Institute of Architecture of Application Systems

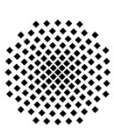
Implementation and Evaluation of a Multi-tenant Open-Source ESB

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:

```
@inproceedings{StrauchAGSL,
  author    = {Steve Strauch and Vasilios Andrikopoulos and Santiago Gómez Saéz
               and Frank Leymann},
  title     = {Implementation and Evaluation of a Multi-tenant Open-Source
               ESB},
  booktitle = {Proceedings of the European Conference on Service-Oriented and
               Cloud Computing, ESOCC'13, 2013},
  year      = {2013},
  pages     = {79--93},
  doi       = {10.1007/978-3-642-40651-5_7},
  series    = {Lecture Notes in Computer Science (LNCS)},
  volume    = {8135},
  publisher = {Springer Berlin Heidelberg}
}
```

University of Stuttgart
Germany

Implementation and Evaluation of a Multi-tenant Open-Source ESB

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

Institute of Architecture of Application Systems, University of Stuttgart, Germany
Universitätsstraße 38, 70569 Stuttgart, Germany
lastname@iaas.uni-stuttgart.de

Abstract Offering applications as a service in the Cloud builds on the notion of application multi-tenancy. Multi-tenancy, the sharing of application instances and their underlying resources between users from different organizational domains, allows service providers to maximize resource utilization and reduce servicing costs per user. Realizing application multi-tenancy however requires suitable enabling mechanisms offered by their supporting middleware. Furthermore, the middleware itself can be multi-tenant in a similar fashion. In this work we focus on enabling multi-tenancy for one of the most important components in service-oriented middleware, the Enterprise Service Bus (ESB). In particular, we discuss the prototype realization of a multi-tenant aware ESB, using an open source solution as the basis. We then evaluate the performance of our proposed solution by an ESB-specific benchmark that we extended for multi-tenancy purposes.

Keywords: Multi-tenancy, Enterprise Service Bus (ESB), ESB benchmarking, JBI specification, Platform as a Service.

1 Introduction

The Enterprise Service Bus (ESB) technology addresses the fundamental need for application integration by acting as the messaging hub between applications. As such, in the last years it has become ubiquitous in service-oriented enterprise computing environments. ESBs control the message handling during service invocations and are at the core of each Service-Oriented Architecture (SOA) [12]. Given the fact that the Cloud computing paradigm [16] is discussed in terms of the creation, delivery and consumption of services [5], it is therefore essential to investigate into how the ESB technology can be used efficiently in a Cloud-oriented environment.

For this purpose, in our previous work we focused on investigating how to make ESBs *multi-tenant aware* [21]. In this context, making an ESB multi-tenant aware means that the ESB is able to manage and identify multiple tenants (groups like companies, organizations or departments sharing the application) and their users, providing tenant-based identification and hierarchical access

K.-K. Lau, W. Lamersdorf, and E. Pimentel (Eds.): ESOCC 2013, LNCS 8135, pp. 79–93, 2013.

control to them. In other words, the ESB should provide the appropriate mechanisms that allow (multi-)tenant applications to seamlessly interact with it while sharing one (logical) instance of the ESB. Given the role of the ESB middleware in the technological stack, there are two fundamental aspects of multi-tenancy awareness: communication (i.e. supporting message exchanges isolated per tenant and application), and administration and management (i.e. allowing each tenant to configure and manage individually their communication endpoints at the ESB).

Multi-tenancy has been previously defined in different ways in the literature for SOA and middleware, see for example [10], [17], [14], [23]. Such definitions however do not address the whole technological stack behind the different Cloud service models as defined in [16] (i. e. IaaS — Infrastructure as a Service, PaaS — Platform as a Service, SaaS — Software as a Service). For these reasons in [21] we define 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 users.*

Multi-tenancy is one of the key enablers that allow Cloud computing solutions to serve multiple customers from a single system instance (the other being virtualization of the application stack). 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 Cloud service consumer side, the fundamental assumption in using multi-tenant applications is that tenants are well isolated from each other, both in terms of data and computational resources. This ensures that the operation of one tenant does not have any discernible effect on the efficacy and efficiency of the operation of the other tenants. Ensuring tenant isolation in Cloud solutions however is a notoriously difficult problem and remains largely an open research question, see for example [10] and [14].

Towards this direction, in this work we investigate the performance of a multi-tenant aware ESB implementation from both perspectives, i.e. service providers and consumers. For this purpose we first present in detail the realization of the ESB^{MT} architectural framework [21] based on the Java Business Integration (JBI) specification [11] into a multi-tenant aware ESB solution. We then evaluate the performance of our solution in terms of response time as experienced by the service consumer (i.e. the application tenant), and CPU and memory utilization (that are of particular interest to the service provider). For this purpose we extend and modify an industry benchmark for ESBs in order to make it suitable for driving multi-tenant, in addition to non multi-tenant, interactions. Our contributions can be summarized as follows:

- A detailed presentation of the realization of a multi-tenant aware ESB solution implementing the ESB^{MT} framework [21] by extending the open source Apache ServiceMix solution [3].
- The creation of an ESB benchmark which allows evaluating the performance and utilization of multi-tenant aware solutions by extending an existing benchmark [2].

– An analysis of the performance and utilization characteristics of our proposed implementation compared against a baseline, non multi-tenant aware ESB solution (Apache ServiceMix).

The remaining of the paper is structured as follows: Section 2 briefly summarizes the JBI specification and the ESB^{MT} framework which is based on JBI. Section 3 discusses the realization of this framework using Apache ServiceMix as a proof-of-concept implementation for our proposal, together with the technologies involved. Section 4 introduces the benchmarking tool that we developed as part of this work, discusses the benchmarking environment, and presents the results of this evaluation. Section 5 discusses the key findings of our evaluation. The paper closes with Section 6 and Section 7 summarizing related work, and concluding with some future work, respectively.

2 Background

Java Business Integration Environment. The Java Business Integration (JBI) specification defines a standards-based environment for integration solutions by specifying the interaction of JBI components installed in a JBI container [11]. A number of middleware technologies like ESBs (e.g. Open ESB[1], Apache ServiceMix [3]) and application servers (like GlassFish[2]) implement the JBI specification. By basing our approach on the JBI specification we therefore ensure that we produce a generic and reusable solution that can be replicated across different ESB solutions (and other technologies that implement the JBI specification).

Figure 1 provides an overview of the JBI environment, based on [11]. JBI-compliant components are deployed in the container and interact through a *Normalized Message Router* (NMR). The components consume or provide services described in WSDL 2.0[3]. Two types of JBI components are specified: *Binding Components* (BCs), providing connectivity to external services and mediating between external protocols and the NMR, and *Service Engines* (SEs), offering business logic and message transformation services inside the JBI container. Configuration of the components is achieved by a management framework based on Java Management Extensions (JMX). The framework allows the installation of JBI components, deployment and configuration of service artifacts called *Service Units* (SUs), and controlling the state of both individual SUs and the JBI container. Different SUs are usually packaged in *Service Assemblies* (SAs), as shown in Fig. 1, in order to solve larger integration problems.

ESB^{MT}: A Multi-tenant ESB Architecture. In our previous work [21] we identified the requirements for enabling multi-tenancy in ESB solutions and categorized them into *functional* and *non-functional* requirements. Functional

[1] Open ESB: `http://openesb-dev.org`
[2] GlassFish: `http://glassfish.java.net`
[3] WSDL 2.0 Specification: `http://www.w3.org/TR/wsdl20/`

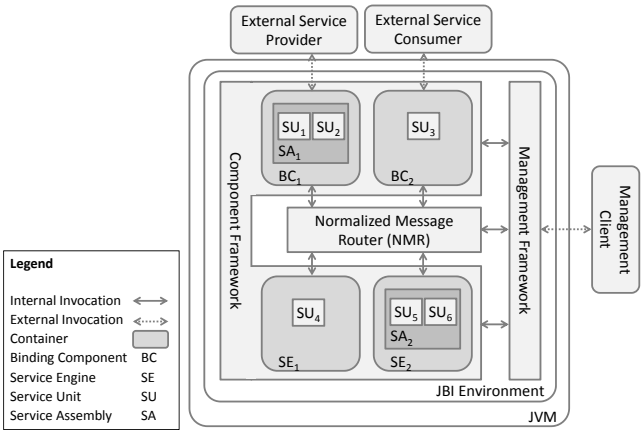


Fig. 1. Overview of the JBI environment

requirements can be further classified as *tenant-related* and *integration-related*. Tenant-related requirements ensure the fine-grained management of both tenants and their corresponding users. In addition, the functionality of the ESB should be provided for each tenant in a transparent manner, without integration effort on behalf of the tenants. Integration-related requirements ensure that other PaaS components or external applications that might not be multi-tenant can also interact with the system in order to share, e.g. the tenant or service registry maintained by the ESB. Non-functional requirements ensure *tenant isolation* and *security* as well *reusability* and *extensibility*. Tenant isolation requirements include data (preventing tenants to access data belonging to other tenants) and performance isolation (ensuring tenants have access only to their assigned computational resources). Security requirements describe the need for appropriate mechanisms for authorization, authentication, integrity, and confidentiality to be in place. Finally, reusability and extensibility requirements define the technology- and solution-independence of the proposed architecture.

Based on these requirements, in [21] we proposed ESB$^{\mathrm{MT}}$, a JBI-based ESB architecture that satisfies these requirements. Figure 2 provides an overview of ESB$^{\mathrm{MT}}$. The three layer architecture consists of a *Presentation* layer, a *Business Logic* layer, and a *Resources* layer. The purpose, contents, and implementation of each layer is discussed in the following.

3 Implementation

For purposes of implementing ESB$^{\mathrm{MT}}$ we extended the open source ESB Apache ServiceMix version 4.3.0 [3], hereafter referred to simply as ServiceMix. All artifacts required to install and setup the ESB$^{\mathrm{MT}}$ realization including a manual are publicly available at `http://tiny.cc/ESB-MT-install`. The presentation

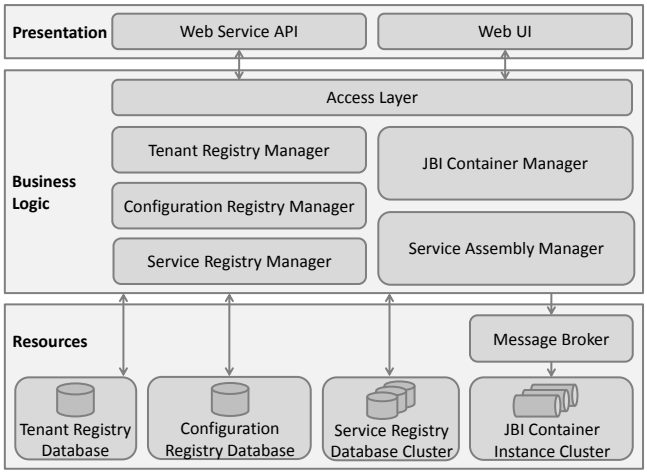


Fig. 2. Overview of ESB^{MT}

of the implementation follows the ESB^{MT} architecture as illustrated in Fig. 2. More specifically:

Resources Layer. The Resources layer consists of a *JBI Container Instance Cluster* and a set of registries. The JBI Container Instance Cluster bundles together multiple JBI containers (in the sense of Fig. 1). Each one of these instances performs the tasks usually associated with traditional ESB solutions, that is, message routing and transformation. For purposes of performance, instances are organized in clusters, using an appropriate mechanism like the one offered by ServiceMix. Realizing multi-tenancy on this level means that both BCs and SEs are able to:

- handle service units and service assemblies containing tenant and user specific configuration information, and
- process such deployment artifacts accordingly in a multi-tenant manner. For example, a new tenant-specific endpoint has to be created whenever a service assembly is deployed to this JBI component in order to ensure data isolation between tenants.

The installation/uninstallation and configuration of BCs and SEs in a JBI Container Instance is performed through a set of standardized interfaces that also allow for backward compatibility with non multi-tenant aware components.

In terms of implementation technologies, ServiceMix is based on the OSGi Framework[4]. OSGi bundles realize the ESB functionality complying to the JBI specification. The original ServiceMix BC for HTTP version 2011.01 and the original Apache Camel SE version 2011.01 are extended in our prototype in order to support multi-tenant aware messaging. These components are able to marshal,

[4] OSGi Version 4.3: `http://www.osgi.org/Download/Release4V43/`

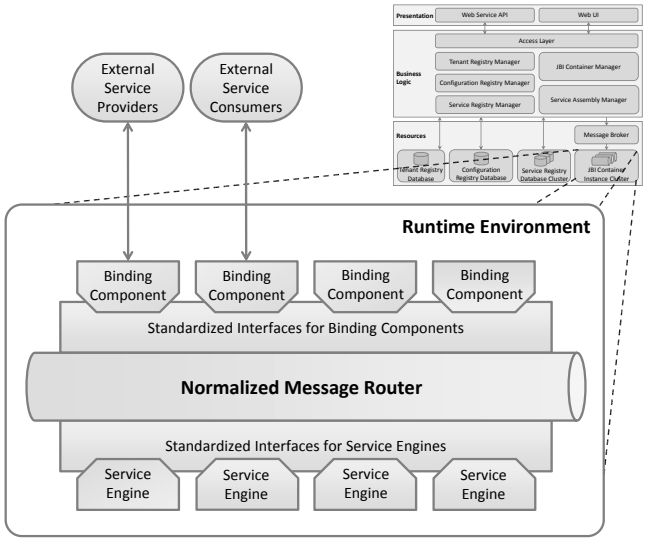


Fig. 3. Architecture of an ESB Instance

demarshal, and process messages with the tenantID and userID included as part of their SOAP header. Our ServiceMix extension also implements an OSGi-based management service which listens to a JMS topic for incoming management messages sent by the Web application.

The Resources layer also contains three different types of registries (Fig. 2): the *Service Registry* stores the services registered with the JBI environment, as well as the service assemblies required for the configuration of the BCs and SEs installed in each JBI Container Instance in the JBI Container Instance Cluster in a tenant-isolated manner [7]; the *Tenant Registry* records the set of users for each tenant, the corresponding unique identifiers to identify them, as well all necessary information to authenticate them; finally, the *Configuration Registry* stores all configuration data created by tenants and the corresponding users, except from the service registrations and configurations that are stored in the Service Registry. Due to the fact that tenant or user actions affect more than one registries at the time, all operations and modifications on the underlying resources are implemented as distributed transactions based on a two-phase commit protocol [9] to ensure consistency. The *ServiceRegistry*, *TenantRegistry*, and *ConfigurationRegistry* components are realized based on PostgreSQL version 9.1.1 [19]. Figure 4 shows the entity-relationship diagram of the information stored in the Configuration Registry.

Business Logic Layer. The Business Logic layer contains an *Access Layer* component, which acts as a multi-tenancy enablement layer [10] based on role-based access control [20]. Different categories of roles can be defined based on their interaction with the system: system-level roles like administrators, and tenant-level roles like operators. The system administrator configures the whole

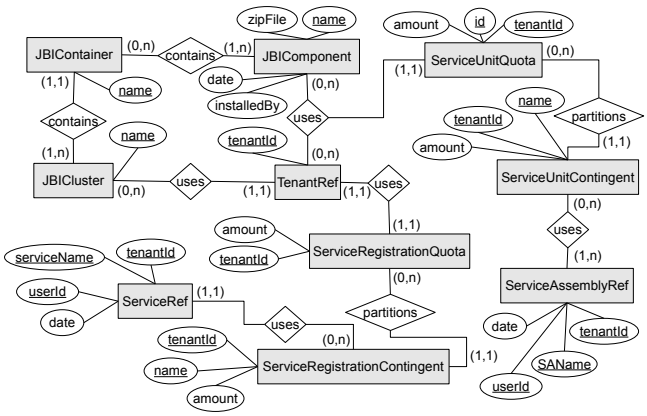


Fig. 4. ER diagram of JBI components in the Configuration Registry using (Min,Max) notation

system and assigns quotas of resource usage. The tenant users consume the quotas of resource usage to deploy service assemblies or to register services. This information is stored in the Configuration Registry (see quota and contingent entities in Fig. 4). A tenant administrator can partition the quota of resource usage obtained from the system administrator. It is important that the system administrator assigns a default tenant administrator role to at least one tenant user to enable the corresponding tenant to perform actions. This default tenant administrator can then appoint other tenant administrators or assign tenant operator roles to tenant users. The tenants and their corresponding users have to be identified and authenticated once when the interaction with the JBI environment is initiated. Afterwards, the authorized access is managed by the Access Layer transparently. The identification of tenants and users is performed based on unique *tenantID* and *userID* keys assigned to them by the Access Layer.

The various *Managers* in this layer (Fig. 2) encapsulate the business logic required to manage and interact with the underlying components in the Resources layer: *Tenant Registry*, *Configuration Registry*, and *Service Registry Managers* for the corresponding registries, *JBI Container Manager* to install and uninstall BCs and SEs in JBI Containers in the cluster, and *Service Assembly Manager* for their configuration through deploying and undeploying appropriate service artifacts.

The Business Logic layer of the proposed architecture is implemented as a Web application. In order to ensure consistency, the application is running in the Java EE 5 application server JOnAS version 5.2.2 [18], 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.

Presentation Layer. The Presentation layer contains the *Web UI* and the *Web service API* components which allow the customization, administration, management, and interaction with the other layers. 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. The Web service API offers the same functionality as the Web UI, but also enables the integration and communication of external components and applications. It is realized based on the JAX-WS version 2.0 [5]. For both interface mechanisms, security aspects such as integrity and confidentiality of incoming messages must be ensured by appropriate mechanisms, e.g. Secure HTTP connections and WS-Security[6]. As a result, signing and encryption of SOAP messages is supported by the implementation. Furthermore, authentication is implemented by using a custom SOAP header element named TenantContext. The Tenant Context contains the tenantID and userID both represented as UUIDs, and the password of the user. This header element is encrypted and signed. Thus, users of other tenants are prevented to act on behalf of the sending user.

4 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 backward compatibility feature of ESB^{MT} as non multi-tenant aware version of the ESB, because the functionality in this case is the same as of the original non multi-tenant ServiceMix that we based our implementation on. The following sections discuss the method, workload, experimental setup and results towards this goal.

4.1 Method

Our investigation showed that there is no commonly agreed benchmark for ESBs, see for example [23]. For this reason we chose to use the industrial ESB benchmark by AdroitLogic [2] 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 [2].

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 the other benchmarking scenarios, however,

[5] `http://jcp.org/aboutJava/communityprocess/final/jsr224/`
[6] `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

we were able to execute the benchmark normally. Secondly, this benchmark was not designed for multi-tenancy, using only one communication endpoint with multiple users of the same tenant sending concurrent requests. Thus, we had to adapt the AdroitLogic Benchmark Driver accordingly, as described in the following sections.

4.2 Workload

For purposes of evaluation we derived three test scenarios from the Direct Proxy Service scenario in AdroitLogic's benchmark [2]. 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 service on the provider side. Starting from this point, we defined the following scenarios:

1. a non multi-tenant ESB deployment (backward compatibility feature of ESB^{MT}) 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* [22], 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.

4.3 Experimental Setup

Figure 5 provides an overview of the experimental setup realizing our adaptation of the Direct Proxy Service Scenario including message flow, control, and measurement points. The test cases were run on Flexiscale[7] 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,

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

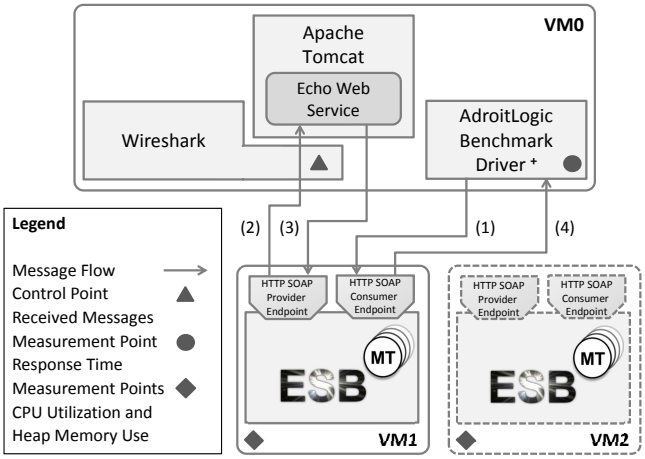


Fig. 5. Overview of the Experimental Setup

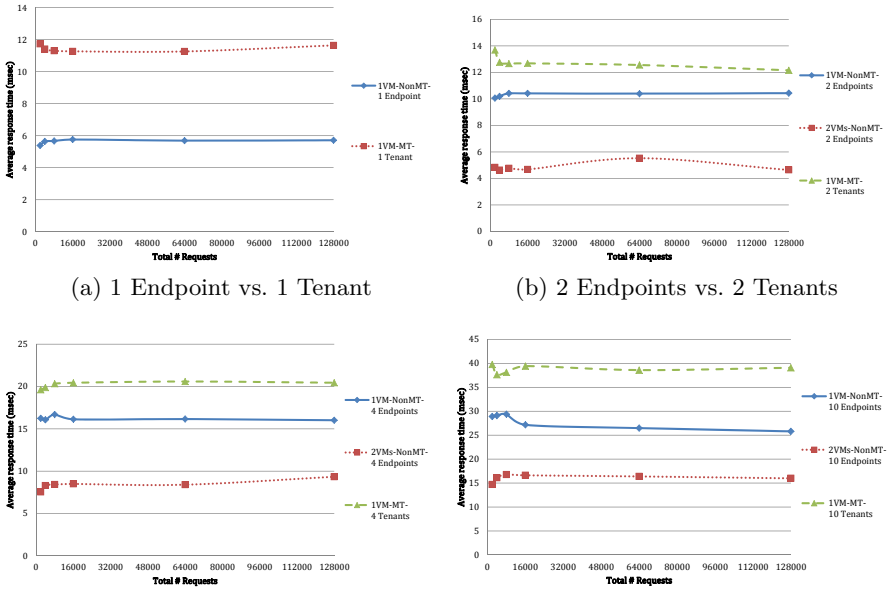
an Apache Tomcat 7.0.23 instance was deployed with the Echo Web service, the adapted AdroitLogic Benchmark Driver, and Wireshark 1.2.7 for monitoring HTTP requests and responses. In VM1 and VM2, the ESB^{MT} implementation is deployed, which required also the deployment of PostgreSQL 9.1.1 database (for 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, see Fig. 5. Scenarios 1 and 2 are using the backward compatibility feature of ESB^{MT} for non multi-tenant operation.

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 latency. 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.

4.4 Experimental Results

Performance: Figure 6 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. 6). As shown in the figure, our proposed multi-tenant aware implementation of the ESB exhibits a performance decline of around 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. 6b), 24, 68% more for 4 tenants/endpoints (Fig. 6c) and 39, 44% increase for 10 tenants/endpoints (Fig. 6d).

When comparing 1 tenant against 1 endpoint (Fig. 6a) an 50% reduction of response time is observed, showing that the performance decrease is actually



(a) 1 Endpoint vs. 1 Tenant

(b) 2 Endpoints vs. 2 Tenants

(c) 4 Endpoints vs. 4 Tenants

(d) 10 Endpoints vs. 10 Tenants

Fig. 6. Average response time (latency) for 1KB size messages

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).

Utilization: The measurements for CPU and memory utilization for the same loads are summarized by Table 1. The reported CPU utilization is normalized over the number of CPUs of the VMs containing the ESB implementation (Fig. 5). 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 Table 1, 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, Table 1 shows also an overall increase of around 100% across the three cases of ESB$^{\mathrm{MT}}$ (2, 4 and 10 tenants). The low standard deviation, and the small

Table 1. CPU and Memory utilization

		1/2E	**2/2E**	**1/2T**	**1/4E**	**2/4E**	**1/4T**	**1/10E**	**2/10E**	**1/10T**
CPU (%)	Average	10,77	8,55	47,01	11,71	8,46	54,42	14,99	9,69	66,33
	Median	11,00	9,42	50,00	12,00	10,00	58,33	16,33	11,83	76,00
	Max	12,00	16,33	51,67	13,00	10,67	60,33	19,00	12,67	78,33
	σ	1,63	3,21	9,03	2,07	3,14	11,38	3,72	4,09	21,39
Memory (%)	Average	18,47	15,99	37,67	23,90	15,22	42,93	20,54	13,26	47,06
	Median	17,71	15,66	36,09	23,37	15,47	43,57	20,70	13,36	46,51
	Max	35,43	22,98	67,31	36,09	20,00	70,50	29,06	18,34	80,78
	σ	0,05	0,05	0,13	0,05	0,02	0,14	0,04	0,02	0,15
Legend:	***1/iE***: 1 VM, non multi-tenant, i endpoints;									
	2/jE: 2 VMs, non multi-tenant, j endpoints in total;									
	1/kT: 1 VM, multi-tenant aware, k tenants									

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.

5 Discussion

The results presented in the previous section show that performance reduction in our implementation is significant (one third of the baseline performance) w.r.t. to system latency. However, it has to be noted that we have not introduced any optimization techniques in our multi-tenant aware ESB solution, or tried to implement performance isolation between tenants. As such, there is much space for improvement in this respect. CPU utilization on the other hand increases more than threefold and remains high for the most part of the benchmark, while memory utilization doubles but remains well below the 50% of the maximum allowed size on average. Our ESB$^{\mathrm{MT}}$ therefore has a relatively small impact on memory requirements, but incurs high computation resource demands. Computational resources are relatively cheap (compared, e.g., to storage space) and continue to grow cheaper over time [4]. The actual cost of using our approach must therefore be evaluated against the possibilities opened by the fine granularity of administration and management on the level of both tenants and users.

Horizontal scaling the ESB produces the desired results, i.e. 50% improvement for adding one VM, only for 2 endpoints distributed between the 2 VMs. Adding another VM produces diminishing returns as the number of endpoints (representing applications using the same ESB) increases. This needs to be weighed against the cost of deploying and operating multiple VMs. Furthermore, the measurements presented in Section 4 for the horizontal scaling scenario assume that the requests are evenly distributed between the two VMs, emulating the effect of a load balancer operating on the front end of the ESB. Actually implementing such a solution will incur additional development and operating costs that need to be considered. In principle therefore we can conclude that the realization of our

proposal achieves its envisioned goal as far as service providers are concerned, i.e. increasing CPU utilization, while imposing a relatively small memory footprint. Performance on the service consumer side however is impacted negatively and further work towards the direction of ameliorating this effect is necessary.

6 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 [10]. As discussed also in [23] 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. [23] 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 [1] 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[8] and WSO2 ESB[9] 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 Axis 2 [10] 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 [8], on Cloud-related features such as elasticity [6] and performance isolation [13], or on virtualization technology [15]. 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 [2] the 6th round of public ESB performance benchmarking 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.

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

7 Conclusions 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. 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 [14]. 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.

Acknowledgments. 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).

References

1. 4CaaSt Consortium: D7.1.1 – Immigrant PaaS Technologies: Scientific and Technical Report. Deliverable (July 2011), http://www.4caast.eu/wp-content/uploads/2011/09/4CaaSt_D7.1.1_Scientific_and_Technical_Report.pdf
2. AdroitLogic Private Ltd.: Performance Framework and ESB Performance Benchmarking, http://www.esbperformance.org

3. Apache Software Foundation: Apache ServiceMix, http://servicemix.apache.org
4. Armbrust, M., et al.: Above the Clouds: A Berkeley View of Cloud Computing. Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley (2009)
5. Behrendt, M., et al.: Introduction and Architecture Overview IBM Cloud Computing Reference Architecture 2.0 (February 2011), http://www.opengroup.org/cloudcomputing/uploads/40/23840/CCRA.IBMSubmission.02282011.doc
6. Brebner, P.: Is your Cloud Elastic Enough?: Performance Modelling the Elasticity of Infrastructure as a Service (IaaS) Cloud Applications. In: Proceedings of ICPE 2012, pp. 263–266 (2012)
7. Chong, F., Carraro, G., Wolter, R.: Multi-Tenant Data Architecture. MSDN (2006), http://msdn.microsoft.com/en-us/library/aa479086.aspx
8. Cooper, B.F., et al.: Benchmarking Cloud Serving Systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
9. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed Systems: Concepts and Design. Addison Wesley (June 2005)
10. Guo, C., et al.: A Framework for Native Multi-Tenancy Application Development and Management. In: Proceedings of CEC/EEE 2007, pp. 551–558. IEEE (2007)
11. Java Community Process: Java Business Integration (JBI) 1.0, Final Release (2005), http://jcp.org/aboutJava/communityprocess/final/jsr208/
12. Josuttis, N.: SOA in Practice. O'Reilly Media, Inc. (2007)
13. Krebs, R., Momm, C., Kounev, S.: Metrics and Techniques for Quantifying Performance Isolation in Cloud Environments. In: Proceedings of the 8th International ACM SIGSOFT Conference on Quality of Software Architectures, pp. 91–100. ACM (2012)
14. Krebs, R., Momm, C., Kounev, S.: Architectural Concerns in Multi-Tenant SaaS Applications. In: Proceedings of CLOSER 2012. SciTePress (2012)
15. Makhija, V., et al.: VMmark: A Scalable Benchmark for Virtualized Systems. Tech. Rep. VMware-TR-2006-002, VMware, Inc. (2006)
16. Mell, P., Grance, T.: The NIST Definition of Cloud Computing (September 2011), http://www.nist.gov/customcf/get_pdf.cfm?pub_id=909616
17. Mietzner, R., Unger, T., Titze, R., Leymann, F.: Combining Different Multi-Tenancy Patterns in Service-Oriented Applications. In: Proceedings of EDOC 2009, pp. 131–140. IEEE (2009)
18. OW2 Consortium: JOnAS: Java Open Application Server, http://wiki.jonas.ow2.org
19. PostgreSQL Gobal Development Group: PostgreSQL, http://www.postgresql.org
20. Sandhu, R.S., et al.: Role-based Access Control Models. Computer 29, 38–47 (1996)
21. Strauch, S., Andrikopoulos, V., Leymann, F., Muhler, D.: ESB^{MT}: Enabling Multi-Tenancy in Enterprise Service Buses. In: Proceedings of CloudCom 2012, pp. 456–463. IEEE (2012)
22. Vaquero, L., Rodero-Merino, L., Buyya, R.: Dynamically Scaling Applications in the Cloud. ACM SIGCOMM Computer Communication Review 41(1), 45–52 (2011)
23. Walraven, S., Truyen, E., Joosen, W.: A Middleware Layer for Flexible and Cost-Efficient Multi-Tenant Applications. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 370–389. Springer, Heidelberg (2011)