backward compatibility*: The ESB solution should be able to used seamlessly and transparently by services and applications that are not multi-tenant aware.

Non-functional requirements. In addition to the required functionalities, multi-tenant ESBs should also respect the following properties:

- NFR_1 *Tenant Isolation*: Tenants must be isolated to prevent them from gaining access to other tenant's data (i.e., *data isolation*) and computing resources (i.e., *performance isolation*). Data isolation can be further decomposed into *communication isolation*, referring to keeping the message exchanges for each tenant separate, and *application isolation*, referring to preventing applications and services of one tenant from accessing data of another tenant's applications or services.

NFR$_2$ *Security*: The necessary authorization, authentication, integrity, and confidentiality mechanisms must consider and enforce tenant- and user-wide security policies when required.

NFR$_3$ *Reusability & extensibility*: The multi-tenancy enabling mechanisms and underlying concepts should not be solution-specific and depend on specific technologies to be implemented. ESB components should therefore be extensible when required and reusable by other components in the PaaS model (as for example in the case of the shared registries functional requirement).

Both functional and non-functional requirements are taken into consideration for the design of the architecture we present in Section 5.

4. ESB TENANT AWARENESS

In this section we investigate what is the impact of the requirements identified in the previous section to the core functionality of ESBs. As part of this discussion, we introduce the novel concept of *Tenant Context* to enable multi-tenant aware messaging and investigate its impact on the message processing inside the ESB.

4.1 Tenant Context

With respect to multi-tenancy, two different granularities of consumers have to be considered: *tenants* and *users*. Tenants are used to separate the consumers using multi-tenant aware messaging into disjoint groups. In the case of the 4CaaSt Taxi Scenario for example (Section 2), the different taxi companies using the multi-tenant aware ESB hosted on the 4CaaST platform are different tenants. Users enable the identification and differentiation between consumers potentially belonging to more than one tenant, and therefore introduce a finer level of granularity. In our example, the customers registered with and using the service of one or more taxi companies are the users of the corresponding tenants. In the following we present the requirements for multi-tenant aware communication that we have identified in collaboration with the industry partners in 4CaaSt.

Firstly, all required data to be provided within the tenant context for tenants, and their corresponding users, have to be represented in a structured format. Hence, the relation between the tenant and the user is included in the representation of our Tenant Context. Furthermore, the concept of tenant context must be supported by various communication protocols. Their message metadata must therefore be extended accordingly. Finally, as our goal is to provide a general concept to enable multi-tenant aware messaging, which is not limited to realization in one specific ESB solution and is applicable across the application stack, we also have to consider reusability (NFR$_3$). Nowadays for example, business processes are one established approach to implement business logic. Thus, multi-tenant aware messaging is also required when moving workflow engines executing these business processes to the Cloud. In order to enable reuse of the Tenant Context, the content has therefore to be extensible.

Figure 2 presents the schema of our Tenant Context. More specifically, a Tenant Context consists of a *mandatory* and an *optional* part. The mandatory part contains a *tenantID* and a *userID*. The Tenant Context is uniquely identified by the combination of *tenantID* and *userID*. In order to ensure the uniqueness of *tenantID* and *userID* we use Universally Unique Identifiers (UUIDs) [Network Working Group 2005] for their representation, see Figure 2. In addition to the mandatory part, the optional part may contain additional information such as the name of the tenant. For this purpose optional entries can be defined as key-value pairs, which makes the Tenant Context schema of Figure 2 extensible. Figure 3 provides an example for an instance of the Tenant Context schema including an optional entry.

As our proposal is independent from the technology or protocol used, it is required to support both messaging protocols that allow integration of structured information into the message metadata such as SOAP [World Wide Web Consortium (W3C) 2007], as well as ones that do not support these metadata by default, such as SMTP [The Internet Engineering Task Force

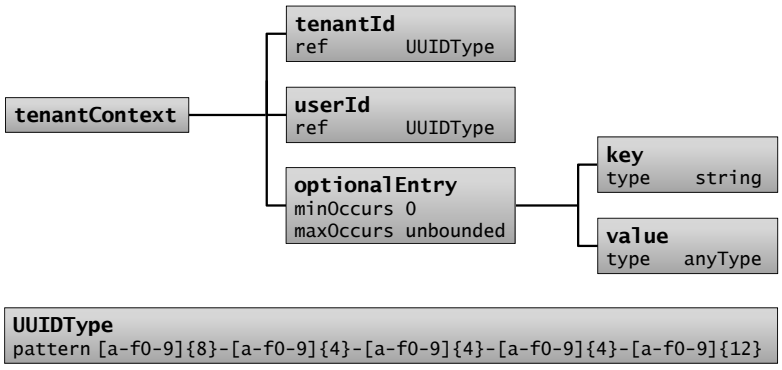


Figure 2: Schema of Tenant Context

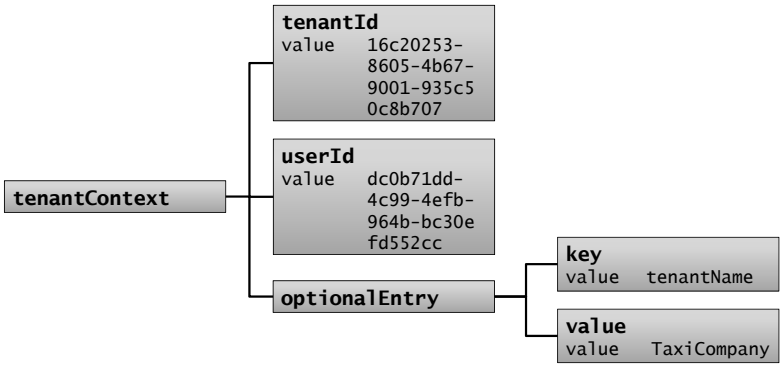


Figure 3: Example of Tenant Context Representation

(IETF) 2008] or the Java Message Service (JMS) [Oracle 2002] (FR_5). Thus, if the communication protocol does not support structured metadata we put the tenant context in XML format as string in the message metadata, in order to avoid negative impact on the performance, e.g., by looking up and retrieving the tenant information from a registry, when a message containing tenant information encoded as a key to be used for the lookup is received by the ESB.

4.2 Message Lifecycle

In this section, we focus on how we use the Tenant Context concept we defined above in order to support multi-tenant aware communication for ESB solutions (NFR_1). The proposed approach relies solely on the message processing cycle, which is common across many ESB solutions. It is therefore applicable to many different ESB implementations.

By its nature, an ESB solution has to support various communication protocols, e.g., with respect to message transformation and message routing. Any extensions of the ESB in order to enable multi-tenant aware message processing have therefore to be independent from the communication protocol. Moreover, apart from the integration of the Tenant Context into the messages exchanged with the ESB, the multi-tenant communication awareness mechanisms have to be transparent to the tenants and their users. In other words, the user must have the impression that he is the only one using and communicating with one concrete (multi-tenant aware) ESB instance. Furthermore, the ESB extended for multi-tenant aware communication has to be able to process also messages that do not contain a Tenant Context (FR_5).

Figure 4 shows a conceptual view of the message processing cycle of ESBs. While in the *Receive Message* phase, the ESB can receive a message in a specific communication protocol. As the message processing cycle differs only slightly depending on the communication protocol used,

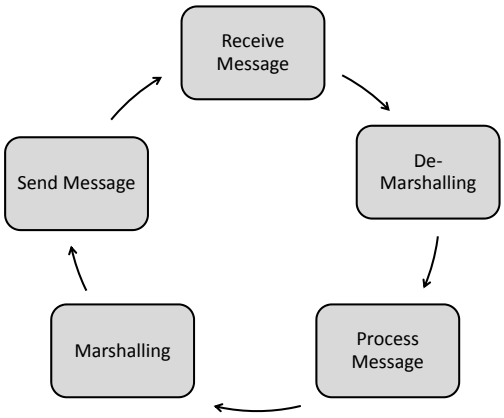


Figure 4: Message Processing Cycle of an ESB

we do not limit this discussion to a specific protocol but, we will emphasize the differences where necessary. In order to fulfill the requirement of transparent multi-tenancy from the point of view of the tenants and their users, we propose to include the Tenant Context introduced above in the metadata of the message. Thus, the payload of the message (independent of the protocol) is still exclusively used for carrying application data and therefore it is not necessary to modify existing application implementations using the ESB.

In the *De-Marshalling* phase, the communication protocol-specific format of the received message is mapped and transformed into an internal *Normalized Message Format (NMF)*, which is independent from the communication protocol. The NMF eases the ESB internal message processing. As we integrated the *Tenant Context* into the metadata of the communication protocol-dependent message, the NMF therefore has also to include the information provided by the Tenant Context after the mapping. As a result, the mapping and transformation to and from the NMF has to be extended accordingly. Therefore, it has to be considered whether the communication protocol used supports structured message metadata. In case structured metadata is supported the business logic can directly access the information in the Tenant Context for the mapping to the NMF. If the Tenant Context information are contained as one XML string in the message metadata when structured metadata is not supported, the *tenantID*, *userID*, and optional information have to be extracted before mapping and transforming it into the NMF.

After the message has been de-marshalled and all required information for multi-tenant aware processing of the message, e.g., routing, is available, the message is processed by the internal business logic of the ESB in the *Process Message* phase. For this purpose, the ESB business logic accesses the required information from the message available in the NMF format and includes the processing results in the message, if required. When the processing is finished, the message in NMF containing the results has to be mapped and transformed to the concrete format of the communication protocol defined for the specified recipient of the message (*Marshalling*). This protocol may or may not be the same as the one that the original message was sent to the ESB in. As in the case of de-marshalling, the marshalling function has also to be extended to consider the Tenant Context information. Finally, the resulting message is sent to its target in the *Send Message* phase.

For the purposes of this discussion we assume that both the sender and the receiver of the message are multi-tenant aware and know how to produce and process Tenant Context information as required. Furthermore, before each party interacts with the ESB in a multi-tenant way, it is assigned globally unique identifiers, e.g., during a registration phase. Finally, supporting backward compatibility is realized by deploying in parallel the default mechanisms for marshalling and de-marshalling of non-multi-tenant aware messages. In this case, the tenant-specific processing of messages is simply circumvented.

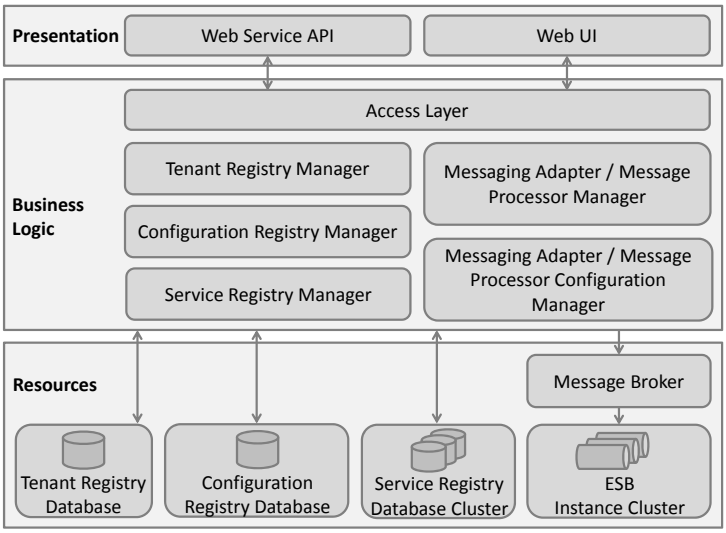


Figure 5: Overall ESB^{MT} Architecture

5. A MULTI-TENANT AWARE ESB ARCHITECTURE

Figure 5 provides an overview of our proposal for a generic multi-tenant aware ESB architecture (ESB^{MT}), which fulfills the requirements identified in the previous sections. More specifically, the three layer ESB^{MT} architecture consists of a *Presentation* layer, a *Business Logic* layer, and a *Resources* layer. In the following we present in a bottom-up fashion the components required for each layer of the architecture.

5.1 Resources layer

The Resources layer consists of an *ESB Instance Cluster* and a set of registries. The ESB Instance Cluster bundles together multiple *ESB Instances*. Each one of these instances perform the tasks usually associated with traditional ESB solutions, that is, message routing and transformation. In the simplest case, the ESB Instance Cluster may consist of only one (running) ESB Instance handling all tenants and users using an ESB^{MT} implementation. Since this however may create performance issues, in the ESB^{MT} architecture a clustering mechanism similar to the one provided for example by Apache ServiceMix[6] is recommended.

Each ESB Instance consists of three main components: a *Normalized Message Router*, *Messaging Adapters*, and *Message Processors* (Figure 6, zooming in on the bottom right part of Figure 5). Messaging Adapters are responsible for handling the communication with external services and applications (*External Service Providers* and *Consumers* in Figure 6) and converting to and from a normalized internal format for all incoming and outgoing messages, respectively. Message Processors provide additional ESB internal business logic related to message processing such as routing. The Normalized Message Router takes care of the internal routing between Messaging Adapters and/or Message Processors. These components appear under different names in many existing ESB solutions.

In order to enable multi-tenancy in any ESB solution, these components, and all other components in the ESB architecture, must be *multi-tenant aware*, i.e. able to operate with multiple tenants and users using the same instance of the ESB. For an ESB Instance in particular, this means that Adapters and Processors are able to handle messages containing tenant and user information, and process such messages accordingly in a multi-tenant manner (FR$_1$). For example, a message may be routed to different endpoints based on the tenant information contained in the

[6]Apache ServiceMix: `http://servicemix.apache.org`

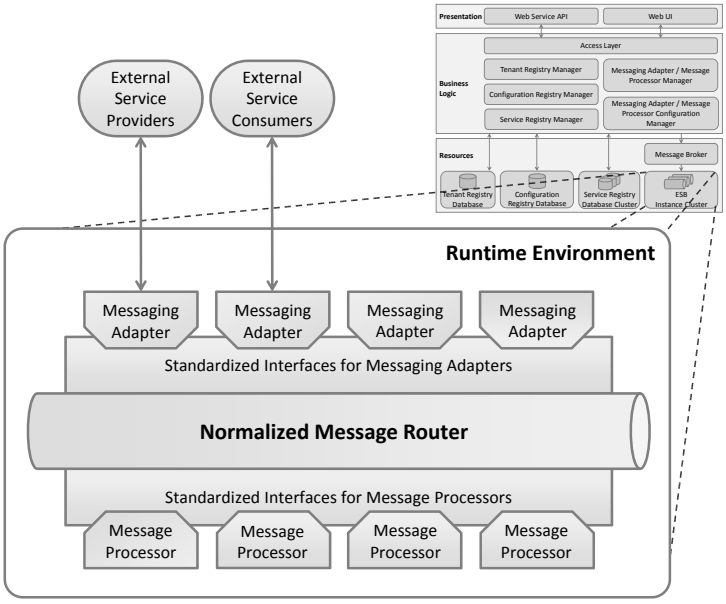


Figure 6: Architecture of an ESB Instance

message. Additionally, message flows of tenants and users when communicating with an ESB Instance, as well as message flows inside the ESB Instance, must be isolated from the message flows of other tenants and users (NFR_1). Furthermore, the Messaging Adapters and Message Processors have to be able to support tenant- and user-specific configurations when required (FR_2). This enables for example, that for each tenant a new endpoint for communication with the protocol specific adapter is created during configuration (NFR_1). The deployment/undeployment and configuration of Messaging Adapters and Message Processors in an ESB Instance is performed by means of a set of *standardized interfaces*. While these interfaces, and all other components of the ESB Instance, are multi-tenant aware, special care has to be taken to ensure backward compatibility (FR_5). This means that installation and configuration of non multi-tenant aware Adapters and Processors must still be possible. Processing and routing of non multi-tenant aware messages has to be performed normally, by still supporting the deployment of custom consumer and provider endpoints in a non multi-tenant manner.

Going back to Figure 5, within the Resource layer we also introduce three different types of registries. The *Service Registry* stores the services registered with the various ESB Instances, as well as the configuration of the Messaging Adapters and Message Processors installed in each ESB Instance in the ESB Instance Cluster (Figure 6) in a tenant-isolated manner [Chong et al. 2006] (FR_2). Currently we are focusing on the approach that each ESB instance of the ESB Instance Cluster has the same messaging adapters and message processors installed. As the messaging adapters and message processors are common, and in order to offer the possibility of horizontal scalability support [Pritchett 2008], a load balancer (not shown in Figure 5) must retrieve the required configurations from the Service Registry and deploy them when starting an additional ESB instance, e.g., to cover increased load. As we propose to share the Service Registry with other PaaS components, e.g., composition engines (FR_4), and for the sake of reusability (NFR_3), we recommend to realize the Service Registry as a database cluster to avoid performance bottlenecks.

The *Tenant Registry* stores a set of users for each tenant and the corresponding unique identifiers (FR_1). Additionally, each tenant and user may have associated properties such as tenant or user name represented as key-value pairs (NFR_3). The *Configuration Registry* stores all configuration data created by a tenant and the corresponding users, except from the service registrations and configurations stored in the Service Registry. The Configuration Registry stores for example

the configuration of ESB Instances (FR_2), the mapping of ESB Instances to tenants (FR_1), and the mapping of permissions to roles according to the role-based access control mechanisms offered by the Access Layer component in the next layer (NFR_1). When a tenant or user interacts with the management component of the multi-tenant ESB system, the data in more than one registry might have to be changed. Consequently, all operations and modifications on the underlying resources have to be handled as distributed transactions based on a two-phase commit protocol [Coulouris et al. 2005] so that a consistent state of all resources is ensured (NFR_1). As many JBI components from several JBI Container Instances in the cluster might participate in the distributed transaction and this might lead to performance bottlenecks, we recommend to decouple them from the distributed transactions using messaging with guaranteed delivery [Gregor Hohpe and Bobby Woolf 2003], e.g., a Message Broker (see Figure 5).

5.2 Business Logic layer

The Business Logic layer contains an *Access Layer* component, encapsulating the functionality that ensures tenant awareness and security (FR_1 and NFR_2, respectively). The Access Layer acts as a multi-tenancy enablement layer [Guo et al. 2007] based on role-based access control [Sandhu et al. 1996]. The tenants and their corresponding users have to be identified and authenticated once when the interaction with the ESB is initiated. Afterwards, the authorized access is managed by the Access Layer transparently. Prior to authentication and identification of tenants and users, the Access Layer component handles authorization by registering tenants and users and granting them access to ESB Instances (NFR_2). Therefore, in case of a multi-tenant aware interaction with the system, each tenant and user has to identify themselves by providing a unique *tenantID* and *userID* (FR_1).

In addition to the Access Layer component, the Business Logic layer also contains a set of *Managers* (Figure 5) encapsulating the functionality to manage and interact with the underlying components in the Resources layer. The *Tenant Registry*, *Configuration Registry*, and *Service Registry Managers* implement the business logic required to retrieve and store data in the corresponding registries in the Resources layer. The *Messaging Adapter/Message Processor Managers* deploy and undeploy Messaging Adapters and Message Processors in each ESB Instance in the Cluster, while the *Configuration Managers* take care of configuring them appropriately. Both managers are using the standardized interfaces provided by each ESB Instance for this purpose (Figure 6), as discussed in the Resources layer.

5.3 Presentation layer

The Presentation layer contains two components allowing the customization, administration, management, and interaction with an ESB^{MT} implementation: the *Web UI* and the *Web service API*. The Web UI offers a customizable interface for human and application interaction with the system, allowing for the administration and management of tenants and users (FR_3). The Web service API offers the same functionality as the Web UI, but also enables the integration and communication of external components and applications (NFR_3). For both interface mechanisms, security aspects such as integrity and confidentiality of incoming messages must be ensured (NFR_2) by, for example, using Web Services Security (WS-Security) for the Web Service API and Secure HTTP connections for the Web UI. A discussion on the particular mechanisms to be used for this purpose is outside of the scope of this work.

6. REALIZATION

In this section we provide a summary of the implementation of the architecture introduced in the previous section. A detailed discussion on the implementation can be found in [Strauch et al. 2012; Strauch et al. 2013]. A proof-of-concept realization of the ESB^{MT} architecture is provided as a deployment diagram in Figure 7. The realization is based on the open source ESB Apache ServiceMix version 4.3.0 (hereafter referred to simply as ServiceMix) and components and libraries being reused are marked in gray. ServiceMix is based on the OSGi Framework [OSGi Alliance

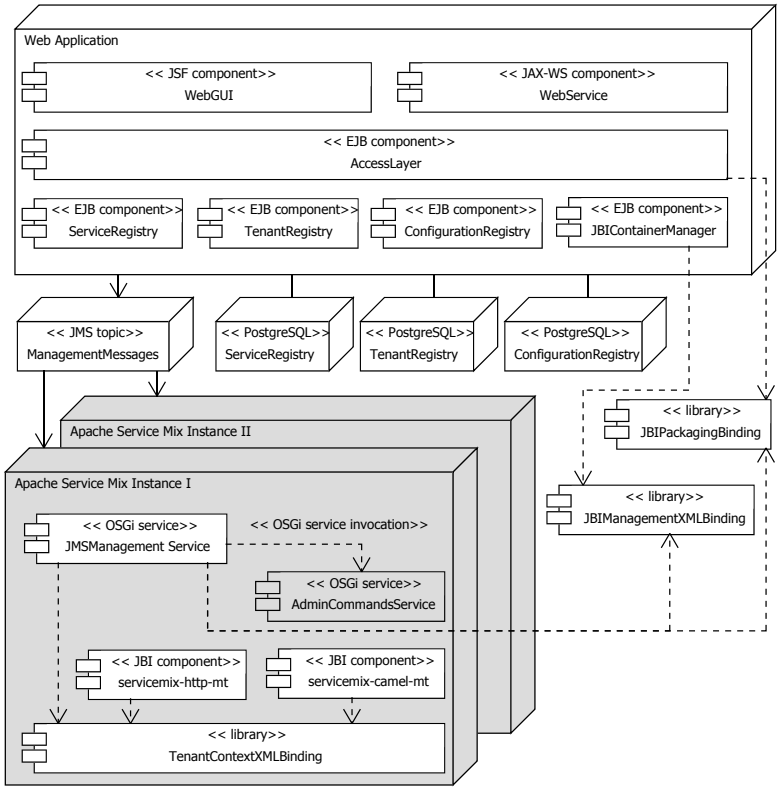


Figure 7: Deployment Diagram of Prototype Realization of ESB^{MT}

2011]. OSGi bundles realize the ESB functionality complying to the JBI specification [Java Community Process 2005].

ServiceMix is provided with several JBI components. Binding Components (BCs) are Messaging Adapters (in the sense of Figure 6) supporting various protocols such as SOAP over HTTP, FTP, or JMS. Service Engines (SEs) are JBI components providing additional business logic within the ESB. For example, the SE for Apache Camel [Apache Software Foundation 2011] enables usage of Enterprise Integration Patterns [Gregor Hohpe and Bobby Woolf 2003]. In this sense they serve as the Message Processors in our architecture.

The original ServiceMix BC for HTTP version 2011.01 and the original Apache Camel SE version 2011.01 were extended in our prototype in order to support multi-tenancy (see the *servicemix-http-mt* and *servicemix-caml-mt* components in Figure 7), as were the original BC for JMS version 2011.01 and the original BC for e-mail version 2011.01 (not shown in Figure 7). In addition, ServiceMix was extended by an OSGi-based management service (*JMSManagement Service* component), which listens to a JMS topic for incoming management messages sent by the Web Application (Figure 7). As the Web Application might modify more then one resources, all operations are handled within distributed transactions. The Web Application itself implements the Presentation and Business Logic layers of ESB^{MT} (Figure 5) and is running in the Java EE 5 application server JOnAS version 5.2.2, which can manage distributed transactions. As the management components of the underlying resources are implemented as EJB components, we use container-managed transaction demarcation, which allows the definition of transaction attributes for whole business methods, including all resource changes [Java Community Process 2006a].

As many JBI containers deployed on several ServiceMix instances can be involved in the distributed transactions, and the distributed transaction can contain many JBI containers, this might lead to a performance bottleneck. Hence, the Web application subdivides the transac-

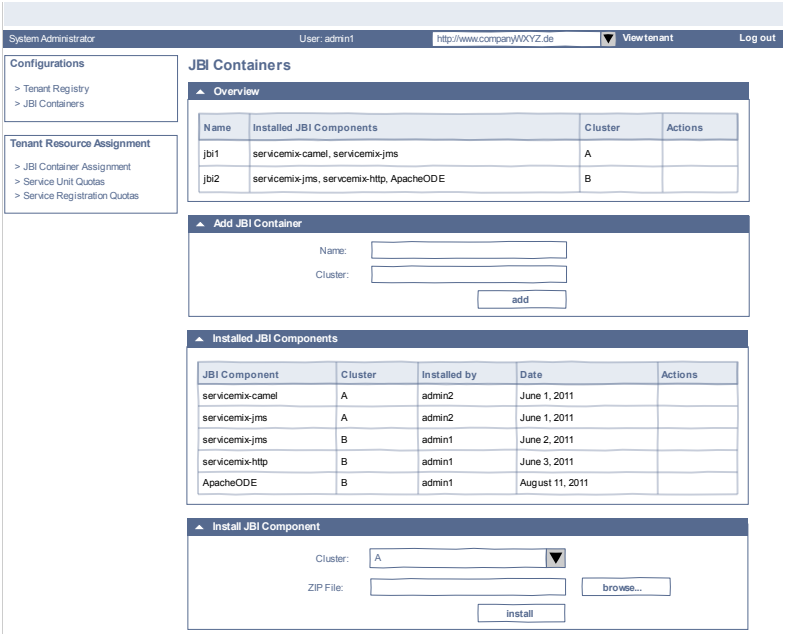


Figure 8: WebGUI: Content Panel to add JBI Containers and to Install JBI Components

tion to the JBI containers using messaging with guaranteed delivery [Gregor Hohpe and Bobby Woolf 2003]. If the message is persistently stored in the message topic, the distributed transaction will commit. Afterwards, each JBI container acts as selective, transactional, and durable subscriber. A transaction between each corresponding JBI container and the topic ensures that the message is successfully processed before being deleted from the topic. For JMS messaging we use Apache ActiveMQ version 5.3.1[7]. The *ServiceRegistry*, *TenantRegistry*, and *ConfigurationRegistry* components are realized based on PostgreSQL version 9.1.1[8]. The *AccessLayer* of the Web Application applies the Session Façade pattern [Marinescu 2002], which is a design pattern for EJB projects encapsulating business logic in order to minimize the number of calls to the EJB container. The Web Service API of the Web Application is based on the Java API for XML-Based Web Services version 2.0 [Java Community Process 2006c]. The *WebGUI* has been specified and designed based on JavaServer Faces version 1.2 [Java Community Process 2006b], but the implementation is still ongoing. A screenshot of the WebGUI is shown in Figure 8. This content panel is used by the system administrator to add JBI Containers to the JBI Container Instance Cluster and to install JBI Components to concrete JBI Containers.

The evaluation of the realization of the architecture within the context of 4CaaSt is based on the Taxi Scenario introduced in Section 2. For this purpose, we implemented the motivating scenario discussed in Section 2 for two taxi companies (tenants). Both companies are using the same taxi management application hosted by the 4CaaSt platform. The application provides an interface for their registered customers and drivers (users) that is customizable by the companies on demand. Using this interface, the customer can request a taxi by providing the necessary information through, e.g., a smartphone device.

The customer request is then forwarded to the two nearest drivers and pops up in their GUIs (as shown in Figure 9). The first driver that confirms the request is assigned to the customer. The driver further has the option to get routing information to the designated pick up location through an integration with Google Maps Web Services. Based on the distance between the

[7]Apache ActiveMQ: `http://activemq.apache.org`
[8]PostgreSQL: `http://postgresql.org`

Figure 9: Screenshot of the GUI Used by the Taxi Drivers of one Company

driver and the pick up location, the customer receives a notification containing the estimated pick up time. All messaging between services in the scenario, as shown in Figure 1b, is handled by our realization of the ESB^{MT} architecture discussed above. A video introducing the architecture of the taxi application and demonstrating the taxi application in action is available at `http://tiny.cc/taxidemo`.

7. EVALUATION

As discussed in the opening of this paper, multi-tenancy of Cloud solutions can be decomposed into two perspectives: *performance*, as experienced by the ESB users, and *resource utilization*, of primary concern to the ESB provider. These two perspectives are the focus of our evaluation of the ESB^{MT} implementation. In order to provide a baseline against which we evaluate our proposal we use the original, non multi-tenant aware ServiceMix version that we based our implementation on. The following sections discuss the method, workload, experimental setup, and results towards this goal.

7.1 Method

Our investigation showed that there is no commonly agreed benchmark for ESBs, see for example [Walraven et al. 2011]. For this reason we chose to use the industrial ESB benchmark by AdroitLogic [AdroitLogic Private Ltd. 2013] as a basis. This benchmark has been in development since 2007, and a number of open source ESB solutions have been evaluated in six rounds, with the latest round results coming out in August 2012. All information about the benchmark, as well as the results of each evaluation round are publicly available at [AdroitLogic Private Ltd. 2013].

We had to deal with two major obstacles in adopting this benchmark. Firstly, ServiceMix version 4.3.0 failed to pass smoke testing by AdroitLogic for one of the benchmarking scenarios and as a result ServiceMix has not been included in their evaluation. By using one of other benchmarking scenarios, however, we were able to execute the benchmark normally. Secondly, the benchmark did not support multi-tenant aware messaging and concurrent load generation

between more than one endpoint. Thus, we had to adapt the AdroitLogic Benchmark Driver accordingly, as described in the following sections.

7.2 Workload

For purposes of evaluation we derived three test scenarios from the Direct Proxy Service scenario in AdroitLogic's benchmark [AdroitLogic Private Ltd. 2013]. The Direct Proxy Service scenario demonstrates the ability of an ESB to act as a virtualization layer for back-end Web services, operating as a proxy between a client (the AdroitLogic Benchmark Driver) and a simple Echo Web service on the provider side. Starting from this point, we defined the following scenarios:

(1) a non multi-tenant ESB deployment on one Virtual Machine (VM) image, acting as the baseline for comparisons;

(2) the same non multi-tenant ESB deployed across 2 VMs, in order to simulate the effect of *horizontal scaling* [Vaquero et al. 2011], i.e. adding another application VM when more computational resources are required; and,

(3) our ESB^{MT} implementation deployed on 1 VM.

Following the test parameters set by the benchmark we configured in each ESB deployment with $1, 2, 4$, and 10 endpoints per scenario. The message size used by the Benchmark Driver is fixed to 1KB, composed out of random characters. The original Benchmark Driver steadily increases the number of concurrent users of the ESB ($2000, 4000, 8000, 16000, 64000$, and 128.000) and sends a fixed number of requests per user for each round of the benchmark. Since in our case we have multiple endpoints and tenants, we distribute these requests between the different endpoints (or tenants in the third scenario) and we send them concurrently across each endpoint. In the first round of the benchmark for example, and for 4 endpoints/tenants, we send $2000/4 = 500$ requests per endpoint or tenant for a total of 2000 requests; in the next round we send $4000/4 = 1000$ requests, and so on. Each endpoint or tenant receives in any case 10K messages as a warm-up before any measurements.

7.3 Experimental Setup

Figure 10 provides an overview of the experimental setup realizing our adaptation of the Direct Proxy Service Scenario including message flow and measurement points. The test cases were run on Flexiscale[9] and three Virtual Machines: VM0 (6GB RAM, 3 CPUs), VM1 (4GB RAM, 2 CPUs), and VM2 (4GB RAM, 2 CPUs). All three VMs run Ubuntu 10.04 Linux OS and every CPU is an AMD Opteron Processor with 2GHz and 512KB cache.

In VM0, an Apache Tomcat 7.0.23 instance was deployed with the Echo Web service, the adapted AdroitLogic Benchmark Driver, and Wireshark 1.2.7 for monitoring HTTP requests and responses. In VM1 and VM2, the ESB^{MT} implementation is deployed, which required also the deployment of PostgreSQL 9.1.1 database (for the registries), and Jonas 5.2.2 server for the Web application implementing the Business Logic layer. The endpoints deployed in ServiceMix are using HTTP-SOAP communication protocol, see Figure 10.

The total time in receiving the receipt acknowledgment by the Echo Web service for each message was measured at the AdroitLogic Benchmark Driver, in order to calculate the response time and the throughput. The CPU utilization for the ServiceMix process and the Java Virtual Machine (JVM) heap memory use was measured directly in VM1 and VM2. The maximum JVM heap memory size was set to 512MB before the warm-up phase for both VM1 and VM2.

7.4 Experimental Results

In this section we provide the measurement results for performance and utilization.

[9]Flexiant Flexiscale: `http://www.flexiscale.com/`

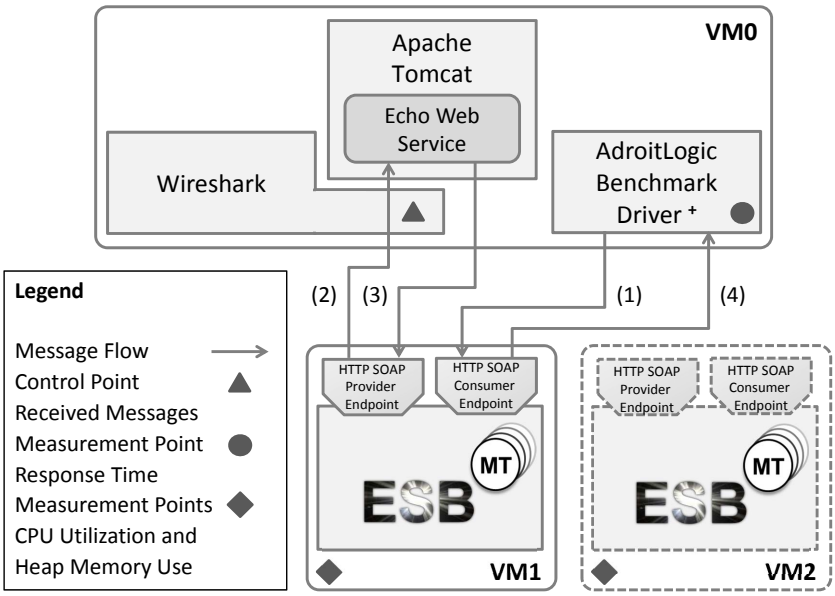
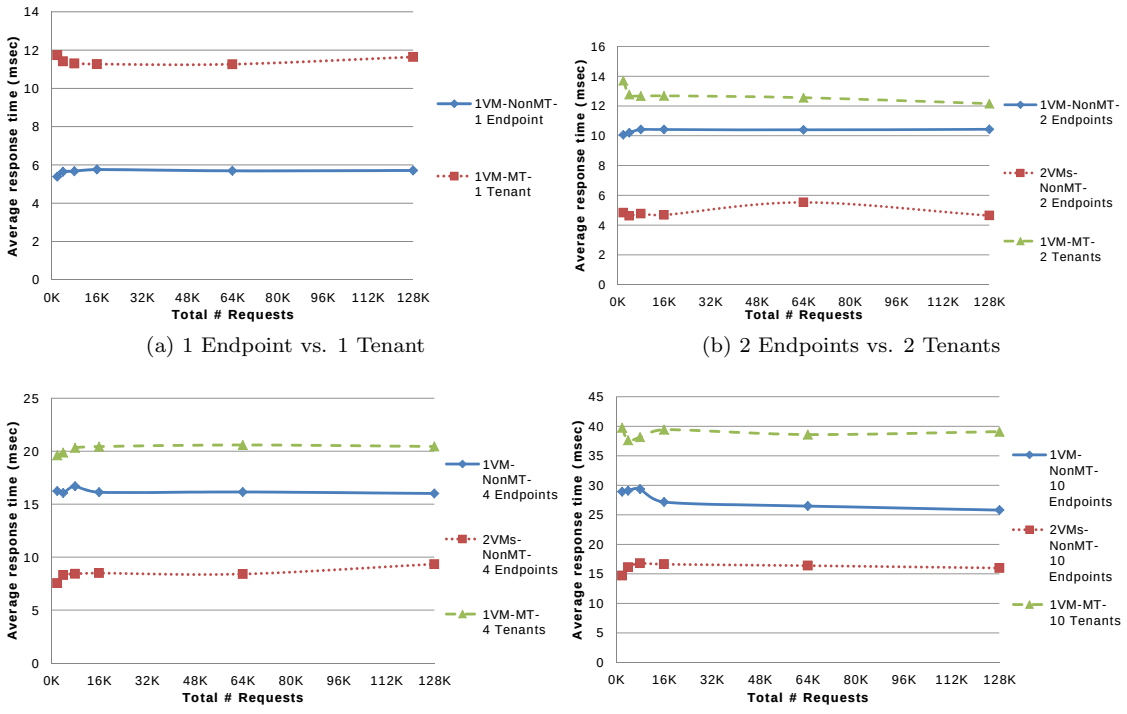


Figure 10: Overview of the Experimental Setup



(a) 1 Endpoint vs. 1 Tenant

(b) 2 Endpoints vs. 2 Tenants

(c) 4 Endpoints vs. 4 Tenants

(d) 10 Endpoints vs. 10 Tenants

Figure 11: Average response time (latency) for 1KB size messages

7.4.1 *Performance.* Figure 11 summarizes and presents the latency recorded for all scenarios and work loads. The baseline for the presentation is the non multi-tenant aware implementation of the ESB on one VM (1VM-NonMT-* Endpoints in Fig. 11). As shown in the figure, our proposed multi-tenant aware implementation of the ESB exhibits a performance decline of around

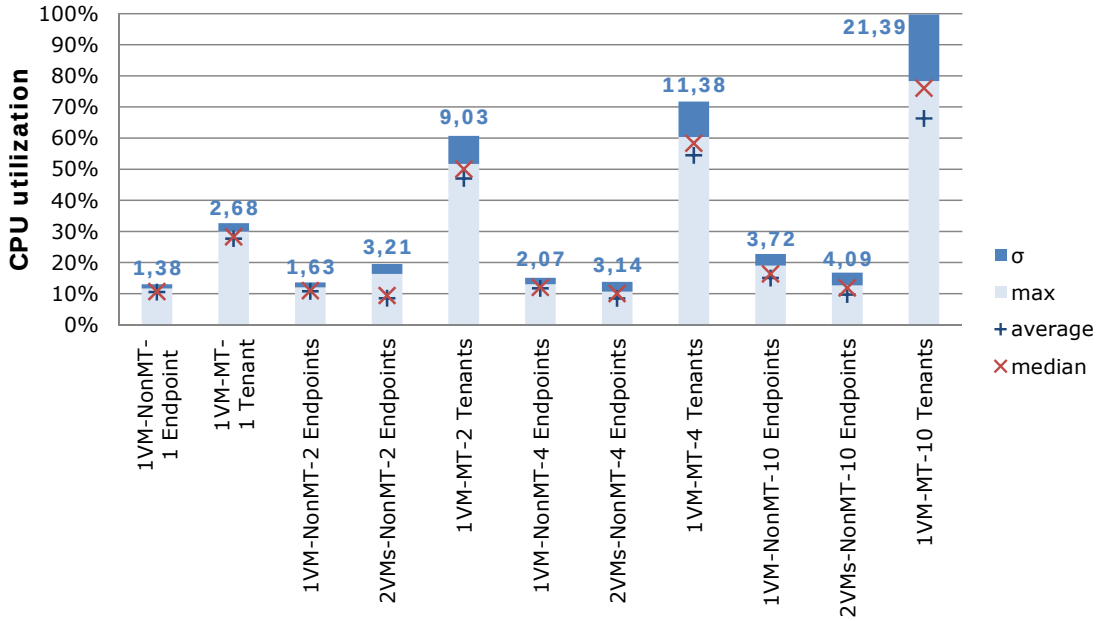


Figure 12: CPU Utilization

30% across the different cases when comparing the same number of endpoints and tenants in the other scenarios. The same load across 2 tenants instead of 2 endpoints, for example, results in 23, 57% more latency on average (Fig. 11b), 24, 68% more for 4 tenants/endpoints (Fig. 11c) and 39, 44% increase for 10 tenants/endpoints (Fig. 11d).

When comparing 1 tenant against 1 endpoint (Fig. 11a) an 100% reduction of response time is observed, showing that the performance decrease is actually ameliorated when more tenants/endpoints are added. Also of particular interest is the fact that adding a VM and distributing the requests between those VMs — essentially reducing the number of active endpoints by half — improves response time by 50% *only for 2 endpoints* (53, 07%), degrading from there with the number of endpoints (48, 10% for 4, and 42% for 10).

7.4.2 *Utilization.* The measurements for CPU and memory utilization for the same loads are summarized by Figures 12 and 13. The reported CPU utilization is normalized over the number of CPUs of the VMs containing the ESB implementation (Fig. 10). Memory utilization is presented as a percentage of the maximum heap size for the JVM containing the ESB (approximately 455MB). In both cases, the figures for the 2VMs scenario are calculated as the average of the utilization of each VM.

As shown in Figure 12, the overall utilization of system resources increases with the introduction of multi-tenancy. The additional computation required for processing the tenant and user information, and routing the messages accordingly, translates into more than 300% increase in CPU utilization compared to the baseline, non multi-tenant aware implementation. With respect to the same scenario, standard deviation σ is increasing with the number of tenants introduced. However, given the proximity of the average and median values to the maximum CPU utilization in all cases, this can be interpreted as a distribution heavily concentrated towards the maximum utilization. With respect to memory utilization, Figure 13 shows also an overall increase of around 100% across the three cases of ESB^{MT} (2, 4 and 10 tenants). The low standard deviation, and the small differences between average and median values show that memory consumption is relatively steady over all work loads. Similar behavior is observed also for the other two (non multi-tenant) scenarios.

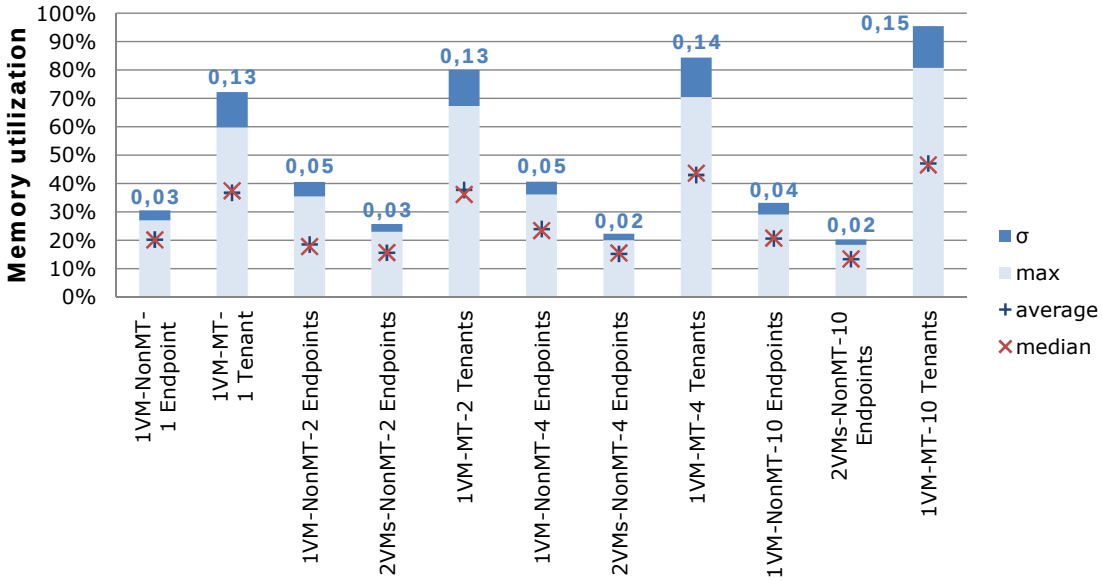


Figure 13: Memory Utilization

8. RELATED WORK

Existing approaches on enabling multi-tenancy for middleware typically focus on different types of isolation in multi-tenant applications for the SaaS delivery model, see for example [Guo et al. 2007]. As discussed also in [Walraven et al. 2011] however, only few PaaS solutions offer multi-tenancy awareness allowing for the development of multi-tenant applications on top of them. The work of Walraven et al. [Walraven et al. 2011] follows a similar approach to ours; our work however proposes a more generic approach built around any ESB technology that complies with the JBI specification, and does not require the implementation of a dedicated support layer for these purposes.

Focusing on ESB solutions, in [4CaaSt Consortium 2011] we surveyed a number of existing ESB solutions and evaluated their multi-tenancy readiness. Our investigation showed that the surveyed solutions in general lack in support of multi-tenancy. Even in the case of products like IBM WebSphere ESB[10] and WSO2 ESB[11] where multi-tenancy is part of their offerings, multi-tenancy support is implemented either based on proprietary technologies like the Tivoli Access Manager (in the former case), or by mitigating the tenant communication and administration on the level of the message container (Apache Axis2 [12] in the latter case). In either case, the used method can not be applied to other ESB solutions and as a result no direct comparison of the applied multi-tenancy enabling mechanisms can be performed. The presented approach differs from existing approaches by integrating multi-tenancy independently from the implementation specifics of the ESB.

The different benchmarks and metrics developed in the domain of Cloud computing in the recent years focus on a particular type of Cloud services such as databases [Cooper et al. 2010], on Cloud-related features such as elasticity [Brebner 2012] and performance isolation [Krebs et al. 2012], or on virtualization technology [Makhija et al. 2006]. To the extent of our knowledge, there is no commonly agreed approach and benchmark for the evaluation of the performance of multi-tenant PaaS middleware components such as an ESB. AdroitLogic completed in August 2012 [AdroitLogic Private Ltd. 2013] the 6th round of public ESB performance benchmark-

[10]IBM WebSphere ESB: `http://tiny.cc/IBMWebSphereESB`
[11]WSO2 ESB: `http://wso2.com/products/enterprise-service-bus/`
[12]Apache Axis2: `http://axis.apache.org/axis2/java/core/`

ing since June 2007. This round included eight free and open source ESBs including Apache ServiceMix version 4.3.0 — for which however they were not able to execute for all defined scenarios. Our ESB performance evaluation approach reuses, but adapts and extends, the AdroitLogic Benchmark Driver and our test scenarios are derived from the Direct Proxy scenario, but extended in order to consider multi-tenancy.

9. OUTLOOK AND FUTURE WORK

Multi-tenancy allows Cloud providers to serve multiple consumers from a single system instance, reducing costs and increasing their return of investment by maximizing system utilization. Realizing multi-tenancy requires making both the application and the underlying middleware components multi-tenant aware. Making therefore ESB solutions, a critical piece of middleware for the service-oriented enterprise environment, multi-tenant aware is essential. Multi-tenancy awareness manifests as the ability to manage and identify multiple tenants (organizational domains) and their users, and allow their applications to interact seamlessly with the ESB. Allowing multiple tenants however to use the same ESB instance requires to ensure that they are isolated from each other. There is therefore a trade-off between the benefits for the ESB provider in terms of utilization and their impact on the performance of applications using the ESB that needs to be investigated.

Toward this goal, in the previous sections we present the realization of our proposal for a generic ESB architecture that enables multi-tenancy awareness based on the JBI specification. We first provide the necessary background and explain our proposed architecture across three layers based on previous work. We then discuss in detail the realization of this architecture by extending the open source Apache ServiceMix ESB solution. In the next step we adapt the ESB benchmark developed by AdroitLogic to accommodate multi-tenancy and we use it to measure the performance and resource utilization of our ESB solution.

Our analysis shows that our current, not optimized in any manner implementation of a multi-tenant aware ESB solution succeeds in increasing the CPU utilization while having a relatively small impact on the memory footprint. In this sense it succeeds as far as the ESB provider is concerned. On the other hand, there is a significant reduction in performance experienced by the ESB consumers which needs to be ameliorated by re-engineering and fine-tuning our implementation accordingly. Techniques for performance isolation have also to be brought into play [Krebs et al. 2012]. In the scope of this work, this is a direction that we want to investigate in the future. We also plan to take advantage of using the JBI specification as the basis of our architectural framework and apply the same techniques and architectural solutions to other ESB solutions, as well as non-ESB solutions, like for example application servers, that comply with this specification.

ACKNOWLEDGEMENTS

The research leading to these results has received funding from projects 4CaaSt (grant agreement no. 258862) and Allow Ensembles (grant agreement no. 600792) part of the European Union's Seventh Framework Programme (FP7/2007-2013). The authors would like to thank Dominik Muhler for his valuable contribution to the development of ESB^{MT}.

REFERENCES

4CaaSt Consortium. 2011. Immigrant PaaS Technologies: Scientific and Technical Report D7.1.1. Deliverable. `http://www.4caast.eu/wp-content/uploads/2011/09/4CaaSt_D7.1.1_Scientific_and_Technical_Report.pdf`.

AdroitLogic Private Ltd. 2013. Performance Framework and ESB Performance Benchmarking. `http://www.esbperformance.org`.

Alves, A. et al. 2007. Web Services Business Process Execution Language Version 2.0. Committee Specification.

Apache Software Foundation. 2011. *Apache Camel User Guide 2.7.0.*

Azeez, A., Perera, S., Gamage, D., Linton, R., Siriwardana, P., Leelaratne, D., Weerawarana, S., and Fremantle, P. 2010. Multi-tenant SOA Middleware for Cloud Computing. In *Proceedings of IEEE CLOUD'10*. 458–465.

Brebner, P. 2012. Is your Cloud Elastic Enough?: Performance Modelling the Elasticity of Infrastructure as a Service (IaaS) Cloud Applications. In *Proceedings of ICPE'12*. 263–266.

Chappell, D. A. 2004. *Enterprise Service Bus*. O'Reilly Media, Inc.

Chong, F., Carraro, G., and Wolter, R. 2006. Multi-tenant data architecture. MSDN. `http://msdn.microsoft.com/en-us/library/aa479086.aspx`.

Cooper, B. F. et al. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of ACM SoCC'10*. ACM, 143–154.

Coulouris, G., Dollimore, J., and Kindberg, T. 2005. *Distributed Systems: Concepts and Design*. Addison Wesley.

Gregor Hohpe and Bobby Woolf. 2003. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley Professional.

Guo, C., Sun, W., Huang, Y., Wang, Z., and Gao, B. 2007. A Framework for Native Multi-Tenancy Application Development and Management. In *Proceedings of CEC/EEE'07*. IEEE, 551–558.

Java Community Process. 2005. Java Business Integration (JBI) 1.0, Final Release. JSR-208.

Java Community Process. 2006a. Enterprise JavaBeans (EJB) 3.0, Final Release. JSR-220.

Java Community Process. 2006b. JavaServer Faces Specification (JSF) 1.2, Final Release. JSR-252.

Java Community Process. 2006c. The Java API for XML-Based Web Services (JAX-WS) 2.0, Final Release. JSR-224.

Josuttis, N. 2007. *SOA in Practice*. O'Reilly Media, Inc.

Krebs, R., Momm, C., and Konev, S. 2012. Architectural Concerns in Multi-Tenant SaaS Applications. In *Proceedings of CLOSER'12*. SciTePress, 426–431.

Krebs, R., Momm, C., and Kounev, S. 2012. Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments. In *Proceedings of ACM QoSA'12*. ACM, 91–100.

Makhija, V. et al. 2006. VMmark: A Scalable Benchmark for Virtualized Systems. Tech. Rep. VMware-TR-2006-002, VMware, Inc.

Marinescu, F. 2002. *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc.

Mell, P. and Grance, T. 2011. The NIST Definition of Cloud Computing. `http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909616`.

Mietzner, R., Unger, T., Titze, R., and Leymann, F. 2009. Combining Different Multi-Tenancy Patterns in Service-Oriented Applications. In *Proceedings of EDOC'09*. IEEE, 131–140.

Network Working Group. 2005. A Universally Unique IDentifier (UUID) URN Namespace.

Oracle. 2002. Java Message Service (JMS) Version 1.1, Specification.

OSGi Alliance. 2011. OSGi Service Platform: Core Specification Version 4.3.

Pritchett, D. 2008. BASE: An ACID Alternative. *Queue 6,* 3, 48–55.

Sandhu, R. S., Coyne, E. J., Feinstein, H. L., and Youman, C. E. 1996. Role-based Access Control Models. *Computer 29*, 38–47.

Strauch, S., Andrikopoulos, V., Gómez Sáez, S., and Leymann, F. 2013. Implementation and Evaluation of a Multi-tenant Open-Source ESB. In *Proceedings of ESOCC 2013*. Lecture Notes in Computer Science, vol. 8135. Springer, 79–93.

Strauch, S., Andrikopoulos, V., Gómez Sáez, S., Leymann, F., and Muhler, D. 2012. Enabling Tenant-Aware Administration and Management for JBI Environments. In *Proceedings of SOCA'12*. IEEE Computer Society Conference Publishing Services, 206–213.

Strauch, S., Andrikopoulos, V., Leymann, F., and Muhler, D. 2012. ESB^{MT}: Enabling Multi-Tenancy in Enterprise Service Buses. In *Proceedings of CloudCom'12*. IEEE, 456–463.

The Internet Engineering Task Force (IETF). 2008. RFC 5321 - Simple Mail Transfer Protocol.

Vaquero, L., Rodero-Merino, L., and Buyya, R. 2011. Dynamically Scaling Applications in the Cloud. *ACM SIGCOMM Computer Communication Review 41,* 1, 45–52.

Walraven, S., Truyen, E., and Joosen, W. 2011. A Middleware Layer for Flexible and Cost-Efficient Multi-Tenant Applications. In *Proceedings of Middleware'11*. Springer, 370–389.

World Wide Web Consortium (W3C). 2007. SOAP Version 1.2. W3C Recommendation (Second Edition).

Steve Strauch works as a research associate and PhD student at the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart since April 2008. His research interests are data migration, data hosting, as well as data security and privacy in the area of Cloud computing, with an emphasis on their architectural aspects. Steve has contributed to the European projects COMPAS (`http://www.compas-ict.eu`), 4CaaSt (`http://www.4caast.eu`), and ALLOW Ensembles (`http://www.allow-ensembles.eu`).

Dr. Vasilios Andrikopoulos is a senior researcher at IAAS, University of Stuttgart. His research is in the areas of services science, cloud computing and infrastructures, and software engineering with an emphasis on evolution and adaptation. He received his PhD cum laude in 2010 from Tilburg University, the Netherlands, where he was also a member of the European Research Institute in Service Science (ERISS). He has experience in research and teaching Database Systems and Management, Software Modeling and Programming, Business Process Management and Integration, and Service Engineering. He has participated in a number of EU projects, including NoE S-Cube and 4CaaSt.

Santiago received his Dipl.-Inf. degree from the University of Stuttgart in 2013. Currently, he is a PhD student and research associate in the Institute of Architecture of Application Systems at the University of Stuttgart. His experience and research interests include Service Oriented Architecture and EAI frameworks, focusing on distributed data and workload analysis and management. Santiago has contributed to the ESB-MT project (`http://www.iaas.uni-stuttgart.de/esbmt/`) and currently contributes to the European project ALLOW Ensembles (`http://www.allow-ensembles.eu`).

Frank Leymann is a full professor of computer science and director of the Institute of Architecture of Application Systems (IAAS) at the University of Stuttgart, Germany. His research interests include service-oriented architectures and associated middleware, workflow- and business process management, cloud computing and associated systems management aspects, and patterns. The projects he is working on are funded by the European Union, the German Government, or directly by industry partners. Frank is co-author of about 300 peer-reviewed papers, more than 40 patents, and several industry standards (e.g. BPEL, BPMN, TOSCA). He is invited expert to consult the European Commission in the area of Cloud Computing. Before accepting the professor position at University of Stuttgart he worked for two decades as an IBM Distinguished Engineer where he was member of a small team that was in charge of the architecture of IBMs complete middleware stack.