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

Master Thesis No. 3671

Dynamic Deployment of Specialized ESB instances in the Cloud

Roberto Jiménez Sánchez

Course of Study:	Infotech (Non-Degree)
Examiner:	Prof. Dr. Frank Leymann
Supervisors:	Santiago Gómez Sáez
	Johannes Wettinger
Commenced:	May 22, 2014
Completed:	November 21, 2014
CR-Classification:	C.2.4 ; D.2

Abstract

In the last years the interaction among heterogeneous applications within one or among multiple enterprises has considerably increased. This fact has arisen several challenges related to how to enable the interaction among enterprises in an interoperable manner. Towards addressing this problem, the Enterprise Service Bus (ESB) has been proposed as an integration middleware capable of wiring all the components of an enterprise system in a transparent and interoperable manner. Enterprise Service Buses are nowadays used to transparently establish and handle interactions among the components within an application or with consumed external services. There are several ESB solutions available in the market as a result of continuously developing message-based approaches aiming at enabling interoperability among enterprise applications. However, the configuration of an ESB is typically custom, and complex. Moreover, there is little support and guidance for developers related to how to efficiently customize and configure the ESB with respect to their application requirements. Consequently, this fact also increments notably the maintenance and operational costs for enterprises. Our target is mainly to simplify the configuration tasks at the same time as provisioning customized ESB instances to satisfy the application's functional and non-functional requirements. Similar works focus on optimizing existing ESB configurations based on runtime reconfiguration rather than offering customized light-weight middleware components.

This Master thesis aims at providing the means to build customized and specialized ESB instances following a reusable and light-weight approach. We propose the creation of a framework capable of guiding the application developer in the tasks related to configuring, provisioning, and executing specialized ESB instances in an automatic, dynamic, and reusable manner. Specialized ESB instances are created automatically and provided to application developers that can build an ESB instance with a specific configuration which may change over time. The proposed framework also incorporates the necessary support for administering, provisioning, and maintaining a clustered infrastructure hosting the specialized ESB instances in an isolated manner.

Contents

List of Figures

List of Tables

List of Listings

1. Introduction

In the last years, the expansion of computing and the number of different devices and computing solutions have grown enormously. The number of components within enterprise systems has become bigger and bigger with the addition of computing in the different areas of the company in the chase for efficiency. With this rising, the heterogeneity of the systems has grown considerably and developers must address new and increasing integration problems. This brings a scenario, where a new integration solution must be created for each pair of components in the system, and hence, the complexity of the system and the number of solutions grows exponentially with the total number of components in the system. Moreover, if one of the components in the system changes, all the components which communicate with it must be modified in order to keep the system working. In this scenario is where a new solution for these integration problem was proposed, the Enterprise Service Bus (ESB) [4].

An ESB provides a centralized architecture with a central middleware is used as a bus for wiring all the components of enterprise system. The ESB comes with features to wire different components and to support several protocols (i.e. HTTP/HTTPS, SQL, SOAP, etc.). It includes also features to handle the exchanged messages between the components and to adapt its shape (e.g. changing the message format from JSON to XML). A wiring solution must be implemented inside of the ESB for each element in the system to connect it with the rest of the components. However, once this wiring solution is implemented, multiple identical components can use it. For example, if a server is deployed in the system, a wiring solution is implemented to connect it to the ESB, but in case more of these servers must be connected too, they would make use of the same solution. The other case that can be addressed with the ESB model is when a determinate component changes. In this case, if an ESB is used, only the wiring solution of the modified particular component must be updated, the rest of components remain identical. For example, the component which initially sent the data in a JSON object, now it sends it in XML. In this case, the ESB converts the incoming XML to JSON format, and the rest of components do not notice this change.

During the last years, there has been a wide research conducted in the ESB field, and nowadays there are multiple projects which provide ESB solutions (i.e. ServiceMix, Mule, Fuse ESB,etc.). Although these solutions are powerful and include many features, its configuration must be performed manually and it can be considerably complex. Moreover, its posterior reconfiguration may be as complicated or even more. On the other hand, we find that each feature installed in an ESB consumes many system resources and normally if a feature is not longer required in the system, it stays to avoid the probably time-consuming reconfiguration process.

In this scenario is where we propose the creation of a platform to configure and launch ESB instances in an automatic, dynamic and reusable way. Our target is enabling the creation of specialized ESB instances with all the necessary features and configuration in a reusable way to provide automatically to multiple developers. In case the developer needs an ESB with another configuration, he can remove the existing one and request a new one with the desired configuration and features. Therefore, the developer does not waste time reconfiguring his ESB instance.

In this document we discuss how such a platform could be designed and implemented. We investigate how Cloud Computing and container virtualization could help to create the desired system. In Chapter 2, these two important concepts plus other main concepts, ideas and relevant information, are explained with detail in order to establish the background for this work. Later in Chapter 3, some works related with this thesis and how they match with our proposal are described. In Chapter 4, the requirements and use cases our platform must achieve are shown. In Chapter 5, the system architecture we propose and its components are described in detail. In Chapter 6 we can find a description of the prototype implemented during our research. In Chapters 7 and 8 we expose a validation and evaluation of our system implementation respectively. Finally in Chapter 9, we present our concluse and present future lines of works.

2. Fundamentals

2.1. Cloud Computing

In the last few years, the development around *Cloud Computing* [6] has increased considerably. The promise of a huge costs reduction, flexibility and scalability is drawing the attention of companies, anxious to find something which helps them to deal with a really increasing number of users, resources, or data processing in a world with high competitiveness. For this work, Cloud Computing must necessarily the first idea to be presented because it is the origin and final purpose of most of other concepts explained in this document and sets the basis of the system we want to create.

2.1.1. Historical Context & Trends

Since the born of the informatics in the 60s with the launch of the first computers, the world and specially this sector has suffered a complete revolution as big as the industrial revolution in the beginning of the 20th century. These first systems were extremely costly and just few companies could afford them. These vast prices force costumers to rent them during a limited time in order to reduce the charge. Indeed, some companies which did business renting these big computers were founded as Electronic Data Systems (EDS) that rented IBM computers per hour. This concept was called *Utility Computing* [7]. In Utility Computing, all the physical resources of a provider are used by different users during a certain amount of time and subsequently the ownership and maintainance costs are reduced.

In the 90s and the beginning of 21th century, universities started to analyse the possibility of resolving complex problems by big amounts of the standard hardware x86 and open source software. This mechanism was used to find solution in multiple sectors, for example, to solve engineering problems or 3D animation design. This was the born of the concept *Grid computing* [7], which is not efficient for big computers because they can solve these big problems themselves, but some of its ideas still remain inside Cloud Computing [8] like using economic hardware and open source software to resolve complex problems and that way, decrease the operational costs.

In the 90s the development of Internet networks and the increase of data rates enables the burst of new Internet users and likewise the born of more efficient and bigger data centers with thousands of computers in the same location. The speed development linked to the increasing demand of data processing worldwide created a need to find something new and revolutionary in this landscape to face these problems. And then it was when the Cloud Computing appeared on the scene. Now we must take a look at a scenario without Cloud and a scenario running an application in the Cloud, and in that way, understanding why the introduction of Cloud Computing is relevant nowadays.

Scenario without Cloud

In our initial scenario without using Cloud computing, a company must maintain its own infrastructure where its applications run. These applications are normally permanently allocated in the same server, because a manual configuration must be performed to install it in another machine. This stability does not help with management and hence, it increases highly the costs.

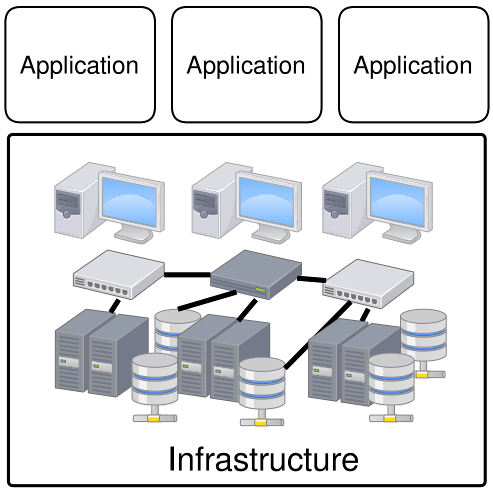


Figure 2.1.: Scenario without Cloud

On the other hand, the company must make an initial invest, normally huge, to create a new system. Commonly a high availability is desired, and hence, duplicated location for this kind of installation is necessary because physical security. For example, in case a building in one of the locations is ablaze, the company could still use the system, because its traffic would be redirected to another location (maybe working slower because of the overload).

Additionally, if we want to have different locations, we must pay the ground cost and something really important is where the installations should be. Normally, they should be in a cheap zone, and with an important communication node in order to get faster data transfer rates. Another important aspect to take account is that processing units expend a lot of energy and therefore, they create a big amount of heat too. Hence, as they cannot overheat, efficient cooling systems must be installed in each one of the locations as well and therefore, costs increase. Finally, we should remark that the servers are usually underutilized, and companies have more capacity they require in significant time intervals, and if a way to make a efficient use of their power is found, the costs decrease enormously.

Scenario with Cloud

Now we have taken a look in some of the problems companies must face in the initial scenario we must know how a Cloud Computing infrastructure could help. Commonly this is called Infrastructure as a Service (IaaS). Using the IaaS model organizations source additional capacity over the web as a service, something faster than installing and configuring new components in their private systems. This gives a flexibility not found previously, companies can add or remove elements to their infrastructure rapidly. This flexibility may be really important nowadays where predicting capacity requirements has become more and more difficult with the increasing number of users. Services that one day have 200 users, next day they may have 10000 thousand, and if these services are built over Cloud, they can react much faster to these variations and without doing any new investment, only renting new additional capacity. Otherwise, a service which was supposed to hold thousands of users, after a costly setup of huge infrastructure and it is deployed, it used uonly by hundreds. Therefore, just a small portion of the system is used and consecuently, its owner looses money. Consequently as we can see, Cloud infrastructures scale better compared to traditional ones.

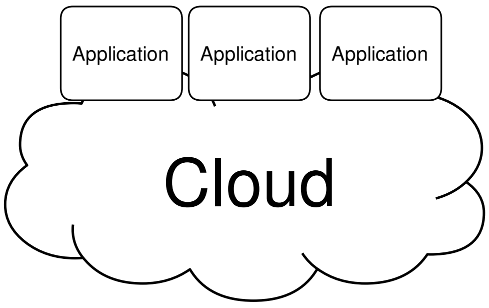


Figure 2.2.: Scenario with Cloud

An idea present nowadays in all companies that offer Cloud environments is "pay as you go", which means users pay just for the resources that they use during the time that they use them. Therefore, users pay really for the processing capacity, memory or network resources that their systems really need. In order to explain this scenario clearer we are going to make an analogy with transport. When you want to buy your own car, you have to pay a big amount of money initially, and later you can travel with it during some years until it breaks or gets obsolete. However, you could not buy that car and use a taxi or public transport, and in that way you would not be force to make such an initial investment to be able to travel, just you would pay for the ticket. Besides, if you buy a car, you get it in days or weeks not when you want to use it for first time. These things are what Cloud brings, users pay just for what they use and when they need.

Then what we can see is the cost of Cloud computing is an operational expenditure, not a capital expenditure, and the main advantage of operational expenditure is that making a decision is much simpler that in a capital expenditure, because the risks are lower and usually external funding is not necessary. For example, taking the decision of buying a car can take weeks, months and even years because it is a big amount of money to be at least concerned.

Besides, for many people is inevitable borrowing money from banks or their families, because they do not have such money at the moment, but they can pay perfectly along the following 5 or 10 years. That is the reason why the companies prefer variable costs over fixed costs.

On the other hand, in the first scenario without Cloud, companies must take care of the management of the underlying infrastructure, and additional fixed costs and unexpected events may appear. But if the company uses an IaaS offering, the IaaS provider must manage this infrastructure with its consequences (e.g. failures, maintenance, backups, availability, etc.). This is like when you have a car, if it breaks, you must pay for its repair for cleaning it every month. By contrast, if a train breaks, you do not have to take charge for it, the train company does and there will be another train waiting for you on the station. Besides, if your car breaks, you have to find another way to go wherever you want to go. So if you are working with Cloud applications, you do not worry about management or availability of your system, at least in a infrastucture level.

In a different case, imagine that the car has a design problem and the engine breaks regularly, and after a few years of desperation, you decide to buy another new and gorgeous car. In this case, the initial investment would have been totally wasted. Talking again about the matter that concerns us, if the implementation of the system, where a big overlay has been done, does not work properly and it is never used again after the first two years, the company looses a huge amount of money. So a Cloud solution is long-term cheaper and less risky, overall in the always changing IT world.

One important benefit is that if an update or change in the application is required, it must be done just once and probably without end users aware. End users, who use Cloud applications, do not have to worry about updates or patches, when the application is always ready to be used. For example, if a train, which carries hundreds of people every day, is painted at night when is out of its duties, and in the morning travellers use it as every day.

Also another important advantage beyond the cost is the added value in the Cloud applications. The new applications include not only functionalities, but also information or data that conventionally the user should introduce himself. And that enables a platform to develop a really complex and powerful system. For example, in Google Maps [9], users can find lot of maps information, places, traffic data or routes, always up-to-date and ready to be used that are not stored locally. Moreover, Google Maps can be integrated with third party applications and therefore, new complex applications may appear making use of this relevant information.

2.1.2. Essential Characteristics

The term Cloud Computing was first used by Prof. Kennenth K Chellepa in 1997 at the Informs Conference in Dallas as "a computing paradigm where the boundaries of computing will be determined by economic rationale rather than technical limits" [10]. This definition is less specific than the definitions in circulation today.

The most accepted definition today is given by NIST that defines *Cloud computing* [8] as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. This cloud model of five essential characteristics (On-demand self-service; Broad network access; Resource pooling; Rapid elasticity; Measured Service), three service models (Cloud Software as a Service (SaaS), Cloud Platform as a Service (PaaS). Cloud Infrastructure as a Service (IaaS) and four deployment models (Private Cloud; Community Cloud; Public Cloud and Hybrid Cloud)". These services and deployment models are explained further in the remaining of this section.

Cloud is not a new invention, but more of a "practical innovation", combining several earlier inventions into something new and compelling. It requires many of the modern available technologies like high bandwidth networks, automation or virtualization, concept that will be explained further in following sections. In fact, some years ago none would have though about Cloud Computing because the networks were too slow and other technologies so primitive. The main features [8] that describe Cloud Computing are:

- *On-demand self-service*. A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.

- *Broad network access*. Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).

- *Resource pooling*. Providers computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer's demand. There is a sense of location independence where customers generally have no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or data center). Examples of resources include storage, processing, memory, and network bandwidth.

- *Rapid elasticity*. Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

- *Measured service*. Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

2.1.3. Service Delivery Models

In Cloud Computing some more service models [8] [11] apart from IaaS are described. These models are Software as a Service (SaaS) and Platform as a Service (PaaS). Each one of these models is destined to different collectives, and in terms of its value to the end users there are some differences. Figure 2.3 shows that the PaaS is the most valuable to the end users due to it provides new functionalities and services the end user can use. In return, the IaaS systems are more valuable for system architects than for end users.

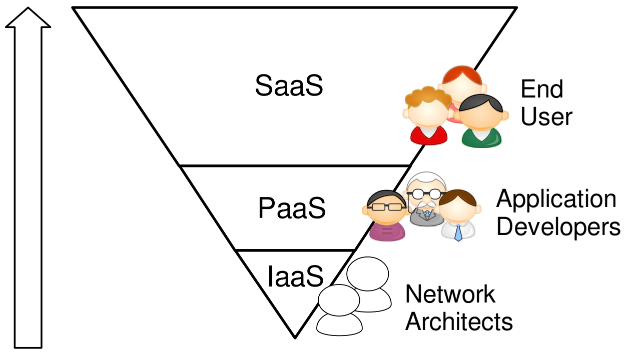


Figure 2.3.: Value visibility to end users [1]

Infrastructure as a Service

This is the simplest of the three models, it provides to the consumer the capability of processing, storage, network broadcast, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying Cloud infrastructure but has control over virtual servers including their operating systems, storage or deployed applications. Besides, consumers can restrict the network access to this kind of systems to improve the security configuration. Examples of providers of this kind of services are IBM (Softlayer [12]), Amazon EC2 [13] or Rackspace [14].

Platform as a Service

In comparison PaaS is more sophisticated. It enables consumers to deploy their own applications over Cloud and as result, over a resilient platform. The consumer does not have to worry about management or control over the underlying Cloud infrastructure. Things like operating systems, hosts, servers, network, communication or storage, now are managed by the Cloud provider, offering an additional and important abstraction level. The deployed application adapts automatically to the number of users without human interaction and with

[1] `http://hrushikeshzadgaonkar.files.wordpress.com/2011/05/cloud_stack.gif`

a more than acceptable Quality of Service (QoS). As a result, if the application is used by a few people, a few money is paid, but if it is used by lot of them, the consumer must pay an extra charge because of the resource usage. In this Cloud the developer only concerns about his application and configuration settings for the application-hosting environment if it is required. In this kind of platforms, the provider enables his own API, libraries and tools to create the services and to communicate with the Cloud. These APIs usually support different conventional programming languages like Python, Javascript, Java or PHP. Messaging and database are also usually included in order to make easier the things to the programmers. Hence, one important point is that if the developer wants to move his application to another platform, infrastructure or Cloud, he will be forced to write most of the code again.

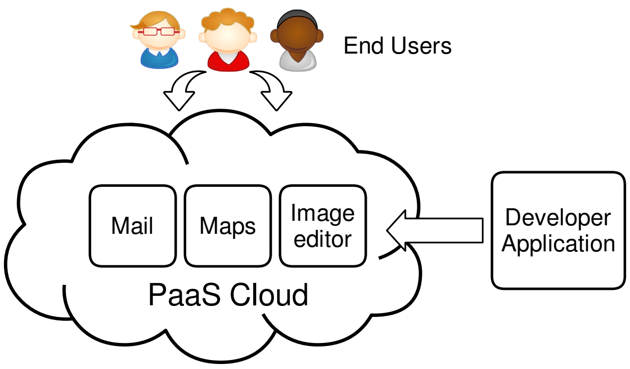


Figure 2.4.: Cloud computing - PaaS

In this thesis we focus on this kind of model and how it can be provided in the most efficient way possible at the moment. Here the provider is responsible of the whole middleware platform which enables users to deploy their own applications. Consequently, providers pursue to streamline their platforms and in that way serving more users and obtaining a higher benefit. As we have already discussed, PaaS makes use of virtualization widely and therefore the key to improve the efficiency of this pattern may be finding a lighter and more flexible virtualization. in the following sections, we discuss about the different virtualization alternatives at the moment in order to find the one which fits better with the needs of the system we must implement. Some examples of PaaS are Google App Engine [15] and Microsoft Azure [16], but IaaS can become also PaaS making use of an open source platform as OpenStack [17].

Software as a Service

In SaaS the end users pay for using provider's applications running on a Cloud infrastructure instead of a software licence installed it in their own machine. Users pay just for the use of the applications, usually monthly. The applications are accessible from various client devices through either a thin client interface, such as a web browser or a program interface. The consumer does not manage or control the stack below, he wants only to use the application and he does so, anything else is running without his concern. SaaS is the most abstract of the three models, all the network, configuration, resource management, hosting, updates, failures or storage are administered by the Cloud service provider with total transparency. Examples of this kind of software are Google Docs [18], Lucidchart [19] or Babbel [20].

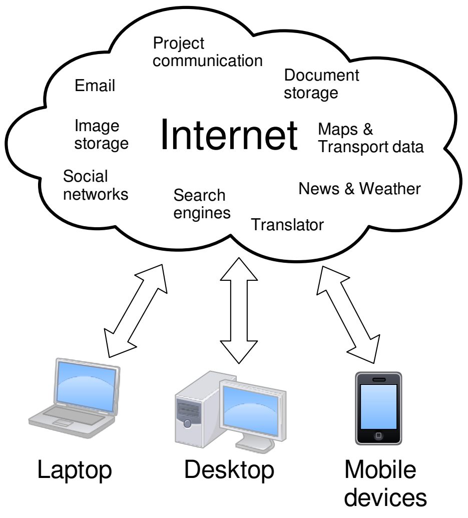


Figure 2.5.: Cloud computing - SaaS

2.1.4. Deployment Models

On the another hand, different kinds of Clouds related to location and ownership are available to satisfy the needs of more diverse costumers. A brief introduction to the main ones is given in the following sections.

Public Cloud

The *Public Cloud* [8] is the most important configuration at the moment because its flexibility, scalability, simplicity and price. The Cloud platform is provided for the use of general public, and it is managed, exploited and owned by a private or public company. With this kind of Cloud users can choose the services model that fits better with their needs and the resources required for the correct performance of their systems. In public Clouds the user pay for what he use whatever it is: storage, services, network, servers, communication, application hosting, etc. Also if an application does not require some capacity in a determinate moment, some of the resources can be stopped immediately without any extra charge, or viceversa.

Community Cloud

A *Community Cloud* [8] is equal as a public cloud but for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, compliance considerations, etc). It may be supervised and operated by one or more of the organizations in the community, a third party, or a combination of them.

Private Cloud

A *Private Cloud* [8] is created for exclusive use by a single organization comprising multiple consumers. It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises. This kind of Cloud is commonly used by big corporations, overall in the financial section because they want to ensure their data privacy unbroken using really robust security tools [21]. Its cost is usually high, and a big initial investment and experts are required. For that reason, the adoption of this kind of technology is taking place slowly and gradually. However, the tend is changing due to some hardware manufacturers like IBM or Oracle have started putting on the market new integrated tools for easily management of small private networks destined to small and medium-size companies that right now prefer public Clouds because their scalability and price.

Hybrid Cloud

Finally, we find the *Hybrid Cloud* [8]. This Cloud may be made-up of more than two distinct Cloud platforms (private, community or public clouds) whose autonomy remains invariable. However, these Clouds are linked in their bounds to each other by standardized or proprietary technologies for communication and integration.

A technique used in the last times is *Cloud bursting*, which combines internal resources with public Cloud resources. This method was born as a response for some scenarios, when a the demand of resources increases and the application must be moved to a public Cloud which follows the model *pay as you go*. With the use of hybrid Clouds the organization only must pay for the resources when are required and the application is able to deal with peaks of traffic. This approach is recommended overall for applications which must achieve high performance, high availability, failover recovery, and which do not handle sensitive information.

2.2. Virtualization

2.2.1. The Origin and Definition of Virtualization

One of the most famous adages in computer science is that “any problem in computer science can be solved by another level of indirection” said by David Wheeler [22]. Time has proved this assertion unequivocally.

The virtualization despite it sounds as a modern concept, dates from 1960s and 1970s when IBM developed the Control Program/Cambridge Monitor System (CP/CMS) [23] which led into VM/370 [24]. These computers run isolately from each other, but all inside a timeshared computing environment.

The language virtualization was firstly introduced in 1980s to support application-level portability and isolation. But one of these language which late to our days, appeared in the 1990s, the Java Virtual Machine (JVM) [25] . The JVM, which firstly was created to program washing machines, offered developers an opportunity to add content to the Web in a portable and secure manner (this Java components are called Java Applets). Besides, with Java [26] the same application can run in a portable way between different platforms, for example in Windows XP, Mac OS X, or any Linux. They used the slogan "write once, run everywhere", something really attractive for headached developers.

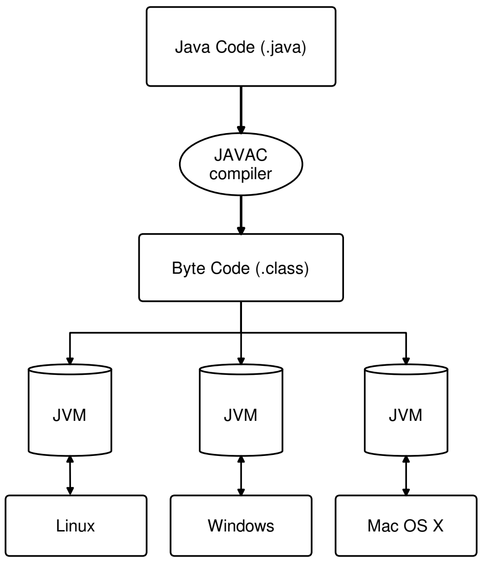


Figure 2.6.: Java Virtual Machine in different platforms [2]

However, despite of the huge repercussion of virtualization, there was a gap between a Virtual Machine (VM) for a programming language runtime and one for an entire Operating System. In this scenario, the Stanford University appeared on the scene with its research project Disco [27] that led to the present head of the virtualization, VMware [28].

In this thesis we focus on Cloud Computing and one of its building blocks is, of course, virtualization. Virtualization is the basic pillar that enables a more efficient use of the resources.

In the 1974 article "Formal Requirements for Virtualizable Third Generation Architectures" [29] Robert P.Goldberg proposed the “Hardware Virtualizer,” in which a virtual machine would communicate directly with hardware instead of going through the host software. It set the conditions sufficient for a computer architecture to support system virtualization efficiently.

[2]`http://javapapers.files.wordpress.com/2011/11/java-program-execution2.png`

A *virtual machine (VM)* is an abstraction layer or environment between hardware components and the end user. A host operating system can run many virtual machines and shares system hardware components such as CPUs, controllers, disk, memory, and I/O among virtual servers. In virtualization the user does not use the the physical resources, instead of that, he uses a "virtual" one rather than the actual.

This key mechanism can be in different levels and shapes. For example we can find network, storage, server, application or desktop virtualization. Increasingly, that level of indirection takes the form of virtualization, in which a resource's consumers are provided with a virtual rather than physical version of that resource. This layer of indirection has helped address myriad problems, including efficiency, security, high availability, elasticity, fault tolerant, mobility, and scalability. Virtual servers generate hardware cost savings by a better utilization of physical resources.

The guest software runs just as if it were installed on a stand-alone hardware platform. Usually, many virtual machines run on a single physical machine; their number is limited by the host hardware capability, such as core number, CPU power, RAM resources, etc. There is no requirement for a guest operating system to be the same as the physical host one. The guest system is able to access specific peripheral devices, exploiting interfaces to those devices, e.g. hard disk drive, network interface card, graphic and audio card.

In these days virtualization is an area with intensely development and interest for companies in order to reduce costs. One of the leader vendors of virtualization is VMWare [28] which offers different virtualization products in multiple shapes and levels. However, the huge data centers make use of the Xen Hypervisor [30], a paravirtualization solution that is re-purposed at runtime to serve either as a host or guest Operating System and offers a good performance.

Now we want to mention the principal aspects and motivations that nowadays has brought virtualization into scene and which will lead to a huge expansion of this field in the next few years.

Server Consolidation

One of the premises nowadays is avoiding not necessary replication or use of resources in a efficiency chase. Following this premise, many small physical servers are substituted by a larger one, which is able to run several virtual machines and by this way increasing the utilization of the expensive hardware resources.

Simplified Management

A virtual machine can be more easily managed, configured and controlled from outside than a physical one, and for example, it can include a snapshot system in order to restore quickly and easily a previous state of the Operating System (OS). Other advantage is the possibility to switch on and off remotely the VM, something not possible with physical ones.

On the other hand, these days a huge number of servers with the same configuration are deployed, and with virtualization only one must be configured. The other ones are launched from the same VM image.

Virtual Machine Migration

Virtual machines can be moved with no trouble from a physical machine to another one. These machines can be reallocated, cloned or launched in other host systems without any typical additional configuration like hardware drivers might be. This aspects give a high mobility to the VMs and with it they could be used in high availability and failover scenarios.

Cloud Computing Outbreak

The increment of the number of devices and the arrival of new kinds of devices or use cases has force the developer community to find a solution to deal with this widespread growth and keep providing QoS to the end users. The spectrum of these devices is wide, and many of them cannot perform some tasks that might be interesting in some cases, and in this way, this task should be realized somewhere else. In that scenario is where the ship of Cloud Computing has appeared and, taking virtualization as one of his basic pillars, has lead to the increase of VMs and new kinds of virtualization solutions.

Unlimited Compute Power

One of the most significant transformations that virtualization has led to is IaaS Cloud computing. Users can access virtually unlimited computing power whenever they need it, paying for only what they use. Producers can efficiently support many users while they are isolated from each other. Consumers must not know about resource allocation or physical capacity. For example, the VM can be moved to another physical or virtual machines or even Clouds.

2.2.2. Types of Virtualization

Now we must talk about some different virtualization levels and types. In the following sections you can find a short explanation of the most important ones.

Full Virtualization

In full virtualization [31], the virtual machine simulates enough hardware to allow an unmodified guest operating system to run in isolation. This approach was pioneered in 1966 with the IBM mainframes. Full virtualization fully abstracts the guest operating system from the underlying hardware (completely decoupled). The guest operating system is not aware if it is being virtualized and requires no modification. Full virtualization offers the best isolation and security for virtual machines, and allows simple procedures for migration and portability as the same guest operating system instance can run virtualized or on native hardware.

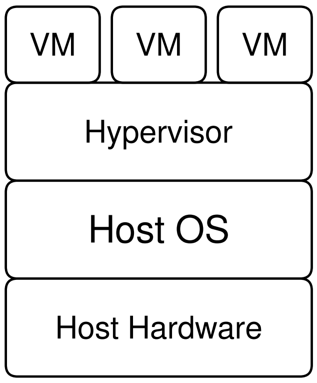


Figure 2.7.: Full virtualization stack [1]

Paravirtualization

Paravirtualization [31] is a variant of full operating system virtualization. Paravirtualization avoids "drawbacks of full virtualization by presenting a virtual machine abstraction that is similar but not identical to the underlying hardware"

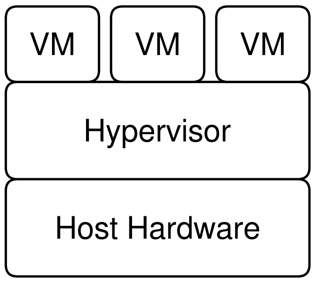


Figure 2.8.: Paravirtualization stack [1]

In a paravirtualized architecture, a light software layer called *Hypervisor* runs directly over the hardware. The Hypervisor is able to allocate the resources needed by the virtual machines. A privileged operating system instance runs over the Hypervisor in order to manage all the active virtual machines. Hypervisors have specialized management functions that allow multiple VMs to coexist peacefully as they share real machine resources.

Paravirtualization enables multiple isolated and secure virtualized servers running over the same physical host. It also has a lower virtualization overhead. The performance can be improved over full virtualization, but it can vary greatly depending on the workload. In contrast to full virtualization, the host and guest operating systems must be modified in order to replace the privileged operations with Hypervisor calls, and therefore, the guest is aware it is being virtualized in comparison.

Shared Kernel

Shared kernel virtualization [32], also known as Operating system level virtualization, is a virtualization method where the kernel of an operating system is used by multiple isolated user-space instances, instead of just one. Such instances, called containers, may look like a real server from the point of view of the end users. These containers can share also libraries and Operating System with any kind of collision, also with the host physical machine.

In addition to isolation mechanisms, the kernel is able to provide resource management features to limit the impact of one container's activities on the other containers. In this way multiple instances can run exactly as multiple physical servers. More details of shared virtualization and containers are given in Section 2.3 where we also talk about the benefits over other kinds of virtualizations.

2.2.3. Virtualization Levels

We have already introduced the most used types and techniques of virtualization in the previous section, and we want to go into more detail with all the different levels where virtualization can be present, from hardware virtualization to desktop virtualization.

Although virtualization is often associated with processors, there is virtualization in all the tiers of the OSI model [33]. For example, virtualization is really important in networking where Virtual Private Network (VPN) or Virtual Local Area Network (VLAN) have been use for years in order to provide isolated virtual networks with the same physical structure.

In this thesis we take care of Cloud Computing, and the nowadays approach goes though virtualization as a mean to enable the usage of resources by as many users as possible. The virtualization in Cloud Computing can be present in all the levels and shapes as it helps to achieve the desired systems efficiency. For instance, it is relevant in Cloud platforms as Amazon EC2 where the user can rent a virtual server instead of a physical one that the provider can allocate or free where, how or when he wants, otherwise a physical one should be reserved per each user and the costs would grow.

Server Virtualization

Server virtualization industry is the most active segment nowadays. Each virtualized server runs its own OS with its own virtual CPU, memory or peripheral devices in secure and isolated way. Typically each server has only one job or function (i.e, database, mail, dns, etc.) and it uses just a fraction of the total host machine capacity.

A mainstream tendency these days is the execution of multiple small virtual servers on a big host one, and in this way, increasing the hardware exploitation and efficiency. Also server virtualization contributes to streamline another aspects like performance, management or security.

Desktop Virtualization

In this kind of virtualization a separate OS environment runs over the existing running OS on the desktop. This enables for example the execution of non compatible programs like legacy ones or others that must be executed in another OS. Two variants of desktop virtualization are distinguished:

- *Remote Desktop Virtualization*. The operating environment runs in another location and it is accessible across the network. The end users must only install a client application in their machines that establish the communication with the remote host.
- *Local Desktop Virtualization*. In this variety, the operating environment runs locally in the user's physical computer. A middleware software must be installed in other to support the execution of the new OS.

Likewise this virtualization approach enhances security, makes easier the management or control tasks and end users have on-demand access to their virtual desktop anytime from anywhere (i.e, from home, mobile phones, tablets, office, etc).

Storage Virtualization

In Storage virtualization a group of independent storage resources are combined in order make believe they are just one single entity. Taking use of this, this collection can be centrally managed as an unit and in this way increasing the flexibility and enabling the possibility of a total storage expansion in a easy way.

Network Virtualization

Decoupling between the network infrastructure and service provisioning is the key of this virtualization. The responsibility of the traditional Internet Service Provider (ISP) is divided into: Infrastructure Provider (InP), who take care of the underlying physical infrastructure, and the Service Provider (SP), who offers network services to the end users.

With this decoupling more complex services can be created because SP can forget the underlying infrastructure. It also enables that multiple VN (VN) use the same physical subrate with isolation. A VN is composed of a set of virtual links and nodes, which can be considered as a subset of the total network.

An example of Network virtualization is the VLAN where a collection of hosts with the same interest can be logically connected as they would be in the same Local Area Network (LAN), despite of they may be located in different buildings or floors.

Application Virtualization

Application virtualization provides specific applications to the end users that are virtualized from the desktop OS and which are not installed as usual. The application is encapsulated inside a container which interacts with the host system and can isolate it in order to avoid undesirable interaction with other systems or application components.

2.3. Container Virtualization

Nowadays, hypervisor-based virtualization is the most popular virtualization technology used in the main large datacenters of huge companies. It is flexible, works for almost any guest operating system and, as we have already said, has several performance advantages over full virtualization. However, the most modern approach shared kernel virtualization or container virtualization can have more additional performance or flexibility benefits in some cases [32].

A container is an operating system-level virtualizaton method for running multiple systems on a single host. Normally, containers are constructed over the linux kernel and they are commonly called Linux Container (LXC) [34]. The Linux kernel comprises cgroups for resource isolation (CPU, memory, block I/O, network, etc.) that does not require starting any virtual machine or hardware emulation. Cgroups also provides namespace isolation to completely isolate application's view of the operating environment, including process trees, network, user ids and mounted file systems.

The key drawback of virtualization is that it requires the system to run multiple copies of the guest OS. The memory overhead used for that hits directly the performance of the whole system. Containerization on the other hand shares a single host OS, and single kernel, but creates containers inside that OS which include all of the software, environment and libraries for a particular application. For example, in this kind of virtualization the hardware calls are not duplicated and the host operating system is responsible of all the hardware management. This property increases the performance of the system, which can hold more VMs [32] [35].

On the other hand, a challenge in this kind of virtualization is the isolation, despite there is an isolation between containers is not complete and they still share memory and kernel, and

[3]`http://msopentech.com/wp-content/uploads/Docker-containeraization.jpg`

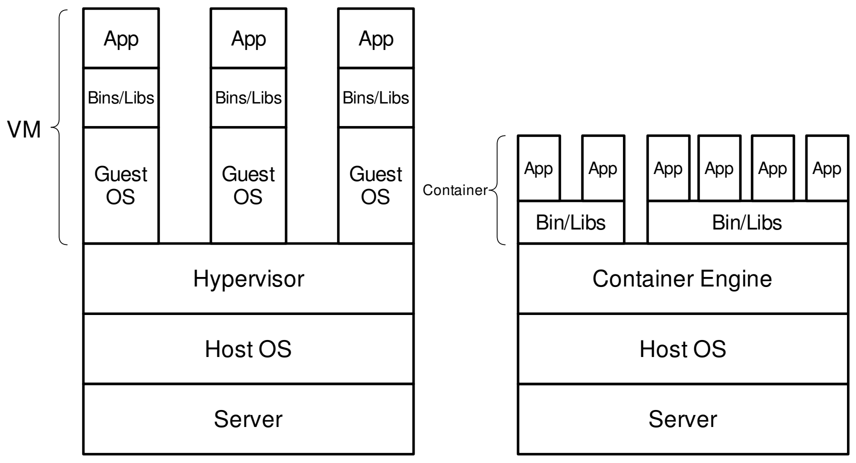


Figure 2.9.: Comparison VMs and Containers [3]

cross application may be performed. For example, this can be dangerous in case of malware attack in one of the containers, and in that case the attacker could have full access to all data or running applications in the other containers.

One of the first container virtualization approaches was implemented by Sun some years ago, but this LXC proposal did not have success because it was a really complex solution. In the last years some new alternatives have appeared, but overall during the last year the open source project Docker has hit the virtualization and Cloud computing environment. Docker is the most attractive alternative at the moment and it has originated a lot of research projects all around the world in both university departments and huge companies as IBM, Amazon or Google. Some of the main features and interesting aspects of this relevant project are given next.

2.3.1. Docker

In the previous sections the shared-kernel virtualization was presented. An open-source project has change the way that community looks to this kind of virtualization, Docker. Docker [36] is an open-source engine based LXC which try to automate the deployment of any application in a lightweight, portable and self-sufficient way to run anywhere. Its first stable version was released in April 2014 and with it a widely research and development in Cloud Computing by a lot of different companies, organizations or universities. For example, important companies like Google, IBM, Ebay or Spotify have started to integrate Docker in their complex platforms in order to streamline them.

The automation of deployment is realized by *containers* which encapsulate any payload. They already include all the application dependencies and therefore are ready to run anywhere in anytime. This containers are basically LXCs that provide a high-level lightweight virtualization that run processes in isolation. Moreover, this container virtualization framework enables automating packaging and deployment of application in many different famous platforms as OpenStack, Amazon EC2, etc. In case you would like to use other alternatives to Docker, these might be Chef [37], and Puppet [38]. Somo features which may be relevant for developers are:

- *Run Anything*. All the application code can be moved inside a container and can be run in any host with Docker installed and Linux kernel as the Figure 2.10 shows.
- *Container isolation*. The containers run in isolation, and they are secure from outside access. Besides, if one container goes down, the rest of the containers and therefore applications are not affected.
- *Portability*. Docker containers are simply directories, they can be compressed and copied anywhere. That same docker container can run anywhere on any linux system on any cloud without any change. It brings with it all required libraries, file versions and environment variables needed. Therefore, they must be configured only once.
- *Fast and lightweight*. Containers do not need much more resources than what each application needs and the complete engine is though to save as many resources as possible. For instance, the images, which hold all the necessary libraries the application uses, are saved a differential storage, they only save the difference between the new images and old ones.
- *Social sharing*. Docker provides a repository community platform to share container images in a public or private way, something that enables and encourages the development in the platform. The images of the new containers can be based in other ones built by other developers. For example, there are a lot of images with Ubuntu, Debian, Python or Node.js already configured and ready to run a new application.

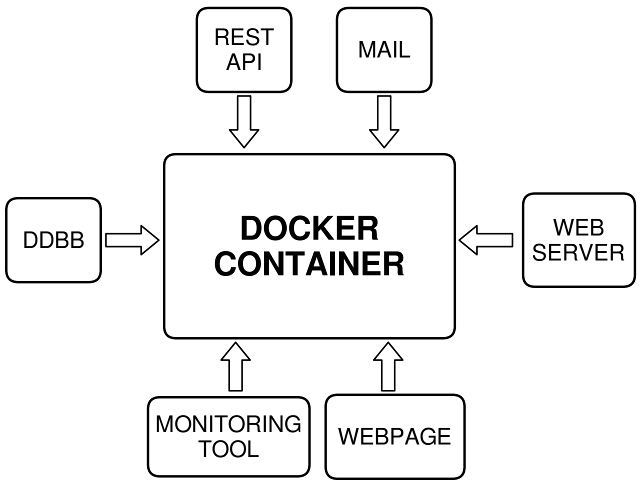


Figure 2.10.: Application encapsulation in Docker

Docker can be relevant overall in projects which include packaging and deployment of applications automation, creation of lightweight PaaS environments, automated testing and integration, scaling web apps deployments, databases and backend services. The Docker engine runs a daemon service in the host machines that runs the containers and store the images locally. The daemon can be used by a command line client tool or by a remote API. Besides, other components of the docker engine are explained: Docker images, the base images which contain the environment to launch containers, and Docker registries, repositories where these images are stored.

Docker Images

As we have already said, *Docker images* are simply templates which contain everything required by the application and from them, containers are launched. An image is a read-only template and it could contain an Ubuntu operating system with a Database, Web server or other kind of application application already installed. Docker images can be created, updated or removed. Moreover, they can be stored locally or inside public or private repositories (Docker registries), Figure 2.11. Besides images can be uploaded to these Docker registries where also other people's images can be downloaded and used. In another way we could say that images must be built in order to hold applications, containers run from images, and images can be stored in public or private Docker registries [39]. This structure is represented in Figure 2.11.

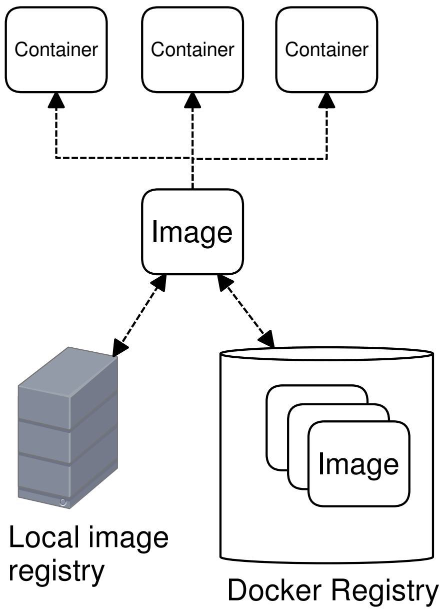


Figure 2.11.: Image storage in Docker

Docker Containers

Docker containers are the main component in Docker and they basically more or less like a directory which contains everything that it is required to run a determinate application.

These containers are created from Docker images, which are the base templates where the dependencies have been previously installed. They can be run, stopped, restarted or deleted anytime and each one runs in isolation with the rest of containers. Moreover, containers has a mechanism to share environment variables between two or more containers. They can be linked and share these variables if it would be necessary.

The complete Docker engine is designed to spend as few resources as possible [32, 40]. For instance, with a traditional VM, each application, each copy of an application, and each slight modification of an application requires creating an entirely new VM, whereas all of the containers share the same operating system, binaries and libraries, and do not need a new VM. For example, if several copies of the same application must run on a host, the shared binaries or OS must not be copied again, with a few megabytes a new VM is launched. In addition, due to they utilize the host operating system, restarting a VM does not mean restarting or rebooting an entire operating system. Hence, containers are much more portable and much more efficient for many use cases, and they can be compressed and migrated to other hosts rapidly.

Docker Registry

Docker registries are repositories where container images are stored. A registry can be public or private and it allows the collaboration with the community or a team. Images can be images which you have built by yourseft or you can make use of images that other people have previously created. Once a Docker image is built, it can be pushed to a public or private registry and then be searched, removed, updated, or be the base of other images.

Docker includes a huge Docker repository called *Docker Hub* [39]. Docker Hub provides both public and private storage for images. Public storage is searchable and can be downloaded by anyone. Private storage is excluded from search results and only authorized people can pull images down and use them to run containers. This huge registry includes at the moment thousands images created by the community with many different shapes and properties that can be really useful for other developments.

2.3.2. CoreOS

CoreOS [2] is a minimal operating system based on Linux that enables the development of resilient large-scale applications. This operating systems is designed to run thousands of Docker containers which contain different kinds of applications in a cluster deployment(i.e. databases, web servers, caching, auth, etc). This new OS allows companies like Google, Facebook, Netflix and Twitter to run their services at high resilience.

After a service has been built inside a Docker container, as we have already mentioned in Section 2.3.1, it can be launched with Fleet [41] within a cluster and connected with other services by Etcd [42], which may store information about the already deployed services, addresses, ports, etc.

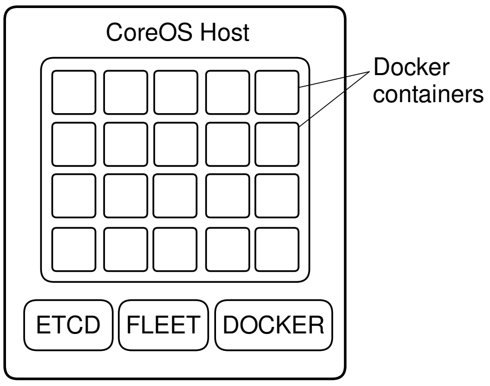


Figure 2.12.: CoreOs host with Docker [2]

In this OS the high availability can be also achieved by deploying services in different machines or regions. This can be easily performed by Fleet [41] which includes many interesting features for Cluster computing and management. It also includes a service discovery feature based on Etcd [42], you can locate easily where services are running within the cluster and be notified when something changes. Essential for a complex, highly dynamic cluster.

CoreOS can run on a local machine or on most cloud providers or platforms including Vagrant [43], Amazon EC2 [13], VMware [28] and OpenStack [17]. Clustering is platform independent and it works across platforms. Hence, services may be moved easily and without any problem from one machine or from a Cloud platform to another one. Before going into detail with Fleet and Etcd, we want to introduce some previous concepts related to cluster deployments: failover and cluster computing.

Failover

Failover [3] is the capacity of a complex system to adapt automatically in case of failures or abnormal situations without human intervention. Load balancing and replication are methods to get this and a continuous monitoring of the system components is required to fault detection in hardware or software. Thanks to them in case of a failure is detected, the system acts consequently and tries to fix such a problem automatically.

Figure 2.13 shows a failover scenario handled by a router, and two servers, a primary server, which processes the requests in a normal situation, and a failover server, which processes these requests in case of failure. Also a shared memory system is installed between the two servers to make accessible the information to both of them. The router forwards the traffic through the primary server, but if it detects that the link or this server are down, it redirects the load through the failover server.

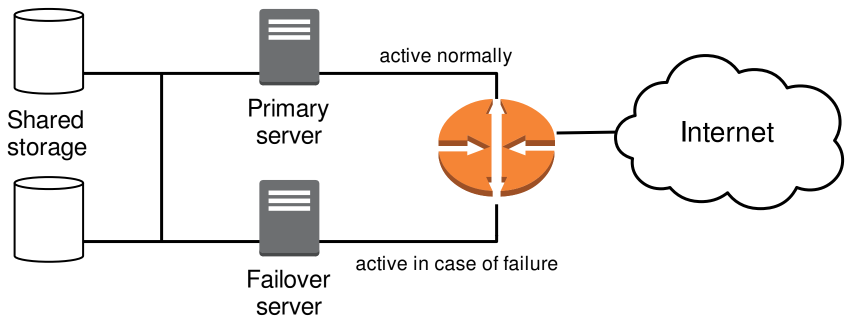


Figure 2.13.: Failover situation with replication and shared storage [3]

Cluster Computing

A cluster [3] is a group of connected systems working together as a single entity. Each one of the cluster members are called *nodes*. This is actually handy because it gives an abstraction of where the things are been executed and you only need to know if there are running or not. A cluster usually comes linked with the other concepts: replication, load balancing and failover techniques. The aim of using all of these in conjunction is the development of high availability systems.

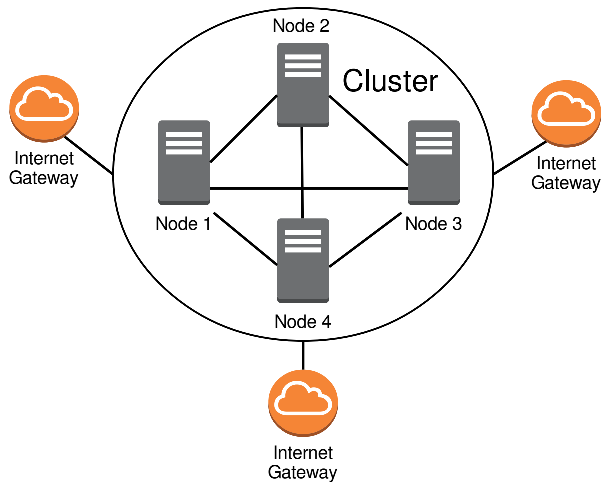


Figure 2.14.: Cluster of servers interconnected [3]

Fleet

Fleet provides the ability to launch and manage containers inside a cluster presented as a single init system. In order to belong to the cluster, containers must run inside what is called *units*. Units are just services based on Systemd [44] which specify the Docker containers running configuration and other interesting parameters. More or less Fleet can be seen as an extension of Systemd that operates at the cluster level instead of the machine level. Systemd is a single machine init system whereas Fleet is a cluster init system.

Fleet receives the unit files and schedules them into machines running within the cluster, as we can see in Figure 2.15. Inside of a unit file some rules can be declared, possible conflicts or preferences about where units must be located. For example, you can specify in which machine the unit must run or if a unit cannot run in the same machine than other one. With this feature and others Fleet is able to deploy high availability services. For example, it can achieve this by ensuring that some services are not located on the same machine, availability zone or region.

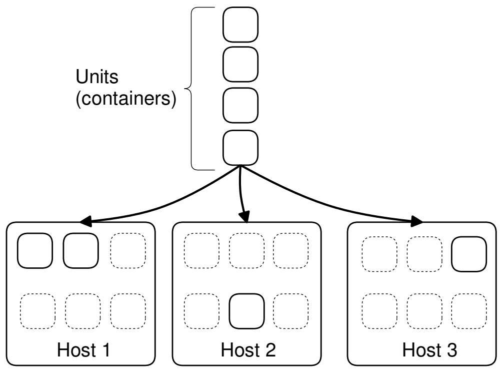


Figure 2.15.: CoreOS cluster scheduling

Fleet has two parts: an engine and an agent. The engine is responsible for job scheduling and bidding. It reacts to changes in cluster size. Scheduling logic is equally distributed between many fleet engines within the cluster and in case of one of the machines fails or is rebooted, the cluster keep going and it redistributes its units into the other nodes of the cluster. A representation of that can be seen in Figure 2.16. As the agent runs on each CoreOS machine and bids for jobs on behalf of the machine. Once a unit is assigned to the cluster, the agent starts the unit file and continually relays the state reported by systemd into fleet.

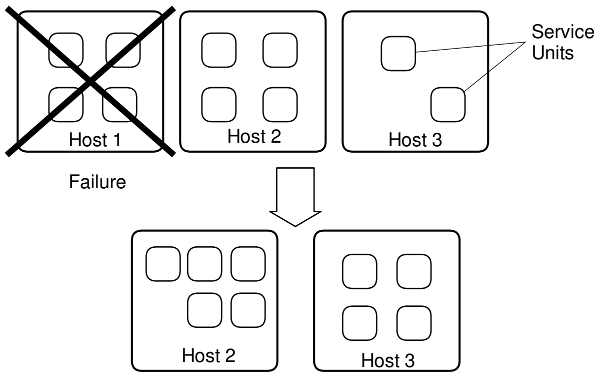


Figure 2.16.: CoreOS failure recovery

We can see that Fleet is a powerful tool included in CoreOS. Fleet easies difficult tasks like discovering the machines running within the cluster, re-scheduling on failure, distributing the units as desired onto the cluster,etc.

Etcd

Etcd [42] is an open-source distributed key value store that enables the communication between the services and machines along the whole cluster. Etcd runs in each machine on the cluster and an endpoint is provided, which is used for service discovery or reading/writing configuration values. In other words, Etcd can be used as the central registry of the cluster. The key-value pairs are available by a simple JSON/HTTP API accesible from the host machines or from inside the deployed containers.

Applications can read and write data into Etcd and therefore, they can share important information for the cluster orchestration like database connection details, cache settings, feature flags, ports, IP addresses, etc. For example, as it is show in Figure 2.17, a system with a database and a frontend server are launched in a CoreOS cluster. The database registers itself into Etcd and in that way, the frontend server is able to locate it checking its location via Etcd. Besides, we can notice that both components are located in different machines.

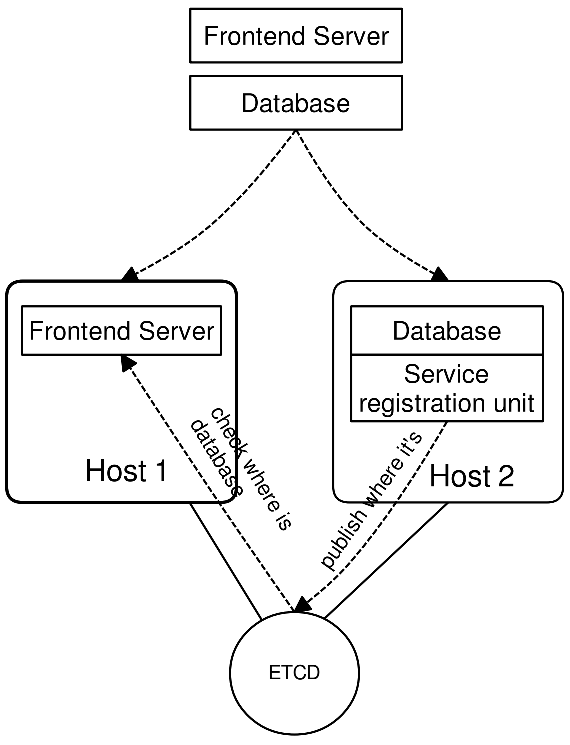


Figure 2.17.: Service discovery with etcd

Figure 2.17 shows an additional component which registers the database service in the Etcd store. This is done in order to keep service registration logic outside of the main service. For example, *sidekick* units can run aside the main unit, which store all the necessary information into Etcd. An application not only write or read an entry, it can listen if a the value of a key has changed. With this possibility sophisticated orchestration can be performed by flags or other mechanisms. The keys support also TTL (Time-to-Live) field, and hence, they can expire after a while. Additionally, Etcd is used for coordination between engines and agents as well.

A more advanced example can be service discovery for a proxy. Each container can announce itself to Etcd after starting up as it is shown in Figure 2.18 and a proxy, which is listening for changes, can adapt automatically to the new situation. The proxy reconfigures itself automatically and it starts to send traffic to the new container. This example shows a small proof of how scalable can be a system running on CoreOS.

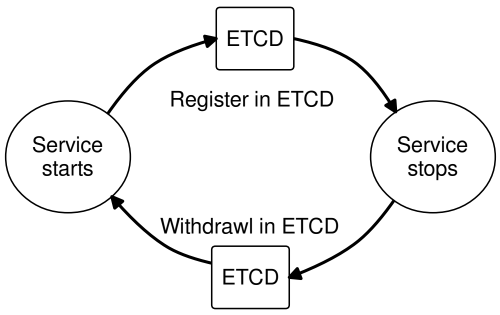


Figure 2.18.: Etcd autoregistration [4]

2.4. Middleware

The terminology *middleware* alludes to an intermediate software layer between the operating systems and distributed applications whose communication is performed via network interaction. The main aim of this model is facilitate the integration between different components running on diverse platforms. This model appeared as a approach to solve enterprise deployment's problems. In this kind of deployments, where many different components, protocols or OS are involved, communication and integration can be really difficult to achieve. The middleware tier tries to mask the underlying networking complexity and heterogeneity of the whole system and in that way, applications in the upper layer can be managed and programmed easily.

For example, huge enterprises usually have more or less autonomous suborganizations, which work independenly and provide different kinds of data sources and applications. The integration between all of them can be really difficult because for instance, they have different data formats or database sources. We could say that protocols are like languages, both speakers must talk the same language, unless they can communicate properly. For example, in large banks thousands of application subsystems must be integrated. The unstructured

[4] `https://coreos.com/assets/images/media/container-lifecycle.png`

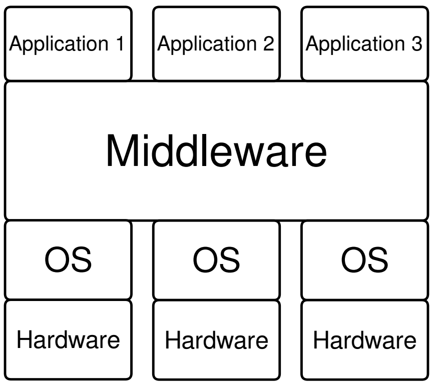


Figure 2.19.: Middleware Layer stack

data can be obtained from internal or external information sources and communication with another enterprises should be realized. In this scenario, a integration solution between every pair of components must be potentially implemented. However, with a middleware solution, just a interface for each one of the protocols is additionally implemented.

By this way, a middleware service is defined by the API and the supported protocols, that gives it the possibility to interconnect different kinds of external services. The middlewares are distributed systems, and they normally include a *client* and *server* parts. The client part supports the service's API running in the application's address space, and the server part, includes the the service's main functionalities and typically runs in another different system. In addition, middleware design includes another aspects, like QoS, information security or intermediary functions which transform or adapt the exchanged data.

The concept of middleware exposed is a really general one, because nowadays its scope is really broad. In the following section, Enterprise Service Bus (ESB), a middleware solution which follows this philosophy, is explained with more detail,

2.4.1. Enterprise Service Bus

An ESB [4] is an architectural pattern created in order to approach integration challenges in a more flexible and constructive way. Its principal functionalities are oriented specially to aspects such routing, transformation, security and orchestration. The predecessor of the ESB was the Enterprise Application Integration (EAI) [45] based on a hub-and-spoke model. This model was a first approach to the problems presented in the point-to-point model. The hub-and-spoke model is a centralized architecture, where the main component is a hub or broker which process all the exchanged data. Unlike in this model, the ESB is a bus model with a distributed architecture, in which each functionality can be implemented by several physical separated functions. Another difference between EAI and ESB systems is the use of open standards. EAI products like WebSphere Message Broker [46], TIBCO BusinessWorks [47], and Sonic XQ were mainly based on proprietary technology to implement messaging functionality and transformation logic. ESB products are based on open standards, such as Java Message Service (JMS), XML, and web services standards.

An ESB can be used as an integration platform that enables existing IT assets and applications to be exposed as services, Service-oriented architecture (SOA). In order to understand well the high-level benefits of a ESB, we are going to describe our initial scenario and its possible problems.

This first scenario is a point-to-point model where every component must communicate with the other components to get data. For instance, an application that provides information of possible public transportation routes to the user. The user can access this data through a Web browser or a smartphone. The application have a main logic component which must communicate to a set of public transportation web services, the Google Maps service, a database and user interfaces. This component must be able to establish this communication with each one of these interfaces in order to provide or get the required information. Besides, the user interfaces must communicate with this logic component. As we can see in Figure 2.20, to enable the communication between each pair of components, a new integration solution must be implemented. In case of one of the interfaces or the system architecture change, several modifications in the rest of the components must be done. Besides between every pair of components the information must be transformed to a data format that both can understand. Therefore, a custom integration solution must be both, developed and maintained. The complexity and maintenance cost increase when a new component is added to this application landscape. Hence, the main problem that we have to face is an integration problem.

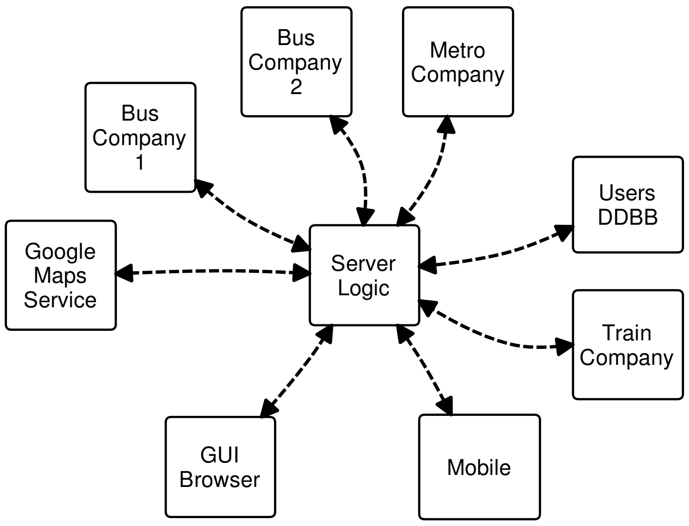


Figure 2.20.: Public transport application with a point-to-point model

In this context is when we think in a centralized solution such as an ESB solution that could increase the flexibility and simplicity of the final products. Another reason is that this landscape is a really heterogeneous when it comes to technologies and protocols. When you have to deal with many different protocols, like JMS, FTP, HTTP, SOAP, SMTP, and

Problem	Solution
Integration between application	Integration solutions can be time-consuming and costly. With an ESB this integration is performed just once with one of the different interfaces that it offers.
Heterogeneous environment	An ESB can deal with a lot of different technologies and protocols in a centralized way.
Cost of ownership	The maintenance and management costs are the most important in software development and with a central solution both could decrease notably.

Table 2.1.: Problems and solutions by ESB

TCP, it is difficult to implement new integration solutions between applications. An ESB provides protocol or technology adapters, which make it easy to deal with a heterogeneous IT environment. Another important reason is the reduction of the total costs of the whole application. In a point-to-point model, the management and maintenance of all the integration points can be really time-consuming and therefore, expensive.

In a environment with the proposed ESB centralized model, all enterprise components are connected to the ESB bus and communicate though it. Each component must implement at the very beginning a solution to communicate to the bus. However, if afterwards another of the components changes, the rest do not have to create a complete new integration solution. E.g. the Google Maps changes the JSON response messages for a XML ones. In this case, the ESB transforms the messages from XML to a JSON and the rest of the components of the system do not notice the change.

In Section 2.4.2 we explain further an ESB implementation, ServiceMix [48] and cite some of the most interesting alternatives nowadays of ESBs. However, after explaining some benefits an ESB solution can provide, the most important aspects an ESB solution must include are explained next:

- *Location transparency*. An ESB helps with decoupling the service consumer from the service provider location. An ESB provides a central platform to communicate with any necessary application without coupling the message sender to the message receiver.
- *Transport protocol conversion*. An ESB should be able to seamlessly integrate applications with different transport protocols like HTTP/HTTPS to JMS or SMTP to TCP.
- *Message transformation*. An ESB provides functionality to transform messages from one format to the other based on open standards like XSLT and XPath.
- *Message routing*. Determining the ultimate destination of an incoming message is an important functionality of an ESB that is categorized as message routing.

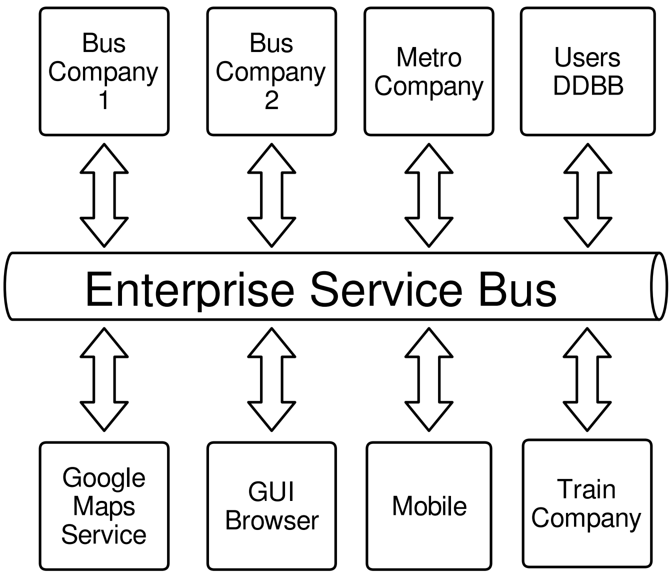


Figure 2.21.: Public transport application with a centralized ESB model

- *Message enhancement*. An ESB should provide functionality to add missing information based on the data in the incoming message by using message enhancement.
- *Security*. Authentication, authorization, and encryption functionalities must be provided by an ESB for securing incoming messages to prevent malicious use of the ESB as well as securing outgoing messages to satisfy the security requirements of the service providers.
- *Monitoring and management*. A monitoring and management environment is necessary to configure the ESB to be high-performing and reliable and also to monitor the runtime execution of the message flows in the ESB.

2.4.2. Apache ServiceMix

Apache ServiceMix [48] is an open-source ESB project with all the desirable features for a ESB such as routing, transformation, messaging, reliability, clustering, remoting, distributed failover, etc. It combines the functionality of a Service Oriented Architecture (SOA) [49] and modularity in order to enable the decoupling of applications and to reduce dependencies. ServiceMix enables application wiring using different protocols without new integration issues. It contains many powerful components including support to HTTP, JMX [50], CXF [51], BPEL [52], SOAP [52], SSH, FTP, etc. This ESB also provides a simple way to create your own components and services and to establish communication between them. Besides, ServiceMix can be embedded into a JEE application server such as JBoss [53], Oracle Weblogic [53] or IBM Websphere [54] .

As an ESB, ServiceMix is basically a bus which enables the communication between enterprise applications, and its internal structure follows the same pattern. Essentially, there is a internal router or broker used as a communication mean between the different components or bundles. In this ESB two different components are distinguished: Service Engine (SE) and Building Component (BC). BCs are used overall to external communication and SEs to add new services within the ESB. In Figure 2.22, this interaction between the different pieces of ServiceMix can be perceived.

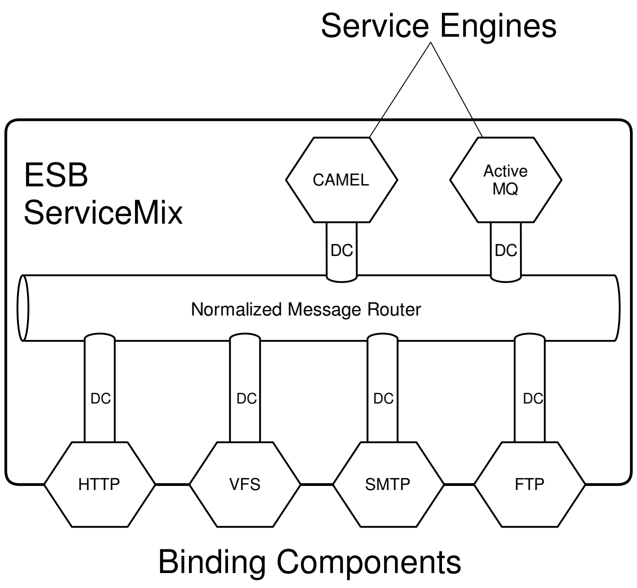


Figure 2.22.: Binding components, Service Engines and Normalized Message Router [4]

Throughout the next sections, we introduce the fundamental concepts involved in ServiceMix, its architecture and its most important components.

Binding Components and Service Engines

A *Service Engine (SE)* provides functionality to other components and can consume services by other components. Hence, it is an internal component in the ESB architecture. In contrast, a *Building Component (BC)* provides connectivity to existing applications and services that are located outside the JBI environment. For instance, if you want to communicate with existing applications, you must do that by using binding components. The same is true if you want to integrate with non-Java-based protocols or communication protocols (e.g., HTTP/HTTPS). Apart from providing access to external services, BCs can be used to expose internal JBI services to the outside world.

Normalized Message Router

The JBI components (SE or BC) do not directly communicate with each other, they communicate through the NMR. The components do not connect directly to this NMR, but instead

Concept	Description
Binding component	A JBI component used to consume and provide services to services outside the JBI container.
Service engine	A JBI component that can provide services to other JBI components and can also consume services provided by other JBI components.
Normalized Message Router	This component of a JBI environment takes part in delivering a message from one component to another component. This exchange always follows one of the standard message exchange patterns.
Delivery channel	The delivery channel connects a JBI component (a service engine or a binding component) to the normalized message router.
Service unit	This is an artifact that can be deployed into a running service engine or binding component.
Service assembly	A group of service units is called a service assembly.

Table 2.2.: ServiceMix main concepts

they use a Delivery channel (DC). It is the NMR's job to make sure that the messages are exchanged correctly among the multiple components in the JBI environment.

Endpoint and Services

ServiceMix distinguishes like other SOA solutions between two roles, consumers and providers, and each component can be a consumer or provider in a determinate moment if it makes use of a service or provides it. In ServiceMix a service cannot be accessed directly, a endpoint must be specified first. Each service must have at least one endpoint, but it can have many more. So when you want to consume a service provided by a JBI component, you need to know the name of the service and the name of the endpoint to invoke. This combination of a service and a specific endpoint on that service is called a service endpoint.

Service Unit and Service Assembly

The whole ServiceMix ESB works as a container thanks to the Apache Karaf kernel [55], and so the SE and BC do. They can be containers themselves to which resources can be deployed. The resources that you can deploy to such a container are called service units (SU). If you group these service units together, you can create a service assembly (SA). Figure 2.24 shows a service assembly can contain multiple service units. Once this service assembly is deployed to a JBI container, each one of the service units is deployed to its specific SE or BC.

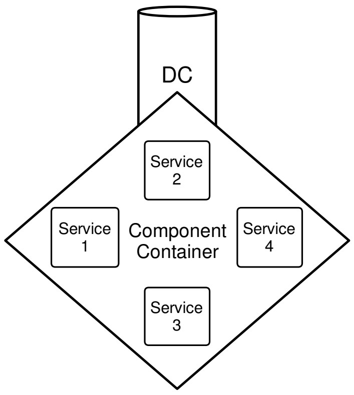


Figure 2.23.: Services executing inside a Component Container [4]

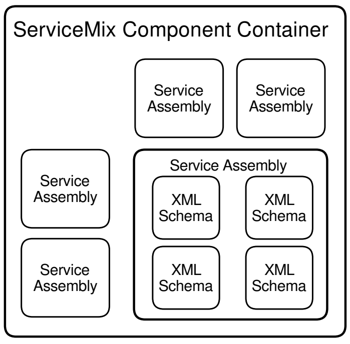


Figure 2.24.: Service Units and Service Assembly

Architecture

Despite of the previous versions of Servicemix were based on JBI, the last version is built over an OSGi-based architecture [56]. In Figure 2.25 we can see the main components that we are going to explain:

- *ServiceMix Kernel*: This kernel is the basis of ServiceMix and which is based on the Apache Karaf project [55] (an OSGi based runtime), handles the core features ServiceMix provides, such as hot deployment, provisioning of libraries or applications, remote access using ssh, JMX management, and more.
- *ServiceMix NMR*: This component, a normalized message router, handles all the routing of messages within ServiceMix and is used by all the other components.
- *ActiveMQ*: ActiveMQ [57], another Apache project, is the message broker that is used to exchange messages between components. Besides ActiveMQ can be used to create a fully distributed ESB.

- *Web component*: enables to start ServiceMix 4.2 embedded in a Web application.
- *JBI Support*: as the previous version of ServiceMix was based on JBI, Servicemix has compatibility with this components yet.
- *Camel NMR*: a couple of different ways for routing are given. You can use the endpoints provided by the ServiceMix NMR, but you can also use more advanced routing engines. One of those is the Camel NMR [58]. This component allows you to run Camel based routes on ServiceMix.
- *CXF NMR*: Besides an NMR based on Camel, ServiceMix also provides an NMR based on CXF [51]. You can use this NMR to expose and route to Java POJOs annotated with JAX-WS annotations.

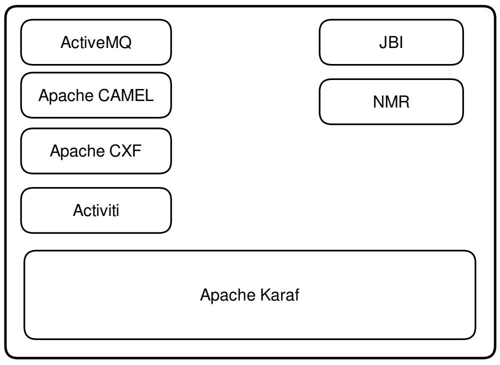


Figure 2.25.: Servicemix architecture

2.5. Monitoring

The monitoring has been an important research field for long time in different systems and levels. It has helped to improve systems with its ability to provide important information that could lead to fault detection, performance measures or QoS agreements compliance. But in the last years it has got a new dimension and challenges because of Cloud Computing rise. Aspects of Cloud Computing business model require a deep data gathering and analysis. For example, the process automation needs something to control the performed tasks, because without human interaction if a problem occurs, it must be detected somehow. Besides, the pay-as-you-go of Cloud Computing requires a resource usage measurement, because the billing is based on the amount used resources per time or capacity. Therefore, the monitoring is primordial for an efficient large-scale Cloud management and maintenance of the QoS and not only important for providers, but also for consumers.

In the article [59] monitoring is defined from a capability perspective as "*a process that fully and precisely identifies the root cause of an event by capturing the correct information at the right time and at the lowest cost in order to determine the state of a system and to surface the status in a timely and meaningful manner.*". As we can extract from this definition that monitoring of operational services should only impose a small performance overhead or workload and it should be non-intrusive to the business logic, as far as possible. The monitoring tool gathers information to make informed decisions, dynamical analysis and model calibration. Monitoring can enable early detection of QoS problems, such as performance degradation, and can deliver usage data for resource management. Such monitoring information is also required to check the fulfillment of service level agreements (SLAs). Therefore, a system's runtime behavior should be monitored and analyzed continuously.

A wide spectrum of monitoring tools can be found from general-purpose infrastructure (i.e. Nagios [38], Collectd [60] or Zabbix [61]) to cloud specific monitoring tools (i.e. the provider independent Nimsoft [62] or the provider dependent Amazon Cloud Watch [63] or Azure Watch [16]) and on all layers of a software system.

General purpose infrastructure monitoring tools typically follow a client-server model by installing an agent in every system to be monitored. Figure 2.26 shows this general architecture. Monitoring agents measure metric values from monitored components and send them to the monitoring server. The server stores the collected metrics into a database, analyses them, and sends alerts as e-mail or SMS. It may generate graphs, trending reports and SLA reports based on the monitored metrics retrieved from the database. Additionally some of them may use a hierarchical structure, organizing the agents in different levels. The child nodes can send their data to other parent node which stores or show them in a generated graph.

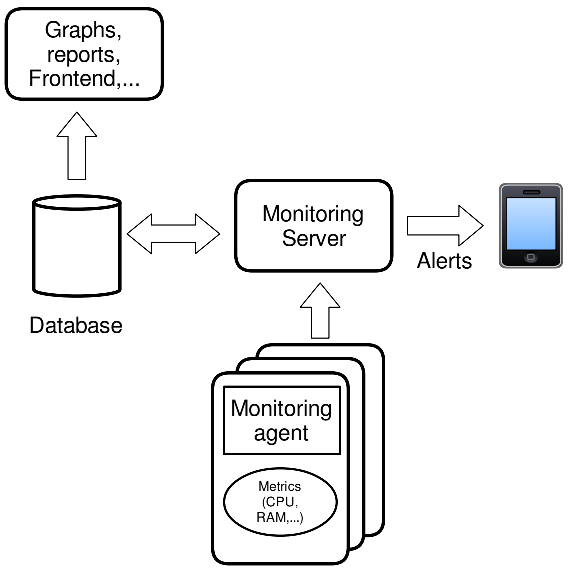


Figure 2.26.: General Monitoring tool with Client-Server model [5]

2.5.1. Desirable Cloud Monitoring Capabilities

In this thesis, we must find a monitoring tool which fits with the whole system environment requirements. Hence, the main features [59], which must be present in a Cloud general monitoring tool, should be cited in order to distinguish which exactly this projects requires.

- *Scalability*. In Cloud Computing thousands of nodes are usually deployed and in this large-scale scenario a way to manage the resources and a tool which adapts automatically to the grow of the system should be achieved. The monitoring tool must not only be able to grow with the system, but it should be able to send the information as fast as possible.
- *Portability*. Due to the heterogeneous nature of Cloud environments a monitoring tools should be able to be moved from a platform to another different one.
- *Non-intrusiveness*. As we have already mentioned, the monitoring tool should not have a big impact in the current system performance and should be not-intrusive to the business logic, because overall the pay-per-use billing of Cloud.
- *Robustness*. The dynamical and changing environment of the Cloud infrastructure where elements are added or removed continuously forces the monitoring tool to be able to face these new situations and keep providing a reliable monitoring data.
- *Multi-tenancy*. One of the basics in Cloud Computing is multi-tenancy, where many tenants access to the same virtual or physical resource. To support this, the monitoring tools must include a mechanism to maintain the isolation and concurrency between tenants. For instance, tenants only must be able to access the information that is addressed to them, but the administrator must be able to access to the whole data.
- *Interoperability*. Nowadays, there are lot of different Cloud providers, and communication between independent and heterogeneous systems must be performed. Therefore, the monitoring tool should be able to exchange information with different Cloud platforms.
- *Customizability*. There are presently numerous Cloud service offerings and many providers are seeking ways to deliver unique services to their customers by allowing them high customization flexibility. Considering the large number of customers, providers must be able to manage the service customisation of each customer, for example, by granting customers the ability to choose the metrics to be monitored for their service. Thus, to realize this goal, efficient monitoring tools should possess this capacity.
- *Extensibility*. With the rapid growth of Cloud computing, there are continuous changes and extensions to technologies. Monitoring tools should be extensible and be able to adapt to new environments or collect new metrics which could be interesting in the future.
- *Shared resource monitoring*. Cloud computing make extensively use of virtualization, and because of that, the monitoring tool should collect information from the virtual and the physical resources or systems.
- *Usability*. A monitoring tool should be highly usable and facilitate deployment, maintenance and human interaction.

- *Affordability*. Due to Cloud Computing chases a reduction of the total costs, the monitoring tool should not increase noticeably the costs. Here is where open source monitoring tools and their positive impact in the final costs enter the scene.
- *Archivability*. The data should be stored in a persistent way for analysing and identifying the origin of issues. Hence, the monitoring tool should be equipped with a permanent database.

3. Related Works

In the following chapter, similar works related to the fundamental concepts involved in this thesis are presented. However, we must first explain briefly the behaviour and features of the system we aim to build.

Our system must be a framework to provision specialized middleware components. As we want to enable the use of the framework by different users, multitenancy [64] must be necessarily supported. Moreover, the ESB instances must run in isolation and in a lightweight way. These instances must be preconfigured ESB instances encapsulated inside Docker containers and must be accessible to be launched at anytime. Hence, they must include all the required dependencies and libraries. In addition, they must be reusable to be used by different users in order to enhance the performance of the whole system.

On the other hand, monitoring is important in order to ensure a specific QoS to the end users and to control the performance in Cloud systems. Therefore, our framework must include a monitoring tool able to collect the data from both, VMs and host machines. It must gather data about the CPU usage, memory, etc.

In summary, we must take a further look in similar projects related to virtualization deployment (overall those related to container deployment), middleware deployment and characterization, and cluster deployment of containers. Additionally, the main Cloud providers and how they enable the container virtualization deployment in their infrastructures is presented. These platforms are Google App Engine (GAE) [15] and Amazon Elastic Beanstalk [65], explained respectively in Sections 3.8 and 3.9.

3.1. Kubernetes

The first related project we want to introduce is Kubernetes [66]. This project is an open source implementation of container cluster management and orchestration. It has been developed by Google and it is a similar approach to Omega [67], the project used within Google but not open source.

Kubernetes enables the deployment of Docker containers into a fleet of machines and provides health management and replication capabilities. These features look interesting, but one that has drawn our attention is the introduction of *pods*, a logical service composed of several containers running in the same machine as a single entity (see Figure 3.1).

Another interesting feature is that the pods can be organized by labels or keys defined by the user, and in that way a specific group of pods can be found. For example, if hundreds of pods

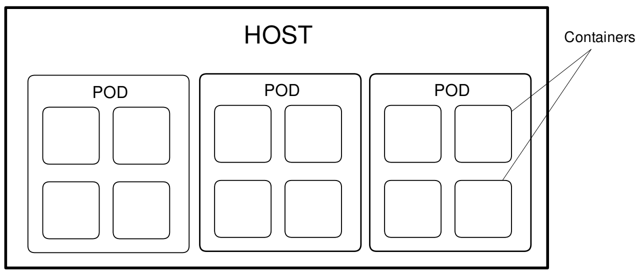


Figure 3.1.: Containers running inside pods

are running at the same time with different functionalities (i.e. Web server, databases, DNS, etc.), the user could find the pods which include a database by their label *type=database*.

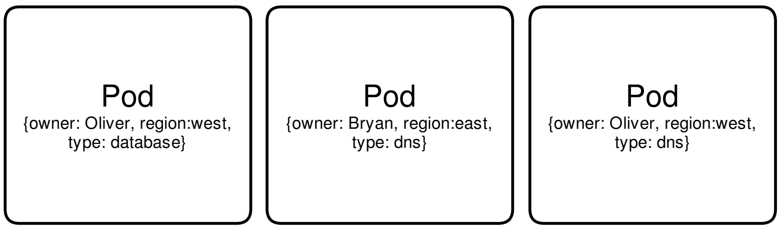

Pod
{owner: Oliver, region:west, type: database}
Pod
{owner: Bryan, region:east, type: dns}
Pod
{owner: Oliver, region:west, type: dns}

Figure 3.2.: Pods organized by labels

As CoreOS does, Kubernetes leverages ETCD (see Section 2.3.2) to achieve an advanced level of cluster orchestration. This distributed key-value store is used to coordinate all the components present in its architecture. Despite of using ETCD, Kubernetes does not run over CoreOS and it does not use Fleet (see Section 2.3.2) either. It contains another components to do a similar job as Fleet for orchestration. Architecturally Kubernetes is also interesting. It is built as a collection of pluggable components with different responsibilities (i.e. schedulers, storage systems, load balancers, etc.) running within services as Fleet units in CoreOS.

This project is powerful enough to enable the Container deployment we want, but we think that some small tasks must not be performed necessarily inside a container. For example, in our approach explained later, a container with a preinstalled ESB instance has aside another component to register all the important information about it in the cluster. With Kubernetes, this would require another entire container running including its OS and dependencies. Besides, the configuration and installation process on Kubernetes is more complex than on CoreOS, which is used in our work in other to enable a quick deployment of new host machines

3.2. Deis

Another interesting project is Deis [68], an open source lighweight PaaS able to deploy, scale and manage applications on different servers. Deis packs, ships and runs applications automatically inside Docker containers running within a CoreOS cluster. It can deploy any language or framework using Docker, but it also includes Heroku [69] build packs for multiple languages (i.e. Ruby, Python, Node.js, Java, Go, etc.). Besides, it can run on any system which supports CoreOS including the user computer and public and private clouds of main Cloud providers.

Deis is able to build and pack applications automatically into a Docker image, and it distributes them as Docker containers across the cluster. Besides, it enables to change the environmental variables of the application. It also includes a routing and load balancing component which routes the traffic into different Docker containers which hold the same application. Something really useful for scaling an application automatically.

This project is really powerful and its philosophy fits with what we want. However, it seems that Deis only allows to publish one port per container, and if we want to encapsulate our ESB instances inside containers several ports are required. Besides, it seems that you cannot specify in which host or aside which container it must be located, something that could be important in some cases.

3.3. GeMS

GeMS [70] faces the problem about general-purpose specialization process. A general middleware process may add excessive footprint, latency and resource usage in general. Therefore, a specialization process of the middleware is crucial to deploy an efficient middleware instance. This work focuses on identifying the components the application might require at the moment and predicting which might use in the future, because the middleware configuration process normally can be extremely tedious and complex. This project tries to discover the perfect middleware specialization for an application and in that way, not being forced to create and configure the middleware in the future.

In contrast with this project, the platform we want to implement skips this problem, because our platform is able to build and provide any specialized middleware at anytime. In our platform, the developer must only identify which components his application requires (i.e. HTTP, SQL, etc.) and he gets automatically a middleware instance (ESB in our case) with these component already installed. In case of his application changes, he must only ask for a new instance with the features he needs.

On the other hand, in case of the middleware does not require a component anymore, the middleware is not normally configured again because it can be really costly. But in our case, this action can be perform quickly just removing the previous instance and creating a new one without the component, and hence, it may save resources and run more efficiently.

3.4. DPRS. Dynamically Programmable and Reconfigurable Software

The approach of this work [71] is based on what they refer as *middleware externalization*. The externalization is a technique to explicitly externalize the state, logic and internal component structure of middleware services. This is used to enhance the configurability, update ability, and upgradeability of the middleware.

They have built a tool to control the middleware state, logic and add or remove component even in runtime execution. They call these components Micro Building blocks (MBB), which include the smallest functionally of the system and can be loaded dynamically too.

The idea about externalization is really interesting also for this thesis, because it might be a first step in order to create the non-reconfigurable ESB instance. If we identified all the variability points of the middleware, we could externalize them to create dynamically ESB instances.

In the document [71] is also specified that despite the flexibility, a reconfiguraable middleware offers equivalent performance to a non-reconfigurable middleware. Therefore, if we were able to create a bunch of non-reconfigurable middleware instances, we could get a similar performance.

3.5. Automatic Middleware Deployment Planning on Clusters

After the Cloud computing burst and the presence of computing in every corner of the world, a way to adapt the middleware deployment to manage heterogeneous resources must be found. The middleware must not only be mapped to existing resources , but it must offer the required QoS in every moment through the entire system.

In the paper [72], they do not face this problem directly, the focus on how to carry out an adapted deployment on a cluster with hundreds of nodes. An approach for automatically determining an optimal deployment is presented. This solution is for hierarchically distributed middleware services on clusters and its goal is to optimize steady-state request throughput. Moreover, it is only valid for a homogeneous resource platforms with a given size, and it tries to determine the optimal middleware deployment, how many nodes should be used and in which hierarchical organization must be disposed in order to maximize a steady throughput.

In contrast with this solution, our approach should be able to run in a heterogeneous platforms and with a dynamic number of nodes, because at the present moment, two of the premises in Cloud is the efficient usage of resources and the elasticity of the systems. For example, the business model of Cloud computing is *pay per use*, and if some of the nodes of the system are not used, the provider looses money.

On the other hand, their solution depends on a predicted workload, and after the initial middleware deployment, the system cannot adapt to new scenarios. The solution we propose enables the deployment of new components in case the workload changes, or their removal if additional resources are not longer required.

3.6. Project Atomic

Project Atomic [73] is another multi-host container deployment project similar to CoreOS [2] released in April 2014 and based on a Operating System called *Project Atomic Host*. Its core component is called *Geard*, an open-source project for installing and linking Docker containers into Systemd, and for coordinating those Docker containers across hosts. The most interesting function is that it is able to wire containers between hosts. GearD introduces a ip-based container linking framework which allows the container to talk to a specific IP address that is afterwards remapable. The IP addresses can be re-routed but they containers remain wired.

As Deis [68] does, Geard allows developers to simply point at their application source in a repository or a Docker image to get a newly built Docker container whichs runs the application. Another interesting features is the OS updates, its updates are applied atomically in one operation and it allows to rollback to older versions if necessary.

This projects looks really promising, but it is under high development and no-production ready. However, it does not fit so well with what we want to do in our framework. For example, wiring between ESB instance is not necessary for us by the moment and it does not include any distributed registry to store our system state or image storage. In addition, Project Atomic does not come with any component to schedule and reschedule the containers.

3.7. Panamax

Panamax [74] is an open source project which enables a entire container base application deployment. It is based on CoreOS and its most interesting idea is what they called *templates*, a group of linked Containers working together that can be organized by categories or tiers. It contains a Web UI to create and control these templates and it works jointly with the Docker Hub. The user can search among all the images in the Docker Hub or in the Panamax marketplace to launch new containers or create new applications. In this interface the user is able to control container parameters such as environment variables or linked containers.

The target of this project is different to ours. Panamax tries to easy the whole application deployment, but we only need to deploy individual ESB instances in a elastic way. Besides, it does not include any of the additional components we need in our framework for the ESB instance management (i.e. registry, controller, etc.). Moreover, this project is still only in a Beta phase.

3.8. Google App Engine

In the Gluecon conference, the senior staff software engineer, Joe Beda claimed :"Everything at Google runs in a container. We start over two billion containers per week." [75]. Therefore, it seems Google is working hard with container virtualization solutions. Some time later, Google enabled the deployment of Docker images inside Google App Engine (GAE) [15]. This enables developers to package their application with all, including OS and dependencies and run in its platform. A serial of VMs, which are only Debian images with a preinstalled Docker, are used to host the container execution.

As we have said, Kubernetes in Section 3.1, is a project by Google, and it is optimized to run in GAE. It is able to deploy containers into a fleet of machines, and to assure health and replication capabilities in the cluster, and as well as allows containers to connect to another one and the outside world.

Additionally, Google released another interesting open source project called cAdvisor [76]. This project enables detailed statistics, both instantaneous and historical, regarding resource usage for containers and host machines. This can be really useful for this thesis, and in the following sections we take a further look into cAdvisor in order to determine if it offers all features the monitoring tool of our framework must contain.

3.9. Amazon Beanstalk

Amazon Elastic Beanstalk [65] now supports deployment of Docker containers. The applications inside Docker containers can be launched with all the features that Beanstalks enables like scalability, load balancing, etc. These containers can be launched from existing images, from the Docker Hub, or from new images built inside the Amazon platform. The developer must just upload his code in a zip file with his application and the build file of the Docker image, or in case of using an existing one, just a description file pointing to the desired image and parameters.

4. Concept and Specification

For this work, we must develop a *"framework which enables the creation, provisioning, and runtime of light weight specialized ESB instances to fulfill a given set of application communication requirements"*. Such a framework can be potentially used in PaaS offerings enabling the provisioning of the necessary underlying communication infrastructure required by the application. The requirements that should be achieved in this work for the proposed framework are detailed next, and in the same way, the use cases that it must cover. Additionally, a system architecture is proposed.

4.1. System Requirements

In this section we expose the different requirements the framework must fulfill. We distinguish between functional and non-functional requirements, and later we explain further the requirements that the monitoring part of the framework must include.

4.1.1. Functional Requirements

- *ESB instance configuration*. The system must provide a way to run different ESB instances with different configurations. The configuration must include parameters to define the already installed components in the ESB and resource usage (i.e. maximal memory, maximal CPU, or ports that the ESB instance could use).
- *Coarse grained ESB instance configuration*. The information about configuration, IP addresses, state of the ESB instance or ports should be accessible.
- *ESB instance migration*. The instance must be encapsulated, stored and relaunched rapidly. This is also important for some data that must be persistent in case of a posterior deployment.
- *ESB instance based management*. A mechanism to manage individually these instances should be enabled.
- *Wiring*. The provisioned ESB must support the integration with the main communication technologies and protocols (i.e. Web Services, SQL, SMTP, etc.) in order to provide a desirable platform for the application.
- *Remote access and control*. Remote management and usage of the framework must be also enabled. The administrator must be able to control the deployed system with simple but powerful tools.

- *Administration interfaces*. A REST API or Command Line Interface must be provided for the system administrator.
- *Global and coarse grained monitoring*. As we must deploy several ESB instances inside of containers running in a determinate host machine, the system must monitor in a container level and in a host level by a light and non-intrusive way.

4.1.2. Non-functional Requirements

- *Web scalability*. The system must be able to scale and work with a dynamic number of ESB instances and host machines.
- *Failure resilient applications that isolate themselves from common failure scenarios*. For example if one of the hosts goes down, the system must be able to recover itself and adapt to the new situation.
- *Multitenancy*. The must support multitenancy [64], defined in our case as *"the sharing of the platform at the same time by different tenants and their corresponding users without any kind of collision and with the same QoS"*.
- *Isolation between ESB instances*. The ESB instances should run in isolation in order to ensure multi-tenancy.
- *Easy installation*. The software must be easily installed and configured in order to deploy new host machines rapidly.
- *Low resource intensity*. Each component of the system should use the minimal necessary resources (storage, network, CPU, memory, storage, etc.). This is important because the new system must be able to have as many current users as possible and it might be deployed in another Cloud provider like Amazon EC2 where the user pays for the resources he uses.
- *Built with well supported technologies*. This feature must be included for future modification or extensions of the source code.

4.1.3. Monitoring Requirements

On the other hand, we want to specify further the monitoring requirements that our monitoring tool should fulfill. The monitoring tool we need in our system must gather the information about both, host machines and deployed VMs. It must be lightweight and not-intrusive with the rest of the framework and it must be able to collect and provide monitoring information about all the machines and VMs deployed in the cluster. All the features that must be included are the following:

- *Operating system level monitoring*. The monitoring will be present in the Operating System level, and it must collect information about CPU, memory and network usage.

- *Monitoring host and guest*. The monitoring tool must get information about the physical hosts and about the virtual machines which encapsulate the ESB instances.
- *Scalability*. The monitoring software must be able to add new machines rapidly without disturbing the rest of the system.
- *Light monitoring*. The monitoring overhead must not affect the performance of the whole system.
- *Export monitoring data*. It must be able to export in standard formats like JSON or XML.

4.2. Use Cases and Roles

In our system two kind of roles are defined, administrator and developer roles. The administrator is the provider of the platform and the developer is the one who specializes an ESB instance based on the application requirements. This section first provides a description of what each one of these roles can do within the platform. Finally, all the use cases that must be covered in the framework are described in detail.

4.2.1. Administrator Role

The administrator makes use of the following functionalities with total power, no matter who is the owner of the instance or image.

- *Image control*. The administrator must be able to create and remove the images which contain the specialized ESB instances.
- *Instance control*. The administrator can remove, create, run or stop instances with no restriction. Additionally, the administrator is able to migrate these instances to another machines in the cluster.
- *Information*. The administrator can list all the running machines and ESB instances in the cluster and their information. For example, he can see the location of machines and instances. The monitoring information must be also included.
- *User control*. The administrator can remove an user and all his elements (i.e. instances, configurations, etc.). He can add or update users in the system too.

4.2.2. Developer Role

The developer owns a set of ESB instances and configurations. He has complete control only over his own instances and configurations, not over other user's elements.

- *Instance control*. He can create, destroy, run or stop his own ESB instances.

- *Backups download*. The possibility of downloading the ESB instance data must also be included.
- *Information*. The ESB instances data containing ports, addresses, configuration and other parameters are given to the developer. The monitoring information about running ESB instances must be also included.
- *Remote access*. A method to access remotely to the instances must be provided. Through this remote access, the developer can manage his ESB instances.

4.2.3. Use Cases Description

From all the functional and non-functional requirements of the system in Section 4.1, we extract the uses cases that our framework must support. Figure 4.1 depicts a use case diagram and detailed information about each one of the referred use cases is given afterwards.

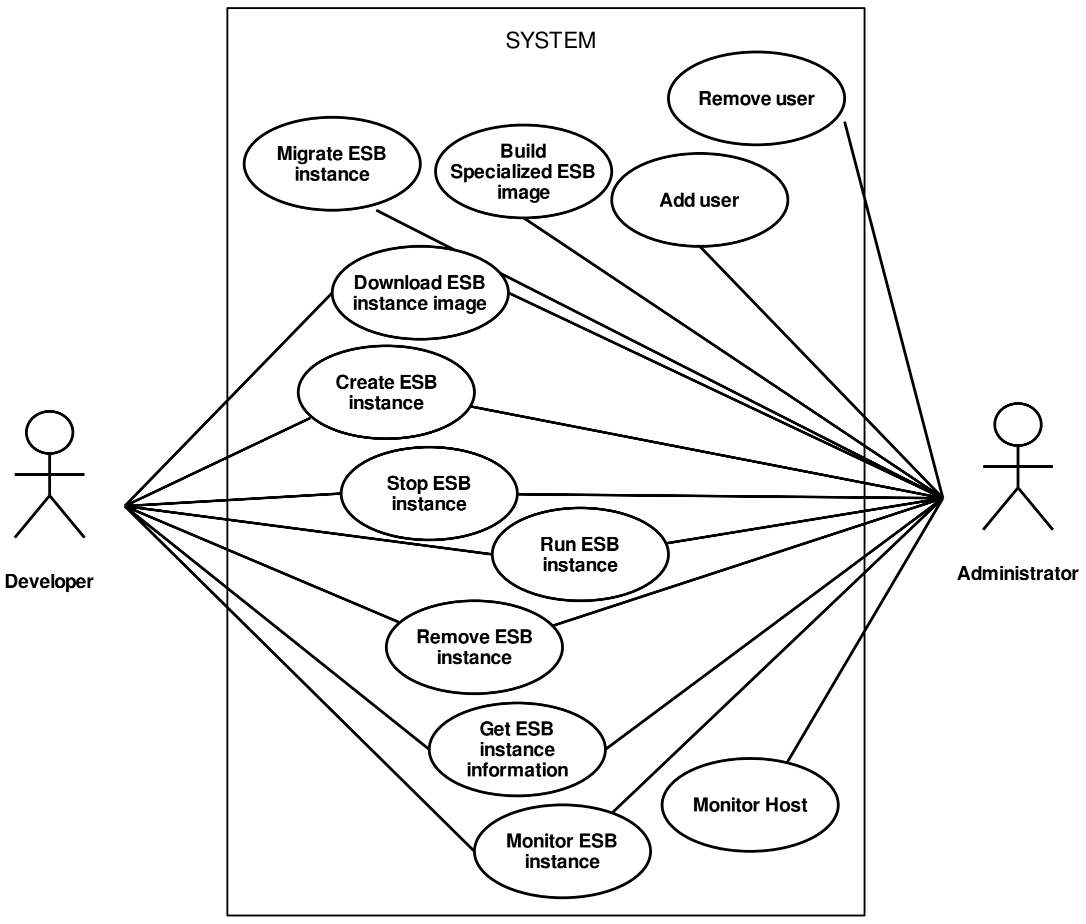


Figure 4.1.: Specialized ESB framework use cases

Name	**Add User**
Goal	The administrator wants to add an user to the system
Actor	Administrator
Pre-Condition	The administrator must provide the user credentials
Post-Condition	A confirmation message is returned
Post-Condition in Special Case	The request is refused and no other users have been added
Normal Case	1. The system receives a request with new user credentials. 2. The system checks if the user already exists. 3. The system adds the new user to the database.
Special Cases	1a. The user already exists. a) The system shows an error message and aborts. 2a. The credentials are not correct. a) The system shows an authentication error and aborts.

Table 4.1.: Description of Use Case *Add User*.

Name	**Remove User**
Goal	The administrator wants to remove an user from the system
Actor	Administrator
Pre-Condition	The administrator must provide the user name
Post-Condition	A confirmation message is returned
Post-Condition in Special Case	The request is refused and the user has been removed
Normal Case	1. The system receives a request with the user name 2. The system checks if the user exists. 3. The user is removed.
Special Cases	1a. The user does not exist. a) The system shows an error message and aborts. 2a. The credentials are not correct. a) The system shows an authentication error and aborts.

Table 4.2.: Description of Use Case *Remove User*.

Name	**Monitor Hosts**
Goal	The administrator wants to get the monitoring information of a running machine (i.e. CPU usage, memory usage,etc)
Actor	Developer or Administrator
Pre-Condition	The administrator must provide the ID of the machine
Post-Condition	A message with the monitoring information of the requested machine is returned
Post-Condition in Special Case	The request has been refused and no information is returned
Normal Case	1. A request with the machine ID is received. 2. The system checks if the machine exists and if the requester has permission over it. 3. The system shows the monitoring information about the requested machine.
Special Cases	1a. The specified machine does not exist. a) The system shows an error message and aborts. 2a. No correct credentials are introduced. a) The system shows an authentication error and aborts.

Table 4.3.: Description of Use Case *Monitor Hosts*.

Name	**Build Specialized ESB Image**
Goal	The administrator wants to build an image with some preinstalled components.
Actor	Administrator
Pre-Condition	The administrator must provide the desired configuration parameters
Post-Condition	A confirmation message is returned
Post-Condition in Special Case	The request has been refused and the ESB Image has not been created
Normal Case	1. A request with the specified configuration 2. The system builds the image with the specified configuration 3. The system shows a confirmation message
Special Cases	1a. The configuration is not valid a) The system shows an error message and aborts. 2a. The credentials are not correct. a) The system shows an authentication error and aborts.

Table 4.4.: Description of Use Case *Build Specialized ESB Image*.

Name	**Create ESB Instance**
Goal	The developer or administrator wants to create a specialized ESB instance in the platform
Actor	Developer or Administrator
Pre-Condition	If it is the developer, he must be already registered in the system. Otherwise, the administrator should specify the user to associate to the new ESB instance. A correct configuration for the new instance must be specified
Post-Condition	A new ESB specialized has been created in the system and associated to the specified user.
Post-Condition in Special Case	The request has been refused and the system has not built or added any new ESB instance
Normal Case	1. The developer or administrator request a new ESB instance with a specific configuration. 2. The system checks if there is a already image built for this instance in the whole system. In that case, it will associated one to user. Otherwise, the system builds the image first. 3. The new image is stored. 4. A new ESB Instance is created from the image, stored and assigned to its owner .
Special Cases	1a. The configuration specified is not correct. a) The system shows an error message and aborts. 2a. The requester does not have the necessary rights in the system or the credentials introduced are incorrect a) The system shows an authentication error and aborts.

Table 4.5.: Description of Use Case *Create ESB Instance*.

Name	**Get ESB Instance Information**
Goal	The developer or administrator wants to get the information of an existing ESB instance (i.e. ports, IP address, configuration, etc.)
Actor	Developer or Administrator
Pre-Condition	The requester must specify the ID of the ESB instance and he must have permission over it
Post-Condition	A message with the information of the requested ESB instance is returned
Post-Condition in Special Case	The request has been refused and the ESB instance is not built
Normal Case	1. A request with the ESB instance ID is received. 2. The system checks if the ESB instance exists and if the requester has permission over it. 3. The system shows the information about the requested ESB Instance.
Special Cases	1a. The specified ESB instance does not exist. a) The system shows an error message and aborts. 2a. The requester does not have the necessary rights in the system or the credentials introduced are incorrect a) The system shows an authentication error and aborts.

Table 4.6.: Description of Use Case *Get ESB Instance Information*.

Name	**Run ESB Instance**
Goal	The developer or administrator wants to run an existing ESB instance.
Actor	Developer or Administrator
Pre-Condition	The requester must specified the ID of the ESB instance and he must have permission over it
Post-Condition	The instance starts and it is ready to used on the system. The requester gets a response with the execution parameters.
Post-Condition in Special Case	The request has been refused and the ESB instance is not launched.
Normal Case	1. A request with the ESB instance ID is received. 2. The system checks if the ESB exists and if the requester has permission to launch it. In that case the instance starts.
Special Cases	1a. The specified ESB instance does not exist. a) The system shows an error message and aborts. 2a. The requester does not have the necessary rights in the system or the credentials introduced are incorrect. a) The system shows an authentication error and aborts.

Table 4.7.: Description of Use Case *Run ESB Instance*.

Name	**Monitor ESB Instance**
Goal	The developer or administrator wants to get the monitoring information an existing ESB instance (i.e. CPU usage, memory usage,etc.)
Actor	Developer or Administrator
Pre-Condition	The requester must specify the ID of the running ESB instance and he must have permission over it
Post-Condition	A message with the monitoring information of the requested ESB instance is returned
Post-Condition in Special Case	The request has been refused and no information is returned
Normal Case	1. A request with the ESB instance ID is received. 2. The system checks if the ESB exists and if the requester has permission over it. 3. The system shows the monitoring information about the requested ESB instance.
Special Cases	1a. The specified ESB instance does not exist or it is not running. a) The system shows an error message and aborts. 2a. The requester does not have the necessary rights in the system or the credentials introduced are incorrect. a) The system shows an authentication error and aborts.

Table 4.8.: Description of Use Case *Monitor ESB Instance.*

Name	**Migrate ESB Instance**
Goal	The administrator wants to move a running ESB instance to another machine.
Actor	Administrator
Pre-Condition	The administrator must provide ID of the running instance
Post-Condition	A confirmation message is returned
Post-Condition in Special Case	The request has been refused and the ESB instance is not migrated.
Normal Case	1. The system receives a request with the ID of the instance and machine. 2. The system checks if the instance exists. 3. The system stops the running instance. 4. The instance state is saved. 5. The system chooses the new destination machine. 6. The instance is downloaded in the new machine. 7. The system starts the instance in the new machine.
Special Cases	1a. The ESB instance is not found. a) The system shows an error message and aborts. 2a. The credentials are not correct. a) The system shows an authentication error and aborts. 2b. The ESB instance is not running. a) The system shows an error message and aborts.

Table 4.9.: Description of Use Case *Migrate ESB Instance*.

Name	**Stop ESB Instance**
Goal	The developer or administrator wants to stop a running ESB instance.
Actor	Developer or Administrator
Pre-Condition	The requester must specify the ID of the ESB instance and he must have permission over it
Post-Condition	The instance is stopped.
Post-Condition in Special Case	The request has been refused and the ESB instance is not stopped.
Normal Case	1. A request with the ESB Instance ID is received. 2. The system checks if the ESB exists and if the requester has permission over it, the instance state is saved. 3. The instance is stopped. 4. The system shows a confirmation message.
Special Cases	1a. The specified ESB instance does not exist. a) The system shows an error message and aborts. 2a. The requester does not have the necessary rights in the system or the credentials introduced are incorrect a) The system shows an authentication error and aborts. 2b. The ESB instance is not running. a) The system shows an error message and aborts.

Table 4.10.: Description of Use Case *Stop ESB Instance*.

Name	**Download ESB Instance Image**
Goal	The developer or administrator wants to download the image of an existing ESB instance.
Actor	Developer or Administrator
Pre-Condition	The requester must specify the ID of the ESB instance and he must have permission over it
Post-Condition	The instance is downloaded in a compressed file.
Post-Condition in Special Case	The request has been refused and the ESB instance cannot be downloaded
Normal Case	1. A request with the ESB instance ID is received. 2. The system checks if the ESB exists and if the requester has permission over it. In that case the requester receives a file containing the instance.
Special Cases	1a. The specified ESB instance does not exist. a) The system shows an error message and aborts. 2a. The requester does not have the necessary rights in the system or the credentials introduced are incorrect a) The system shows an authentication error and aborts.

Table 4.11.: Description of Use Case *Download ESB Instance Image*.

Name	**Remove ESB Instance**
Goal	The developer or administrator wants to remove an existing ESB instance.
Actor	Developer or Administrator
Pre-Condition	The requester must specify the ID of the ESB instance and he must have permission over it
Post-Condition	The instance is removed.
Post-Condition in Special Case	The request has been refused and the ESB instance is not removed.
Normal Case	1. A request with the ESB instance ID is received. 2. The system checks if the ESB exists and if the requester has permission over it. 3. The ESB instance is stopped if it is running. 4. The ESB instance and all its information in the cluster are deleted. 5. The system shows a confirmation message.
Special Cases	1a. The specified ESB instance does not exist. a) The system shows an error message and aborts. 2a. The requester does not have the necessary rights in the system or the credentials introduced are incorrect a) The system shows an authentication error and aborts.

Table 4.12.: Description of Use Case *Remove ESB Instance*.

4.3. System Overview

After describing all the desired use cases and requirements, we propose a system architecture which aims at fulfilling all of them. The presented system is a framework able to deploy multiple ESB instances with a determinate configuration for multiple tenants. The architecture of this system is shown in Figure 4.2 and each one of its components are described in the remaining of this section. A more detailed description is given in Chapter 5.

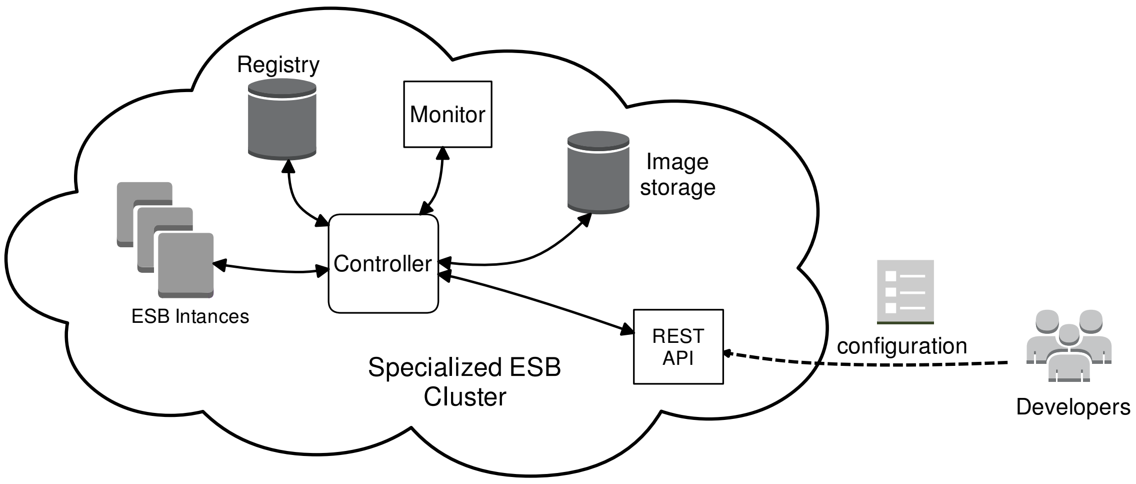


Figure 4.2.: Specialized ESB framework architecture

ESB Image

An *ESB Image* is the VM template which contains a preconfigured ESB package inside. These images must be built first with a determinate configuration specified by a configuration description file. After they are built, the ESB images include all the required ESB features or components and other parameters regarding the resource usage. These images are ready to be launched and they are used to launch new ESB instances.

ESB Instances

We refer as *ESB Instance* to the component able to contain the ESB package (i.e. ServiceMix) and run in the system. Therefore, a ESB instance can contain several subcomponents inside in order to be able to run in the cluster and handle some additional tasks. For example, it can include a component to report if the ESB software is running or not or a component to store the ESB instance data periodically. Besides, each ESB instance is launched from an existing and preconfigured base image from the image storage component. A cluster of hosts is deployed in other to enable resource scalability and adaptability to the system. The ESB instances are distributed through the cluster and they can run in any machine.

Image Storage

The specialized ESB images are stored in an *Image Storage* registry, including base images and images of running ESB instances. These images can be created, pulled or pushed from each host of the cluster, and therefore, the image storage must be reachable from all the host machines.

Registry

The *Registry* holds all the information about the system execution (existing ESB instances, monitoring information, ESB images, etc.). The registry must be a synchronized distributed system also reachable from all the hosts in the cluster.

Controller

The *Controller* component coordinates the cluster deployment. It distributes the ESB instances through the different nodes of the cluster, and it migrates the running instances in case a machine shutdowns to the remaining machines in the cluster. The main function of this component is the ESB instance scheduling and rescheduling. However, it controls also that each a specific ESB instance is actually running or not and its launch and removal from the system.

Monitor

The *Monitor* gathers all the information about the resource usage of the running ESB instances and the hosts in the cluster (i.e. CPU, memory, etc.). It makes this information available to the rest of the components.

REST API

The *REST API* is the interface to interact with the system. It must provide methods to get information and to interact with the monitor, image storage and controller. Through this interface the administrator and developers must be able to manage all the ESB instances in the cluster (i.e. building, creation, removal, etc.). It is the interaction point between the administrator and the developer with the system.

5. Design

This work adopts and uses as its basis a container virtualization approach proposed in [32]. The reason is that all the containers we deploy are ESB instances which have the same OS and libraries (i.e. Java, Maven, etc.) and with container virtualization libraries and OS are shared between different VMs. Hence, they are shared between all the ESB instances in a machine and consequently, the provisioning overhead (and therefore the resource usage due to provisioning tasks) is much lower. Another reason to select container virtualization is that containers usually can start and stop faster than other traditional virtualization technologies, something relevant for our work where plenty of ESB instances must be started and stopped continuously and rapidly. Moreover, a rapid migration of ESB instances is necessary in our system. Containers can be migrated and run in other machines in an easier and lighter way than other kinds of virtualization such as hypervisor-based virtualization or full virtualization. For instance, as containers share libraries and OS, in their migration, libraries and OS must be moved only once, no each time a container is migrated.

In the remaining of this section we depict the components present in the system architecture, and how they must work to achieve our objectives. We start from the ESB characterization, going though all the artifacts to deploy ESB instances in a reusable and efficient way, and we finish presenting the tools to interact with the system.

5.1. ESB Instances Characterization and Configuration

ESB solutions are not typically shipped as light weight packages, as they are constituted by multiple components or features that are plugged to the middleware container. However, each one of these components increments the overhead and the resource usage of the whole system. Therefore, the provisioning overhead of a non-characterized ESB is directly proportional to the number of components that constitute such a middleware and using only the required components is a important aspect for the system efficiency. Therefore, during this section we focus on ESB characterization and later we enable the means to configure rapidly ESB Instances.

5.1.1. ESB Characterization

In the first part of this work we aim to investigate the means to simplify the process of building specialized ESB instances. The number of artifacts included can be huge and the task of identifying the ESB components that an application really requires can be a really complex process. For this purpose, we first build a taxonomy for each one of the supported ESBs. The

Figure 5.1 depicts the main categories that after our analysis we propose to be the first step in the ESB features classification. Additionally, we want to give a brief description of each one of these general categories:

- *Validation*: includes all the components related to validation of the exchanged data (i.e. JSON Schema, XML schema, etc.).
- *Management & Orchestration*: contains all the features associated with the ESB management and orchestration of its different components.
- *Message handlers*: includes components to control the exchanged messages inside of the ESB.
- *Communication*: is the widest category, it includes components destined to allow the communication based on different protocols or shapes (i.e. Email, Jabber, HTTP/HTTPS, etc).
- *Message transformers*: holds components to change the shape of the exchanged messages (e.g. JSON to XML marshal).
- *Storage*: involves all the features related to the storage management (i.e. SQL, NoSQL, Cache, etc.).
- *Routing*: features destined to route messages or requests are contained here (i.e. Apache Camel, NMR, etc.).
- *Security*: all the features related to security aspects must be located in this category (i.e. signatures, cryptography, etc.).

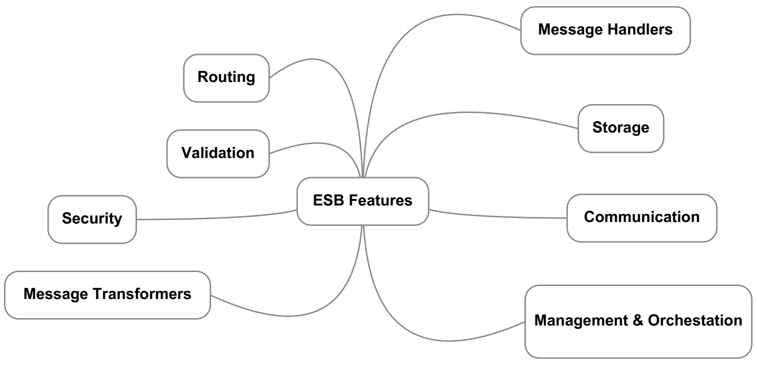


Figure 5.1.: ESB taxonomy main categories

The amount of potential components within an ESB is unlimited with all the already included in the package and the ones created by the users, and it is completely necessary to assist developers in the task of selecting which concrete components are needed for their applications. Therefore, this previous classification is a crucial aspect to simplify the task of ESB configuration.

5.1.2. Characterized ESB Instance Creation Sequence

This section shows how a specialized ESB instance can be created and configured. We propose a sequence of steps to perform such configuration. We distinguish several aspects to configure an ESB as already installed components, parameters related to resource usage or network access options. The steps we propose are detailed next:

1. *Identify the system requirements*. The developer must investigate which components and communication protocols are present in his application. Additionally, there may be some resource limitation in his system he should identify.
2. *ESB software*. The second step is selecting the base ESB software that it will be used between all the supported ones in the platform (e.g. ServiceMix, Mule, Fuse ESB, ect.).
3. *Features configuration*. Each instance can start up with a series of preinstalled ESB features or components. They must be installed during the image building. For example, an ESB can contain features to support SQL or HTTP.
4. *Custom components configuration*. The user can install also components created by himself in the base images. These components must be only accessible by their owner.
5. *Resources configuration*. Some properties about resource usage can be configured usually in configuration files (i.e. JVM memory usage, features repositories, etc.).
6. *Labels*. The labels are only a list of tags for the ESB instance in order to identify the ESB instances rapidly. For example, a ESB instance could be tagged as "region=west, department=sales".
7. *Network access configuration*. The exposed container ports of the instance can be defined also by the user. All the ports the developers wants to use must be first registered to be able to use them. These ports are mapped with host ports when the instance is running. The user must check this mapping in order to discover which container ports are associated with which host ports. A new IP address is assigned to ESB instance when it starts.

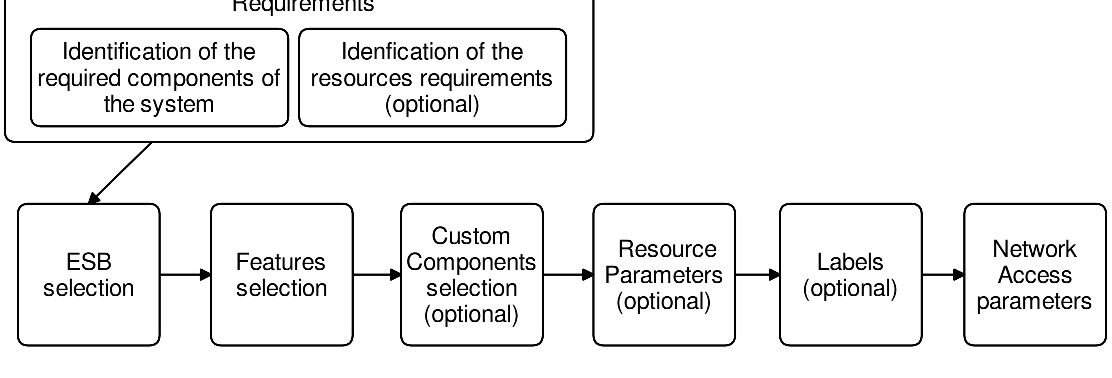

Requirements
Identification of the required components of the system
Idenfication of the resources requirements (optional)
ESB selection
Features selection
Custom Components selection (optional)
Resource Parameters (optional)
Labels (optional)
Network Access parameters

Figure 5.2.: The characterization process

5.1.3. ESB Image Building

After the configuration identification process, an image is built from a configuration and stored in the image storage. The definition of ESB image is given in Section 4.3. After they are built, developers can launch as many ESB instances from these images as they want. In Figure 5.3 we can see how the image is created from a specific configuration and used to launch several containers with the ESB package already installed and configured.

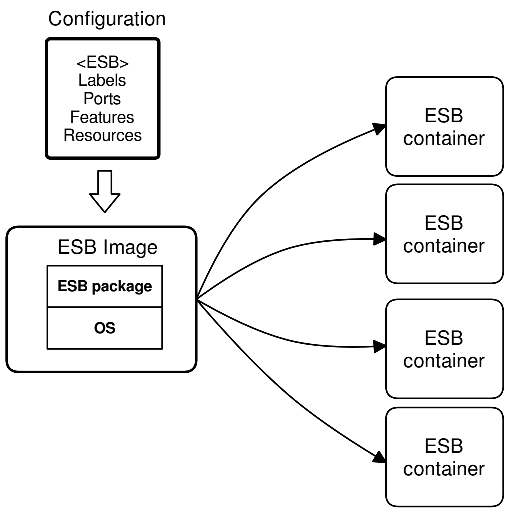


Figure 5.3.: Running several specialized ESB instances from an ESB image

5.2. ESB Instance

An *ESB instance* described in Section 4.3 has two states: stopped and running. When an ESB instance is stopped, it is only stored as an ESB image in the image storage, otherwise it is constituted by two different components, a container and a *minion*. The container runs in the first launch as a copy of an ESB base image in the image storage, and when it is stopped, its state is committed and pushed to the image storage again (see Figure 5.5). The state of an ESB instances must be stored after it is stopped in order to launch it again in same conditions. Besides, its state must be stored during its execution to avoid data lost in case of failure.

On the other hand, we have the *minion* component. The minion component has been created in order to manage all the tasks to run the ESB instance in a system with a cluster deployment. It performs tasks such as ESB instance registration, network data publication, or periodically committing the state of the running container which holds the ESB package. The minion is an internal component of the system and it is totally invisible to the developer.

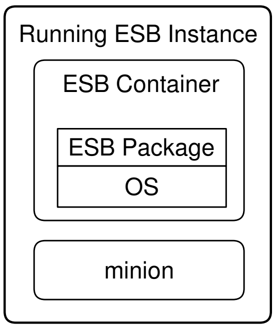


Figure 5.4.: Running ESB instance components

5.3. Image Storage

The *Image Storage* holds all the ESB images, both base images and images of existing ESB instances state. These images can be created, pulled or pushed from the each host of the cluster, and it works as a repository storing only the difference between images to avoid the transference of duplicated data. The image storage must be aside of the rest of the components, or in a remote location, but it must be reachable from all the machines in the cluster.

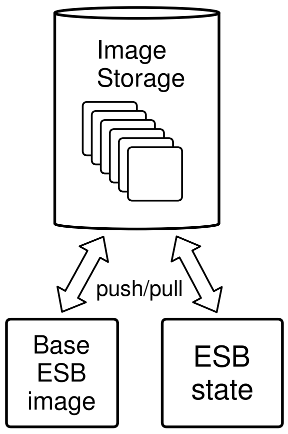


Figure 5.5.: Image storage with ESB images and ESB state images

5.4. Host Machines & Cluster Deployment

The host machines are organized in a cluster which must be deployed in order to enable resource scalability and adaptability of the system. A machine can host multiple ESB instances, which can be migrated from one machine to another in the cluster. The cluster is constituted by a *leader* machine and *follower* machines. Commonly the term *master* is used for what we call leader, and *slaves* for the followers. The leader is the one which performs the management tasks (i.e. scheduling, service control, etc.). A new leader is selected periodically between all the running machines in order to avoid the overload of one of them. The idea is that all the machines should be identical to enable the elasticity of the system. They have the same configuration and responsibility in the cluster. If new machines are added into the cluster, they are also selectable to be the next leader. In case the leader machine shutdowns, a new leader is selected and the system keeps working without any problem. In addition, there is not a dedicated machine for orchestration, all the machines in the cluster run ESB instances and perform such orchestration and have the same responsibility to ensure the good performance of the system. For example, all the machines perform scheduling and rescheduling tasks when they are the leader. In contrast, if a centralized-solution would be used, we should find new means to face the failure of the master machine, which would be reserved for orchestration tasks and which should be known for the rest of the nodes in the cluster.

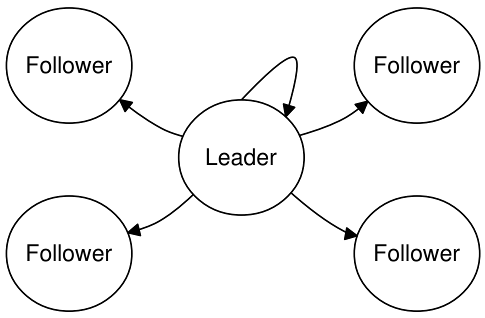


Figure 5.6.: Roles of machines in the cluster

On the other hand, to allow the fast deployment and plugging of new machines into the cluster, all the cluster management components of the system are deployed as distributed applications running in each one of the machines. Each one of these artifacts running in the machine are called *agents*. Each machine of the cluster has different agents installed to interact with the distributed components. Agents to interact with each one of the components of the system (i.e. Controller, Monitor, Registry, etc) are included. These agents can be seen in Figure 5.7. Additionally, the REST API is installed and runs in all the hosts.

5.5. Registry

The *Registry* is a distributed key-value stored which contains all the information related to the entire system. It is accessible and synchronized through the whole cluster. It includes the information about existing ESB instances, ESB images, machines, etc. The relationship and attributes of these entities can be seen in Figure 5.8.

An *ESB instance*, as it is shown in Figure 5.8, is the central component also of the data model which represents our system. It has some properties as its creation time or state. The state of

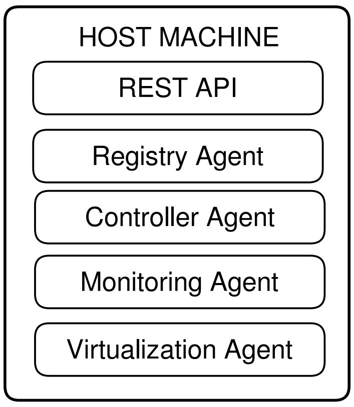


Figure 5.7.: Agents and components installed in a host machine

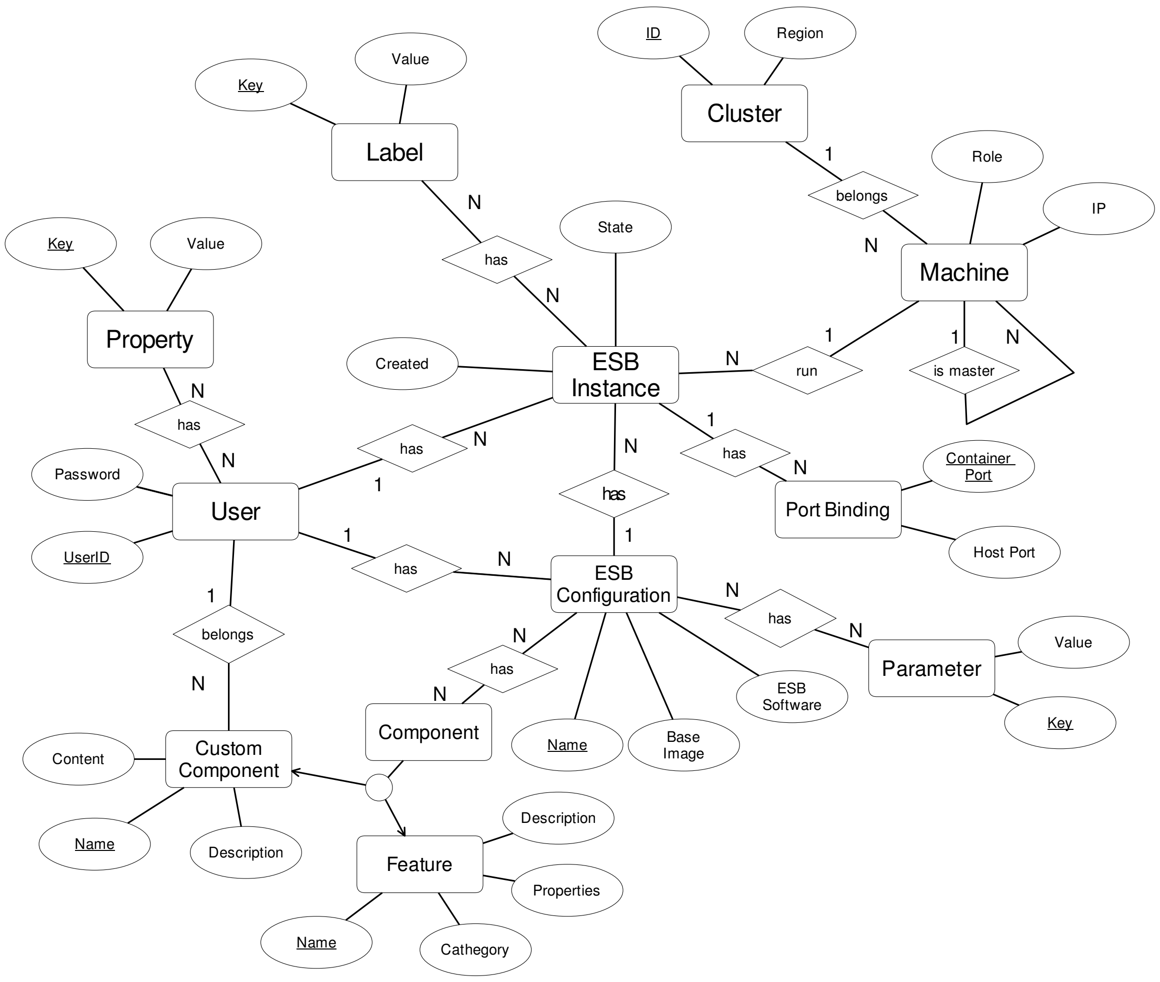


Figure 5.8.: Entity-Relationship diagram

one ESB instance only admits the following values: stopped, starting, running and stopping.

An ESB instance can be tagged with several *Labels*, defined by their key and value. For example, an ESB instance can be tagged as "country=spain" or "department=sales". An ESB instance is created from an *ESB Configuration*, which can be assigned to many instances. In addition, a ESB instance contains a collection of *Port Bindings*, which contain two attributes: a *Container Port*, and a *Host Port*. A container port defines the port designated to the ESB instance running inside a container. In contrast, the host port is the port assigned to this container port in the running host machine.

An *ESB Configuration* is defined by the *User*, and exclusively belongs to him. It includes a set of parameters and a field to identify the base ESB image which is used when a new ESB instance is created. An ESB configuration also includes a set of components, which are the features that are installed in the base image of the ESB instance by default. A *Component* can be shipped by the ESB software provider, or can be developed and customized by an user. A *Custom Component* belongs to a specific user and it must include name and description to be identified unequivocally by its owner. Additionally, its content must be saved inside of the registry component to include it in the new ESB images. For its part, a *Feature* belongs to the ESB software provider, it is defined by its name, category, properties and description. The Features must be installed in the base image building. Their category and properties are defined by the ESB taxonomy, as we mention in Section 5.1.1. They must also include a description to facilitate their selection in the configuration process.

Regarding the machines running in the cluster, all the running ESB instances are located in one machine and can be migrated to other one at any time. A *Machine* is defined by its IP address and role. It belong to a *Cluster*. A cluster has a leader machine and the other machine are its followers. This leader/follower relationship defines the attribute role in the machine entity and it can be used for cluster orchestration, where the leader can manage the rest of the machines to assure a good system performance. A *User* is determined by its user ID, password and other parameters (i.e. email, city, country, etc.). It also owns a set of ESB configurations, custom components and ESB instances.

In summary, an user owns a set of ESB configuration and custom components, which are used to build and launch new ESB instances. The ESB instances, which include all the necessary components and parameters, run in a specific machine. Finally, the machines are organized in a cluster where one of them is the leader, which can perform orchestration tasks. A description of the most relevant entities in the ER Diagram, Figure 5.8, is given in form of JSON schemas in Appendix A.

Figure 5.9 depicts the registry which persists the monitoring information. The *Monitoring Data* defined by its timestamp includes a set of *Parameters* related with monitoring data (i.e. CPU, memory, network,etc.). This monitoring information comes from a running ESB instance or a host machine.

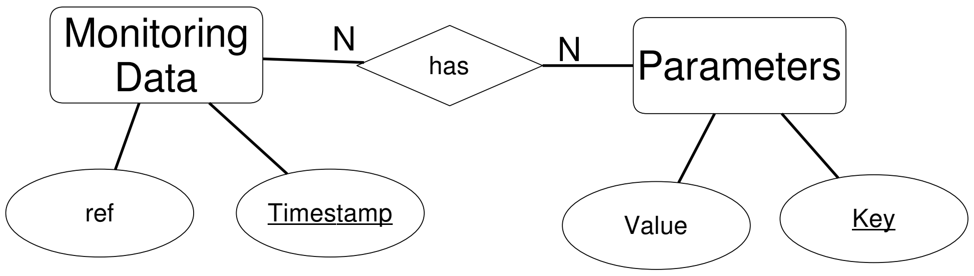


Figure 5.9.: Monitoring Entity-Relationship diagram according to the Chen's notation

5.6. Controller

The *Controller* is the most complex component in the system, it performs different tasks such as scheduling or service management, and it interacts with the other components in the system. It contains three different components to achieve these tasks: *Scheduler*, *Manager* and *Health*. These components are shown in Figure 5.10 and their behavior is explained in the following sections.

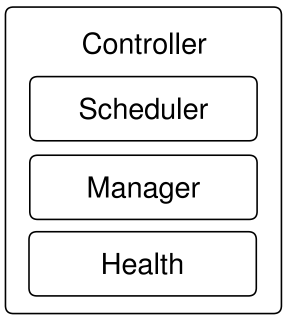


Figure 5.10.: Controller structure

5.6.1. Health

The *Health* component is responsible for analyzing machine monitoring information retrieved from the monitoring component (see Section 5.7) and communicating the machine *state*. This component decides in which state is the machine based on data about CPU, memory or network usage, and it broadcasts the machine state to the manager and scheduler components. The health component identifies four different states for a machine:

- *Idle*: the usage of the machine is low, and therefore the usage of the underlying cluster resources is not maximized. If a machine is in this state, it should be shutdown to streamline the use of resources and hence, the costs.
- *Normal*: this state appears when a machine uses a correct amount of resources, but it can hold more ESB instances.

- *Busy*: a machine is busy when it has enough spare resources for the currently running ESB instances but can not hold any more.
- *Critical*: this state appears when the machine cannot assure the QoS of the running ESB instances. This state could be used to perform tasks as rescheduling, migrations or machine provisioning.

The health component must update machine state periodically, and in that way, the scheduling and management components can make use of that information to do their corresponding tasks. Additionally, the health component also updates the information about if the host machine is the leader or not.

5.6.2. Manager

The *Manager* component is responsible for the supervision of the ESB instances state. It supervises if they are actually running or if any kind of problem appears. For instance, if there is problem with the ESB instance and it stops, the manager component restarts it again. Other case is when the host machine fails, then the Manager relaunch the ESB instance in other machine in the cluster. Besides, it handles the restart or permanent removal of ESB instances. Additionally, the manager component interacts with other artifacts in the system. For example, it removes an ESB image if it is not longer required from the image storage.

5.6.3. Scheduler

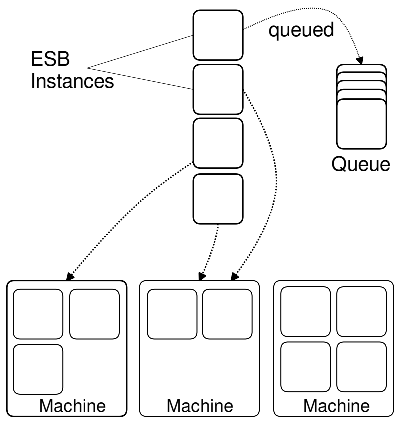


Figure 5.11.: Scheduling of ESB instances

The *Scheduler* component handles the distribution of the ESB instances through the different machines which constitute the cluster. The scheduling algorithm is a combination of different criteria. In order to select the destination machine, the scheduler component decides which of the machine is the least loaded as it is shown in Figure 5.11 where the first instance is assigned to the second machine because it has less ESB instances running. In Figure 5.11, we can also observe that if there are not enough available resources to allocate an ESB instance, this is queued and launched later when the situation changes. The queue is potentially infinite and if this situation appears new VMs resources might be created. However, our system does not address this situation, but it enables the inclusion of such a solution in the future. The scheduling choice is performed in two phases:

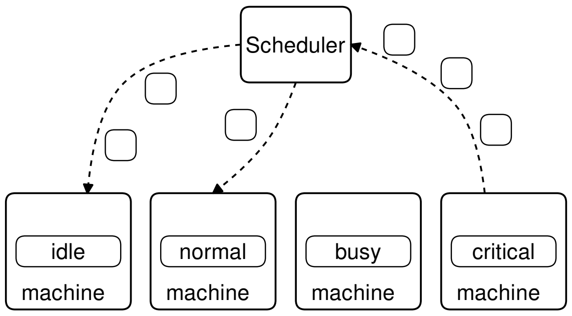


Figure 5.12.: Scheduling based on machine states

1. The scheduler component checks the state of the machines, as we defined in Section 5.6.1, and in case there are machines with resources to launch a new ESB instance (machines in state *idle* or *normal*) the instances are allocated in one of these machines (see Figure 5.12). Otherwise, if all the machines are in *busy* or *critical* states, the instance is queued until one of the machine is again available, Figure 5.11.
2. The scheduler component chooses between all the available machines the one with more free resources. This one would be the one with the most relaxed state, and if there are more than one with this state, the one which has less ESB instances already deployed as it is shown in Figure 5.11.

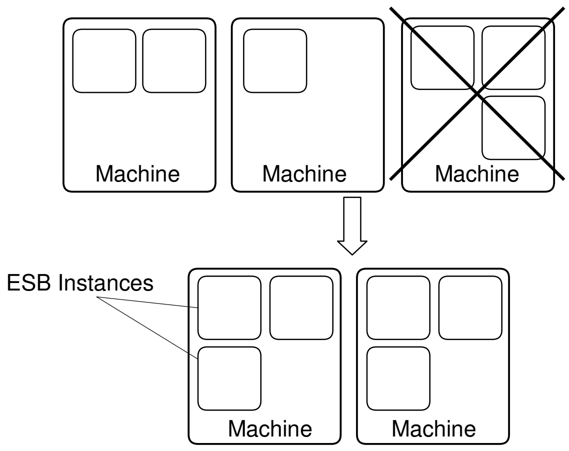


Figure 5.13.: Migration of ESB instances in case of machine overloaded

The scheduler component can react in some problematic cases rescheduling the running ESB instances. Two different cases are addressed, when failure or when a machine runs out of resources. In case of a machine shutdowns or fails, the running ESB instances are scheduled to the other machines in the cluster, as we can see in Figure 5.13. The other case when the instances are migrated, it is when a machine is out of resources, in a *"critical"* state as we have defined (see Figure 5.14).

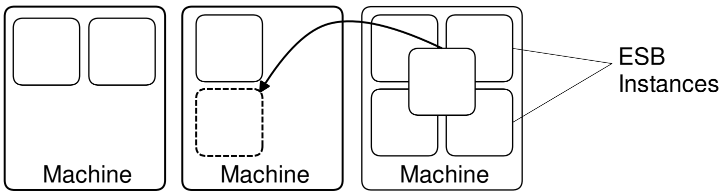


Figure 5.14.: Migration of ESB Instance in case of machine overloaded

5.7. Monitor

The *Monitor* gathers resource usage information (CPU, memory, network, etc.) from running containers which host ESB instances and from host machines. The monitor component is installed in each one of the machines and the collected data is stored in a local database inside of them. The recorded information is provided through a REST API and in that way, other components can use it as the health and manager components.

There is not a central database with all the monitoring information of the machines. In contrast the information must be aggregated by the components which want to use it. For example, the Health component, as we have already mentioned, collects data from each one of the hosts, and processes it to discover the state of the machines in the cluster.

5.8. REST API

The REST API makes possible the control of the whole system by remote HTTP requests. There are two different REST APIs, one for the administrator and other one for the developers. A summary of the supported operations in the REST API is shown in Figure 5.1.

Name	User	Description
List Features	Administrator and Developer	This outputs a list of available features.
Features Tree	Administrator and Developer	Search through the feature characterization tree.
List Machines	Administrator	This outputs a list with all the running machines in the system.
Get Machine	Administrator	This outputs the machine information.

Get Machine Monitoring Info	Administrator	This outputs the machine monitoring information.
Add New User	Administrator	This creates a new user in the system.
Update User	Administrator	This updates the properties of an existing user.
Get User	Administrator	This outputs the information of the user.
List Users	Administrator	This outputs a list with all the users in the system.
Remove User	Administrator	This removes an existing user and all his belongings.
Build Image	Administrator	Builds an image with a specialized ESB instance.
Add New Configuration	Developer	This creates a new configuration associated with the developer.
Update Configuration	Developer	This updates the properties of an existing configuration.
Get Configuration	Developer	This outputs a configuration.
List Configurations	Developer	This outputs a list with all the configuration of the developer.
Remove Configuration	Developer	This removes an existing configuration.
Create New Instance	Administrator and Developer	This creates a new instance with the specified configuration associated with the specified Developer.
Get Instance	Administrator and Developer	This outputs the information of the ESB instance.
List Instances	Administrator and Developer	This outputs a list with all the ESB instances of the user or the system.
Start Instance	Administrator and Developer	This starts an existing instance.
Get Instance Monitoring Info	Administrator and Developer	This outputs the monitoring information of the ESB instance.
Move Instance	Administrator	Moves a running instance to another machine with more resources.
Stop Instance	Administrator and Developer	This stops a running instance.
Download Instance	Developer	With this method a tar file with the instance image can be downloaded.
Delete Instance	Administrator and Developer	This removes an existing ESB instance.

Table 5.1.: REST API summary

5.8.1. Administrator REST API

Method	**List features**
Authorized users	Administrator
Description	This outputs a list with the available features
HTTP Request	GET /admin/feature
URI params	-
Query params	• *value*: value to search • *criteria*: specifies which attribute of the feature must be checked. If not criteria is defined, the search takes a look along all the attributes.
Post params	-
Response	Array with the matching features
Error responses	• *500 Internal Server Error*

Table 5.2.: Description of REST method *List features* for *Administrator*.

Method	**Features Tree**
Authorized users	Administrator
Description	Search though the feature characterization tree
HTTP Request	GET /admin/feature
URI params	-
Query params	• *node*: name of the tree node
Post params	-
Response	Array with the children of the node
Error responses	• *404 Not Found* • *500 Internal Server Error*

Table 5.3.: Description of REST method *Features Tree* for *Administrator*.

Method	**List machines**
Authorized users	Administrator
Description	This outputs a list with all the running machines in the system
HTTP Request	GET /admin/machine

URI params	-
Query params	-
Post params	-
Response	Array with all the machines in the cluster
Error responses	• *500 Internal Server Error*

Table 5.4.: Description of REST method *List machines* for *Administrator*.

Method	**Get machine**
Authorized users	Administrator
Description	This outputs the machine information
HTTP Request	GET /admin/instance/{id}
URI params	• **id** : identification of the requested machine
Query params	-
Post params	-
Response	Returns machine object
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.5.: Description of REST method *Get machine* for *Administrator*.

Method	**Get machine monitoring info**
Authorized users	Administrator
Description	This outputs the machine monitoring information
HTTP Request	GET /admin/machine/{id}/stats
URI params	• **id** : identification of the machine
Query params	-
Post params	-
Response	Returns monitoring data
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.6.: Description of REST method *Get machine monitoring info* for *Administrator*.

Method	**Add new user**
Authorized users	Administrator
Description	This creates a new user in the system.
HTTP Request	POST /admin/user
URI params	-
Query params	-
Post params	User object
Response	*201 Created*
Error responses	• *500 Internal Server Error* • *400 Bad Request*

Table 5.7.: Description of REST method *Add new user* for *Administrator*.

Method	**Update user**
Authorized users	Administrator
Description	This updates the properties of an existing user
HTTP Request	PUT /admin/user/{id}
URI params	• **name** : identification of the user
Query params	-
Post params	User object
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.8.: Description of REST method *Update user* for *Administrator*.

Method	**Get user**
Authorized users	Administrator
Description	This outputs the information of the user

HTTP Request	GET /admin/user/{name}
URI params	• **name** : identification of the user
Query params	-
Post params	-
Response	Returns an user object
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.9.: Description of REST method *Get user* for *Administrator*.

Method	**List Users**
Authorized users	Administrator
Description	This outputs a list with all the users in the system
HTTP Request	GET /admin/user
URI params	-
Query params	-
Post params	-
Response	Array with all the users
Error responses	• *500 Internal Server Error*

Table 5.10.: Description of REST method *List Users* for *Administrator*.

Method	**Remove User**
Authorized users	Administrator
Description	This removes an existing user and all his belongings.
HTTP Request	DELETE /admin/user/{name}
URI params	• **name** : identifier of the user.
Query params	-
Post params	-
Response	*200 OK*

Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.11.: Description of REST method *Remove User* for *Administrator*.

Method	**Build Image**
Authorized users	Administrator
Description	Builds an image with a specialized ESB instance
HTTP Request	POST /admin/image
URI params	-
Query params	-
Post params	Configuration object
Response	Returns a confirmation response with the ID of the new image and *200 OK*
Error responses	• *500 Internal Server Error* • *400 Bad Request*

Table 5.12.: Description of REST method *Build Image* for *Administrator*.

Method	**Create New Instance**
Authorized users	Administrator
Description	This creates a new instance with the specified configuration associated with the specified Developer
HTTP Request	POST /admin/action/new/instance
URI params	-
Query params	• **owner** : user name to be the owner of the instance
Post params	Configuration object
Response	Returns a new instance object
Error responses	• *500 Internal Server Error* • *400 Bad Request*

Table 5.13.: Description of REST method *Create New Instance* for *Administrator*.

Method	**Get Instance**
Authorized users	Administrator
Description	This outputs the information of the ESB instance
HTTP Request	GET /admin/instance/{id}
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	Returns an ESB instance object
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.14.: Description of REST method *Get Instance* for *Administrator*.

Method	**List Instances**
Authorized users	Administrator
Description	This outputs a list with all the ESB instances in the system
HTTP Request	GET /admin/instance
URI params	-
Query params	-
Post params	A query with the user labels can be specified in order to filter the search
Response	Array with all the intances objects of the user
Error responses	• *500 Internal Server Error*

Table 5.15.: Description of REST method *List Instances* for *Administrator*.

Method	**Start Instance**
Authorized users	Administrator
Description	This starts an existing instance
HTTP Request	PUT /admin/instance/{id}/start
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-

Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.16.: Description of REST method *Start Instance* for *Administrator*.

Method	**Get Instance Monitoring Info**
Authorized users	Administrator
Description	This outputs the monitoring information of the ESB instance
HTTP Request	GET /admin/instance/{id}/stats
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	Returns monitoring data
Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.17.: Description of REST method *Get Instance Monitoring Info* for *Administrator*.

Method	**Move Instance**
Authorized users	Administrator
Description	Moves a running instance to another machine with more resources
HTTP Request	PUT /admin/instance/{id}/move
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.18.: Description of REST method *Move Instance* for *Administrator*.

Method	**Stop Instance**
Authorized users	Administrator
Description	This stops a running instance
HTTP Request	PUT /admin/instance/{id}/stop
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.19.: Description of REST method *Stop Instance* for *Administrator*.

Method	**Delete Instance**
Authorized users	Administrator
Description	This removes an existing instance
HTTP Request	DELETE /admin/instance/{id}
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.20.: Description of REST method *Delete Instance* for *Administrator*.

5.8.2. Developer REST API

Method	**List Features**
Authorized users	Developer
Description	This outputs a list of available features.
HTTP Request	GET /feature

URI params	-
Query params	• *value*: value to search • *criteria*: specifies which attribute of the feature must be checked. If no criteria is defined, the search takes a look along all the attributes.
Post params	-
Response	Array with the matching features
Error responses	• *500 Internal Server Error*

Table 5.21.: Description of REST method *List Features* for *Developer*.

Method	**Features Tree**
Authorized users	Developer
Description	Searchs through the feature characterization tree.
HTTP Request	GET /feature/tree
URI params	-
Query params	• *node*: name of the tree node
Post params	-
Response	Array with the children of the node
Error responses	• *404 Not Found* • *500 Internal Server Error*

Table 5.22.: Description of REST method *Features Tree* for *Developer*.

Method	**Add New Configuration**
Authorized users	Developer
Description	This creates a new configuration associated with the developer
HTTP Request	POST /configuration
URI params	-
Query params	-
Post params	Configuration object
Response	*201 Created*

Error responses	• *500 Internal Server Error* • *400 Bad Request*

Table 5.23.: Description of REST method *Add New Configuration* for *Developer*.

Method	**Update Configuration**
Authorized users	Developer
Description	This updates the properties of an existing configutation
HTTP Request	PUT /configuration/{id}
URI params	• **name** : name of the configuration.
Query params	-
Post params	Configuration object
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.24.: Description of REST method *Update Configuration* for *Developer*.

Method	**Get Configuration**
Authorized users	Developer
Description	This outputs a configuration
HTTP Request	GET /configuration/{name}
URI params	• **name** : name of the configuration
Query params	-
Post params	-
Response	Returns a Configuration object
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.25.: Description of REST method *Get Configuration* for *Developer*.

Method	**List Configurations**
Authorized users	Developer
Description	This outputs a list with all the configuration of the developer
HTTP Request	GET /admin/user
URI params	-
Query params	-
Post params	-
Response	List with all the configurations
Error responses	• *500 Internal Server Error*

Table 5.26.: Description of REST method *List Configurations* for *Developer*.

Method	**Remove Configuration**
Authorized users	Developer
Description	This removes an existing configuration
HTTP Request	DELETE /configuration/{name}
URI params	• **name** : name of the configuration.
Query params	-
Post params	-
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.27.: Description of REST method *Remove Configuration* for *Developer*.

Method	**Create New Instance**
Authorized users	Developer
Description	This creates a new instance with the specified configuration
HTTP Request	POST /instance
URI params	• *configuration*: name of the configuration
Query params	-

Post params	-
Response	Returns a new ESB instance object
Error responses	• *500 Internal Server Error* • *400 Bad Request*

Table 5.28.: Description of REST method *Create New Instance* for *Developer*.

Method	**Get Instance**
Authorized users	Developer
Description	This outputs the information of the ESB instance
HTTP Request	GET /instance/{id}
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	Returns an ESB instance object
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.29.: Description of REST method *Get Instance* for *Developer*.

Method	**List Instances**
Authorized users	Developer
Description	This outputs a list with all the ESB instances of the user
HTTP Request	GET /instance
URI params	A query with the user labels can be specified in order to filter the search
Query params	-
Post params	-
Response	Array with all the instances objects of the user
Error responses	• *500 Internal Server Error*

Table 5.30.: Description of REST method *List Instances* for *Developer*.

Method	**Start Instance**
Authorized users	Developer
Description	This starts an existing instance
HTTP Request	PUT /instance/{id}/start
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.31.: Description of REST method *Start Instance* for *Developer*.

Method	**Get Instance Monitoring Info**
Authorized users	Developer
Description	This outputs the monitoring information of an ESB instance
HTTP Request	GET /instance/{id}/stats
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	Returns monitoring data
Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad request*

Table 5.32.: Description of REST method *Get Instance Monitoring Info* for *Developer*.

Method	**Stop Instance**
Authorized users	Developer
Description	This stops a running instance
HTTP Request	PUT /instance/{id}/stop

URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	*200 OK*
Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.33.: Description of REST method *Stop Instance* for *Developer*.

Method	**Download Instance Image**
Authorized users	Developer
Description	With this method a tar file with the instance image can be downloaded.
HTTP Request	GET /instance/{id}/download
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	Returns a tar file with the instance image.
Error responses	• *500 Internal Server Error* • *404 Not Found*

Table 5.34.: Description of REST method *Download Instance Image* for *Developer*.

Method	**Delete Instance**
Authorized users	Developer
Description	This removes an existing instance
HTTP Request	DELETE /instance/{id}
URI params	• **id** : identification of the requested instance
Query params	-
Post params	-
Response	*200 OK*

Error responses	• *500 Internal Server Error* • *404 Not Found* • *400 Bad Request*

Table 5.35.: Description of REST method *Delete Instance* for *Developer*.

5.9. Command Line Interface

A Command Line Interface (CLI) is installed in all the machines in the cluster and it can be used by the administrator to manage the system instead of doing it though the REST API. All the commands and options are described in the following tables.

Command	**features**
Description	List of features
Command	**esb features**
Options	• *j*: output in a JSON format • *q* : search by a query beetween all atributes the features • *n* : search features by their name • *p* : search feature by their category path. Ex: esb features -p communication/email • Default: return the complete list.
Params	• *query*: value to be search along the features.
Response	List of requested elements

Table 5.36.: Description of CLI command *features*.

Command	**machines**
Description	This returns a list with the running machines in the cluster
Command	**esb machines**
Options	• *j*: output in a JSON format
Params	-
Response	List of requested elements

Table 5.37.: Description of CLI command *machines*.

Command	**users**
Description	List of users in the system. Add or remove users.
Command	**esb features**
Options	• *add*: creates a new user in the system. • *del* : removes an user from the system. • *update* : updates the properties of an user. • Default: returns the complete list of users.
Params	• –add *user password*: user and password for the new user. • –del *user*: name of the user. • –update *user password*: user and password of a existing user.
Response	-

Table 5.38.: Description of CLI command *users*.

Command	**build**
Description	This builds a specialized ESB image with the specified configuration
Command	**esb build** {filename}
Options	-
Params	• **filename** : path to the configuration file
Response	Id of the created image

Table 5.39.: Description of CLI command *build*.

Command	**create**
Description	This creates a specialized ESB Instance with the specified configuration
Command	**esb create** -u {username} {filename}
Options	• *-u*: Owner of the new instance
Params	• **filename** : path to the configuration file
Response	Id of the created ESB instance

Table 5.40.: Description of CLI command *create*.

Command	**start**
Description	This starts a stopped ESB instance
Command	**esb start** {id}
Options	-
Params	• **id** : Id of the stopped ESB instance
Response	Id of the starting ESB instance

Table 5.41.: Description of CLI command *start*.

Command	**inspect**
Description	This returns the information of an existing ESB instance
Command	**esb inspect** {id}
Options	-
Params	• **id** : Id of the existing ESB instance
Response	ESB instance information

Table 5.42.: Description of CLI command *inspect*.

Command	**instances**
Description	This returns the list of ESB instances
Command	**esb machines**
Options	• *j*: output in a JSON format
Params	-
Response	List of requested elements

Table 5.43.: Description of CLI command *instances*.

Command	**search**
Description	Return a list of elements which match with the labels conditions
Command	**esb search** {label}={value}
Options	-

Params	• **label**: label name • **value**: expected value for the specified label.
Response	List of requested elements

Table 5.44.: Description of CLI command *search*.

Command	**move**
Description	This migrates a running ESB instance to another machine with more resources.
Command	**esb move** {id}
Options	-
Params	• **id** : Id of the existing ESB instance
Response	Id of the moved ESB instance

Table 5.45.: Description of CLI command *move*.

Command	**stop**
Description	This stops a running ESB Instance
Command	**esb stop** {id}
Options	-
Params	• **id** : Id of the running ESB instance
Response	Id of the stopped ESB instance

Table 5.46.: Description of CLI command *stop*.

Command	**destroy**
Description	This destroys an existing ESB instance
Command	**esb destroy** {id}
Options	-
Params	• **id** : Id of the existing ESB instance
Response	Id of the destroyed ESB instance

Table 5.47.: Description of CLI command *destroy*.

6. Implementation

The first step in our implementation must be deciding which Enterprise Service Bus project we should use. For this prototype we offer support for the ESB solution ServiceMix (see Section 2.4.2). ServiceMix includes not only support for many communication technologies and protocols (HTTP/HTTPS, SQL, SOAP, etc.), but it also enables to create and plug new components to the existing ESB solution. They are what we previously defined as *Features* and *Custom Components*, respectively. It has been used in multiple investigations because of these and more properties as it is shown in the articles [64] and [4]. Moreover, the expertise gained during the last years by using and extending ServiceMix in several projects has contributed to deciding among which ESB solution to use in this thesis.

However, before starting with the ServiceMix characterization we should also decide the virtualization technology involved in our work. As previously discussed, our approach is based on container virtualization. Regarding our prototypical implementation, the virtualization technology selected for its features has been Docker (see Section 2.3.1). Docker is a well known and emerging container virtualization project with a simpler and lighter solution than previous approaches (see Section 2.3). The resource usage of Docker containers is efficient and their management is also simple. Docker v1.2 allows us to deploy multiple ESB instances in the same machine which share OS and libraries in a efficient way.

6.1. ServiceMix Characterization

As we said previously, a taxonomy and characterization of the supported ESBs must be performed from the beginning to facilitate the use of the platform to the developers. Therefore, the ServiceMix features must be revised and classified starting from the general categories we propose in Section 5.1.1. The information about our classification with the most relevant ServiceMix features can be found in Appendix B. The developer should use the taxonomy tree to find better which features may support a determinate protocol or tool he needs and afterwards, check its properties and description in the features table in Figure B to make a good decision.

The other aspect linked with the ServiceMix selection is the *resource configuration*. Each ESB solution has a different set of parameters that can be configured to limit the resource usage of the instance. In ServiceMix, we have identified the parameters that can be configured and what should be included in a particularization of the ESB configuration schema that we present later. All the configuration points in ServiceMix are included in configuration files in the /etc folder of the ServiceMix package.

Another aspect which depends on our ESB selection is how the developer can manage each instance. In ServiceMix, there are two ways that might be sufficient for developers:

- *Karaf console*. The developer can connect to the Karaf console via SSH to manage the ServiceMix instance. This console enables a CLI interface to manage the instance and also a SSH server to copy files into the deploy folder.

- *Webconsole*. ServiceMix includes a Web interface for management where developers can install or uninstalls features and bundles or other components.

After extracting the parameters dependent on ServiceMix, the ESB instance configuration file can therefore be described. The description of the complete ESB configuration is based on a JSON file, and it includes all the aspects described in Section 5.1.2 plus the properties identified after the ServiceMix analysis. The JSON schema of a ServiceMix is shown in Figure A.5. There we can see the fields related to the "labels", container ports, features installed and access credentials to the ServiceMix instance. It includes also parameters to control the CPU and memory usage of the container which holds the ServiceMix instance and other parameters related to the resource usage.

Despite of the fact that the ServiceMix JSON schema contains a big amount of parameters to be defined, many of them are optional. In Listing 6.1, a basic ServiceMix configuration file can be seen. It includes some arbitrary ports and features to make the example clearer.

```
{
   "Name":"system-configuration",
   "Configuration":{
      "ESB":"servicemix",
      "Ports":{
         "mycomponent1":4444,
         "mycomponent2":5555
      },
      "Features":[
         "camel-sql",
         "camel-http"
      ]
   }
}
```

Listing 6.1: Minimal configuration file for and ServiceMix instance

We want to mention that for our prototype, the *Custom Components* installation in the building process is not supported, but the developers can include their own components afterwards in the existing ServiceMix instances though the webconsole or the Karaf console.

6.2. ServiceMix Images Generation

Before describing the generation of the ServiceMix images, we must mention that with Docker some of the parameters can be externalized and specified during both building and running phases of the image and container, respectively. Hence, we must check which parameters specified in our ServiceMix configuration can be externalized. Normally, this includes all the parameters which describe variable values that can be modified from configuration files (i.e. JVM memory, number of threads, credentials, etc.). The ones that cannot be externalized are the installed features and custom components, because they are heavy files which must be downloaded and added to the middleware. As we want to launch ESB instance rapidly, the features and custom components must be directly included in the ServiceMix images to avoid a slow ESB instance launch. Otherwise these components would be downloaded in the very first start. The remaining ones can be introduced afterwards when the container starts as environmental variables.

In our implementation, each specialized ServiceMix is based on what we called *ServiceMix base image*. The ServiceMix base image includes all the libraries, dependencies and Operating System required to run ServiceMix. The reason to create such a base image, is because of how Docker handles the image building and storage. In Docker, the images are build in steps, and in each step a new image is created storing only the difference with the previous one. This enables that other images can be based on others without wasting additional disk space. This is useful and profitable for migrations, where only the difference between the migrated container and the existing ones in the destination machine is transferred. For example, if we have the same base image in all the machines which hold ServiceMix instances, the migrated data contains only the additional components and other files related to the execution of the particular instances. Moreover, if the destination machine contains the specialized ServiceMix image of the transferred ESB instance, only the execution data is transferred.

The creation of new images in Docker is based on what is called Dockerfiles. The Dockerfiles are the building files which include all the steps to create a new image and to configure it. In this file, we can be specified that the new image is based on other one.

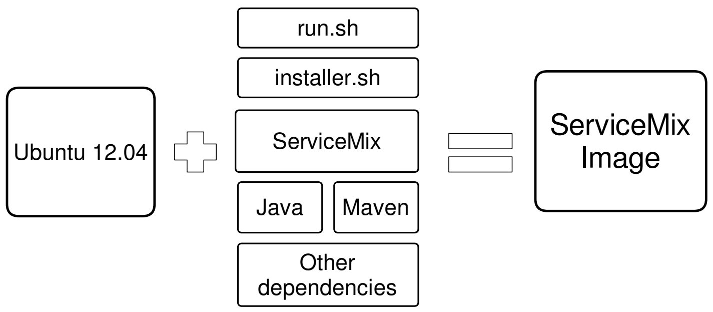


Figure 6.1.: ServiceMix base image components

For the prototypical implementation in this thesis, the ServiceMix base image is created from Ubuntu 12.04 image, because it is the verified and compatible version with ServiceMix 4.5 we have used in the implementation. After the selection of the base image, the dependencies must be installed. In our case it includes Java, Maven and some tools and files to handle the configuration of the parameters when a container is launched. For example, we have included a tool called *envtpl* [1], which renders templates with environmental variables, which can be specified when a container starts. For the creation of our base image, we have included two more additional files, *installer.sh* and *run.sh*. The installer.sh file is used in the image building to install the ServiceMix features that developers needs. For its part, the run.sh file contains all the logic to configure the externalized parameters and run the ServiceMix instance inside of the container. All the components included in the ServiceMix base image can be observed in Figure 6.1.

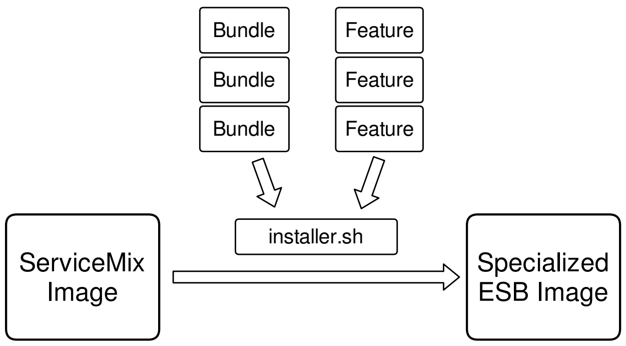


Figure 6.2.: Building a specialized ServiceMix image

The ServiceMix base image includes a minimal installation of the ServiceMix instance with only the necessary basic components but nothing else. Hence, this package includes the lightest configuration for ServiceMix. Later to create specialized ServiceMix images new components and features are installed in this image. After the installation of features the resource usage grows. However, the ServiceMix instance in this case holds only the required additional components and the base installation. No unnecessary components for the final application are included and therefore, no resources are wasted.

In order to create the specialized ServiceMix image our prototype generates a new Dockerfile which installs the features in the ServiceMix instance already included in the image using the ServiceMix base image as a basis. The features are installed by the installer.sh file. It receives an environmental variable with the list of the desired features and adds them to ServiceMix (see Figure 6.2). The Dockerfile template for the specialized images can be observed in Listing 6.2. A new Dockerfile is generated each time a new image must be created.

```
FROM totemteleko/servicemix

MAINTAINER totemteleko totemteleko@gmail.com

ENV features {{.Features}} # List of features list separated by
   comas. Ex: camel-sql,camel-http
```

[1]https://github.com/andreasjansson/envtpl

```
RUN /bin/sh -c /esb/installer.sh
```

Listing 6.2: Dockerfile template for specialized ServiceMix instances building

6.3. Image Storage

After creating the ESB images, these need to be persisted in the *Image Storage* registry. For our image storage implementation, we use an already built container with a Docker Registry inside, whose configuration and deployment are explained with more detail later. A Docker registry works as a repository and it stores the images and its previous versions. The images can be pushed and pulled from that Docker registry.

The command to run our image storage component on any Linux distribution with Docker installed is shown in Listing 6.3. The port binding is also specified in this command, 5000:5000, and the folder in the host machine where the images are stored , /registry.

```

$ docker run -p 5000:5000 -v /registry:/tmp/registry --name
    registry registry
```

Listing 6.3: Running the Image Storage

Images can be pushed and pulled from any machine which has a Docker agent installed. The image storage does not include any kind of authentication, hence it must be internal to the system, not accessible from outside. This component can be any Docker Registry running anywhere, so if the system administrator has an existing Docker registry in a remote location, he must only set the system to point to the registry. This particular configuration is explained further in the following sections.

6.4. Monitor

The implementation of the monitor component is based on the Google project cAdvisor. cAdvisor is a single-host monitoring tool which collects the information about CPU, memory and network in a container and host levels. It includes a REST API to expose the data in a JSON format and a web interface which provides a graphical view of the resource usage of the host machine and its running containers. Both are exposed though the port 8080 of the container. The command to run cAdvisor is shown in Listing 6.4.

```
$ docker run -t --volume=/var/run:/var/run:rw --volume=/sys:/sys:
    ro --volume=/var/lib/docker/:/var/lib/docker:ro --publish
    =8080:8080 --name=monitor google/cadvisor:latest
```

Listing 6.4: Running the Monitor

As cAdvisor is a single-host monitoring tool, it must be installed and launched in any machine of the cluster we must deploy. Its information is used by other components to take decisions. To request information the component must perform a HTTP GET request to the cAdvisor REST API to an address with the structure shown in Listing 6.5. The JSON output format must be checked in the cAdvisor website [2]. However, in our system administrator and developers must get this information through REST API using their credentials. Only the system components can access the monitoring information without the REST API.

```
GET http://<hostname>:<port>/api/<version>/<request>

Ex:
GET http://localhost:8080/api/v1.0/containers/foo
```

Listing 6.5: Get data from cAdvisor

6.5. Registry

In our prototype, we make use of CoreOS [2], further explained in Section 2.3.2. CoreOS comes with a distributed key-store called ETCD (see Section 2.3.2). ETCD is present in all the machines in the cluster, and synchronized between all of them. In case a machine is added to the cluster, it has immediately access to the information stored inside. On the other hand, when a machine leaves the cluster there is no loss of information. This is relevant for the dynamism of the cluster we seek. New machines can join to the cluster or leave any moment, and the cluster still works perfectly. For example, if the running machines are overloaded, the cluster could grow and hold more ESB instances. In contrast, if the existing machines in the cluster do not use their full power, they could be shutdown.

ETCD stores the objects by their keys, but it can also create folders to store collection of keys with similar properties. All the keys in the registry are accessible by a REST API, where a path can be specified to store a key. For example the key "foo" can be stored with the path "/system/documents/personal/foo", where "documents" and "personal" are folders which can store more keys. In addition, we want to mention that the keys can contain any string value, in our case we store everything in JSON objects. Listing 6.6 shows an example of how a HTTP request gets the key "mykey".

```
GET http://127.0.0.1:4001/v2/keys/mykey -d value="this is my key"
```

Listing 6.6: Getting the value of key in ETCD

[2] https://github.com/google/cadvisor

As ETCD is a non-relational database, the relations between the data in our prototype are handled in the business logic of the system. In our prototype, we have created three big folders: *image*, *user*, and *machine*. In the image folder we can find all the names and keys of the already built ESB images to identify them from a new configuration file and not building them again. In the machine folder the information about the machines is stored, each machine with an own key and JSON object with the schema described in Listing A.2. The other directory, users, is the most complex of the them, it contains a set of folders which each one belongs to one user. These folders contain a key with the information of the user, and two additional subfolders that contain the user configurations and ESB instances objects with the schemas defined in Listings A.4 and A.3 respectively. The structure of the registry can be observed better in Table 6.1.

Path	*Description*
/_esb/	This folder is the general folder where all our data inside ETCD is stored. It is an invisible directory, in ETCD the invisible directory names start with the character "_" .
/_esb/scheduler	This folder contains all the instances that must be scheduled because the machines in the cluster can hold them. This folder works as a queue, and the scheduler component pushes and pops instances to be launched during the runtime.
/_esb/image/	Here we can find all the names and keys of the already built ESB images to identify them from a new configuration file and not building them again.
/_esb/machine/	In this folder the information about machines is stored. Each machine is stored in a JSON format with a new key in this folder.
/_esb/user/	This folder holds all the information about users and their entities. A new folder is created for each new user.
/_esb/user/{id}/info	This key contains the information about an user in JSON format. It contains his name, password and other properties.
/_esb/user/{id}/instance/	Here all the created ESB instances of the user are stored.
/_esb/user/{id}/configuration/	This folder contains all the configurations added to system by the user

Table 6.1.: Description of the Registry structure based on ETCD

In addition in Table 6.1, a key called *scheduler* is displayed. It is a folder which contains all the instances that must be scheduled later because the machines in the cluster cannot hold them currently. This folder works as a queue, and the scheduler component pushes and pops ESB instances to be launched during the runtime. More details are given in Section 6.6.3

6.6. Controller & Cluster Deployment

For the cluster deployment of the ESB instances, we have chosen CoreOS [2], explained in Section 2.3.2. The reason of this selection is that CoreOS matches with some aspects of our design. Apart from the already mentioned ETCD, it comes with Fleet (see Section 2.3.2), which enables the deployment of Docker containers though different machines ensuring their correct performance and migration from one machine to another. However, CoreOS cannot cover all the aspects of our design. Therefore, we have used CoreOS as a basis for our platform, not only using some of its features in the different components of our design, but also extending its functionalities to face other problems. More information about our implementation based on CoreOS for each particular component in our design is given in the the remaining of this section.

We want to point out that all the components included in the controller component, the REST API and the Command Line Tool have been developed using the Go programming language [77]. There are several reasons for choosing Go as the programming language in our platform. For instance, the Go source code can be compiled into different platforms, Linux, Windows or Mac OS X, and, of course, to work in CoreOS. Fleet and ETCD have been written in Go and there are some libraries that can be used to work directly with them instead of making use of their REST APIs. In addition, Go provides a lot of different libraries and contains libraries to develop easily Command Line Tools, REST APIs or to marshall and unmarshall JSON objects. For our prototype, we have developed a Go library which includes all the functions to control the entire platform. The controller component, REST API and CLI use such library and in that way, we reduce the amount of generated code.

6.6.1. Health

The health component is a complete new component we have added to CoreOS in order to make monitoring of each one of the machines more accessible, and to provide some additional information. It is installed in each one of the machines and collects the host machine date from the monitor component to determine the state of the machine, as it is defined in Section 5.6.1. It also publishes the role of the host machine, whose election is already performed by Fleet. Fleets selects a new leader periodically and the health component gathers this information. All the host information is stored in the system registry as a machine object with the schema shown in Listing A.2.

The health component gathers the information periodically from the monitor component. Later it calculates the average of the CPU and memory during this interval. The interval can be defined in the cloud configuration file as we explain later. After calculating the average values, it decides in which state the machine is based on different thresholds, checks if the host machine is the leader and publishes the data and some more properties as the host IP address into ETCD (see Figure 6.3). It performs these tasks periodically, how often as the parameter which defines the interval specifies.

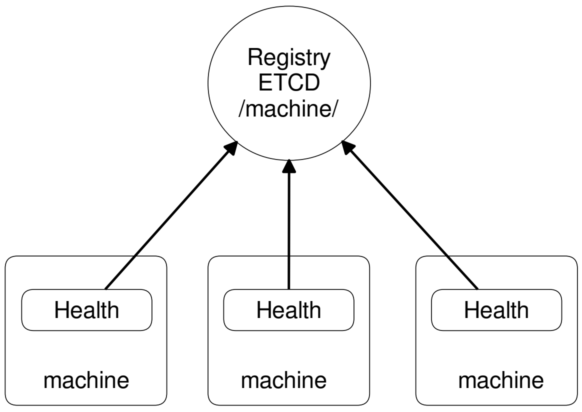


Figure 6.3.: The health component pushing the machine information into ETCD

We want to mention how the health component gets the role of the host machine. In CoreOS the leader is selected periodically between all the machines in the cluster by the Raft consensus algorithm [78]. All the machines bid to be the leader, until they get an agreement of which one must be. This process takes some time, and the information about who is the leader is not available during this time. The health component implementation waits until the new leader is selected if when it updates the status information the leader election is being done. Who is the leader of the cluster is stored in ETCD and Listing 6.7 shows where this information can be found.

```
GET http://127.0.0.1:4001/v2/keys/_coreos.com/fleet/lease/engine-
    leader
```

Listing 6.7: Discovering the cluster leader

The different parameters to configure the interval and thresholds of the different states of the machines are detailed next:

- HEALTH_INTERVAL: defines the periodicity of the information updates.
- HEALTH_MIN_CPU: describes the threshold between the idle and normal states based on the CPU usage.
- HEALTH_MIN_CPU_BUSY: determines the threshold between the normal and busy states based on the CPU usage.
- HEALTH_MAX_CPU: sets the threshold between the busy and critical states based on the CPU usage.
- HEALTH_MIN_MEM: describes the threshold between the idle and normal states based on the memory usage.
- HEALTH_MIN_MEM_BUSY: determines the threshold between the normal and busy states based on the memory usage.

- HEALTH_MAX_MEM: defines the threshold between the busy and critical states based on the memory usage.

Figure 6.4 shows the relationship between these parameters and the states of the machines. How to set up these parameters is shown in Section 6.9.

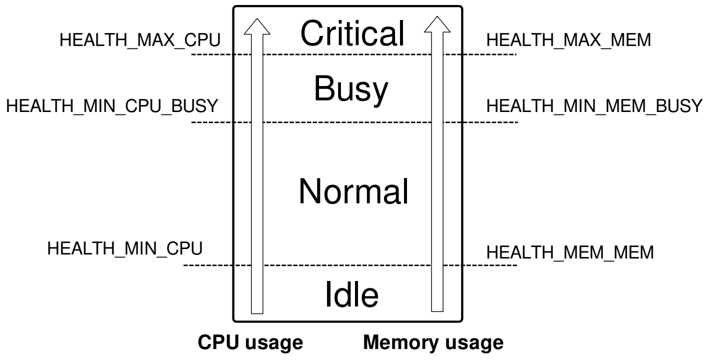


Figure 6.4.: Relationship between the Health parameters with machine states

In addition, the health component must update the metadata of the CoreOS machine with the field "health={newstate}". This information is used by the scheduler component, something that is explained with more detail in Section 6.6.3.

6.6.2. Manager

The manager component, defined in Section 5.6.2, controls if ESB instances are running or not and their behavior when they start or stop. To take care about these tasks, CoreOS uses Fleet, an extension of Systemd [44]. Systemd is a system management daemon which controls a collection of services or processes and how they behave. For its part, Fleet relies on Systemd to run such services in a cluster.

In order to make use of this aspect, our ESB instance must be encapsulated inside a Fleet Unit, which includes a Systemd service inside. Some properties and behavior of the unit can be specified. For example, we can set that when an unit stops, some task must be performed (i.e. removing something from a database, sending a notification, etc.). An example file of how we can run an ESB instance inside one of these units is shown in Listing 6.8, where some of the options of the docker command of the final description file have been excluded for brevity. In that listing, we can also see that some commands can be specified when the unit starts or stops and before starting or after stopping.

```
[Unit]

Description=Launcher for the Instance: myinstance
Requires=docker.service

```

```
[Service]

ExecStartPre=/bin/bash -c "/usr/bin/docker pull esbimage "
ExecStart=/bin/bash -c "/usr/bin/docker run -t -P --name
   myinstance esbimage
ExecStop=/bin/bash -c "/usr/bin/docker stop myinstance "
ExecStopPost=/bin/bash -c "/usr/bin/docker rm myinstance "
```

Listing 6.8: Creating a Fleet Unit for a Docker Container

6.6.3. Scheduler

The scheduler component is the most complex of the components in our framework. For its implementation we make use of CoreOS and Fleet. Fleet covers some of the aspects described in Section 5.6.3 such as the failure recovery and a basic scheduling of the Fleet Units. However, its scheduler is really simple. Fleet scheduling algorithm is only based on which machine has less units deployed. In contrast, in our design we establish that the scheduling must also be based somehow on the amount of resources in use in the host machines. This can be relevant, because the ESB instances consume different amount of resources depending on the installed components, and the processed traffic. Therefore, in our implementation an extension of the default Fleet scheduler has been done. We explain this extension in three parts: scheduling, rescheduling in case of machine is out of resources, and rescheduling in case of failure.

For the scheduling part, we use the default Fleet scheduler but adding two layers before. The first step of our scheduler is checking if there are machines which resources to hold more ESB instances. To achieve this, the scheduler component uses the data provided by the health component. Depending on the state of the machines (see Section 5.6.1), the scheduler sends it to the next layer or enqueues it in a waiting list until there are available machines. The scheduler component enqueues the new ESB instance, if there is no machine in the cluster in "idle" or "normal" state. This tier of the scheduler checks periodically if it can send the ESB instances to the next step of the scheduling.

In the next step, the scheduler component decides which machine may be less loaded, using the information from the health component and Fleet. It assigns the units first to the machines with idle state, after in normal and between the machines with the same state, the one with less running ESB instances. Consequently, the corresponding units for the ESB instances are sent to the Fleet default scheduler. All this procedure can be seen in Figure 6.5. In addition, we want to mention that the second step of this process is done using the metadata of the CoreOS machines. As we said, the health component updates this metadata periodically and a preference order can be specified to select the machine where Fleet sends the units. This can be done using additional parameters in the Fleet Unit as we can see in Listing 6.9.

```
[X-Fleet]
MachineMetadata=health=idle
```

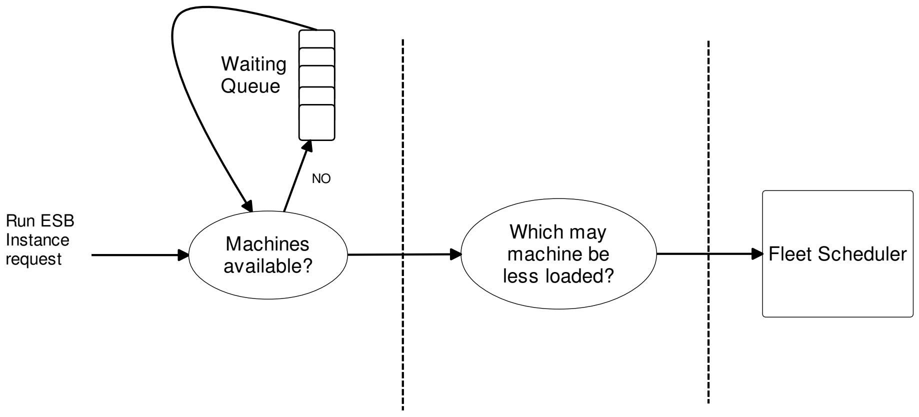


Figure 6.5.: Implementation: Scheduling steps

```
MachineMetadata=health=normal
MachineMetadata=health=busy
```

Listing 6.9: Implementation: Scheduling to less loaded machines

The implemented scheduler supports rescheduling in case of machine out of resources as it is defined in Section 5.6.3. It is a complete new feature for the Fleet scheduler. It uses also the information provided by the health component to check the state of machines periodically and to detect if the ESB instances of a determinate machine must be migrated to another one. This component is installed in all the machines, an each machine decides if its own ESB instances must relocated or not. The scheduler agent present in the machine, if a state *critical* is detected, stops and sends its instances to the scheduler component again. Hence, these instances can stay waiting until another one of machines can hold them.

The last part of the scheduler component covers the recovery of the units in case of failure. These functionality is supported fully by Fleet, so in our implementation we only make use of it. Fleet detects if a machines fails, and in that case, migrates all the instances that where running in that machine to the rest of the machines in the cluster.

Despite of describing the scheduler implementation, and how it migrates the ESB instances in specific cases, we have not mentioned how the platform deals with these situation and the data stored in the instances during its execution. These and other aspects are explained in Section 6.6.4.

6.6.4. Lifecycle & Deployment of ESB instances

After describing such amount of different components and implementations, we want to explain how an ESB instance really works in our prototype. What we define in Section 5.2 as ESB instance has a more or less complex behavior and structure. In contrast with what is said in that section, in our implementation there are two main states, *stopped* and *running*, and two intermediate states to do some tasks in the transition from stopped to running and viceversa. We call these two new states *stopping* and *starting*.

The ESB instance definition and management in our implementation require a combination of different components and engines. Firstly, the ESB package is encapsulated inside a Docker container which runs in a CoreOS machine. Then for the container management we have three different components integrated in the controller component, as we represent in Figure 6.6. In this feature we can see also the components of the controller and the components used by it for the implementation part.

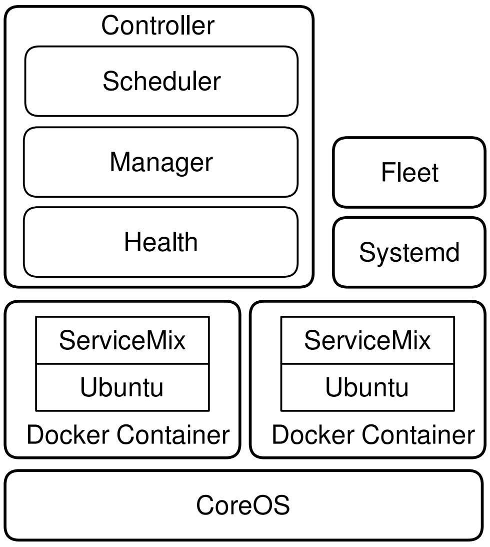


Figure 6.6.: Stack of components for a running ESB instance

Now we want to explain the lifecycle of the ESB instances in the system and its behaviour in all the states and transitions. When an ESB instance is created, a new ESB image is stored in the image storage, in a stopped state. When its owner wants to run it, the ESB instance is sent to the scheduler component which can enqueued or launch it in one of the machines. During the time the ESB instance is in the waiting queue, its state is called starting. Once the scheduler component decides to run this instance in one of the machines, the machine starts pulling its image. When the pull finishes, the ESB instance starts in the machine, state called running. In this state, it is when the ESB instance receives an IP address and its ports are bind to host ports. The owner can access in this moment to the instance for management proposes or for using it in his system. During this state is when the *minion* component appears. The minion in our implementation has two main tasks, updating the information about the running ESB instance (IP address, ports, state, etc.) and committing the execution state into the image storage. When a ESB instance is running, the minion has a mechanism to deal with a possible failure of the host machine. It commits the ESB instance state periodically to assure the data protection. The information the minion publishes of the instance expires in some seconds, and in that way, if a failure appears, the developer do not see the instance in running state when it is not really running. When the developer requests to stop an ESB instance, this goes into stopping state. In this state, the container information is pushed and saved in the image storage again to be able to restart the ESB instance in the same conditions in the future. When this procedure ends, the ESB Instance is again in a stopped state, when it can be removed or restarted again. All this information about the lifecycle of the ESB instance is shown in Figure 6.7.

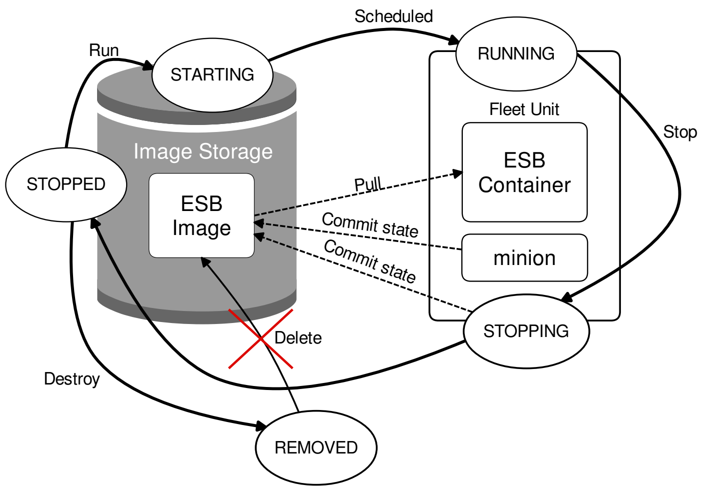


Figure 6.7.: ESB instances lifecycle

The implementation of all that behavior has been done by creating three different Fleet Units for the ESB instance. One includes the Docker container start and stop actions, other one the information updates, and other one to store periodically the ESB instance state. The three files are generated for each one of the ESB instance when these are created. The Fleet files are immutable during its existence. A skeleton of these three Fleet units is given in Listing D.1 in order to give a general idea about how this can be done.

6.7. Command Line Interface

The Command Line Interface provided for the system administrator has been fully developed in Go, and includes all the commands as defined in Section 5.9. It makes uses of the Go library we mentioned previously, and therefore, its code covers aspects related to CLI input and output. The functions which include the action to manage the platform are in the library, and in the CLI implementation are only called. For implementing the commands, we use a Go library *cli*[3] which has helped us enormously because it facilitates several aspects such as options and arguments handling. Listing D.2 shows an example of how a new command can be created for the CLI.

[3] `https://github.com/codegangsta/cli`

6.8. REST API

For its part, the REST API has been developed in Go as well, and it handles all the requests described in Section 5.8. As the CLI, it uses our Go library, and therefore it only deals with the routing and the input/output messages and requests. Another Go library *rest* [4] has been used to implement the REST API in our prototype.

Finally, as we said previously the REST API and the CLI are installed in all the machines by default in order to work with all of them in the same conditions. If the REST API is deployed in all the host machines, machines can be removed without problems. This implementation follows our distributed arquitecture where all the machines are identical. Moreover, this also enables the distribution of requests from the developers through all the machines overall the requests which may produce a new image building and in that way, avoiding that machines may go out of resource because of system tasks.

6.9. Configuration of the Host Machines

One of the thing we seek in our design and implementation is a easy installation of the framework in order to enable a rapid deployment of new machines in the cluster. In our implementation this is done by the Cloud init file of CoreOS. In this file the behaviour of the CoreOS machine when it starts (for installing things, running services, etc.) can be specified. For our implementation we have created one of these files to configure the machines. This file is the same for all the machines in the cluster, and new machines only must be launched with the same file to join into the cluster, including all its components (registry, controller, monitor, etc.). In this file, some parameters can be configured. For example, the ports where the monitor and the REST API run or the defining parameters for the health state detection. A template of this file is shown in Listing E.1, where all the components of our platform are configured and launched. Finally we want to mention that our framework can run in all the platforms that offer support for CoreOS including Vagrant [43], Amazon EC2 [13], VMware [28] and OpenStack [17].

[4]`https://github.com/ant0ine/go-json-rest/rest`

7. Validation

In order to validate our prototype, we are going to follow a sequence of actions that emulates the system administrator and developer behavior when using the proposed framework. This sequence is shown in Figure 7.1.

In terms of environmental setup, we test our prototype launching a cluster of three CoreOS machines on Amazon EC2, and a separate machine running the image storage. After the cluster starts, the administrator checks the running machines in the cluster and starts adding new users who can use the platform. Along this chapter, we perform all the commands though the REST API, including the administrator commands which can also be done by the CLI instead of using the REST API. In addition, we want to mention that all the requests require basic authentication and in case of administrator request the default user and password are admin and admin respectively.

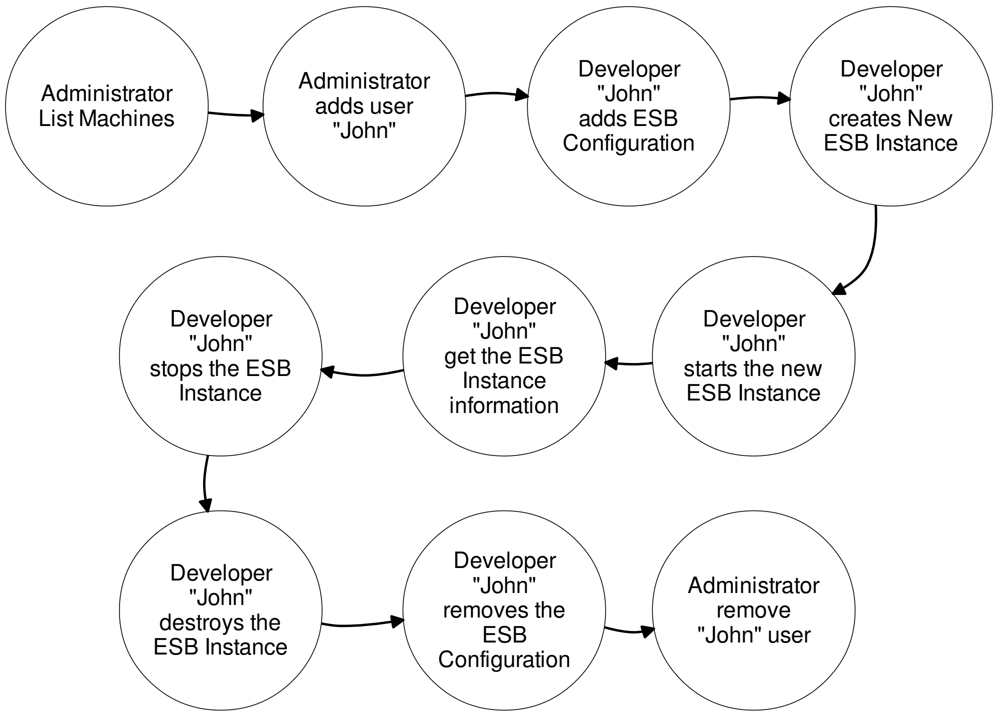

Administrator List Machines
Administrator adds user "John"
Developer "John" adds ESB Configuration
Developer "John" creates New ESB Instance
Developer "John" starts the new ESB Instance
Developer "John" get the ESB Instance information
Developer "John" stops the ESB Instance
Developer "John" destroys the ESB Instance
Developer "John" removes the ESB Configuration
Administrator remove "John" user

Figure 7.1.: Sequence of actions followed for validation

```

GET /admin/machine HTTP/1.1
```

```
Host: 54.172.183.241:4000
Authorization: Basic YWRtaW46YWRtaW4=
Cache-Control: no-cache

Response:

Status: 200 OK
Content-Length: 1145
Content-Type: application/json
Date: Fri, 17 Oct 2014 12:56:19 GMT
X-Powered-By: go-json-rest


[
    {
        "ID": "b8645b9d2f354c8c810fe43137f8d2c2",
        "Ip": "172.31.20.193",
        "Role": "leader",
        "Status": {
            "State": "normal",
            "Cpu_usage": 4.2119918938728285,
            "Mem_usage": 39.67685397466127
        }
    },
    {
        "ID": "2d270a222cda41e5b9192222528d598f",
        "Ip": "172.31.20.194",
        "Role": "follower",
        "Status": {
            "State": "normal",
            "Cpu_usage": 1.507448974349744,
            "Mem_usage": 49.74393475245864
        }
    },
    {
        "ID": "55dbd07081ae455ab2e5977c287ed5c3",
        "Ip": "172.31.20.195",
        "Role": "follower",
        "Status": {
            "State": "normal",
            "Cpu_usage": 1.4300149192578373,
            "Mem_usage": 52.16419876463377
        }
    }
]
```

Listing 7.1: Validation: Get the Machines in the cluster

After the administrator checks that the machines are working as expected (there is one leader, all machines are in a normal or idle state, etc), he creates a new user called "John".

```
POST /admin/user HTTP/1.1
Host: 54.172.183.241:4000
Authorization: Basic YWRtaW46YWRtaW4=
Cache-Control: no-cache

{"Name":"john", "Password":"passw0rd"}

Response:

Status: 201 Created
Content-Length: 24
Content-Type: application/json
Date: Fri, 17 Oct 2014 12:57:13 GMT
X-Powered-By: go-json-rest

{
  "Value": "Created"
}
```

Listing 7.2: Validation: Add a new user

Now that the developer John has access to the system, he can start creating new specialized ESB instances. However, the first thing he must do is adding the configuration of the ESB instance he want to create in the system.

```
POST /configuration HTTP/1.1
Host:  54.172.183.241:4000
Authorization: Basic am9objpwYXNzdzByZA==
Cache-Control: no-cache

{ "Name":"myconfig", "Configuration":{"ESB":"servicemix", "Ports
   ":{ "endpoint1":4444,"endpoint2": 5555 }, "Features":[ "camel-
   sql", "camel-http" ] } }

Response

Status: 201 Created
```

```
Content-Length: 24
Content-Type: application/json
Date: Fri, 17 Oct 2014 12:58:54 GMT
X-Powered-By: go-json-rest
```

Listing 7.3: Validation: The developer adds a new ESB configuration

The developer creates a new ServiceMix instance in the system making a request with the name of the already added configuration.

```
POST /instance?configuration=myconfig HTTP/1.1
Host: 54.172.183.241:4000
Authorization: Basic am9objpwYXNzdzByZA==
Cache-Control: no-cache

Response:

Status: 201 Created
Content-Length: 1066
Content-Type: application/json
Date: Fri, 17 Oct 2014 13:01:02 GMT
X-Powered-By: go-json-rest

{
    "ID": "e-19747eac-f5eb-4053-9091-8306",
    "Labels": {
        "owner": "john"
    },
    "Created": "Sunday, 19-Oct-14 02:04:05 UTC",
    "State": "stopped",
    "Host": "",
    "Configuration": {
        "CPU_SHARED": "",
        "ESB": "servicemix",
        "Features": [
            "camel-http",
            "camel-sql"
        ],
        "ID": "i-7d03dda3-39c3-43a2-8d25-7eb5",
        "JBI_allowCoreThreadTimeOut": "true",
        "JBI_corePoolSize": "4",
        "JBI_keepAliveTime": "60000",
        "JBI_maximumPoolSize": "-1",
        "JBI_queueSize": "1024",
        "JBI_shutdownTimeout": "0",
        "JVM_MAX_MEM": "",
        "JVM_MAX_PERM_MEM": "",
        "JVM_MIN_MEM": "",
        "JVM_PERM_MEM": "",
        "MAX_MEM": "",
        "NMR_allowCoreThreadTimeOut": "true",
        "NMR_corePoolSize": "4",
        "NMR_keepAliveTime": "60000",
        "NMR_maximumPoolSize": "-1",
        "NMR_queueSize": "1024",
        "NMR_shutdownTimeout": "0",
        "Name": "myconfig",
        "Ports": {
            "endpoint1":4444,
            "endpoint2":5555
        },
        "SERVICEMIX_PASSWORD": "smx",
        "SERVICEMIX_USER": "smx"
    },
    "Ports": {
        "4444": "",
        "5555": "",
        "8101": "",
        "8181": ""
    }
}
```

Listing 7.4: Validation: The developer creates a new ESB instance

Once the ESB instance has been created, the developer receives a response with the data of the new instance. He must check the ID field of the new component in order to manage it. After its creation the ESB instance is launched specifying its ID.

```

PUT /instance/e-19747eac-f5eb-4053-9091-8306/start HTTP/1.1
Host: 54.172.183.241:4000
Authorization: Basic am9objpwYXNzdzByZA==
Cache-Control: no-cache

Response:

Status: 200 OK
Content-Length: 26
```

```
Content-Type: application/json
Date: Fri, 17 Oct 2014 13:02:57 GMT
X-Powered-By: go-json-rest
```

Listing 7.5: Validation: The developer starts a stopped ESB Instance

The next step is checking the running attributes of the ESB instance. The developer John checks the IP address where it is running and the port bindings to use and manage it. Now he can connect to the ServiceMix instance and install new bundles though the host ports assigned to the ports, 8101 (Karaf console) and 8181 (webconsole), which are respectively the ports, 49155 and 49156.

```

GET /instance/e-19747eac-f5eb-4053-9091-8306 HTTP/1.1
Host: 54.172.183.241:4000
Authorization: Basic am9objpwYXNzdzByZA==
Cache-Control: no-cache

Response:

Response:

Status: 200 OK
Content-Length: 1262
Content-Type: application/json
Date: Fri, 17 Oct 2014 13:05:30 GMT
X-Powered-By: go-json-rest

{    "Name":"myconfig",
     "ID": "e-19747eac-f5eb-4053-9091-8306",
     "Labels": {
         "owner": "john"
     },
     "Created": "Friday, 17-Oct-14 23:05:04 UTC",
     "State": "running",
     "Host": "54.172.183.241",
     "Configuration": {
         "CPU_SHARED": "",
         "ESB": "servicemix",
         "Features": [
             "camel-sql",
             "camel-http
         ],
         "ID": "i-124691ed-f613-4ea5-ab6b-05c7",
         "JBI_allowCoreThreadTimeOut": "true",
         "JBI_corePoolSize": "4",
         "JBI_keepAliveTime": "60000",
         "JBI_maximumPoolSize": "-1",
         "JBI_queueSize": "1024",
         "JBI_shutdownTimeout": "0",
         "JVM_MAX_MEM": "",
         "JVM_MAX_PERM_MEM": "",
         "JVM_MIN_MEM": "",
         "JVM_PERM_MEM": "",
         "MAX_MEM": "",
         "NMR_allowCoreThreadTimeOut": "true",
         "NMR_corePoolSize": "4",
         "NMR_keepAliveTime": "60000",
         "NMR_maximumPoolSize": "-1",
         "NMR_queueSize": "1024",
         "NMR_shutdownTimeout": "0",
         "Name": "myconfig",
         "Ports": {
             "endpoint1":4444,
             "endpoint2":5555
         },
         "SERVICEMIX_PASSWORD": "smx",
         "SERVICEMIX_USER": "smx"
     },
     "Ports": {
         "4444": "49153",
         "5555": "49154",
         "8101": "49155",
         "8181": "49156"
     }
 }
```

Listing 7.6: Validation: The developer gets the information of a running ESB instance

After using it, the developer proceeds to stop it to restart in the future or to destroy it as it is done in Listing 7.8.

```
PUT /instance/e-19747eac-f5eb-4053-9091-8306/stop HTTP/1.1
Host: 54.172.183.241:4000
Authorization: Basic am9objpwYXNzdzByZA==
Cache-Control: no-cache

Response:
```

```

Status: 200 OK
Content-Length: 24
Content-Type: application/json
Date: Fri, 17 Oct 2014 13:07:14 GMT
X-Powered-By: go-json-rest
```

Listing 7.7: Validation: The developer stops a running ESB instance

```

DELETE /instance/e-19747eac-f5eb-4053-9091-8306 HTTP/1.1
Host: 54.172.183.241:4000
Authorization: Basic am9objpwYXNzdzByZA==
Cache-Control: no-cache

Response:

Status: 200 OK
Content-Length: 26
Content-Type: application/json
Date: Fri, 17 Oct 2014 13:09:12 GMT
X-Powered-By: go-json-rest
```

Listing 7.8: Validation: The developer destroys an existing ESB Instance

After making use of the configuration for launching so many ServiceMix instances as he wants, the developer removes this configuration from the system.

```
DELETE /admin/user/john HTTP/1.1
Host: 54.172.101.91:4000
Authorization: Basic am9objpwYXNzdzByZA=
Cache-Control: no-cache


Response

Status: 200 OK
Content-Length: 24
Content-Type: application/json
Date: Fri, 17 Oct 2014 13:011:14 GMT
X-Powered-By: go-json-rest
```

Listing 7.9: Validation: The developer destroys the previously created configuration

Finally, the administrator removes the user "John" from the system when he does not need longer the platform.

```
DELETE /admin/user/john HTTP/1.1
Host:  54.172.183.241:4000
Authorization: Basic YWRtaW46YWRtaW4=
Cache-Control: no-cache


Response

Status: 200 OK
Content-Length: 24
Content-Type: application/json
Date: Fri, 17 Oct 2014 13:16:57 GMT
X-Powered-By: go-json-rest
```

Listing 7.10: Validation: The administrator removes the user

8. Evaluation

In the ESB configuration some parameters about resource usage can be specified. For this evaluation we have deployed different ESB instances setting maximal values for CPU and memory. Therefore, we focus on how different parameters can affect to the resources usage for different ESB instances running in our deployed system and how the increment of the number of ESB instances running in the same machine can affect. In this chapter, we describe first the environment to run all the scenarios, and afterwards, the two scenario to be tested. Finally, the results of the experiment are shown and some conclusions from this results are exposed.

8.1. Evaluation Setup

For the evaluation, we have tested two different scenarios using Amazon EC2 platform for the deployment of a cluster of CoreOS machines and JMeter [79] for workload generation. The JMeter is deployed in our local workstation and it generates the HTTP traffic for a MediaWiki server which go though the ServiceMix instances, as we can see in Figure 8.1. For both scenarios, we have deployed a CoreOS cluster with three machines in the platform Amazon EC2. The machines are three t2.micro instances with 600mb of memory and one vCPU. The configuration of each one of the machines and therefore, the cluster configuration, is performed by a cloud init file. A template for this configuration can be found in Appendix E. Moreover, all the ESB instances are used to route HTTP requests to a backend MediaWiki Application running in an external machine (m3.xlarge instance in Amazon EC2). The HTTP endpoint to route the traffic through ServiceMix has been created by a Camel route using the feature *camel-jetty*. This feature is one the several alternatives we could have used, which can be found through the REST API.

The different ESB configurations for all the deployed ESB instance contain the same features and ports. However, we have deployed three different ESB instances depending on the resource usage. The first one contains the default resource parameters (we do not need to specify them), the second one uses more CPU resources (8 threads instead of 4 for the ServiceMix Instance) and the third one sets a maximal value for the memory of 256mb. The configuration files to create such instances are shown in Listings 8.1, 8.2 and 8.3.

The measurements for this evaluation are retrieved from the health component and JMeter. The health component gathers every 10 seconds information about the memory and CPU usage in the host machines in order to compare them in different cases, and JMeters collects the thoughput values in both scenarios.

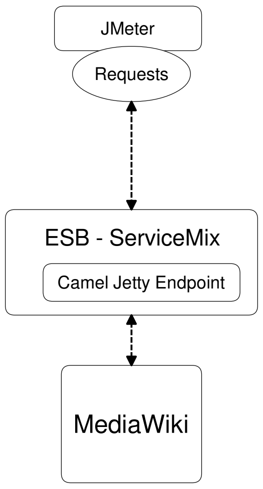


Figure 8.1.: Evaluation - MediaWiki exposed though ServiceMix

```
{
  "Name" :"mediawiki-esb",
  "Configuration":{
    "ESB":"servicemix",
    "Ports":{
      "mediawiki": 8150
    },
    "Features":[
      "servicemix-camel",
      "camel-jetty",
      "servicemix-http"
    ]
  }
}
```

Listing 8.1: Evaluation - Configuration with default resource parameters

```
{
  "Name" :"mediawiki-esb_cpu",
  "Configuration":{
    "ESB":"servicemix",
    "Ports":{
      "mediawiki": 8150
    },
    "Features":[
      "servicemix-camel",
      "camel-jetty",
      "servicemix-http"
    ],
    "JBI_corePoolSize": "8",
    "NMR_corePoolSize": "8"
  }
}
```

Listing 8.2: Evaluation - Configuration with more CPU capacity

```
{
  "Name" :"mediawiki-esb_mem",
  "Configuration":{
    "ESB":"servicemix",
    "Ports":{
      "mediawiki": 8150
    },
    "Features":[
      "servicemix-camel",
      "camel-jetty",
      "servicemix-http"
    ],
    "MAX_MEM": "256M"
  }
}
```

Listing 8.3: Evaluation - Configuration with limited memory

8.2. Description of the Scenarios

In our first scenario, we have deployed three different ESB instances for a single developer as Figure 8.2 shows. Each one of the three instance has a different configuration, configurations described in previous listings. The workload generated by JMeter has been 100000 thousand requests between 10 users going though all the ESB instances. The target of this scenario is the deployment of ESB with different configurations to analyse the different use of resources in the three cases.

Scenario	Hosts	ESBs	ESBs/host	Users/ESB	Requests	Requests/User
Scenario 1	3	3	1	10	100000	3333
Scenario 2	3	6	2	10	100000	1666

Table 8.1.: Evaluation - Summary of the scenarios

For the second scenario, shown in Figure 8.3, we have deployed three different ESB instances for two different developers each one of the configuration shown previously. In total, six ESB instances are deployed, three per each developer. The workload generated by JMeter has been 100000 requests between 10 end users going though all the ESB instances. The target of this scenario is evaluating the increase of the use of resources with multiple instances running on the same machine. A summary with the specific values in both scenarios is shown in Table 8.1.

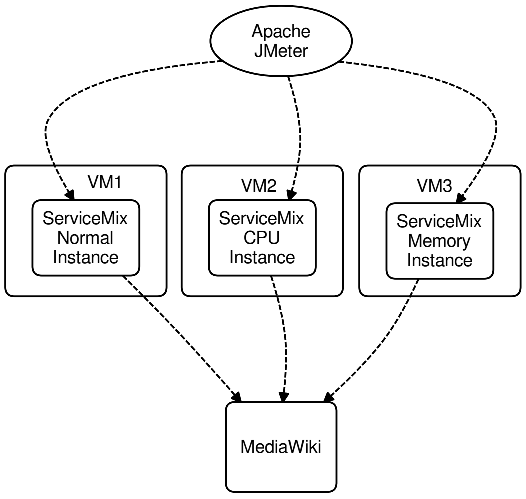


Figure 8.2.: Evaluation - Environment Scenario 1

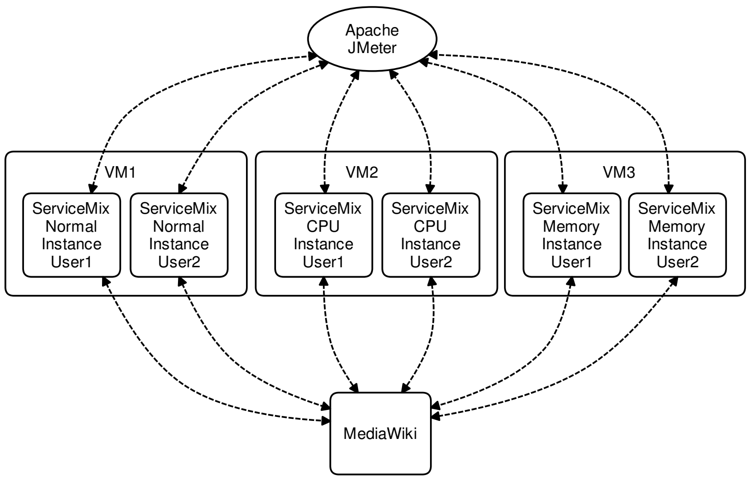

Apache JMeter
VM1
ServiceMix Normal Instance User1
ServiceMix Normal Instance User2
VM2
ServiceMix CPU Instance User1
ServiceMix CPU Instance User2
VM3
ServiceMix Memory Instance User1
ServiceMix Memory Instance User2
MediaWiki

Figure 8.3.: Evaluation- Environment Scenario 2

8.3. Evaluation Results

For each scenario we have collected average throughput shown by all the ESB instances and measurements about CPU and memory usage on the host machines. In both scenarios as we can see in Figures 8.4 and 8.6, CPU usage is slightly superior for the machine which holds ESB instances with more threads than for the other two machine. We want to point that ServiceMix with the components we have used during this work uses few CPU resources, and that's the reason because difference between the machines is not bigger.

On the other hand, in Figures 8.5 and 8.7 we can appreciate that memory usage is almost the same for the three machines despite of one holds limited memory ESB instances. The reason is that the memory usage is stable when comparing instances with the same installed components which receive the same traffic. Another reason is is that the selected maximal value 256mb was enough for a normal performance of the ServiceMix instance which was executed in a machine with only 600mb.

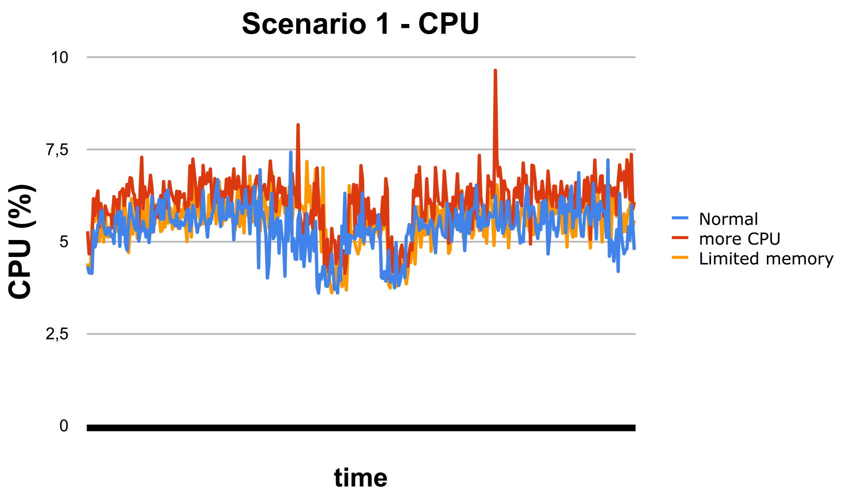


Figure 8.4.: Evaluation - Scenario 1 CPU results

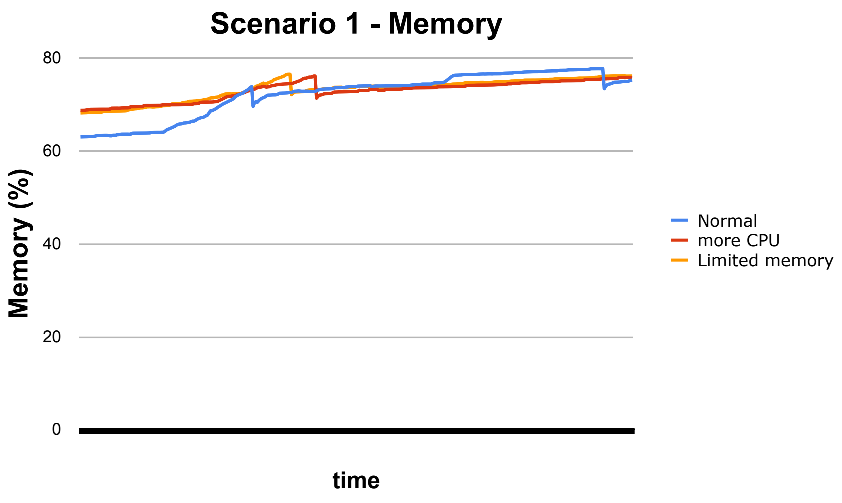


Figure 8.5.: Evaluation - Scenario 1 Memory results

If we compare the figure of both scenarios, we can see that the resource usage with two ESB instances running on the same machines has grown slighty taking into account that ServiceMix uses a considerable amount of memory by default. Therefore, we could say that adding new ESB instances to a machine with an already running ServiceMix instance does not require so many resources as launching an initial one because of the virtualization technology used, Docker.

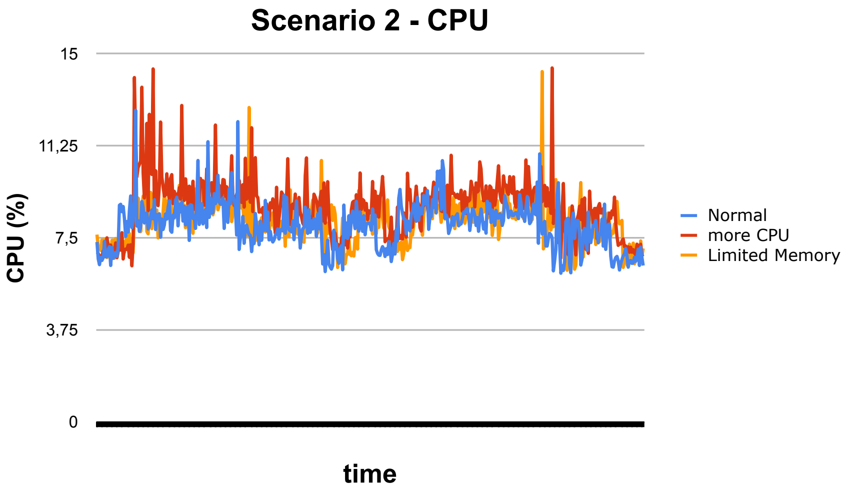


Figure 8.6.: Evaluation - Scenario 2 CPU results

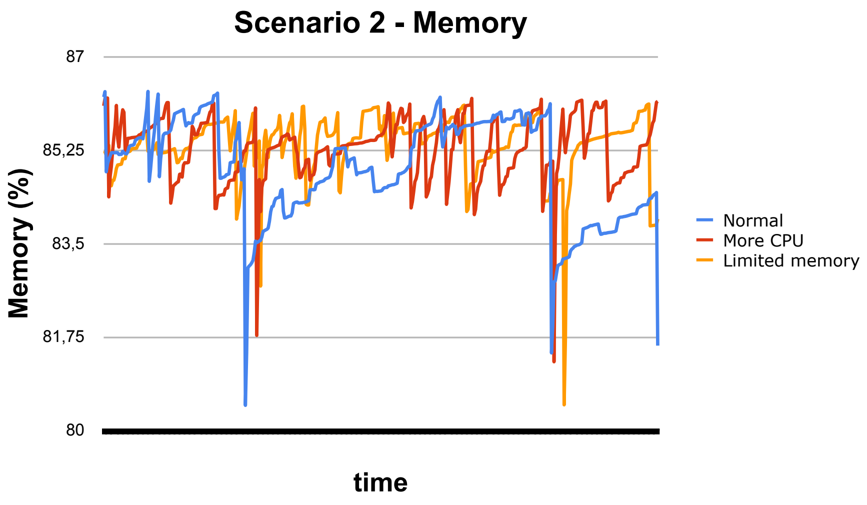


Figure 8.7.: Evaluation - Scenario 2 Memory results

Figure 8.8 depicts the average throughput shown in our two scenarios by all the ServiMix instances. We can see that the throughput has decreased for the second scenario where we had six Servicemix instances running comparing with the first one where we had only three instances. We can conclude that the number of ServiceMix instances running in the same machine can affect slightly the performance of the ServiceMix instances.

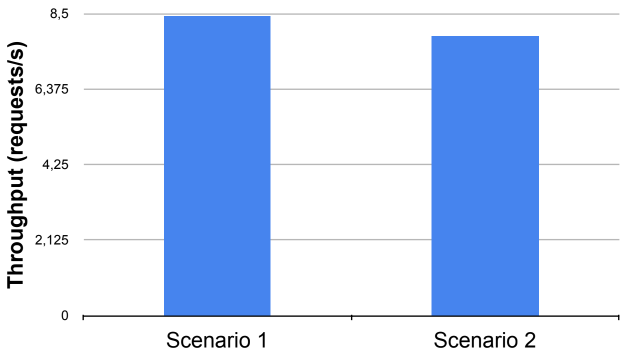


Figure 8.8.: Evaluation - Thoughput comparison

9. Conclusion and Future Work

The characterization and creation of light-weight ESB components significantly reduces the overall resources consumption in an infrastructure. However, the characterization of these ESB instances is commonly a complex process the developers must face. In this thesis we focus on providing the means to assist application developers in the tasks related to the characterization of a customized ESB that satisfies the application requirements. We started by analysing the ESB software in order to identify its variability points. After identifying the variability points, a taxonomy of the features or components in the ESB was performed. The number of features in an ESB is vast and the developer may face challenges in finding out which features are the most appropriate to use for his application.

Chapter 2 exposes main ideas in this work to set the necessary background. The origin of terms such as Cloud Computing and virtualization is explained in more detail. Additionally, more information regarding some relevant projects and tools this work relies on are given (i.e. Docker, CoreOS, ServiceMix, etc.). Chapter 3 presents some similar works related to containers deployment and middleware characterization and deployment. In these works we found that the configuration of an ESB can become a time constraint for the application developer. Moreover, these projects claim that a posterior reconfiguration of the ESB may be even more complex that the initial configuration. Therefore, a characterization of the ESB must be performed to streamline the resource usage. Further investigations showed that a reconfigurable ESB does not offer any advance over a non-reconfigurable ESB. Hence, we focus in this thesis on the characterization of an ESB to provide customized and reusable ESB instances. In the projects related to container deployment we saw aspects related to how our ESB instances can be dynamically provisioned and managed. The proposed solution includes only the deployment of ESB instances inside using a container virtualization approach. Containers share the same OS, libraries and kernel and therefore, memory, CPU and disk usage with this solution can be much more efficient than with other virtualization technologies. Besides, we analysed how we can deploy the ESB instances within a cluster in order to improve the scalability of the entire system.

In Chapter 4 we firstly derived the functional and non-functional requirements of such a framework. After analysing them, we proposed the creation of a framework to create and deploy specialized ESB instances. The proposed framework incorporates a system where all the ESB instances are deployed in a clustered manner and constituted by four main components to handle the different phases of the ESB life-cycle:image creation, storage, provisioning, execution, and monitoring.

In Chapter 5 proposes an architectural design aiming at fulfilling the requirements proposed in Chapter 4. All of the system components are built in a distributed manner in order to enable the addition or removal of new machines in the cluster. The cluster is based on provisioning and administrating pre-configured virtual machines which can be increased based on current and future demands. We defined a system which can face also some difficult situations. For instance, the system can relocate the running ESB instance to other machines in case of a machine fails or it is out of resources.

The prototypical implementation fulfilling the architectural design is proposed in Chapter 6. We used Docker to encapsulate ESB instances into containers and CoreOS was used to run such containers within a cluster. Some new components have been fully implemented to fulfil all the functional and non-functional requirements described in Chapter 4. On the other hand, we used cAdvisor to monitor host machines and containers running ESB instances. The prototype supports ServiceMix ESB as an example of how a characterization, taxonomy and provisioning could be performed for other ESB solutions.

The validation and evaluation of our approach is presented in chapters 7 and 8, respectively. The former consists of validating our framework using a realistic application as a case study. On the other hand, the latter consists of empirically evaluating the performance under different scenarios.In the results we could confirm that the launch of new ESB instances in the same host machine requires much less resources as launching the first ESB instance. In that chapter we could also see that the number of running ESB instances affects slightly the performance shown by the ESB instances.

Despite of our solution shows a good performance and makes more easier the use of ESBs by the developers, the implementation of our approach is based on a concrete ESB solution. Our system is an initial solution to enable the deployment of specialized ESB instances, and in future works it could be improved in several aspects. For example, one of the lacks in our proposal is that if the ESB instance is moved to another machine, the developer must reconfigurate his system to point to the new instance's location. Therefore, it would be necessary to abstract the physical location of the ESB by enhancing our system with logical routing or load balancing. Another feature that could be added would be the reconfiguration of a stopped ESB instance. In our solution, if the developer needs a new component in an instance or a new port, a new instance must be created. Further future works could consider ESB images sharing and configurations among different developers. However, we recommend as next step for this work, the design and realization of a higher layer to dynamically provision and horizontally scale an ESB instance constituted by a group of wired instances running as a single entity. Finally, we want to mention that our prototype does not support the default installation of components created by developers. However, this feature is included in our design because it can enhance the flexibility of the framework.

Appendix A.

Description of the Most Relevant Entities in the Registry

A description of the most relevant entities of the registry are shown in the ER Diagram, Figure 5.9, is given in form of JSON schemas next.

```
{
   "title":"Feature",
   "description":"describes a component of the ESB software to
      perform a determinate task.",
   "type":"object",
   "$schema":"http://json-schema.org/draft-03/schema",
   "properties":{
      "Name":{
         "type":"string",
         "description":"tag of the configuration. Normally is
            defined by the user to identify it easily. "
      },
      "Category":{
         "type":"string",
         "description":"general category the features belongs to"
      },
      "Engine":{
         "type":"string",
         "description":"contains the engine necessary for running
             the feature"
      },
      "Description":{
         "type":"string",
         "description":"gives an overview of the feature"
      },
      "Path":{
         "type":"string",
         "description":"path to find the feature in the taxonomy
            graph which includes all the categories it belongs."
      },
      "Properties":{
         "type":"string",
         "description":"Description of the attributes and
            functionalities of the feature"
      }
   },
   "required":[
      "Name",
      "Category",
      "Engine",
      "Path",
      "Description",
      "Path",
      "Properties"
   ]
}
```

Listing A.1: Feature Entity JSON Schema

```
{
   "title":"Machine",
   "description":"Describes an host machine in the cluster",
   "type":"object",
   "$schema":"http://json-schema.org/draft-03/schema",
   "properties":{
      "ID":{
         "type":"string"
      },
      "Ip":{
         "type":"string",
         "description":"This IP addess would be the public one in
             case of a public and private IP address exist."
      },
      "Role":{
         "type":"string",
         "description":"this property describes if the machine is
             the leader of the cluster or not",
         "enum":[
            "leader",
            "follower"
         ]
      },
      "Status":{
         "type":"object",
         "description":"describes the state of the machine based
            on resource parameters",
         "properties":{
            "State":{
               "type":"string",
               "description":"describes the situation of the
                  machine to host ESB instances",
               "enum":[
                  "idle",
                  "normal",
                  "busy",
                  "critical"
               ]
            },
            "Cpu_usage":{
               "type":"number",
               "description":"CPU usage average during a certain
                  interval of time"
            },
            "Mem_usage":{
               "type":"number",
               "description":"CPU usage average during a certain
                  interval of time"
            }
         }
      },
      "required":[
         "ID",
         "Ip",
         "Role",
         "Status"
      ]
   }
}
```

Listing A.2: Machine Entity JSON Schema

```
{
   "title":"ESB Configuration",
   "description":"Includes all the parameters for configuring and
       ESB Instance",
   "type":"object",
   "$schema":"http://json-schema.org/draft-03/schema",
   "properties":{
      "ID":{
         "type":"string"
      },
      "Name":{
         "type":"string",
         "description":"tag of the configuration. Normally is defined by the user to identify it easily. "
      },
      "ESB":{
         "type":"string",
         "description":"contains the ESB Software use for the instance."
      },
      "Features":{
         "type":"array",
         "description":"List with the features of the ESB which must be already installed in the ESB Image",
         "items":{
            "type":"string"
         }
      },
      "CustomComponents":{
         "type":"array",
         "description":"List with the components created by the developer which must be already installed in the ESB Image",
         "items":{
            "type":"string"
         }
      },
      "Ports":{
         "type":"array",
         "description":"List of the ports of the container that will be publish and mapping to ports in the host machine",
         "items":{
            "type":"object"
         }
      }
   },
   "required":[
      "ID",
      "ESB",
      "Name"
```

```
  ],
  "additionalProperties":true
}
```

Listing A.3: Configuration Entity JSON Schema

```
{
  "title":"ESB Instance",
  "description":"contains the information of an existing ESB in
     the system",
  "type":"object",
  "$schema":"http://json-schema.org/draft-03/schema",
  "properties":{
     "Configuration":{
        "$ref":"ESB Config"
     },
     "Created":{
        "type":"string",
        "description":"timestamp when the instance was created"
     },
     "Error":{
        "type":"string",
        "description":"Displays a error message if it's the case
           ."
     },
     "Host":{
        "type":"string",
        "description":"IP Address of the ESB Instance when is
           running. This property is empty when the instance is
           not running."
     },
     "ID":{
        "type":"string"
     },
     "Labels":{
        "type":"array",
        "description":"list of tags for the instance to classify
            and recognize it better. Pairs of key values can be
           defined.",
        "items":{
           "type":"object"
        }
     },
     "PortBindings":{
        "type":"array",
        "description":"list of the port bindings with the container and host port.",
        "items":{
           "type":"object"
        }
     },
     "UserAccess":{
        "type":"array",
        "description":"Includes information related to how the developer can access to the ESB Instance when it's running",
        "items":{
           "type":"object"
        }
     },
     "State":{
        "type":"string",
        "description":"describes the state where the instance is in its lifecycle",
        "enum":[
           "stopped",
           "starting",
           "running",
           "stopping"
        ]
     }
  },
  "required":[
     "ID",
     "State",
     "Configuration",
     "Created"
  ],
  "additionalProperties":false
}
```

Listing A.4: ESB Instance Entity JSON Schema

```
{
   "title":"ESB Configuration",
   "description":"Includes all the parameters for configuring and ESB Instance",
   "type":"object",
   "$schema":"http://json-schema.org/draft-03/schema",
   "properties":{
      "ID":{
         "type":"string"
      },
      "Name":{
         "type":"string",
         "description":"tag of the configuration. Normally is
            defined by the user to identify it easily. "
      },
      "ESB":{
         "type":"string",
         "description":"contains the ESB Software use for the
            instance.",
         "value":"servicemix"
      },
      "Features":{
         "type":"array",
         "description":"List with the features of the ESB which
            must be already installed in the ESB Image",
         "items":{
            "type":"string"
         }
      },
      "CustomComponents":{
         "type":"array",
         "description":"List with the components created by the
            developer which must be already installed in the ESB
            Image",
         "items":{
            "type":"string"
         }
      },
      "Ports":{
         "type":"array",
         "description":"List of the ports of the container that
            will be publish and mapping to ports in the host
            machine",
         "items":{
            "type":"object"
         }
      },
      "CPU_SHARED":{
         "type":"number",
         "description":"maximum percentage of the total cycles of
             the host used by the container "
      },
      "MAX_MEM":{
         "type":"number",
         "description":"maximum memory capacity of the host
            machine used by the container "
      },
      "JBI_allowCoreThreadTimeOut":{
         "type":"boolean",
         "default":"true"
      },
      "JBI_corePoolSize":{
         "type":"integer",
         "default":"4"
      },
      "JBI_keepAliveTime":{
         "type":"integer",
         "default":"60000"
      },
      "JBI_maximumPoolSize":{
         "type":"integer",
         "default":"-1"
      },
      "JBI_queueSize":{
         "type":"integer",
         "default":"1024"
      },
      "JBI_shutdownTimeout":{
         "type":"integer",
         "default":"0"
      },
      "JVM_MAX_MEM":{
         "type":"integer"
      },
      "JVM_MAX_PERM_MEM":{
         "type":"integer"
      },
      "JVM_MIN_MEM":{
         "type":"integer"
      },
      "JVM_PERM_MEM":{
         "type":"integer"
      },
      "NMR_allowCoreThreadTimeOut":{
         "type":"boolean",
         "default":"true"
      },
      "NMR_corePoolSize":{
         "type":"integer",
         "default":"4"
      },
      "NMR_keepAliveTime":{
         "type":"integer",
         "default":"60000"
      },
      "NMR_maximumPoolSize":{
         "type":"integer",
         "default":"-1"
      },
      "NMR_queueSize":{
         "type":"integer",
         "default":"1024"
      },
      "NMR_shutdownTimeout":{
         "type":"integer",
         "default":"0"
      },
      "SERVICEMIX_USER":{
         "type":"string",
         "description":"username for the interaction with the ServiceMix Instance",
         "default":"smx"
      },
      "SERVICEMIX_PASSWORD":{
         "type":"string",
         "description":"password for the interaction with the ServiceMix Instance",
         "default":"smx"
      }
   },
   "required":[
      "ID",
      "ESB",
      "Name"
   ],
   "additionalProperties":true
}
```

Listing A.5: ServiceMix Configuration JSON Schema

Appendix B.

ServiceMix Features Taxonomy

Name	Category	Type	Engine	Description	Properties
webconsole	Management & Orchestration	BC	ServiceMix	The Karaf web console provides a graphical overview of the runtime.	install and uninstall features start, stop, install bundles create child instances configure Karaf view logging informations
ssh	Management & Orchestration	BC	ServiceMix	SSH server to enable the SSH connection to the Karaf instance	-
servicemix-exec	Management & Orchestration	SE	ServiceMix	The ServiceMix Exec component is used to invoke commands	Commands: executables, binaries, shell commands, shell scripts, etc
camel-exec	Management & Orchestration	SE	Camel	The exec component can be used to execute system commands.	commands: executables, binaries, shell commands, shell scripts, etc
servicemix-osworkflow	Management & Orchestration	SE	ServiceMix	The ServiceMix OSWorkflow component provides workflow functionality to the ESB.	You can specify one or more workflows and it's processing will start when a valid message is received.
servicemix-scripting	Management & Orchestration	SE	ServiceMix	The ServiceMix Scripting component provides support for processing scripts using JSR-223 compliant scripting languages.	Groovy (1.5.6) JRuby (1.1.2) Rhino JavaScript (1.7R1)
servicemix-cxf-se	Management & Orchestration	SE	ServiceMix	ServiceMix CXF SE component is a JBI Service Engine exposing (annotated) POJO as services on the JBI Bus.	It uses Apache CXF internally to perform service invocations and xml marshaling. Features: jsr181 annotations, jaxb2/aegis/xmlbeans databinding,wsdl auto generation, java proxy support, MTOM / attachments support.
camel-quartz2	Message Handlers	SE	Camel	It provides a scheduled delivery of messages.	Quartz Scheduler 2.x . Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).
camel-quartz	Message Handlers	SE	Camel	It provides a scheduled delivery of messages.	Quartz Scheduler 1.x . Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).
servicemix-quartz	Message Handlers	SE	ServiceMix	The servicemix-quartz component is a standard JBI Service Engine able to schedule and trigger jobs using the great Quartz scheduler.	Quartz Scheduler 1.x . Each endpoint represents a different timer (in Quartz terms, a Trigger and JobDetail).
cxf-nmr	Routing	SE	ServiceMix	Base broker in ServiceMix	-
servicemix-camel	Routing	SE	ServiceMix	The servicemix-camel component provides support for using Apache Camel	It provides a full set of Enterprise Integration Patterns and flexible routing and transformation in both Java code or Spring XML to route services on the Normalized Message Router.
camel-nmr	Routing	SE	Camel	The nmr component is an adapter to the Normalized Message Router (NMR) in ServiceMix, which is intended for use by Camel applications deployed directly into the OSGi container.	You can exchange objects with NMR and not only XML like this is the case with the JBI specification. The interest of this component is that you can interconnect camel routes deployed in different OSGI bundles.
servicemix-drools	Routing	SE	ServiceMix	This Service Engine can be used to deploy a rules set that will implement a router or an actual service.	-
camel-fop	Message Transformers	SE	Camel	The FOP component allows you to render a message into different output formats using Apache FOP	Supported formats: pdf, ps, pcl, png, jpeg, svg, xml, mif, rtf, txt, etc.
camel-avro	Message Transformers	SE	Camel	This component provides a dataformat for avro, which allows serialization and deserialization of messages using Apache Avro's binary dataformat.	It provides support for Apache Avro's rpc, by providing producers and consumers endpoint for using avro over netty or http.
servicemix-saxon	Message Transformers	SE	ServiceMix	The servicemix-saxon component is a standard JBI Service Engine for XSLT / XQuery.	This component is based on Saxon and supports XSLT 2.0 and XPath 2.0, and XQuery 1.0.
camel-xmlbean	Message Transformers	SE	Camel	XmlBeans is a Data Format which uses the XmlBeans library to unmarshal an XML payload into Java objects or to marshal Java objects into an XML payload.	-
camel-xmljson	Message Transformers	SE	Camel	This data format provides the capability to convert from XML to JSON and viceversa directly, without stepping through intermediate POJOs.	-
servicemix-validation	Validation	SE	ServiceMix	The ServiceMix Validation component provides schema validation of documents using JAXP 1.3 and XMLSchema or RelaxNG.	-
camel-xmlsecurity	Security	SE	Camel	Validate XML signatures as described in the W3C standard XML Signature Syntax and Processing or as described in the successor version 1.1.	Supports Apache Santuario and the JDK provider for JSR 105
camel-crypto	Security	SE	Camel	The Crypto Data Format integrates the Java Cryptographic Extension into Camel, allowing simple and flexible encryption and decryption of messages using Camel's familiar marshall and unmarshal formatting mechanism.	Encryption algorithms: DES, CBC and PKCS5Padding. Message Authentication Code.

camel-cache	Storage	SE	Camel	The cache component enables you to perform caching operations using EHCache as the Cache Implementation	This component supports producer and event based consumer endpoints.
camel-sql	Storage	SE	Camel	This component allows you to work with databases using JDBC queries.	-
camel-jdbc	Storage	SE	Camel	The jdbc component enables you to access databases through JDBC, where SQL queries and operations are sent in the message body.	This component uses the standard JDBC API.
camel-mongodb	Storage	SE	Camel	The camel-mongodb component integrates Camel with MongoDB allowing you to interact with MongoDB collections both as a producer (performing operations on the collection) and as a consumer (consuming documents from a MongoDB collection).	-
camel-couchdb	Storage	SE	Camel	The couchdb: component allows you to treat CouchDB instances as a producer or consumer of messages.	As a consumer, monitors couch changesets for inserts, updates and deletes and publishes these as messages into camel routes. As a producer, can save or update documents into couch. Can support as many endpoints as required, eg for multiple databases across multiple instances. Ability to have events trigger for only deletes, only inserts/updates or all (default). Headers set for sequenceId, document revision, document id, and HTTP method type.
camel-krati	Storage	SE	Camel	This component allows the use krati datastores and datasets inside Camel	-
camel-jclouds	Communication	SE	Camel	This component allows interaction with cloud provider key-value engines (blobstores) and compute services.	-
camel-aws	Communication	SE	Camel	This component provides connectivity to AWS from Camel	Support for: SQS, SNS, S3, SES, SimpleDB, DynamoDB, CloudWatch and Simple Workfrow.
camel-gae	Communication	SE	Camel	It provides connectivity to GAE's cloud computing services.	Url fetch service, Task queueing service, Mail service, Memcache service, XMPP service, Images service, Datastore service, Accounts service, etc.
camel-mail	Communication	SE	Camel	The mail component provides access to Email via Spring's Mail support and the underlying JavaMail system.	SMTP, SMTPS, POP3, POP3S, IMAP, IMAPS
servicemix-email	Communication	BC	ServiceMix	The ServiceMix Mail component provides support for receiving and sending mails via the enterprise service bus.	POP3, IMAP, SMTP
servicemix-ftp	Communication	BC	ServiceMix	The ServiceMix FTP component provides JBI integration to the FTP servers.	It can be used to read & write files over FTP or to periodically poll directories for new files.
camel-ftp	Communication	SE	Camel	This component provides access to remote file systems over the FTP and SFTP protocols.	FTP, SFTP, FTPS.
servicemix-vfs	Communication	BC	ServiceMix	The ServiceMix VFS component provides support for reading from and writing to virtual file systems via the enterprise service bus by using the Apache commons-vfs library.	-
camel-jsch	Communication	SE	Camel	The camel-jsch component supports the SCP protocol using the Client API of the Jsch project	-
camel-facebook	Communication	SE	Camel	The Facebook component provides access to all of the Facebook APIs accessible using Facebook4J.	It allows producing messages to retrieve, add, and delete posts, likes, comments, photos, albums, videos, photos, checkins, locations, links, etc. It also supports APIs that allow polling for posts, users, checkins, groups, locations, etc.
camel-twitter	Communication	SE	Camel	The Twitter component enables the most useful features of the Twitter API by encapsulating Twitter4J.	It allows direct, polling, or event-driven consumption of timelines, users, trends, and direct messages. Also, it supports producing messages as status updates or direct messages.
camel-xmpp	Communication	SE	Camel	The xmpp: component implements an XMPP (Jabber) transport.	The component supports both room based and private person-person conversations.
servicemix-xmpp	Communication	BC	ServiceMix	The ServiceMix XMPP component is used to communicate with XMPP (Jabber) servers through the JBI bus.	The component supports both room based and private person-person conversations.
camel-rss	Communication	SE	Camel	The rss: component is used for polling RSS feeds.	-
camel-irc	Communication	SE	Camel	You can hang out on IRC with other Camel developers and users.	-
camel-activemq	Communication	SE	Camel	The ActiveMQ component allows messages to be sent to a JMS Queue or Topic or messages to be consumed from a JMS Queue or Topic using Apache ActiveMQ.	-

camel-jms	Communication	SE	Camel	The JMS component allows messages to be sent to (or consumed from) a JMS Queue or Topic.	-
servicemix-jms	Communication	BC	ServiceMix	The JMS component allows messages to be sent to (or consumed from) a JMS Queue or Topic.	JBI compliant Binding Component Usable in a lightweight mode in servicemix.xml configuration files SOAP 1.1 and 1.2 support MIME attachments WS-Addressing support WSDL based and XBean based deployments Support for all MEPs as consumers or providers
servicemix-cxf-bc	Communication	BC	ServiceMix	A JBI compliant HTTP/SOAP or JMS/SOAP binding component named servicemix-cxf-bc which use apache cxf internally.	JBI compliant Binding Component Usable in a lightweight mode in servicemix.xml configuration files SOAP 1.1 and 1.2 support MIME attachments Support for all MEPs as consumers or providers SSL support WS-Security support WS-Policy support WS-RM support WS-Addressing support
camel-rmi	Communication	SE	Camel	The rmi: component binds Exchanges to the RMI protocol (JRMP).	-
camel-http	Communication	SE	Camel	The http: component provides HTTP based endpoints for consuming external HTTP resources (as a client to call external servers using HTTP).	HTTP, HTTPS
camel-http4	Communication	SE	Camel	The http4: component provides HTTP based endpoints for calling external HTTP resources (as a client to call external servers using HTTP).	HTTP, HTTPS
servicemix-cxf-bc	Communication	BC	ServiceMix	A JBI compliant HTTP/SOAP or JMS/SOAP binding component named servicemix-cxf-bc which use apache cxf internally.	JBI compliant Binding Component Usable in a lightweight mode in servicemix.xml configuration files SOAP 1.1 and 1.2 support MIME attachments Support for all MEPs as consumers or providers SSL support WS-Security support WS-Policy support WS-RM support WS-Addressing support
servicemix-http	Communication	BC	ServiceMix	ServiceMix ships with a JBI compliant HTTP/SOAP binding component named servicemix-http.	JBI compliant Binding Component Usable in a lightweight mode in servicemix.xml configuration files Integrated HTTP server based on Jetty 6 HTTP Client using Jakarta Commons HTTP Client Highly performant and scalable using Jetty 6 continuations SOAP 1.1 and 1.2 support MIME attachments WS-Addressing support WSDL based and XBean based deployments Support for all MEPs as consumers or providers SSL support WS-Security support
camel-websocket	Communication	SE	Camel	The websocket component provides websocket endpoints for communicating with clients using websocket.	It supports the protocols ws:// and wss://.
camel-printer	Communication	SE	Camel	The printer component provides a way to direct payloads on a route to a printer.	This component only supports a camel producer endpoint.
camel-apns	Communication	SE	Camel	The apns component is used for sending notifications to iOS devices.	The component supports sending notifications to Apple Push Notification Servers (APNS) and consuming feedback from the servers.

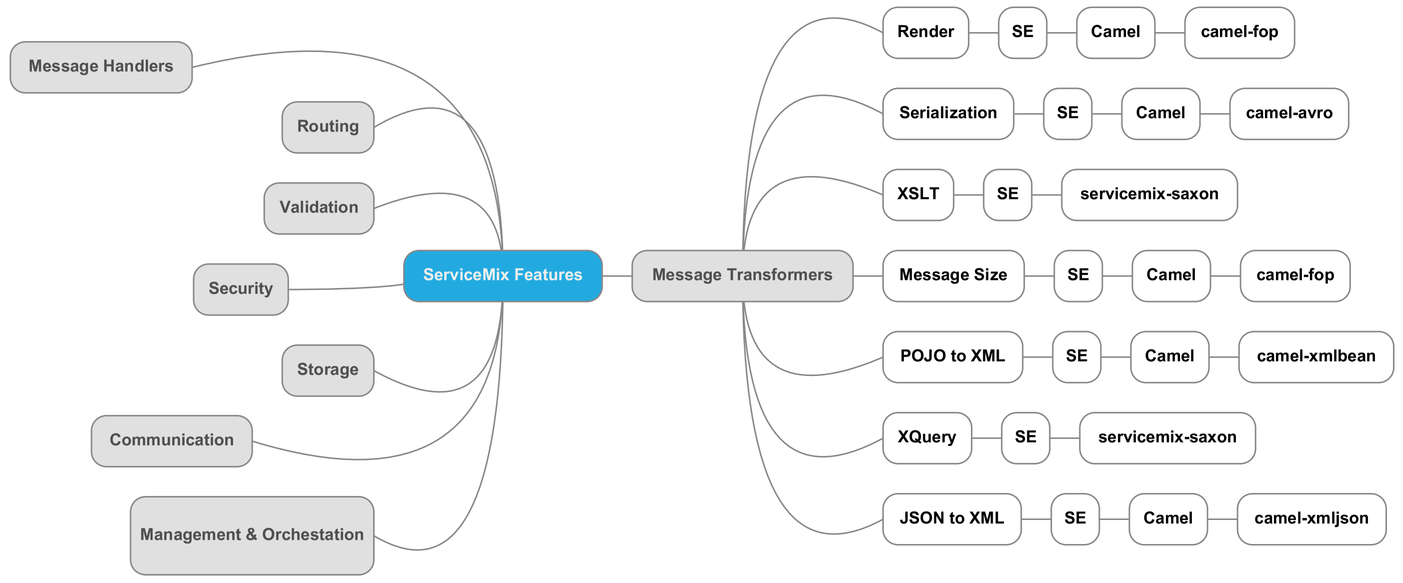


Figure B.1.: ServiceMix features

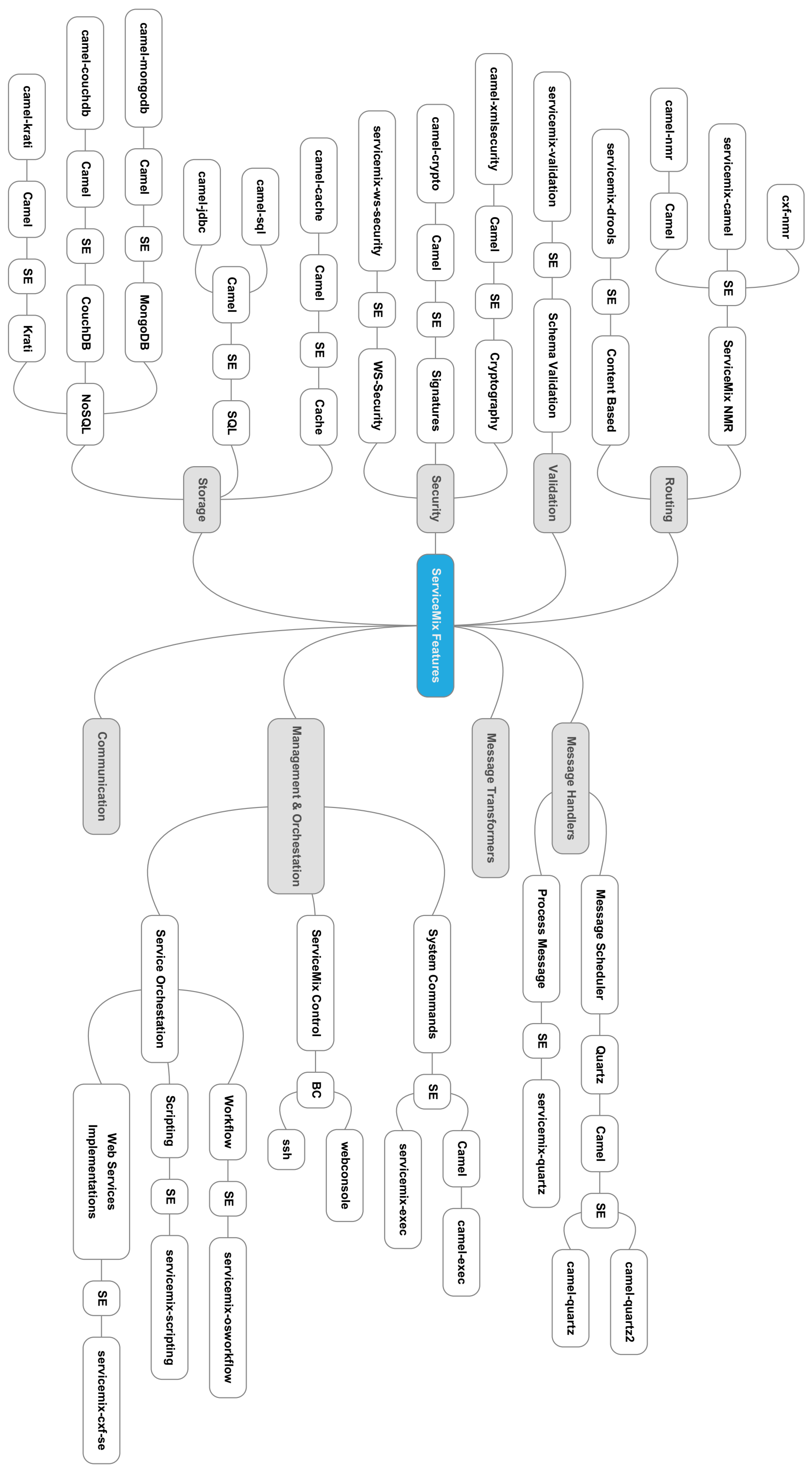


Figure B.2.: ServiceMix features

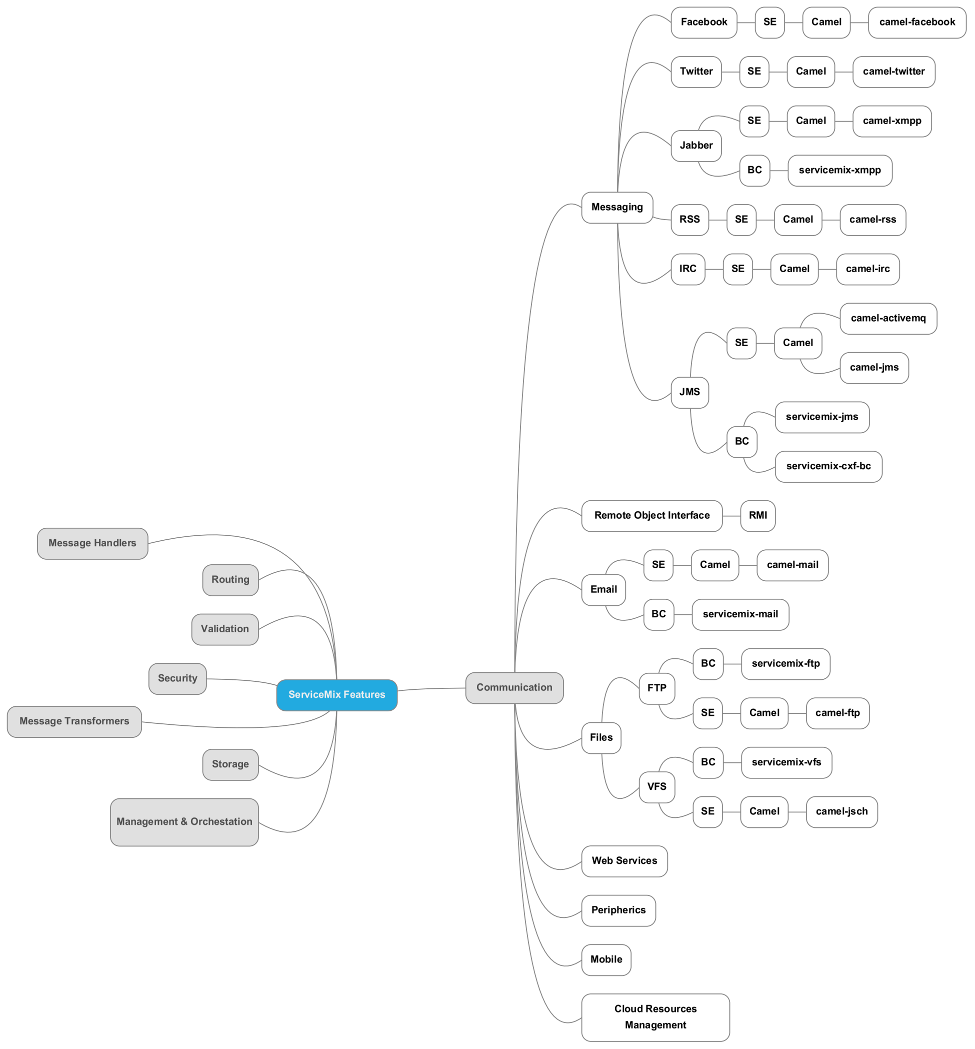


Figure B.3.: ServiceMix features

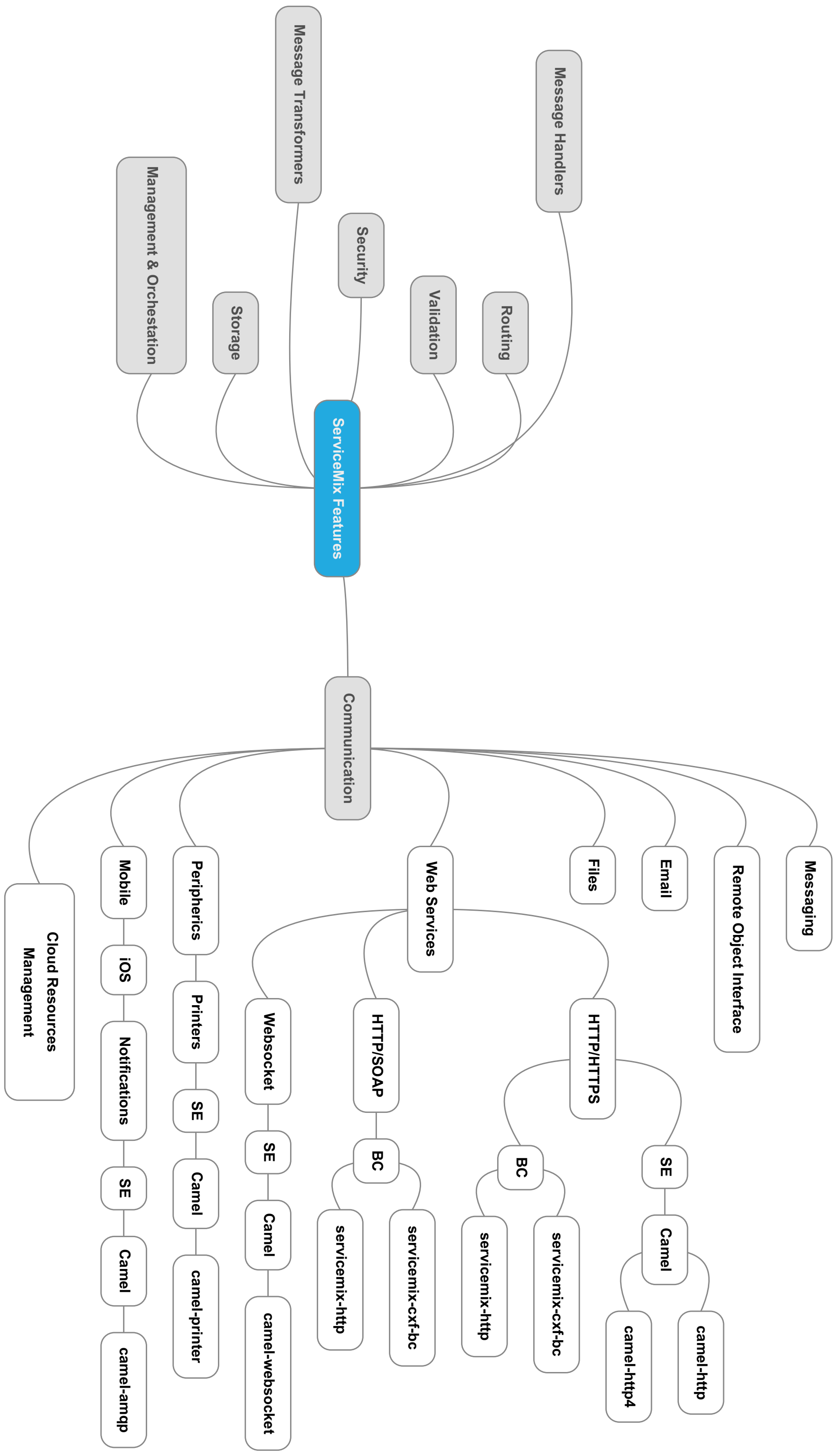
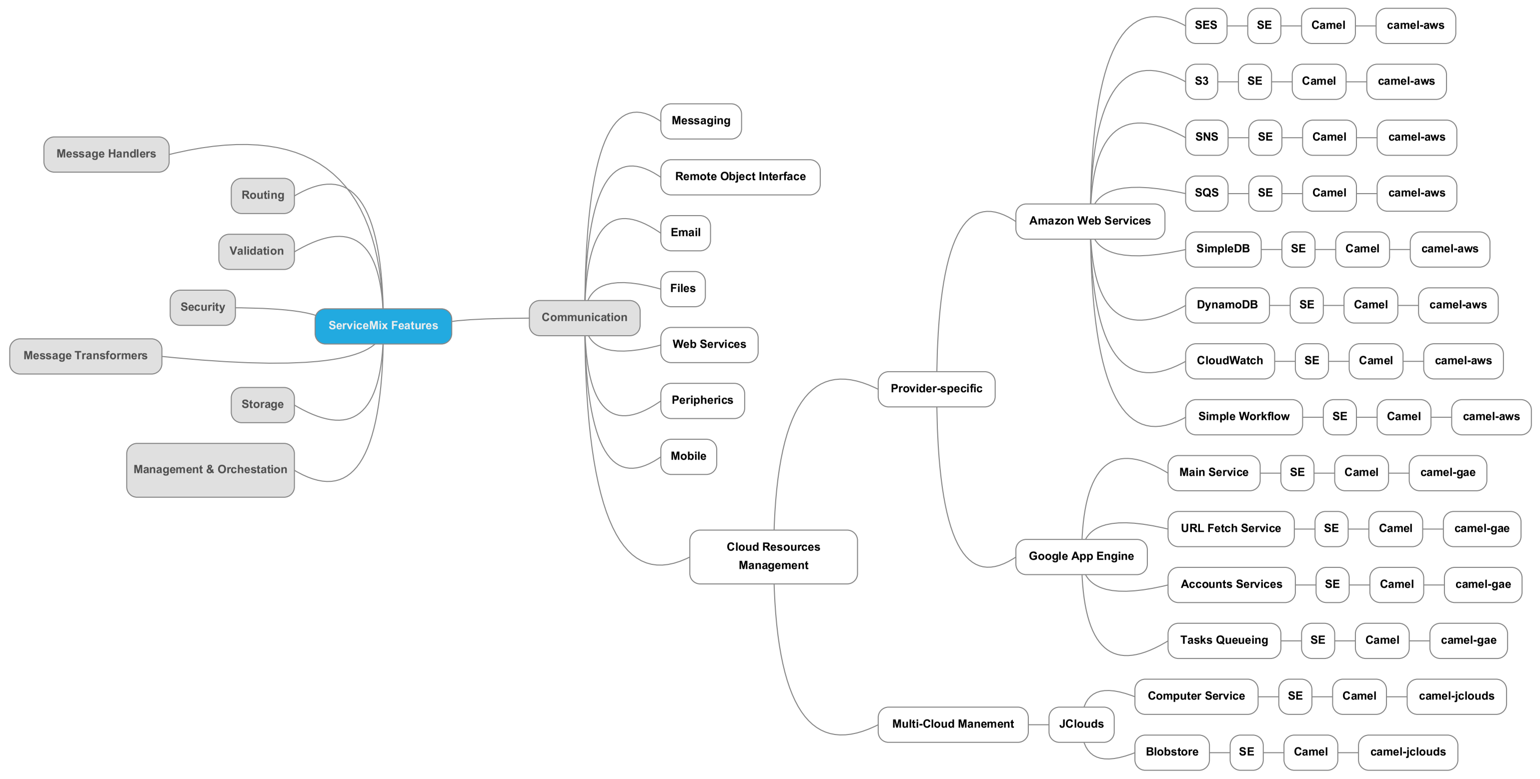


Figure B.4.: ServiceMix features



ServiceMix Features
Message Handlers
Routing
Validation
Security
Message Transformers
Storage
Management & Orchestation
Communication
Messaging
Remote Object Interface
Email
Files
Web Services
Periphercs
Mobile
Cloud Resources Management
Provider-specific
Amazon Web Services
SES — SE — Camel — camel-aws
S3 — SE — Camel — camel-aws
SNS — SE — Camel — camel-aws
SQS — SE — Camel — camel-aws
SimpleDB — SE — Camel — camel-aws
DynamoDB — SE — Camel — camel-aws
CloudWatch — SE — Camel — camel-aws
Simple Workflow — SE — Camel — camel-aws
Google App Engine
Main Service — SE — Camel — camel-gae
URL Fetch Service — SE — Camel — camel-gae
Accounts Services — SE — Camel — camel-gae
Tasks Queueing — SE — Camel — camel-gae
Multi-Cloud Manement
JClouds
Computer Service — SE — Camel — camel-jclouds
Blobstore — SE — Camel — camel-jclouds

Figure B.5.: ServiceMix features

Appendix C.

Taxonomy Validation - Case Studies

C.1. Example: Using ESB to Deploy a Wordpress Application

The first step in this case is identifying which components are included in our Wordpress application. After taking a look in the Wordpress documentation, we can come up with the following main components in a simple Wordpress installation:

- Server with Wordpress. The communication with this service is perform through HTTP/HTTPS requests.
- MySQL database. The Wordpress server gets the data making SQL queries to this database.
- Browser (client). It processes the graphical interface obtained from the server by HTTP/HTTPS requests.

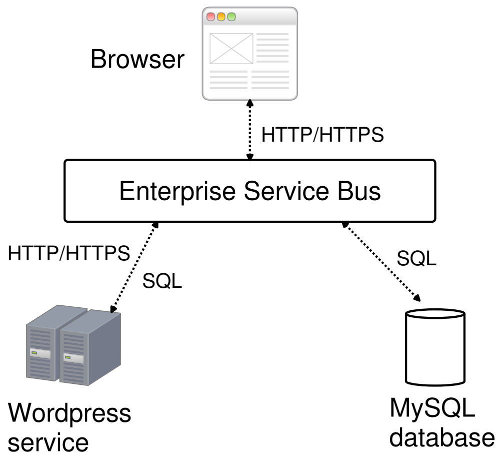


Figure C.1.: Basic components of a WordPress application

The second step is selecting which ESB features we need for the communication of these components. For that, we must make use of our features map. At least we need to find two features: one which supports HTTP/HTTPS, and other one for SQL support. As we can see in Figure C.3, there are different alternatives for HTTP/HTTPS support: camel-http, camel-http4, servicemix-cxf-bc and servicemix-http. For SQL we have also two options: camel-sql and camel-jdbc.

C.2. Example: Using ESB to Deploy a KeystoreJS Application

As in the other configuration, the first step is identifying which components are included in a typical KeystoreJS deployment. After taking a look in the KeystoreJS documentation, we can come up with the following main components in a simple KeystoreJS installation:

- Server with KeystoreJS. It includes a dashboard and a server accessible through HTTP/HTTPS requests. Besides it includes a MongoDB.
- Browser (client). It processes the graphical interface with data obtained from the server by HTTP/HTTPS requests.
- Mobile devices (client). They processes the graphical interface with data obtained from the server by HTTP/HTTPS requests.

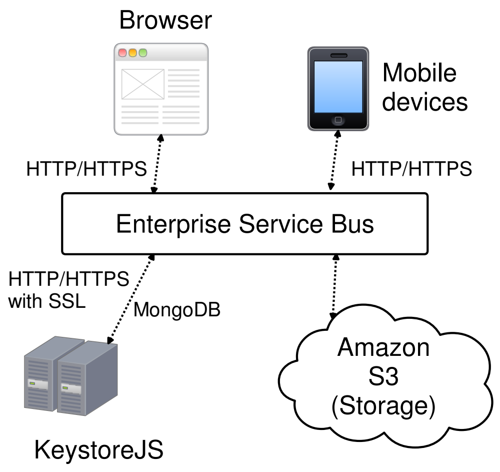


Figure C.2.: Basic components of a KeystoreJS application

The second step is selecting which ESB features we need for the communication of these components. For that, we should make use of our features map. At least we need three features: one which supports HTTP/HTTPS, one for MongoDB and other one for Amazon S3. As we can see in Figure C.4, there are different alternatives for HTTP/HTTPS support: camel-http, camel-http4, servicemix-cxf-bc and servicemix-http. For MongoDB we have only one options: camel-mongodb. Finally, for communication with Amazon S3, we need the feature : camel-aws.

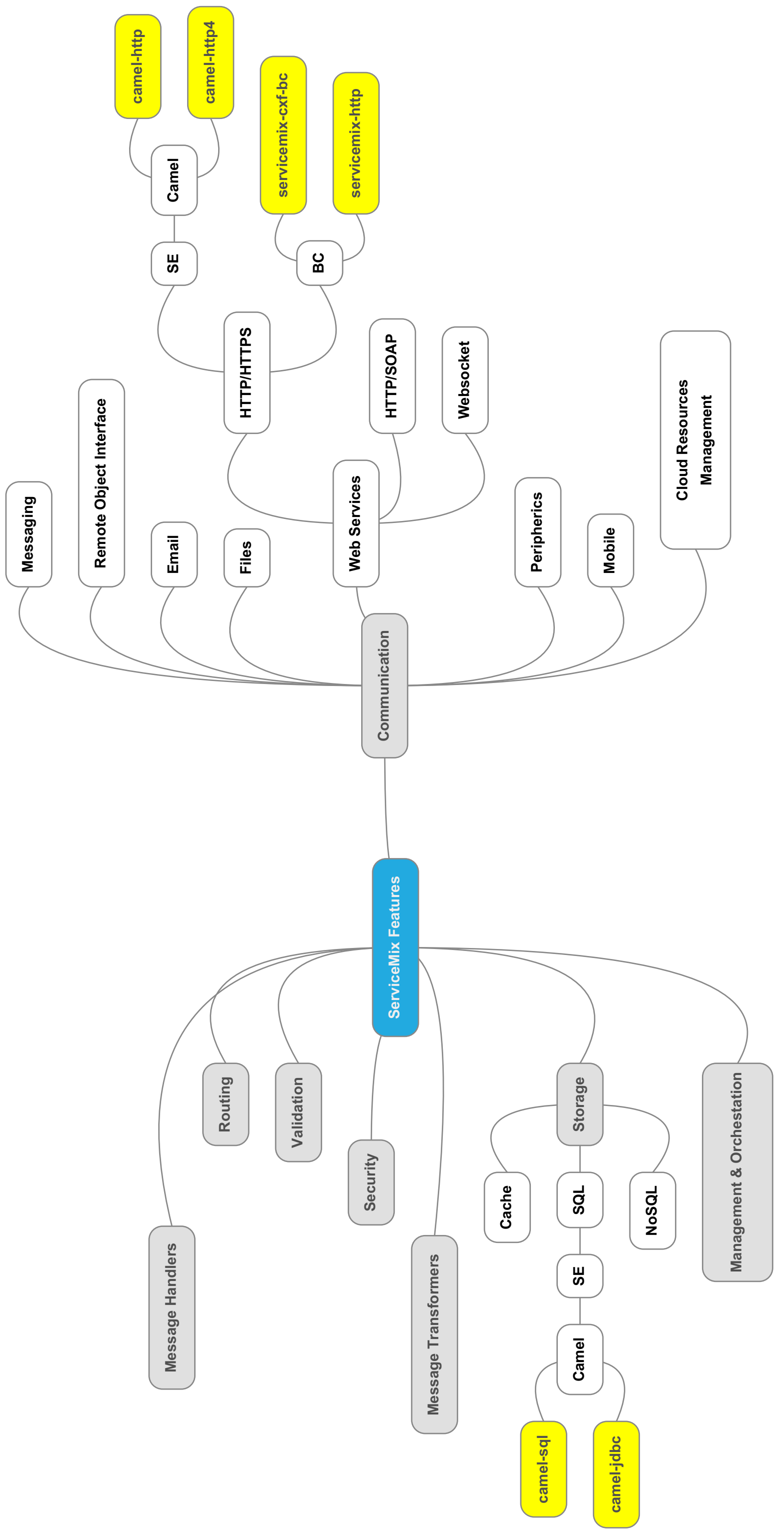


Figure C.3.: Selection of features to deploy a Wordpress Application

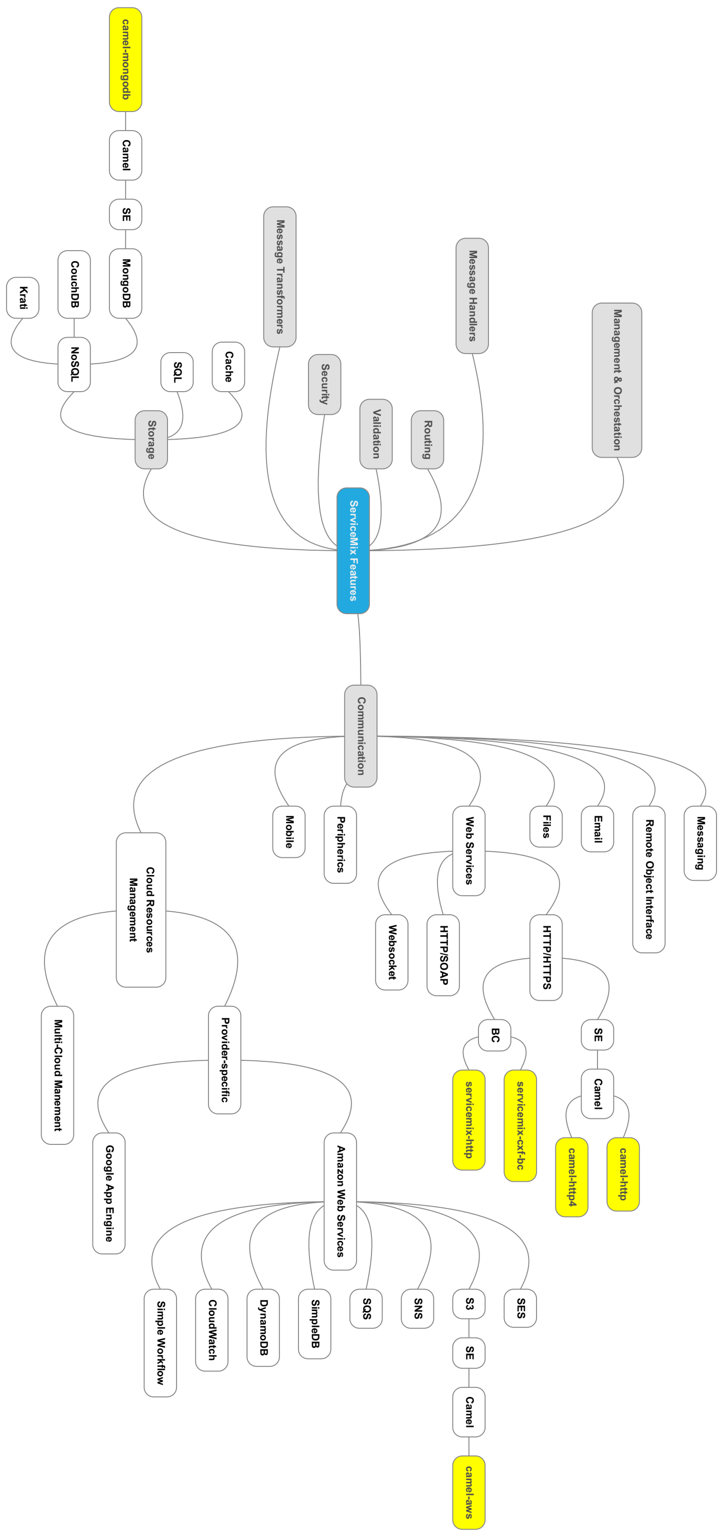


Figure C.4.: Selection of features to deploy a KeystorJS Application

Appendix D. Implementation Details

In order to create a new ESB Instance in the system, three Fleet units must be defined:

- *Launcher*: contains the command to start and stop an ESB instance. It also saves the ESB instance state when it exits. It contains also some fields (X-Fleet fields) to select a machine preference during the scheduling process.
- *Minion*: publishes the information of the running ESB instance into the registry (ETCD). It publishes the IP address and binding ports of such instance and provides the information to access to the instance. It includes also a field *MachineOf* to be located aside the launcher.
- *Committer*: commits periodically the state of the running ESB instance and pushes it to the image storage. It includes also a field *MachineOf* to be located aside the launcher.

A skeleton of these three components can be seen in Listing D.1.

```

## LAUNCHER UNIT

[Unit]

Description=Launcher for ESB Instance {{ IntanceID }}.
Requires=docker.service


[Service]

TimeoutSec=0

## Pull the ESB image

ExecStartPre=/bin/bash -c "/usr/bin/docker pull {{ InstanceImage
    }} "
ExecStart=/bin/bash -c "/usr/bin/docker run -t -P --name {{
    InstanceID}} {{ConfigurationParameters}} {{InstanceImage}} "

```

```

## Execute to stop the container with the ESB Instance

ExecStop=/bin/bash -c "/usr/bin/docker stop {{.Instance.ID}} "

## Sets the stopping state in the registry.

ExecStopPost=/usr/bin/etcdctl set /_esb/user/{{.Owner}}/instance
   /{{.Service}} "{{.StoppingInstance}}"

## Push the state of the ESB instance into the Image Storage

ExecStopPost=/bin/bash -c "/usr/bin/docker commit  {{InstanceID}}
    {{InstanceImage}} ; /usr/bin/docker push {{InstanceImage}}"

## Sets the stopping state in the registry.

ExecStopPost=/usr/bin/etcdctl set /_esb/user/{{Owner}}/instance
   /{{InstanceID}} "{{StoppedInstanceJSON}}"

## Removes the container from the machine

ExecStopPost=/bin/bash -c "/usr/bin/docker rm {{InstanceID}} "

## Parameters for scheduling preferences.

[X-Fleet]
MachineMetadata=health=idle
MachineMetadata=health=normal
MachineMetadata=health=busy

-------------------------------------------------------

## INFO UNIT


[Unit]
Description=Info {{InstanceID}} ESB Instance

# Run after the Launcher unit.

After={{InstanceID}}.service
Requires={{InstanceID}}.service

```

```
[Service]
EnvironmentFile=/etc/environment

# Periodically sets the running state for the instance with a TTL value.

ExecStart=/bin/sh -c "while true; do ; /usr/bin/sleep 3; etcdctl set /_esb/user/{{Owner}}/instance/{{InstanceID}} \"{{RunningInstance}}\" --ttl 90; /usr/bin/sleep 30; done"

# This field tells the scheduler to run this unit aside the Launcher unit.

[X-Fleet]
MachineOf={{InstanceID}}.service

----------------------------------------------------------
## COMMITTER UNIT


[Unit]
Description=Commit state {{InstanceID}} ESB Instance into a Docker Registry
After={{InstanceID}}.service
Requires={{InstanceID}}.service

[Service]
EnvironmentFile=/etc/environment

# Periodically push the state of the running instance into the Image Storage.

ExecStart=/bin/sh -c "while true; do /usr/bin/sleep 300; /usr/bin/docker commit --pause=false {{InstanceID}} {{ImageName}} ; /usr/bin/docker push {{ImageName}} ; done"

# This field tells the scheduler to run this unit aside the Launcher unit.

[X-Fleet]
MachineOf={{.Service}}.service
```

Listing D.1: Implementation of and ESB instance skeleton

In Listing D.2 we can see a skeleton of how a command for the CLI can be done in Golang using the cited library.

```go
package command

import (
  "errors"
  "github.com/coreos/etcdctl/third_party/github.com/codegangsta/cli"
  esb "github.com/totemteleko/goesb"
  "os"
)

// Creates command example

func CreateCommand() cli.Command {
  return cli.Command{
    Name:  "create",
    Usage: "create␣ESB␣instances␣in␣the␣cluster␣from␣a␣JSON␣config␣file",
    // These flags specified the options of the command line tool
    Flags: []cli.Flag{
      cli.StringFlag{"user,␣u", "admin", "Owner␣of␣the␣new␣ESB␣instance"},
    },

    Action: func(c *cli.Context) {
      // Calls the handler function.
      handler(c)

    },
  }
}

func handler(c *cli.Context) {

  // Checks if the number of arguments is correct.

  if len(c.Args()) == 0 {
    printError(errors.New("Config␣file␣required").Error())
    os.Exit(2)
  }
  // Client which includes of the functions to use the platform. Initialization of the Client has been ommitted for brevity.
  client, _ := localESBClient()

  // Loop
  for _, arg := range c.Args() {

    // here a function of the ESB client is called with the
       argument.

    // In case of error, print it.
    if err != nil {
      println("ERROR:␣" + err.Error())
      os.Exit(2)
    } else {
      println("success")
    }
  }

}
```

Listing D.2: Implementing a CLI command

Appendix E.

System Configuration

In order to configure each one of the machines in the cluster, only one file is required. This file is a configuration file which is used when a machine is provision in the supported infrastructures. This file contains the same information and values for all the machines in the system and hence, a configuration must be performed just once. A template is given in Listing E.1.

```
#cloud-config

coreos:
  etcd:
    # generate a new token for each unique cluster from https://
       discovery.etcd.io/new
    discovery: https://discovery.etcd.io/<token>
    # multi-region and multi-cloud deployments need to use
       $public_ipv4
    addr: $private_ipv4:4001
    peer-addr: $private_ipv4:7001
  units:
    - name: etcd.service
      command: start
    - name: fleet.service
      command: start

     # This unit run the Monitor in the specified port of the
        machine
    - name: monitor.service
      command: start
      content: |
          [Unit]
          Description=Monitor for CoreOS host
          After=docker.service
          Requires=docker.service

          [Service]
        Restart=always
        ExecStartPre=/bin/bash -c "/usr/bin/docker pull google/cadvisor"
        ExecStart=/usr/bin/docker run -t --volume=/var/run:/var/run:rw --volume=/sys:/sys:ro --volume=/var/lib/docker/:/var/lib/docker:ro --publish=<MONITOR_PORT>:8080 --name=monitor google/cadvisor:latest
        ExecStop=/bin/bash -c "/usr/bin/docker stop monitor"
        ExecStopPost=/bin/bash -c "/usr/bin/docker rm monitor"

  # This unit install some of the required files for the deployment.
  - name : esb.service
    command: start
    content: |
              [Unit]
              After=docker.service
              After=fleet.service
              After=etcd.service
              Description=Download ESB Binaries
              Documentation=https://github.com/totemteleko/esb
              Requires=docker.service
              Requires=fleet.service
              Requires=etcd.service

              [Service]
              Environment="DOCKER_USER=<ADMIN_NAME >"
              Environment="MAINTAINER=<ADMIN_EMAIL >"
              Environment="public_ipv4=$public_ipv4"
              Environment="private_ipv4=$private_ipv4"
              Environment="ESB_PORT=<ESB_PORT >"
              Environment="MONITOR_PORT=<MONITOR_PORT >"
              Environment="IMAGE_STORAGE=<IMAGE_STORAGE_LOCATION >"
              ExecStartPre=/usr/bin/docker pull totemteleko/servicemix
              ExecStart=/usr/bin/wget -P /home/core/ https://bitbucket.org/totemteleko/esb/downloads/esb-setup.sh
              ExecStart=/usr/bin/sh /home/core/esb-setup.sh
              ExecStart=/usr/bin/rm /home/core/esb-setup.sh

              Type=oneshot

```

```
     # This unit runs the REST API
    - name : esb-rest.service
      command: start
      content: |
               [Unit]
               After=esb.service
               After=docker.service
               After=fleet.service
               After=etcd.service
               Description=Rest API for ESB cloud
               Documentation=https://github.com/totemteleko/esb
               Requires=docker.service
               Requires=fleet.service
               Requires=etcd.service

               [Service]

               EnvironmentFile=/etc/esb-environment
               ExecStart=/opt/bin/esb-rest
               Restart=always
    - name : esb-controller.service
      command: start
      content: |
               [Unit]
               After=monitor.service
               After=esb.service
               After=docker.service
               After=fleet.service
               After=etcd.service
               After=monitor.service

               Description=Rest API for ESB cloud
               Documentation=https://github.com/totemteleko/esb
               Requires=monitor.service
               Requires=docker.service
               Requires=fleet.service
               Requires=etcd.service
               Requires=monitor.service

               [Service]

               EnvironmentFile=/etc/environment
               EnvironmentFile=/etc/esb-environment
               Environment="TEMPLATE_PATH=/var/templates/fleet.
```

```
                    conf"
                Environment="FLEET_CONFIG=/etc/fleet/fleet.conf"
                Environment="HEALTH_INTERVAL=<HEALTH_INTERVAL>"
                Environment="HEALTH_MIN_CPU=<HEALTH_MIN_CPU>"
                Environment="HEALTH_MAX_CPU=<HEALTH_MAX_CPU>"
                Environment="HEALTH_MIN_CPU_BUSY=<
                    HEALTH_MIN_CPU_BUSY>"
                Environment="HEALTH_MIN_MEM=<HEALTH_MIN_MEM>"
                Environment="HEALTH_MAX_MEM=<HEALTH_MAX_MEM>"
                Environment="HEALTH_MIN_MEM_BUSY=<
                    HEALTH_MIN_MEM_BUSY>"
                ExecStart=/opt/bin/esb-controller
                Restart=always

# This create a file to update the metadata of the machine when
    the Controller starts
write_files:
  - path: /etc/fleet/fleet.conf
    content: |
      public_ip="$private_ipv4"
      metadata="health=unknown"
```

Listing E.1: Template for the configuration of the framework inside CoreOS machines

Bibliography

[1] R. P. Padhy, “Virtualization techniques & technologies: state-of-the-art,” *Journal of Global research in Computer science*, vol. 2, no. 12, pp. 29–43, 2012.

[2] CoreOS,Inc., “CoreOS. Minimal operating system for massive server deployments,” 2014. `https://coreos.com/`.

[3] F. Calzolari, “High availability using virtualization,” *CoRR*, vol. abs/0910.1719, 2009.

[4] T. Rademakers and J. Dirksen, *Open-Source ESBs in Action: Example Implementations in Mule and ServiceMix*. Greenwich, CT; London: Manning, 2009.

[5] K. E. Kjær, “A survey of context-aware middleware,” in *Proceedings of the 25th Conference on IASTED International Multi-Conference: Software Engineering*, SE’07, (Anaheim, CA, USA), pp. 148–155, ACTA Press, 2007.

[6] F. Chong and G. Carraro, “Architecture strategies for catching the long tail,” *MSDN Library, Microsoft Corporation*, pp. 9–10, 2006.

[7] C. S. Yeo, M. D. de Assunção, J. Yu, A. Sulistio, S. Venugopal, M. Placek, and R. Buyya, “Utility computing and global grids,” *CoRR*, vol. abs/cs/0605056, 2006.

[8] P. Mell and T. Grance, “The NIST Definition of Cloud Computing,” Tech. Rep. 800-145, National Institute of Standards and Technology (NIST), Gaithersburg,, September 2011.

[9] M. Purvis, J. Sambells, and C. Turner, *Beginning Google maps applications with PHP and Ajax*. Springer, 2006.

[10] R. Chellappa, “Intermediaries in Cloud-computing: A new computing paradigm,” in *INFORMS Annual Meeting, Dallas*, 1997.

[11] G. Petri, “Shedding light on Cloud computing,” *The Cloud Academy*, 2010.

[12] IBM, “IBM Softlayer,” 2005. `http://www.softlayer.com`.

[13] E. Amazon, “Amazon Elastic Compute Cloud (Amazon EC2),” *Amazon Elastic Compute Cloud (Amazon EC2)*, 2010.

[14] R. Cloud, “Cloud Sites—The Power of Cloud Computing & Cloud Hosting by Rackspace,” *retrieved on Aug*, vol. 11, 2009.

[15] A. Zahariev, “Google App Engine,” *Helsinki University of Technology*, 2009.

[16] B. Wilder, *Cloud Architecture Patterns: Using Microsoft Azure*. " O’Reilly Media, Inc.", 2012.

[17] K. Pepple, *Deploying OpenStack*. " O’Reilly Media, Inc.", 2011.

[18] Google,Inc., "Google docs," 2005. `https://docs.google.com/`.

[19] Lucid Software, Inc., "Lucidchart," 2012. `https://www.lucidchart.com`.

[20] Lesson Nine GmbH, "Babbel," 2011. `https://www.babbel.com/`.

[21] Z. Wang, "Security and privacy issues within the Cloud Computing," in *Computational and Information Sciences (ICCIS), 2011 International Conference on*, pp. 175–178, IEEE, 2011.

[22] D. Spinellis, "Another level of indirection," in *Beautiful Code: Leading Programmers Explain How They Think* (A. Oram and G. Wilson, eds.), ch. 17, pp. 279–291, Sebastopol, CA: O'Reilly and Associates, 2007.

[23] E. W. Pugh, L. R. Johnson, and J. H. Palmer, *IBM's 360 and Early 370 Systems.* MIT Press, 1991.

[24] R. J. Creasy, "The Origin of the VM/370 Time-sharing System," *IBM J. Res. Dev.*, vol. 25, pp. 483–490, Sept. 1981.

[25] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.

[26] B. Joy, G. L. Steele Jr, J. Gosling, and G. Bracha, "The java language specification," 1998.

[27] E. Bugnion, S. Devine, K. Govil, and M. Rosenblum, "Disco: Running commodity operating systems on scalable multiprocessors," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 4, pp. 412–447, 1997.

[28] M. Rosenblum, "Vmware's virtual platform^{TM}," in *Proceedings of hot chips*, vol. 1999, pp. 185–196, 1999.

[29] G. J. Popek and R. P. Goldberg, "Formal requirements for virtualizable third generation architectures," *Communications of the ACM*, vol. 17, no. 7, pp. 412–421, 1974.

[30] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[31] "Understanding Full Virtualization, Paravirtualization, and Hardware Assist," *VMware White Paper*.

[32] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, "Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 275–287, Mar. 2007.

[33] I. ISO and I. Standard, "7498-1," *Information Technology–Open Systems Interconnection–Basic reference model*, 1994.

[34] M.-C. Hsiao, "The Study of a Linux Container-Based Cloud Operating System for Platform as a Service," July 24 2014.

[35] M. J. Scheepers, "Virtualization and containerization of application infrastructure: A comparison," 2014.

[36] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014.

[37] M. Marschall, *Chef Infrastructure Automation Cookbook*. Packt Publishing Ltd, 2013.

[38] A. Kosmin, "`Puppet` and `nagios`: a roadmap to advanced configuration," *Linux Journal*, vol. 2012, pp. 3:1–3:??, Apr. 2012.

[39] Docker, Inc., "Docker Hub. Docker public community repository," 2014. `https://hub.docker.com/`.

[40] A. S. Kumar, *Virtualizing Intelligent River R: A Comparative Study of Alternative Virtualization Technologies*. PhD thesis, Clemson University, 2013.

[41] CoreOS, Inc., "Fleet. A distributed init system," 2014. `https://github.com/coreos/fleet`.

[42] CoreOS, Inc., "ETCD. Service discovery," 2014. `https://github.com/coreos/etcd`.

[43] J. Palat, "Introducing Vagrant," *Linux Journal*, vol. 2012, no. 220, p. 2, 2012.

[44] Free Software Foundation, "Systemd," 2010. `http://www.freedesktop.org/wiki/Software/systemd/`.

[45] D. S. Linthicum, *Enterprise application integration*. Addison-Wesley Professional, 2000.

[46] S. Davies, L. Cowen, C. Giddings, and H. Parker, *WebSphere Message Broker Basics*. IBM, International Technical Support Organization, 2005.

[47] P. C. Brown, *TIBCO Architecture Fundamentals*. Addison-Wesley, 2011.

[48] A. ServiceMix, "The agile open source ESB," *The Apache software foundation*.

[49] T. Erl, *SOA: principles of service design*, vol. 1. Prentice Hall Upper Saddle River, 2008.

[50] J. Perry, *Java Management Extensions*. " O'Reilly Media, Inc.", 2002.

[51] N. Balani and R. Hathi, *Apache CXF web service development: Develop and deploy SOAP and RESTful web services*. Packt Publishing Ltd, 2009.

[52] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web services platform architecture: SOAP, WSDL, WS-policy, WS-addressing, WS-BPEL, WS-reliable messaging and more*. Prentice Hall PTR, 2005.

[53] M. Fleury and F. Reverbel, "The JBoss extensible server," in *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pp. 344–373, Springer-Verlag New York, Inc., 2003.

[54] J. Chamberlain, C. Blanchard, S. Burlingame, S. Chandramohan, E. Forestier, G. Griffith, M. Mazzara, S. Musti, S. Son, G. Stump, *et al.*, *IBM Websphere RFID handbook: A solution guide*. IBM, International Technical Support Organization, 2006.

[55] A. Nierbeck, J. Goodyear, J. Edstrom, and H. Kesler, *Apache Karaf Cookbook*. Packt Publishing Ltd, 2014.

[56] O. Alliance, "OSGi-the dynamic module system for Java," *accessed, May*, vol. 25, 2009.

[57] B. Snyder, D. Bosnanac, and R. Davies, *ActiveMQ in action*. Manning, 2011.

[58] C. Ibsen and J. Anstey, *Camel in action*. Manning Publications Co., 2010.

[59] K. Fatema, V. C. Emeakaroha, P. D. Healy, J. P. Morrison, and T. Lynn, "A survey of Cloud monitoring tools: Taxonomy, capabilities and objectives ," *Journal of Parallel and Distributed Computing*, vol. 74, no. 10, pp. 2918 – 2933, 2014.

[60] R. Garcia-Carmona, F. Cuadrado, A. Navas, A. Celorio, and J. C. Duenas, "A multi-level monitoring approach for the dynamic management of private iaas platforms," *Journal of Internet Technology, Accepted for publication*, 2013.

[61] A. Vladishev, "Open Source Enterprise Monitoring with Zabbix," in *Open Source Data Center Conference, Nurnberg*, vol. 60, 2009.

[62] G. Aceto, A. Botta, W. De Donato, and A. Pescapè, "Cloud monitoring: A survey," *Computer Networks*, vol. 57, no. 9, pp. 2093–2115, 2013.

[63] J. Varia, "Best practices in architecting Cloud applications in the AWS cloud," *Cloud Computing: Principles and Paradigms*, pp. 459–490, 2011.

[64] S. Strauch, V. Andrikopoulos, S. G. Sáez, and F. Leymann, "ESB^{mt}: A multi-tenant aware enterprise service bus," *International Journal of Next-Generation Computing*, vol. 4, pp. 230–249, November 2013.

[65] J. Van Vliet, F. Paganelli, S. van Wel, and D. Dowd, *Elastic Beanstalk*. " O'Reilly Media, Inc.", 2011.

[66] Google Inc, "Kubernetes. Container orchestration manager," 2014. `https://github.com/GoogleCloudPlatform/kubernetes`.

[67] M. Schwarzkopf, A. Konwinski, M. Abd-El-Malek, and J. Wilkes, "Omega: flexible, scalable schedulers for large compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 351–364, ACM, 2013.

[68] OpDemand, "Deis. Open Source Application Platform for public and private Clouds," 2014. `http://deis.io`.

[69] N. Middleton, R. Schneeman, *et al.*, *Heroku: Up and Running*. " O'Reilly Media, Inc.", 2013.

[70] A. Dabholkar and A. Gokhale, "A generative middleware specialization process for distributed real-time and embedded systems," in *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, pp. 197–204, IEEE, 2011.

[71] M. Roman and N. Islam, "Dynamically programmable and reconfigurable middleware services," in *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, pp. 372–396, Springer-Verlag New York, Inc., 2004.

[72] P. K. Chouhan, H. Dail, E. Caron, and F. Vivien, "Automatic middleware deployment planning on clusters," *International Journal of High Performance Computing Applications*, vol. 20, no. 4, pp. 517–530, 2006.

[73] Red Hat, "Project Atomic. A cross-host Container deployment," 2014. `http://www.projectatomic.ioc`.

[74] CenturyLink, "Panamax," 2014. `http://www.panamax.io`.

[75] J. Beda, "Containers at scale," 2014.

[76] Google, Inc., "cAdvisor. Cloud Application Platform," 2014. `https://github.com/google/cadvisor`.

[77] R. Pike, "The Go Programming Language," *Talk given at Google's Tech Talks*, 2009.

[78] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," *Draft of October*, vol. 7, 2013.

[79] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

All links were last followed on November 19, 2014

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references that the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Stuttgart, November 19, 2014

(Name)