Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. ????

Bootstrapping Provisioning Engines for On-demand Provisioning in Cloud Environments

Lukas Reinfurt

Studiengang: Informatik

Prüfer: Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer: Dipl.-Inf., Dipl.-Wirt. Ing.(FH) Karolina Vukojevic
begonnen am: ??.??.2013
beendet am: ??.??.2014

CR-Klassifikation: ???

Abstract

Table of Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AMI	Amazon Machine Image, page 18
API	Application Programming Interface, page 42
APT	Advanced Packaging Tool, page 26
AWS	Amazon Web Services, page 14
BPEL	Business Process Execution Language, page 18
BPMN	Business Process Model and Notation, page 18
CLI	command-line interface, page 47
CORBA	Common Object Request Broker Architecture, page 37
CSAR	Cloud Service Archive, page 18
EC2	Elastic Compute Cloud, page 14
ESB	Enterprise Service Bus, page 21
FSM	finite state machine, page 47
IaaS	Infrastructure as a Service, page 13
JPF	Java Plugin Framework, page 52
JRE	Java Runtime Environment, page 52
JSPF	Java Simple Plugin Framework, page 52
NIST	National Institute of Standards and Technology, page 13
OASIS	Organization for the Advancement of Structured Information Standards, page 17
OMG	Object Management Group, page 37
OS	Operating System, page 13

1 Introduction

Workflow technology and the service based computing paradigm were mostly used in a business context until now. But slowly they are extended to be used in other fields, such as eScience, where business centric assumptions that where previously true aren't reasonable anymore. One of these assumptions is that services should run continuously. This made sense in large enterprises where those services are used often. Science, on the other hand, often takes a more dynamic approach, where certain services, for example for simulation purposes, are only used at certain times. In those cases, it would make more sense to dynamically provision services only when they are needed.

1.1 Task of this Diploma Thesis

The task of this diploma thesis is to design a lightweight bootstrapping system that can kick off dynamic provisioning in cloud environments. It should be able to provision various provisioning engines in all kinds of cloud environments. The provisioning engines then handle the actual provisioning of required workflow systems and services. A managing component that keeps track of provisioned environments is also part of this system.

Support for different cloud environments and provisioning engines should be achieved through means of software engineering. A functioning prototype that supports Amazon as cloud environment and TOSCA as provisioning engine should be implemented.

1.2 Structure of this Document

...

2 Fundamentals

This chapter starts with a short overview of the SimTech project, of which this diploma thesis is a part of. Next, bootstrapping is defined in the context of this diploma thesis, since it can have various different meanings. Then, a short overview of the cloud landscape is presented, with focus on Amazons cloud offerings, since these are used in this diploma thesis. Finally, we take a look at provisioning solutions, in particular TOSCA, which is also used later in this thesis.

2.1 SimTech

Since 2005, the German federal and state government have been running the Excellence Initiative[1], which aims to promote cutting-edge research, thereby increasing the quality and international competitiveness of German universities. In three rounds of funding, universities have competed with project proposals in three areas: Institutional Strategies, Graduate Schools, and Clusters of Excellence. In total, the winners received approximately 3.3 billion euros since 2005, split up between 14 Instituational Strategies, 51 Graduate Schools, and 49 Clusters of Excellence [13, pp. 16-18].

Simulation Technology (SimTech) is one of the Clusters of Excellence that are funded by the Excellence Initiative. In a partnership between the University of Stuttgart, the German Aerospace Center, the Fraunhofer Institute for Manufacturing Engineering and Automation, and the Max Planck Institute for Intelligent Systems, it combines over 60 project from researchers in Engineering, Natural Science, and the Life and Social Sciences. The aim of SimTech is to improve existing simulation strategies and to create new simulation solutions [13, pp. 109].

In the SimTech project, seven individual research areas collaborate in seven different project networks, one of which is project network 6: *Cyber Infrastructure and Beyond*. The goal of this project network is to build an easy-to-use infrastructure that supports scientists in their day to day work with simulations [10].

[1] `http://www.dfg.de/en/research_funding/programmes/excellence_initiative/index.html`

As part of this project, the SimTech Simulation Workflow Management System (SWfMS) was developed. It is a tool that enables scientists to easily create, manage and execute simulation workflows [14].

2.2 Bootstrapping

The term *to bootstrap sth.* appears to have originated in the early 19th century in the United States, where phrases like “pulling oneself up over a fence by the straps of one's boots” where used as a figure for an impossible task [32]. In the early 20th century the metaphor's sense shifted to suggest a possible task, where one improves one's situation by one's own efforts without help from others. An example of this can be found in James Joyce's Ulysses from 1922, where he writes about “others who had forced their way to the top from the lowest rung by the aid of their bootstraps” [16]. From there, the metaphor extended to the general meaning it has today, which is the act of starting a self-sustaining process that proceeds without help from the outside.

An early reference to bootstrapping in the context of computing dates back to 1953, describing the “bootstrapping technique” as follows: “Pushing the load button then causes one full word to be loaded into a memory address [...], after which the program control is directed to that memory address and the computer starts automatically. [...] [This] full word may, however, consist of two instructions of which one is a Copy instruction which can pull another full word [...], so that one can rapidly build up a program loop which is capable of loading the actual operating program.” [8, p. 1273].

The term bootstrapping is also used with a similar meaning in a business context, where it refers to the process of starting and sustaining a company without outside funding. The company is started with money from the founders which is used to develop a product that can be sold to customers. Once the business reaches profitability it is self-sufficient and can use the profits it generates to organically grow further [23].

In this diploma thesis, bootstrapping describes the process of starting a simple program, that without further help is able to start much more complex programs. These complex programms might require additional middleware, databases, or other components. During the bootstrapping process, all these dependencies will be set up automatically.

2.3 Cloud Computing

Cloud computing emerged in recent years as an alternative to traditional IT. Compared to traditional IT, it offers customers far more flexibility in terms of short term access to and scalability of resource, such as servers, databases, communication service, etc. This increased flexibility is the result of a combination of certain technologies and business models that, although having been around for a while individually, where combined only in recent years.

Since cloud computing is a relatively new phenomenon, there are many definitions of it scattered around. Vaquero et al. looked at over 20 of them and proposed the following definition:

> "Clouds are a large pool of easily usable and accessible virtualized resources (such as hardware, development platforms and/or services). These resources can be dynamically reconfigured to adjust to a variable load (scale), allowing also for an optimum resource utilization. This pool of resources is typically exploited by a pay-per-use model in which guarantees are offered by the Infrastructure Provider by means of customized SLAs."[2] [29]

The National Institute of Standards and Technology (NIST) also proposes a definition:

> "Cloud computing is 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." [20]

Cloud services can be categorized into different cloud service models, according to what exactly each service encompasses. Infrastructure as a Service (IaaS) is at the lowest level and provides a customer with access to a virtualization environment on top of servers, storage, and networking. Here, the customer has to manage the Operating System (OS), middleware stack, applications, and data him self. Platform as a Service (PaaS) is the next higher tier, which offers a customer access to a fully managed runtime environment in the cloud. Here, the customer only has to manage the Application he wants to execute in the runtime environment and the data. Finally, Software as a Service (SaaS) offers a customer access to a fully managed application running in the cloud. In this case, the user has to manage neither the OSs, nor any middleware, application, or data.

[2] Service Level Agreement (SLA): "An agreement that sets the expectations between the service provider and the customer and describes the products or services to be delivered, the single point of contact for end-user problems and the metrics by which the effectiveness of the process is monitored and approved." [26]

Today, there are many different cloud providers offering a huge selection of services. The range of providers spans from titans like Amazon[3], Google[4], Microsoft[5], and IBM[6] to small, focused providers like Heroku[7] or Jelastic[8] and even solutions to build own clouds, like OpenStack[9]. The next section describes Amazon's cloud services in more detail, since those will be used in this diploma thesis.

2.3.1 Amazon Web Services

In 2006, Amazon started offering cloud resource under the umbrella of Amazon Web Services (AWS). Since then, their offerings steadily increased and do now comprise over 20 different products and services for computing, data storage, content delivery, analytics, deployment, management, and payment in the cloud [1].

The most relevant cloud offering for this thesis is Elastic Compute Cloud (EC2)[10], Amazons IaaS offer. It allows customers to rent virtual server instances at an hourly rate. These servers are freely configurable, so virtually any software can be installed, making EC2 very versatile [2]. In addition to general purpose instances (M3), Amazon offers a wide selection of specialized instances, which are optimized for a specific purpose. These include instances optimized for computation performance (C3), memory-intensive applications (R3), or high storage instances (I2). For this thesis we will be using Amazons low cost micro instances (T1) [3].

Also of interest to this thesis is Elastic Beanstalk[11], Amazons PaaS offering. Customers can upload an application and Elastic Beanstalk takes care of deployment and scaling [5]. This makes it easier and quicker to use than EC2, but also less flexible. It could be used instead of a more manual approach with EC2.

Amazon offers multiple ways to interact with cloud resources. All AWS offerings can be controlled using the AWS Management Console, a web based management interface that allows customers to start, stop, and manage cloud resources on-demand. It also provides access to account and billing information [15]. Additionally, Amazon provides a command line interface, tools for Eclipse and Visual Studio, and Software Development Kits (SDKs) for

[3] http://aws.amazon.com
[4] https://cloud.google.com
[5] http://azure.microsoft.com
[6] http://www.ibm.com/cloud-computing
[7] https://www.heroku.com
[8] http://jelastic.com
[9] https://www.openstack.org
[10] http://aws.amazon.com/ec2
[11] http://aws.amazon.com/elasticbeanstalk

several programming languages, including Java, .Net, Python, Ruby, and the Android and iOS platforms [27]. In this thesis, we will use the AWS Java SDK[12] to interact with Amazons cloud resources programmatically.

2.4 Provisioning

This section provides an overview of provisioning in general and describes some of the provisioning solution available today. It focuses in particular on TOSCA and Open TOSCA, since those are used in the prototypical implementation later on.

2.4.1 Overview

Setting up a complex distributed system with many different components scattered across multiple environments is a time-consuming task if done by hand. For this reason, many provisioning solutions have been created over the years to automate this process. They differ in some areas, which we will discuss later, but their core functionality is in basically identical: They prepare all necessary resources for a certain task. Let's state this core functionality a little more precise with the following definition: Provisioning is, “in telecommunications, the setting in place and configuring of the hardware and software required to activate a telecommunications service for a customer; in many cases the hardware and software may already be in place and provisioning entails only configuration tasks” [11]. Since we are working in a cloud environment, we won't have to deal with hardware directly, but rather with virtual machines (VMs). So for us, provisioning means the creation and deletion of VMs in a cloud environment, as well as the installation, configuration, monitoring, running and stopping of software on these VMs [18].

The are many benefits to using an automated provisioning solution in stead of doing things manually. The manual approach is limited by how much work a single person can do at any time, whereas an automatic approach is able to do much more work, in less time, and potentially in parallel. This makes it possible to manage huge infrastructures with very little resources, which can save money compared to a manual approach. Since every step that needs to be done to provision a system has to be written down, a detailed description of the whole provisioning process is created. This makes the whole process reproducible and less error prone, since the human factor is taken out. Parts of such a description can also be

[12] `https://aws.amazon.com/sdkforjava`

shared in a business or even between businesses, which makes the process of creating such a description potentially much more efficient.

The general process of working with provisioning software is very similar with all the different solutions. It can be described as a two step process. In step one, a description of the whole provisioning process has to be created using the tools provided by the particular solution. In general this involves creating a textual description in a certain format that is understood by the provisioning software that is to be used. In this description, we tell the software what virtual resources we need, what software should be installed on them and how everything should be configured. In step two, we pass this description to the provisioning software, which interprets and executes it.

Many different provisioning solutions exist today. Some cloud providers offer provisioning solutions that are particularly tailored to their cloud offerings, for example AWS CloudFormation[13], which can only be used to provision resources in the Amazon cloud. Then, there are more generally usable provisioning solutions that are not bound to any particular cloud provider. A few popular examples include Ansible[14], Chef[15], Puppet[16], and TOSCA, which we will discuss in detail later.

All these solutions differ in some form or another. A full feature comparison of different solutions is out of scope for this diploma thesis, but what follows is a short overview of some of the differences. As already mentioned, AWS CloudFront is bound to Amazons cloud platform, while the other solutions are not. Chef and Puppet both use a client server architecture, where each node that should be configured by them has to run a client program to communicate with a server node, whereas Ansible executes its command over SSH and therefor doesn't require additional software on the nodes that are configured. The solutions also differ in modularity and flexibility. While Ansible, Chef, Puppet and TOSCA are highly flexible and can be used in a fine grained modular fashion, this also makes them more complex to use, for example compared to AWS CloudFront.

2.4.2 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard that is currently being worked on by the Organization for the Advancement of Structured Information

[13] http://aws.amazon.com/cloudformation
[14] http://www.ansible.com
[15] http://www.getchef.com/chef
[16] http://puppetlabs.com/

Standards (OASIS)[17]. Its development is also supported by various industry partners, which include IBM, Cisco, SAP, HP and others. It's aim is to provide a language that can describe service components and their relations in a cloud environment independent fashion [28].

TOSCA defines an XML syntax, which describes services and their relations in a so called service templates. Figure 2.1 shows that a service template can be made of up of four distinct parts: Topology templates, orchestration plans, reusable entities, and artifact templates.

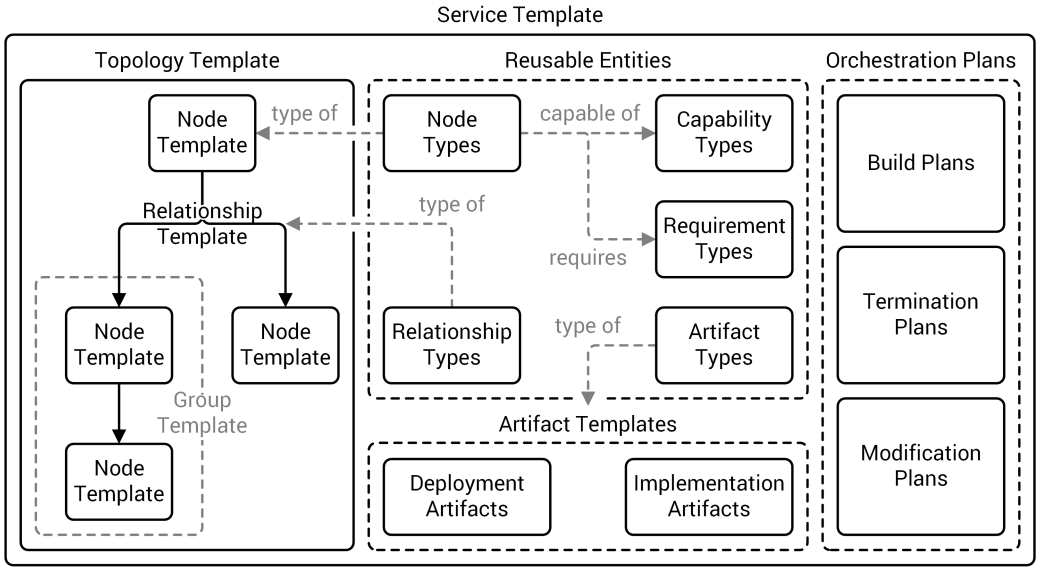

Figure 2.1: TOSCA service template structure [based on 28].

Topology templates, as seen on the left side of Figure 2.1, model the structure of a service as a directed graph. The vertices of the graph represent nodes, which are occurrences of a specific component, for example, an application server or a database. These nodes are defined by node types, or by other service templates. Node types are reusable entities, as shown in the top center of Figure 2.1. They define the properties of a component, as well as operations to manipulate a component, so called interfaces. Additionally, node types can be annotated with requirements and capabilities. These, in turn, are defined by requirement and capability types, which also belong to the group of reusable entities. This allows for requirement and capability matching between different components. The edges of the graph represent connections between nodes, which are defined by relationship templates that specify the properties of the relation. An example for such a connection would be a node A, representing a web service, which is deployed on node B, an application server. Relationship types are also used to connect requirements and capabilities.

[17] https://www.oasis-open.org/

Orchestration plans, shown on the right of Figure 2.1, are used to manage the service that is defined by the service template. TOSCA distinguishes between three types of plans: Build plans, termination plans, and modification plans. Build plans describe how instances of a service are created. Termination plans describe how such a service is removed. Modification plans manage a service during its runtime. These plans consist of one or more tasks, i.e., an operation on a node (via an interface) or an external service call, and the order in which these tasks should be performed. They can be written in Business Process Model and Notation (BPMN) or Business Process Execution Language (BPEL), which are already existing languages to describe process models.

The bottom center of Figure 2.1 shows artifact templates, which represent artifacts. Artifacts are things that can be executed directly (e.g.: scripts, archives) or indirectly (e.g.: URL, ports). TOSCA further distinguishes between two types of artifacts, namely deployment and implementation artifacts. Deployment artifacts materialize instances of a node and are used by a build plan to create a service. An example for this is an Amazon Machine Image (AMI), which creates an Apache server once deployed in a VM. Implementation artifacts represent the interfaces of components. Here, an example would be a node that has an interface for starting the particular component described by the node. This interfaces could be implemented by an implementation artifact like a *.jar* file.

One or more TOSCA service templates are packaged, together with some meta data, into a Cloud Service Archive (CSAR), which is essentially a zip file that contains all files necessary to create and manage a service. CSAR files can then be executed in a TOSCA runtime environment, also called TOSCA container, to create the service described within.

2.4.3 OpenTOSCA

OpenTOSCA is a browser based open-source implementation of a TOSCA container, created at the IAAS at University Stuttgart, which supports the execution of TOSCA CSAR archives. Figure 2.2 shows the architecture of OpenTOSCA. Its functionality is realized in three main components, which are the Controller, the Implementation Artifact Engine, and the Plan Engine. After a CSAR is uploaded to OpenTOSCA it can be deployed in three steps. In the first step, the CSAR file is unpacked and its content is stored for further use. The TOSCA XML files are then loaded and processed by the Controller. The Controller in turn calls the Implementation Artifact Engine and the Plan Engine. The Implementation Artifact Engine knows how to deploy and store the provided implementation artifacts via plugins. Plans are then run by the Plan Engine, which also uses plugins to support different plan formats. OpenTOSCA also offers two APIs, the Container API and the Plan Portability API. The Container API can be used to

access the functionality provided by the container from outside and to provide additional interfaces to the container, like the already existing admin UI, self-service portal, or modeling tool. The Plan Portability API is used by plans to access topology and instance information [7].

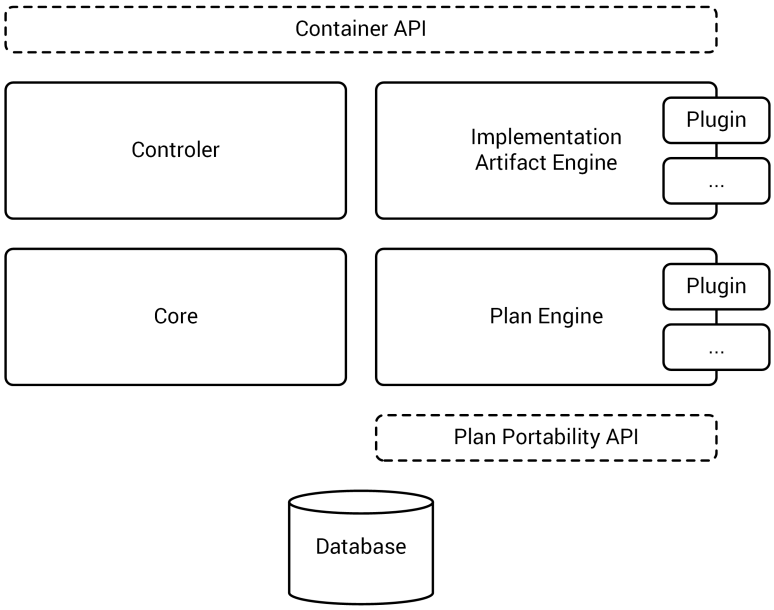

Figure 2.2: OpenTOSCA architecture [based on 7].

3 Related Work

This chapter summarizes related work of other authors that is of interest to this thesis. First, we present the paper that laid the foundation for this diploma thesis. Then we take a look at another diploma thesis which expanded some ideas presented in the first paper. We also take a look at a paper that presents work similar to this diploma thesis.

3.1 On-demand Provisioning for Simulation Workflows

Vukojevic-Haupt, Karastoyanova, and Leymann identify requirements that need to be addressed to make the current approach for scientific workflows used by the SimTech SWfMS more suitable for scientific simulation work. These requirements are: Dynamic allocation as well as release of computing resources, on-demand provisioning and deprovisioning of workflow middleware and infrastructure, and dynamic deployment and undeployment of simulation services and their software stacks. To fulfill these requirements, they propose a new service binding strategy that supports dynamic service deployment, an approach for dynamic provisioning and deprovisioning of workflow middleware, an architecture that is capable of these dynamic deployment and provisioning operations, and, as part of this architecture, the bootware - the subject of this diploma thesis - that kicks of these dynamic processes [30].

The new service binding strategy is necessary, because existing static and dynamic binding strategies, as shown on the left and in the center of Figure 3.1, rely on services that are always running, or, as in the case of dynamic binding with service deployment, only dynamically deploy the service, but not its middleware and infrastructure. The new service binding strategy, shown on the right of Figure 3.1, called *dynamic binding with software stack provisioning*, is similar to the already existing dynamic binding with service deployment strategy, but adds the dynamic provisioning of the middleware and infrastructure required by the service [30].

Their approach for dynamic provisioning and deprovisioning of workflow middleware is separated into six steps. The first step is to model and start the execution of a simulation

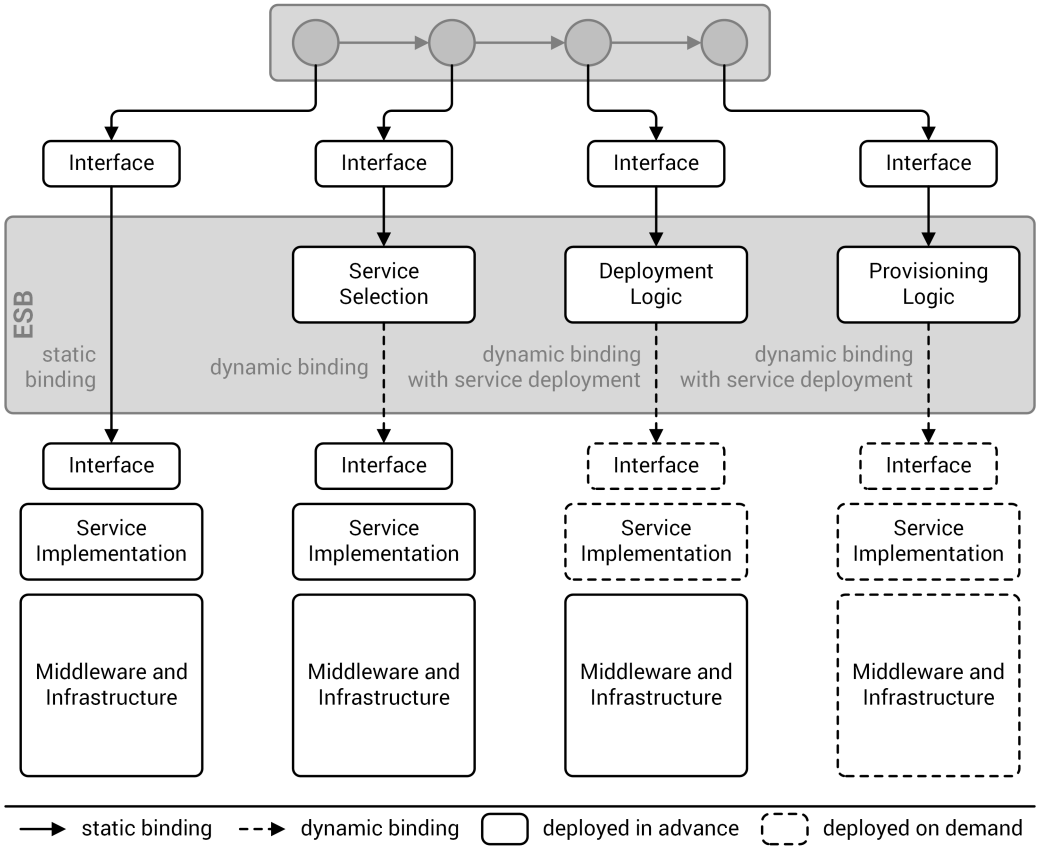

Figure 3.1: Simplified overview of service binding strategies [based on 30].

workflow. For this, the Modeler component shown on the left of Figure 3.2 is used. In the second step, the middleware for executing the worklfow, e.g. the SWfMS, and its underlying infrastructure are provisioned. This is shown in Figure 3.2 as deployment step one and two. In deployment step one, the bootware deploys a provisioning engine into a cloud environment. In deployment step two, this provisioning engine is used to deploy the actual middleware. Now, the workflow can be deployed on this middleware, which is step three. In step four, an instance of this workflow is executed. During this execution, some external service might be required that are not yet available. The Enterprise Service Bus (ESB) determines this by checking the service registry. If the requested service isn't available, the ESB tells the provisioning engine to provision this service. The on-demand provisioning of services is step five, which corresponds to deployment step three in Figure 3.2. The service is also deprovisioned if it is no longer needed. The final step is to deprovision the workflow model and the workflow execution middleware after the execution of the workflow instance is finished [30].

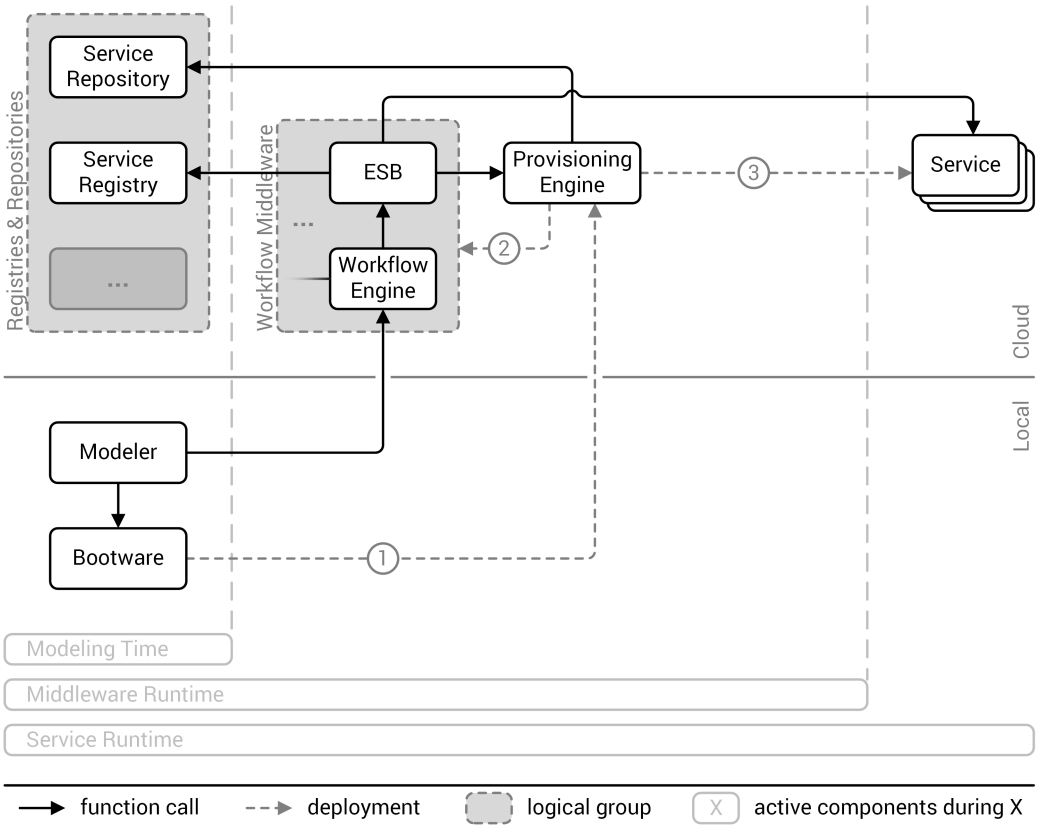

Figure 3.2: Proposed architecture [based on 30].

The architecture they present can be separated into a local part and a cloud part, as well as different phases. Figure 3.2 shows that the only local components are the modeler and the bootware, while all other components are hosted in the cloud. In the modeling phase, a scientist uses local modeling and monitoring tools in combination with cloud hosted repositories and registries to create a workflow. These components are always running. When he deploys the workflow, the local bootware component kicks of the on demand provisioning process and therefore the second phase, called middlware runtime phase. In this phase, the bootloader deploys a provisioning engine in the cloud, which in turn deploys the workflow middleware. Once the middleware is up and running, the workflow can be executed. During the execution, the ESB receives service calls from the workflow engine. Services that are not running at this time can then be provisioned by the provisioning engine. This takes place in the third phase, the service runtime phase. The bars at the bottom of Figure 3.2 show, which components are active during which phase [30].

3.2 Dynamic Provisioning of Web Services for Simulation Workflows

Schneider found some problems with the architecture proposed in section 3.1. The original architecture assumes that only one provisioning engine is used at a time. It neglects situations where services might require another (or multiple other) provisioning engines, because their provisioning descriptions aren't available in a format that the currently used provisioning engine understands. It also assumes that the ESB communicates directly with this provisioning engine to deploy and undeploy other services. This implicates that the ESB understands all manner of interfaces provided by various provisioning engines.

It further assumes that every provisioning engine knows how to communicate with the service repository to get the information and resources it needs to provision a service. While this might be true for some provisioning engines, it's certainly not true for all of them. This problem is further amplified because there are no standards defined for such a service repository.

Another assumption of the original architecture is that a provisioning engine always understands the format of the service packages provided by the service repository. Different provisioning engines use different formats which are in general not compatible. If provisioning engines would all use a standardized format (like CSAR), this would not be a problem, but that isn't the case.

Schneider further refines the previously shown middleware architecture by adding a provisioning manager as intermediary between the ESB and the provisioning engines [24]. This addition improves the original architecture in three aspects.

The ESB can now use the stable interface of the provisioning manager to trigger provisioning engines instead of calling those provisioning engines directly. The provisioning manager handles the differences between the provisioning engines. This makes it also possible to use multiple different provisioning engines during one workflow execution. The provisioning manager also handles the communication with the service repository or possibly multiple service repositories for different provisioning engines. He can provide information to a particular provisioning engine if it cannot get the information it needs from the service repository on its own. The provisioning manager could also translate different service distribution formats so that provisioning engines could be used with formats that they don't support

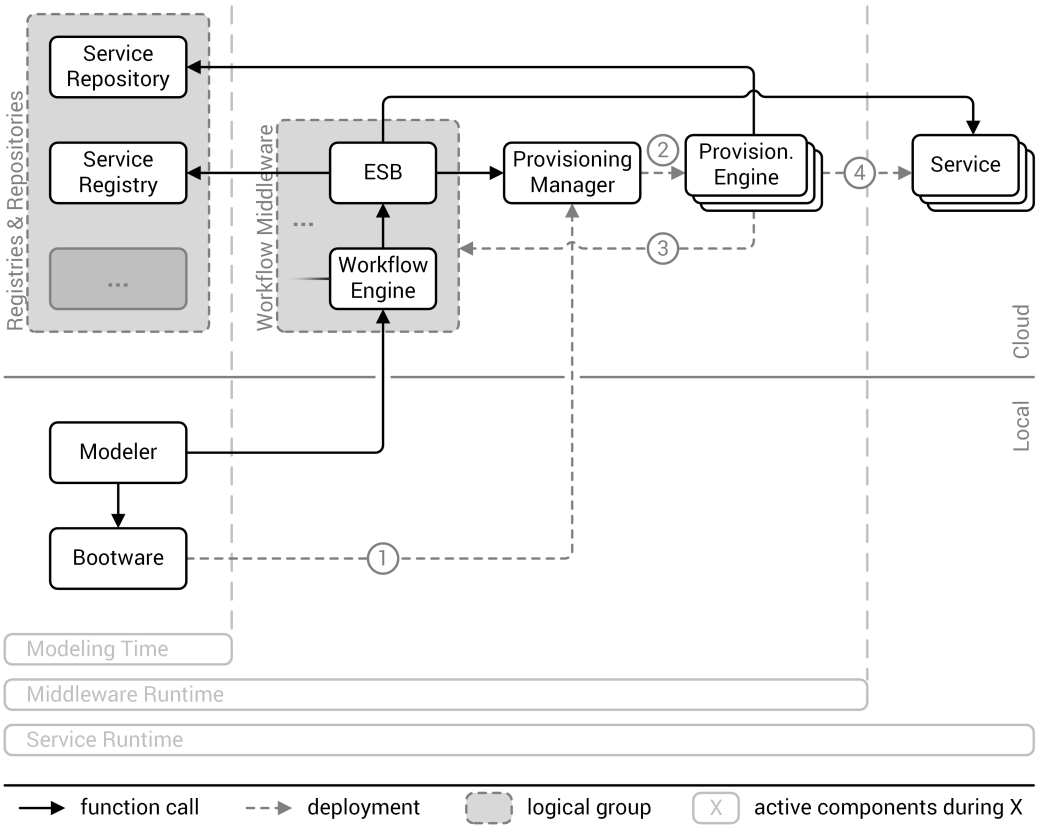

Figure 3.3: Extended architecture with added provisioning manager [based on 24]

3.3 Architecture for Automatic Provisioning of Cloud Services

Kirschnick et al. present an extensible architecture for automatic provisioning of cloud infrastructure and services at different cloud providers. They define a cloud service as a number of software components, their configuration, and the cloud infrastructure they are running on. Therefore, to provision a cloud service, one must first provision the cloud infrastructure, then install the software components and finally configure them. For this process they designed a so called service orchestrator [17].

Figure 3.4 shows a simplified overview of the service orchestrator architecture. Users can submit service models via a service API, shown at the bottom of Figure 3.4. The service model describes the topology of a cloud service, the components it consists of and their configuration. This model is used by the service orchestrator to provision new cloud services

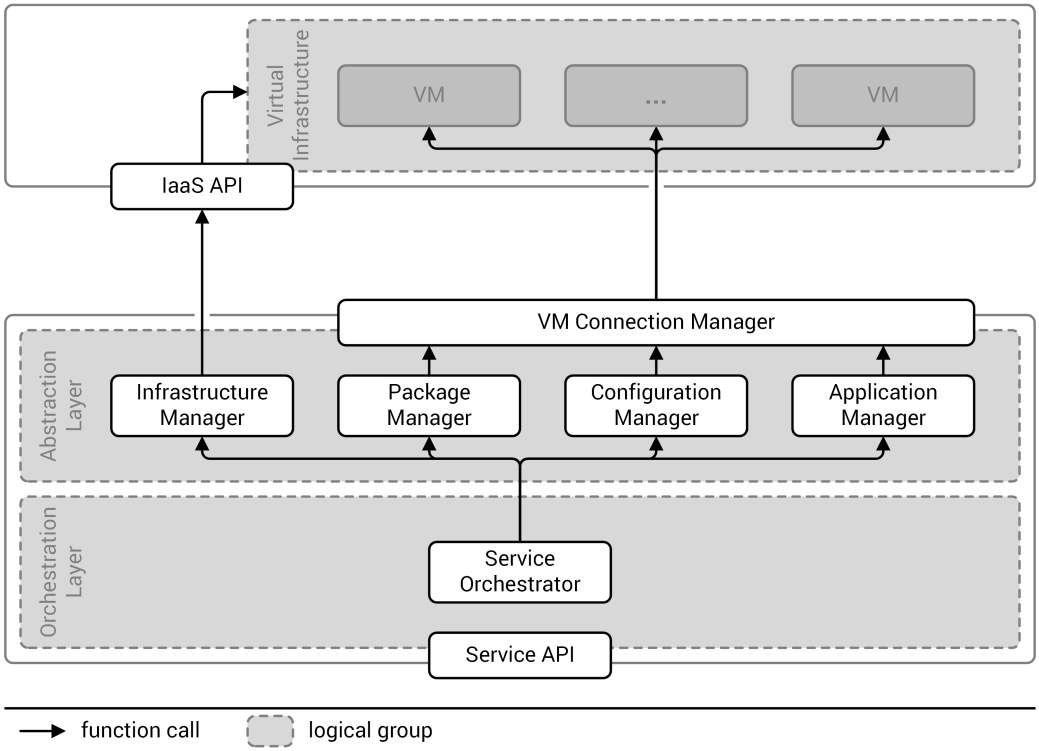

Figure 3.4: Simplified architecture for automatic provisioning of cloud services [based on 17]

and to trigger reconfiguration and topology changes of existing services [17].

The service orchestrator is build in two layers. The orchestration layer, shown at the bottom of Figure 3.4, picks up the service model and delegates the different steps that are necessary to provision the cloud service to the abstraction layer. The deployment steps will be executed in parallel wherever dependencies allow it. The abstraction layer, shown in the middle of Figure 3.4, provides abstract methods to handle the management, installation, configuration, and starting of software. It consists of five different managers for infrastructure, packages, applications, configuration, and VM connections [17].

The infrastructure manager abstracts away different APIs of cloud providers for provisioning cloud infrastructure. It is used to create new VMs at a specific cloud provider, shown at the top of Figure 3.4. These can then can be used by the other managers to create the requested cloud service. The VM connection manager offers a unified interface for different communication methods, for example Secure Shell (SSH), Remote Desktop Connection (RDC), Virtual Network Computing (VNC), and Telnet. The package, configuration, and application managers use the interface provided by the connection manager to send their commands to

the VMs at the cloud provider [17].

The package manager can install software packages in different environments, for example with existing package managers like Advanced Packaging Tool (APT), or directly on the file system. The configuration manager offers a unified interface for component specific configuration methods, such as file templates or configuration APIs. The application manager manages the state of software components, i.e. starting and stopping them. Together, these managers can install, configure, and manage the requested cloud service on the VMs provided by the cloud provider [17].

4 Requirements, Constraints and Design

In this chapter we will talk about the design of the bootware component and the requirements and constraints that shape it. We begin with the requirements, which were explicitly given at the beginning of this thesis. Next, we describe additional constraints which we added to limit the scope of the work. Then, we explain the design that formed as a result of the limitations that we had.

4.1 Requirements

The main goal of this thesis is to lay a ground stone by creating the core design of the bootware component. It was clear from the beginning that, because of the limited time available, not every feature that might be necessary for the full operation can be fully implemented. Instead, the foundation we develop here should keep future needs in mind and make it simple to extend when needed. It is therefore a core requirement to keep the bootware component relatively generic and make it extensible where necessary.

It should be extensible in two key areas, namely the support for different cloud providers and for different provisioning engines. For this thesis, Amazon is the only cloud provider that has to be supported, but it has to be possible to add others in the future. Concerning provisioning engines, only OpenTOSCA has to be supported for now, but again with the possibility to add more in the future.

It is also important that the bootware is easy to use. In fact, it should be practically invisible whenever possible. It should hook into the already existing process of deploying a workflow without adding unnecessary interaction steps when possible. However, It can't be hidden completely, because the user has to specify a cloud provider and the corresponding log-in credentials some where. The user should also get some feedback about the progress of the deployment, because this process might take some time and might seem unresponsive without sufficient feedback.

A further requirement is that the bootware component should be relatively lightweight and open standards should be used where possible.

4.2 Constrains

The Bootware could theoretically be written in any major programming language but we limit our selves to Java. The reason for this is that all the other SimTech components are written in Java, so by also using Java we fit nicely into this already existing ecosystem. Additionally, for things like Eclipse integration we would have to use Java anyway. We also have to keep in mind that the bootware component will not be finished with this thesis. Other people will have to extend it in the future and since Java is common in general, as well as in the SimTech project, it makes sense to use it instead of another, maybe more obscure programming language.

We can further narrow our use of Java by limiting us to Java 1.6. This also has to do with the already existing parts of the SimTech project, that are geared towards this version as well. Using another version of Java could lead to unforeseen incompatibilities.

In the next chapter we will also introduce additional constraints that became necessary during the design process and will therefore be explained at the appropriate times.

4.3 Design

In this section we will develop the design of the bootware. This design is held intentionally abstract. Specific implementation details will be selected and described in section 6.1.

4.3.1 Component Division

As described in section 3.1 on page 20, the proposed architecture initially only envisioned one bootware component. This architecture was expanded with the introduction of the provisioning manager, as described in section 3.2 on page 23. At this stage, the provisioning manager included all the functionality necessary to provision and deprovision provisioning engines in the cloud, in addition to the functionality already mentioned in section 3.2. This was a somewhat convoluted design where multiple responsibilities where mixed into one

component. It was later decided that the provisioning manager should be split into two parts. The actual provisioning manager handles the communication with the service repository and the various provisioning engines, as described before in section 3.2. A separate bootware component handles the provisioning and deprovisioning of the provisioning engines. At the moment, that leaves us with two bootware components, one local and one remote, where the local bootware component kick-starts the remote bootware, which then handles the actual provisioning of provisioning engines. The first question that has to be answered is whether or not this division is reasonable, or if another alternative makes more sense. We will now discuss the viability of four such alternatives.

Single Local Component

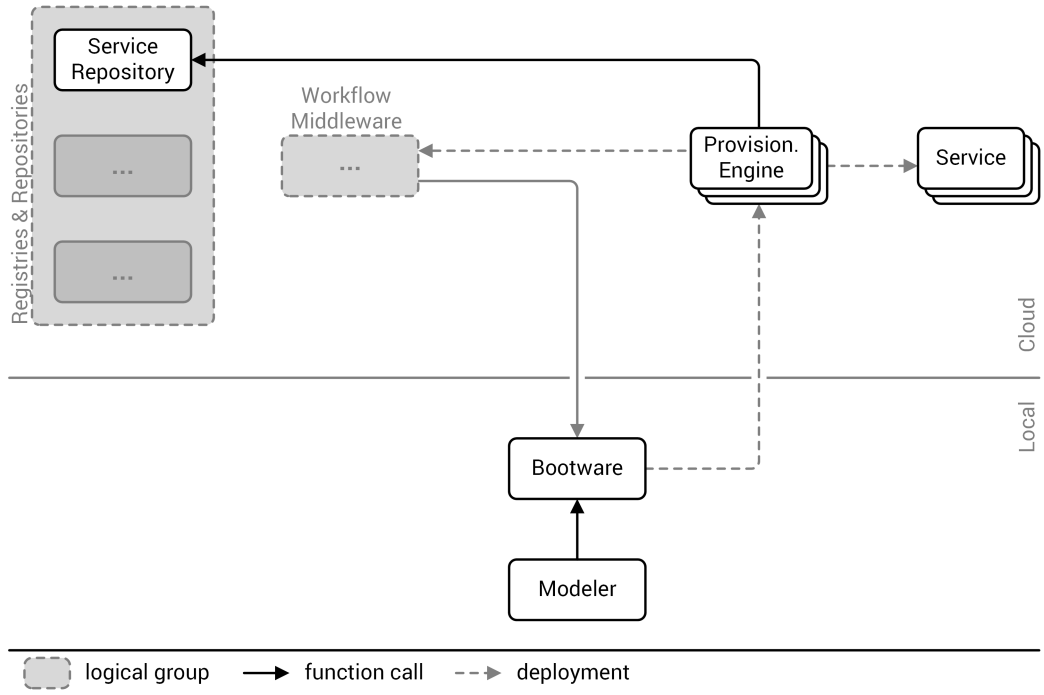

Figure 4.1: Simplified overview of the single local component architecture

First we consider the simplest case: A single local component. In this scenario, all provisioning processes are initiated from a component installed locally on the users machine, alongside or as part of the workflow modeler.

The advantages of this architecture lie in its simplicity. Only one component has to be created and managed. We wouldn't have to deal with any cloud environments and each user would

have his own personal instance, so multi-tenancy wouldn't be an issue. There is no possible overlap in functionality, as it would be the case in a 2-tier architecture (see section 4.3.1) and communication between multiple bootware components doesn't have to be considered.

The disadvantages are caused by the component being local. Since all the functionality is concentrated in one component, it can become quite large and complicated, which is one thing that should be avoided according to the requirements. A much bigger problem however is the remote communication happening in this scenario. All calls to the bootware component from the ESB of the workflow middleware would leave the remote environment. Also, all calls from the bootware component to the provisioning engines would enter the remote environment. This type of split communication can be costly and slow, as shown by Li et al. They compared public cloud providers and measured that intra-datacenter communication can be two to three times faster and also cheaper (often free) compared to inter-datacenter communication [19].

Single Remote Component

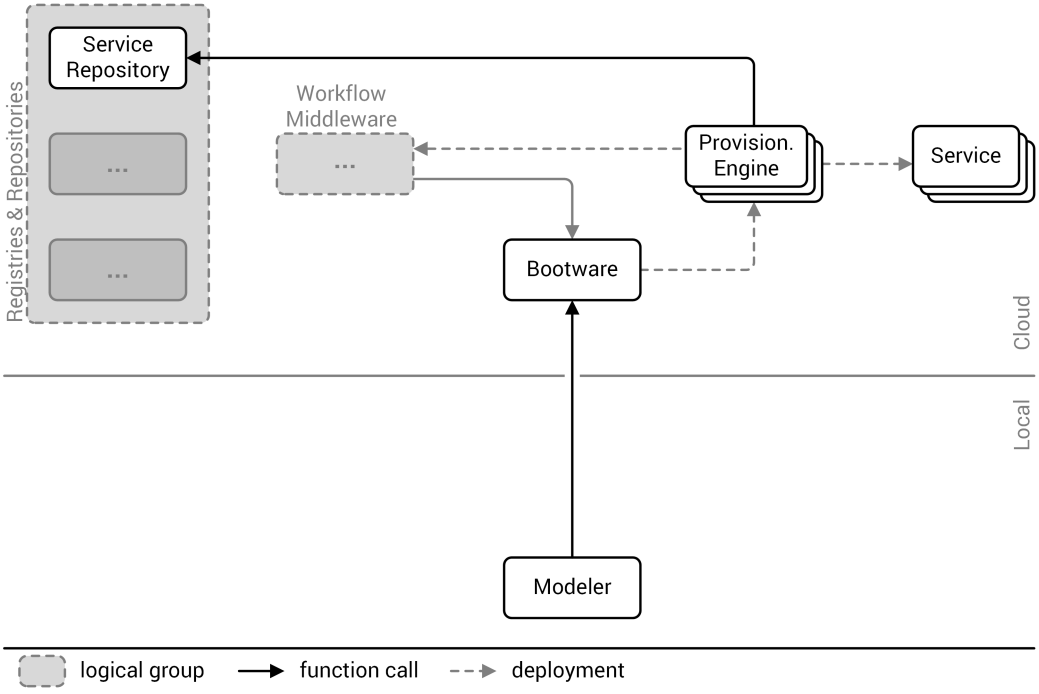

Figure 4.2: Simplified overview of the single remote component architecture

The next obvious choice is to put the single bootware component into a remote environment,

where the disadvantages of local to remote communication would disappear. However, this creates new problems.

Since there aren't any additional components in this scenario that could manage the life-cycle of the remote bootware, the user would have to manage it by hand, which leads to two possibilities. Either the user provisions the bootware once in some cloud environment and then keep this one instance running, or she provision it once she needs it and deprovisions it when she is done.

In the first case the user would only have to provision the bootware once, but this creates a new problem: The user doesn't know where exactly to put the bootware component. Since one requirement is that multiple cloud environments should be supported, it is possible that the bootware component is not located anywhere near the cloud environment where it should provision further components. The communication problem of the single local bootware component can still occur in these cases.

Another problem is that the bootware would be running all the time, even if the user doesn't need it, which would increase costs. This problem could be reduces if this bootware instance is shared with others to assure a more balanced load. But then the user would have to manage some sort of load balancing and the bootware component would have to support multi-tenancy or be stateless to be able to cope with potential high usage spikes. This would further complicate the design and implementation of the bootware component and possibly increase the running costs.

In the second case the user would provision the bootware whenever she needs it. Now the user would be able to pick a cloud environment that is close to the other components that she plans to provision later. This eliminates the two major problems of the first case but increases the effort that the user has to put into a task that she shouldn't have to do in the first place. Life-cycle management of the bootware should be automated completely and hidden away from the user. Therefor, this scenario is not appropriate for our case.

2-Tier Architecture

Next, we take a look at a 2-tier architecture, where the bootware is divided into two components. On the local side we have a small and simple component which has mainly one function: To provision the larger second part of the bootware in a remote environment near to the environment, where other components will be provisioned later.

This eliminates the problems of a single local or remote bootware component. The user no

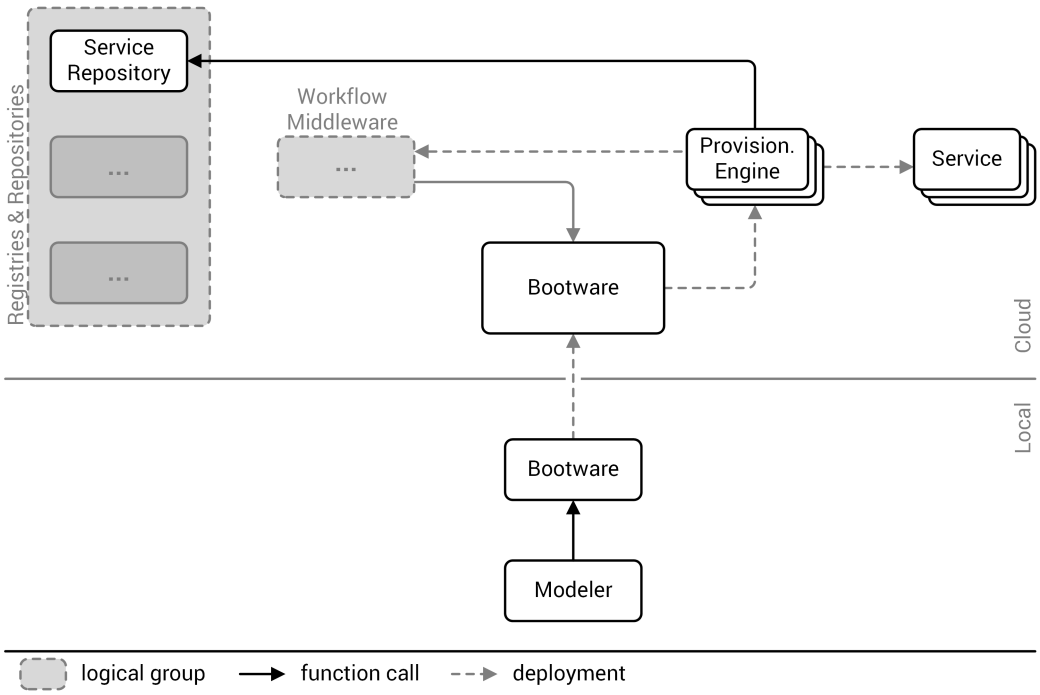

Figure 4.3: Simplified overview of the 2-tier architecture

longer has to be involved in the management of the remote bootware since the local bootware handles all that. Since we provision the remote bootware on demand we now also can position the remote bootware close to other remote components to minimize local/remote communication and the problems resulting of it. We can now keep the local part as simple as possible and make the remote part as complicated as it has to be and since we provision the remote bootware only for one user we don't have to worry about multi-tenancy.

But we also introduce new problems. For one, we now have duplicate functionality between the two components. Both components have to know how to provision a component into multiple cloud environments. The local component has to be able to put its remote counterpart into any cloud environment. The remote component has to be able to provision other components into the same environment in which it runs (ideally, to minimize costs). Since itself can be located in any cloud environment, it has to be able to do this in any cloud environment. Independent from this, it has to be able to provision to any environment that the user/service package chooses. But this problem can be solved by using a plugin architecture, which allows both components to use the same plugins. We discuss plugins in detail in subsection 4.3.2.

A second problem which we can't avoid but can solve is the communication which is now necessary between the different parts of the bootware. More on this in subsection 4.3.3

Cloning

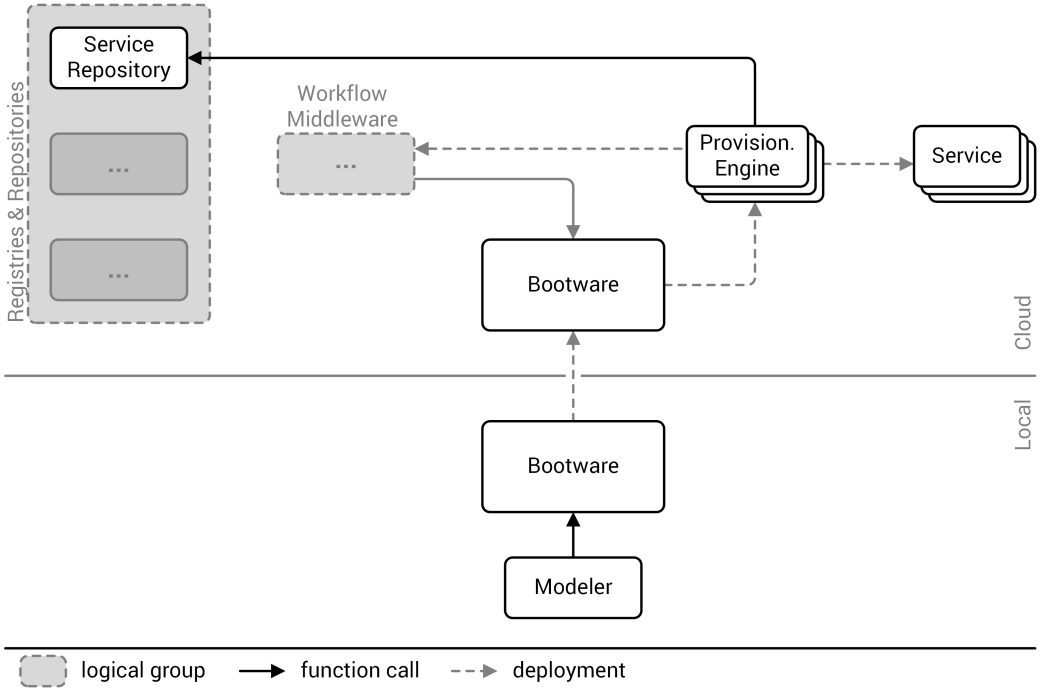

Figure 4.4: Simplified overview of the cloned component architecture

This architecture can be seen as an alternative form of the 2-tier architecture described in section 4.3.1. In this case there are also two bootwares working together and the remote bootware does most of the work. However, the local and the remote bootware are identical. Instead of provisioning a bigger bootware in a remote environment, the local bootware clones itself. Compared to the 2-tier architecture described before, this has the advantage that only one component has to be designed and implemented and that function duplication is not an issue. The disadvantage would be that the local bootware would be exactly as complex as the remote bootware and would contain functionality that it wouldn't require for local operation (e.g. a web service interface). However, since we want to keep the whole bootware, including the remote part, fairly lightweight, it's highly unlikely that the complexity of the remote bootware will reach such heights that it could not be run on an average local machine. In this case, the advantage of only having to design and implement one component seems to outweigh the disadvantage of a slightly more complex local component (compared to the 2-tier variant). Of course, this architecture makes only sense if the functionality of the two separate components in the 2-tier architecture turns out to be mostly identical. Therefore we can't decide yet if this architecture should be used.

Decision

Of the four alternative presented here, alternative three - the 2-tier architecture - makes the most sense. Therefore it is selected as the alternative of choice and used for further discussion. We do however retain the option to transform it into alternative four if we discover that both components share much of same functionality, but this can only be judged at a later stage, when we know exactly how the internal functionality of the bootware will work.

4.3.2 Extensibility

The requirements for the bootware component state that support for different cloud environments and provisioning engines should be achieved through means of software engineering. This requirement is intentionally vague to allow to select a fitting extension mechanism during the design process. In this section we will take a look at different extension mechanisms for Java and pick the one that suits our needs best.

Extension Mechanisms

The simplest way to fulfill the extensibility requirement would be to create a set of interface and abstract classes to define the interfaces and basic functionality that are necessary to work with different cloud environments and provisioning engines. These interfaces and abstract classes would then be implemented separately to support different scenarios and would be compiled, together with the rest of the application, into one executable. At runtime, a suitable implementation would be selected and used to execute the specific functionality required at this time.

This extension mechanism is simple, but restricted by its static nature. The entire executable has to be recompiled if any implementations are changed or added. This may not be a problem if the set of possible extensions that have to be supported is limited and known at the time of implementation or if it changes rarely. If the set of necessary extensions is unknown or changing from time to time, implementing new, or changing existing extensions, can get cumbersome, since a new version of the whole software has to be released each time. Additionally, if a user wanted to add an extension for a specific use case, he would have to get the source code of the bootware, add his extension and then compile the whole bootware. It would be far better if extensions could be implemented separately from the core bootware component and added and removed at will.

A more flexible architecture is needed, for example a plugin architecture. Interfaces for the extension points still exist but the extension are no longer part of the main bootware component. They are compiled separately into plugins that can be loaded into the main bootware component on the fly. There are several possibilities to realize such an architecture.

It is certainly possible to implement a plugin framework from scratch. An advantage of this approach would be that the design of the plugin architecture could be tailored to our use case and would be as simple or complex as needed. But there are also several disadvantages. For one, we would reinvent the wheel, since multiple such frameworks already exist. It would also shift resources away from the actual goal of this thesis, which is designing the bootware component. Furthermore it would require a deep understanding of the language used for the implementation (in this case Java), which is not necessarily given. Therefore it seems more reasonable to use one of the already existing plugin frameworks. Which one exactly will be determined later in subsection 6.1.1.

Plugin Repository

Now that we have introduced plugins we face new problems. Figure 4.5 shows the current architecture, where both bootware components use their own plugins. If a plugin is added or updated, the user has to manually copy this plugin to the right folder of one or both of the bootware components. Furthermore, if both components use the same plugins, which they will (for example plugins for different cloud providers), we will have duplicate plugins scattered around. This is inefficient, probably annoying for the user and can possibly cause errors if plugins get out of sync.

To remedy this situation we introduce a central plugin repository, as shown in Figure 4.6. This repository holds all plugins of both components so it eliminates duplicate plugins. If plugins are added or modified it has only to be done in one place. Plugin synchronization can happen automatically when the bootware components start, so that the user is no longer involved in plugin management. The repository also enables easy plugin sharing, which was cumbersome earlier.

While a central plugin repository is a sensible addition to the proposed bootware architecture, its design and implementation are out of scope of this thesis.

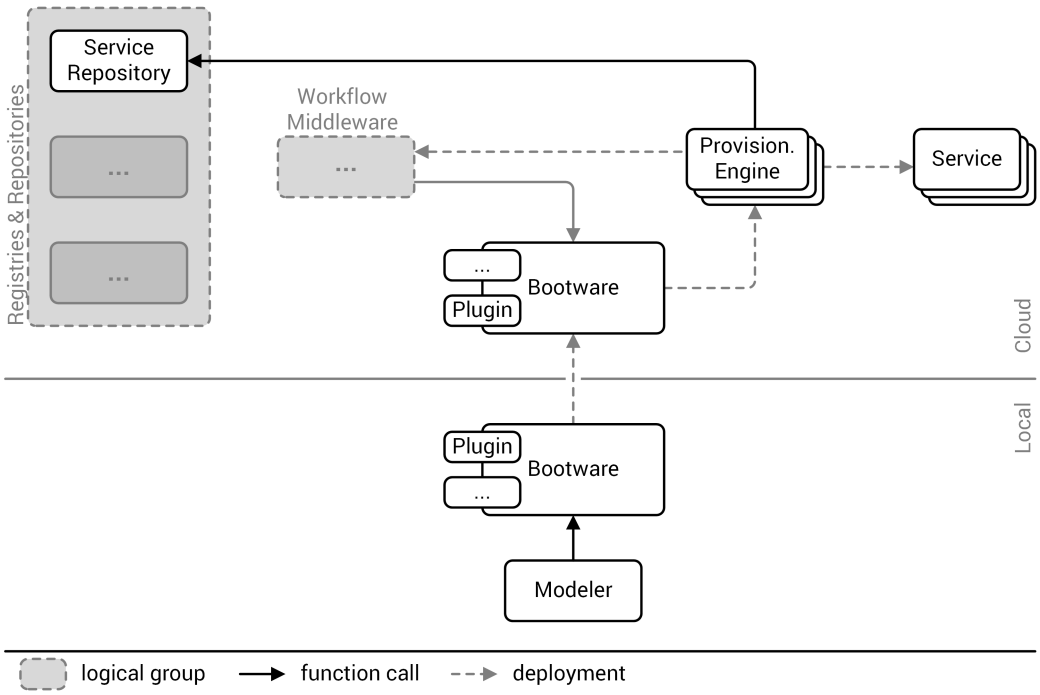

Figure 4.5: Simplified overview of the 2-tier architecture with plugins

4.3.3 External Communication

Since other components will have to call the bootware and since we will use a 2-tiered approach for the bootware component, we now have to decide, how the external communication with the bootware will work. There are several factors that impact this decision. Communication between the components should be as simple as possible, but has to support some critical features. To keep it simple, it would make sense to use the same communication mechanism for communication between the bootware components as well as with other external components, like the ESB.

Since the provisioning processes kicked off by the bootware can potentially take a long time to finish (in the range of minutes to hours), asynchronous communication should be used between the components to avoid timeouts and blocking resources. For the same reason, there should be some mechanism to get feedback on the current status during a long running provisioning process.

The communication with the bootware components will contain sensitive data, for example login information for cloud providers. This information has to be provided from the outside

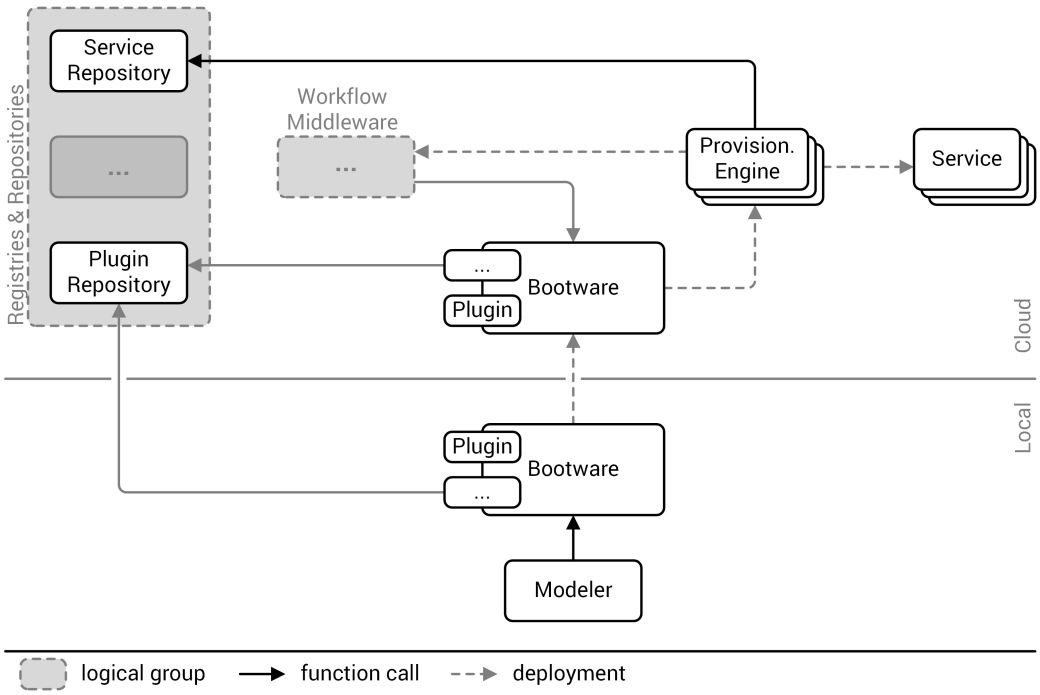

Figure 4.6: Simplified overview of the 2-tier architecture with a plugin repository

on a call to call basis and should be transported securely to prevent malicious or fraudulent attacks. The selected communication method therefore has to support some sort of security mechanism, ideally end-to-end encryption. While these security mechanisms will not be used in this thesis due to time constraints, selecting the right communication method is still critical for future development.

Java provides a package for Remote Method Invocation (RMI), which allows object in one Java VM to invoke methods on objects in another Java VM [22]. But since RMI is limited to Java and we might want to communicate with the bootware from a component written in another programming language, RMI doesn't seem like a good fit. For communication between programs written in different languages we could use the Common Object Request Broker Architecture (CORBA), a standard defined by the Object Management Group (OMG). It supports mappings for common programming languages, like Java, C++, Python, and others. Corba also supports asynchronous method invocation via callbacks [4] and transport layer encryption and other security features [9]. Another alternative are web services via Simple Object Access Protocol (SOAP) or Representational State Transfer (REST). Like CORBA, web services also support asynchronous invocation, as well as security mechanisms [31].

Since the whole SimTech SWfMS already uses SOAP based web service, it would make sense

to also use SOAP based web services as external communication mechanism for the bootware component. The technology and knowledge is already in place and introducing a second mechanism like CORBA would unnecessarily increase the complexity of the project, especially since CORBA doesn't offer any significant advantages over SOAP based web services. Figure 4.7 shows the addition of asynchronous web service call and return communication to the proposed architecture.

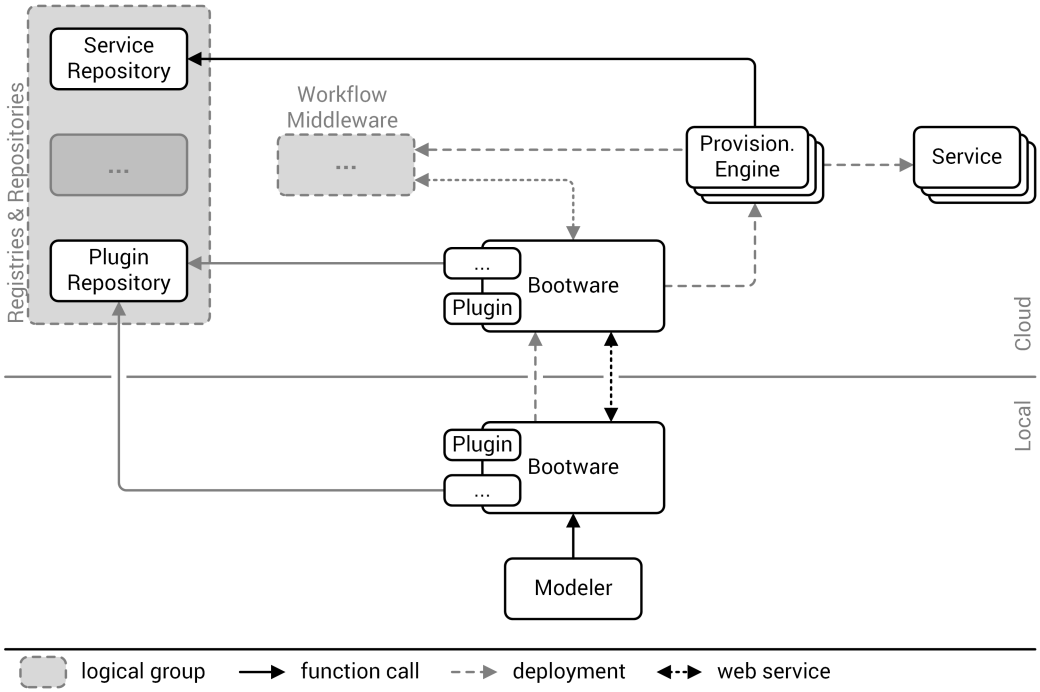

Figure 4.7: Simplified overview of the 2-tier architecture with asynchronous web service communication

With asynchronous communication, long running provisioning processes won't pose a problem. We do however still need information during those long running processes to give the user some feedback. This can't be accomplished by the simple web service request/response pattern. For this, a secondary communication mechanism which supports sending multiple feedback messages has to be used.

Since it is not necessary for the successful use of the bootware it would make sense to implement this secondary communication mechanism as a plugin. This would allow us to add arbitrary communication plugins to the bootware depending on future needs. These secondary communication channels could take any form, but a natural choice for publishing the intermediary state of the bootware would be a message queue system. In this case, the remote bootware component pushes messages to a message queue to which the local

bootware component (and other components if needs be) can subscribe to receive future messages. Figure 4.8 shows the proposed architecture with an additional message queue.

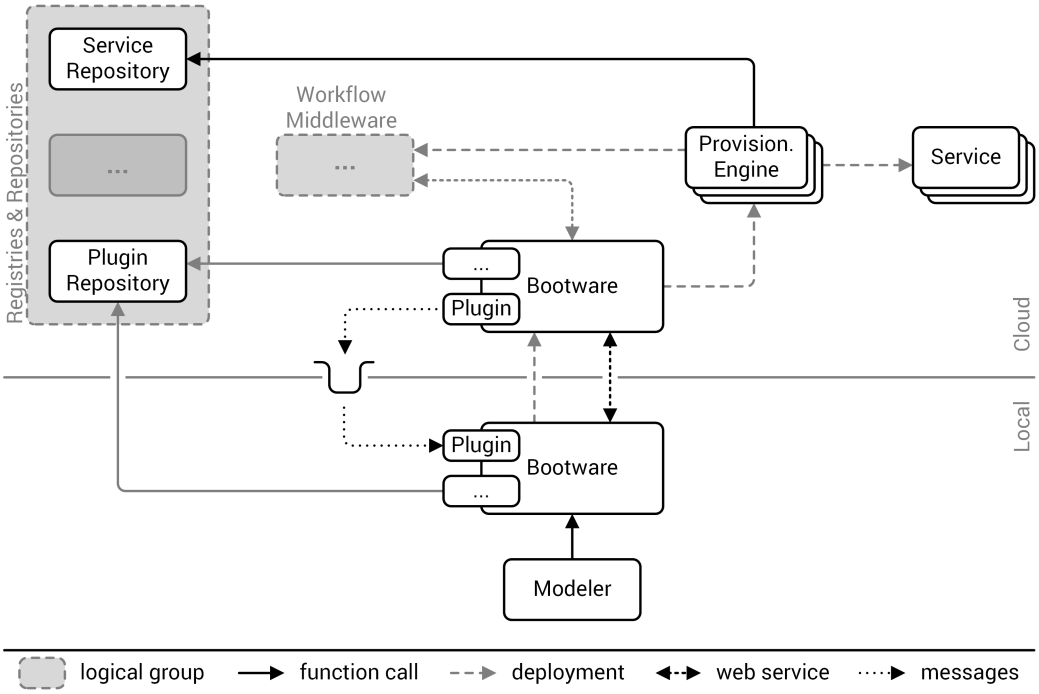

Figure 4.8: Simplified overview of the 2-tier architecture with asynchronous web service and a messaging queue communication

4.3.4 Internal Communication

We also have to consider internal communication between the bootware core and plugins, and possibly also in between plugins. Ideally, every plugin will be able to react to events from the bootware. These event could be triggered by the bootware core or by any plugin, but plugins should be completely independent from each other. Since a plugin doesn't know about other plugins, it can't listen for events at other plugins directly. The only known constant to a plugin is the bootware core. Therefore we need a communication mechanism which allows for loosely coupled communication between the bootware core and the plugins, where plugins can register their interest for certain events with the core and also publish their own events to the core for other plugins to consume. This essentially describes the publish-subscribe pattern [12].

PubSub

The publish-subscribe pattern (PubSub) is a messaging pattern that consists of three types of participant: A event bus (or message broker), publishers, and subscribers. The event bus sits at the center of the communication. He receives messages from publishers and distributes them to all subscribers that have voiced their interest in messages of a certain type by subscribing at the event bus [12].

Using this pattern in our bootware component, we would create an event bus at the bootware core and plugins, as well as other parts of the core, could subscribe at this event bus and also publish messages through this event bus.

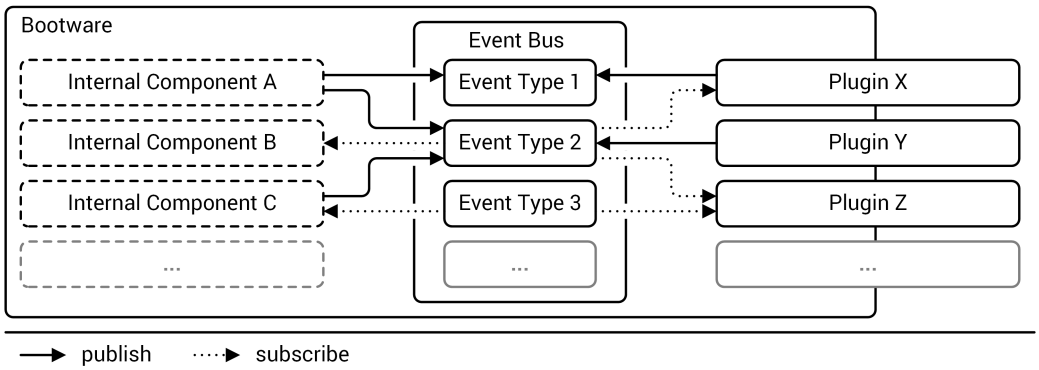

Figure 4.9: Bootware internal communication with PubSub pattern.

4.3.5 Plugin Types

We can already tell from the requirements that we must at least support two different plugin types, one for different cloud providers and one for different provisioning engines. The former are required because we may want to provision into different cloud environments. The latter are required because we might want to use different provisioning engines to do so.

The cloud provider plugins will be responsible for creating and removing resources in cloud environments and making them available for the user to configure and use. This could be barebone VMs (like AWS EC2 instances), or PaaS environments (like AWS Beanstalk). If we think about this for a second, we realize that we don't even have to constrain these plugins to cloud resources and can make them more abstract, as long as we can run the plugin and get an IP address to a computer resource that we can use. For example, we could also provide a

plugin that starts and stops VMs on our local machine, which could be useful for quick and inexpensive local testing. So a better name for these plugins would be *infrastructure plugins*.

The same line of thinking can be used on the provisioning engine plugins. All that we care about is that we can get some software running on any given infrastructure and that we get back an URL where we can find this software once it is up and running. A better name for these plugins would therefore be *payload plugins*.

Now that we have infrastructure plugins and payload plugins, we should be able to start any infrastructure we need and use payload plugins to install and run any software on them. But there is a step in between provisioning the infrastructure and installing the software that we are glancing over: We have to somehow connect to the infrastructure before we can install something. The connection functionality could be part of either the infrastructure plugins or the payload plugins, or it could be separated into independent connection plugins.

For the sake of efficiency and extensibility it would be best to use independent connection plugins. For example, if a user wanted to add a new connection type that should be used to install x applications in y environments, he could do so by writing one new connection plugin, instead of adding the functionality x-times to all payload plugins, or y-times to all infrastructure plugins. This would also reduce code duplication. Therefore, a third plugin type is necessary: The *connection plugins*.

In subsection 4.3.3 we also introduced the notion of secondary communication channels realized by plugins. Together with the use of events that we introduced in subsection 4.3.4, we can see a fourth plugin type emerging: the *event plugins*. These plugins are a bit less specific then the three other types. They allow users to add functionality that reacts to (or creates) events inside the bootware. With this fourth plugin type we have now covered all plugin types we will need. We will describe each plugin type in more detail, but before we do this, we will describe the common operations that all plugin types have to implement.

Operation	Input	Output	Description
init	-	-	Is called by the plugin manager when the plugin is loaded
shutdown	-	-	Is called by the plugin manager when the plugin is unloaded

Table 4.1: Common operations to be implemented by all plugin types

Table 4.1 shows the two common operations that all plugin types must implement. The init

operation is called by the plugin manager when he loads a plugin. This operation can be used by plugin authors to initialize the plugin, for example by creating internal objects that will be used by other plugin operations later on. The shutdown operation is called by the plugin manager when he unloads a plugin. It can be useful to clean up plugin resources before it is removed, for example by deleting temporary files or closing a connection.

Infrastructure Plugins

Infrastructure plugins are responsible for provisioning any infrastructure that the user wants to use during the bootware process. This could be VMs on a local machine, or IaaS or PaaS environments in the cloud. To be able to do this, an infrastructure plugin has to implement a range of functions using some Application Programming Interface (API) or SDK provided by the virtualization software or cloud provider.

Table 4.2 shows the operations a plugins of this type should implement. The deploy operation is responsible for deploying a resource and getting it to a state, where a connection to the resource can be established using a connection plugin. As input it takes login credentials for a cloud provider (if necessary), which are used to authenticate when calling the API or using the SDK. These credentials must be supplied by the user at some point before or during the bootware process. If the deployment was successful, it returns an instance object, which contains information about the created instance, such as its IP address and login information.

The undeploy operation removes a resource that was previously deployed using the deploy operation. In case of a local VM this could mean that it stops the running VM. In case of a cloud resource this could mean that it completely removes the resource so that no further costs are incurred. As input it takes an instance object created earlier by the deploy operation.

Operation	Input	Output	Description
deploy	Credentials	Instance	Given the proper credentials, deploys a connection ready instance of some resource and returns an instance object
undeploy	Instance	-	Completely removes a given instance

Table 4.2: Interfaces to be implemented by infrastructure plugins

Connection Plugins

Connection plugins are responsible for creating a communication channel to previously deployed resources that can later be used by payload plugins to execute their operations on the resource. The connection could be made by using SSH, RDC, VPN, Telnet, or other communication mechanisms supported by the resource. The connection plugins should be implemented in a generic fashion, so that they can be used for all kinds of resources.

Table 4.3 shows the operations that this type of plugin has to implement. The connect operation establishes a connection to a specific resource. The resource is specified by the instance object that is passed as input to the deploy operation. If the connection was established successfully, the operation returns a connection object that can be used later by payload plugins to execute operations through this connection. The disconnect operation closes a connection that was previously established by the connect operation. As input, it takes a connection object that was previously created by the connect operation.

Operation	Input	Output	Description
connect	Instance	Connection	Establishes a connection to the given instance
disconnect	Connection	-	Disconnects a given connection

Table 4.3: Interfaces to be implemented by connection plugins

Payload Plugins

Payload plugins are responsible for installing, uninstalling, starting, and stopping software on a resource. This process can include the uploading of files and the execution of remote orders on a resource.

Table 4.4 shows the operations that plugins of this type should implement. The deploy operation installs a payload on a resource. This can include uploading files from the local machine or downloading files from other machines. To execute this operation, a connection to the resource is necessary, which is supplied as input with the connection object. The undeploy operation removes a payload from a resource. In most cases this will not be necessary, since the resource will be destroyed in the undeploy phase and with it all the payload data (assuming it wasn't installed in persistent storage). This method is provided for completeness and for

special cases. The start operation starts a payload which previously was installed with the deploy operation. If the payload was started successfully, it returns the URL to the running payload. The stop operation stops the execution of a previously started payload. In most cases this will not be necessary, since the payload will be removed together with the resource in the undeploy phase. This method is provided for completeness and for special cases.

Operation	Input	Output	Description
deploy	Connection	-	Deploys the payload over the given connection
undeploy	Connection	-	Undeploys the payload over the given connection
start	Connection	URL	Starts the payload over the given connection
stop	Connection	-	Stops the payload over the given connection

Table 4.4: Interfaces to be implemented by payload plugins

Event Plugins

Unlike the other plugin types, the event plugins don't have any more operations to implement then the init and shutdown operations described in Table 4.1. Instead, they implement their specific functionality by defining one ore more event handlers which will react to specific events when these are published at the event bus. How exactly these event handlers look is dictated by the PubSub library that is used, which will be discussed later.

4.3.6 Execution Flow

Now that we know the different plugin types and their operation that are involved in the bootstrapping process, we can present a step-by-step description of the whole process. Figure 4.10 shows a graph that represents the major steps during the bootware execution as flow diagram. What follows is the description of the whole process.

The bootstrapping process is started by executing the bootware, which is represented by the start state in the top left corner of Figure 4.10. From there, the bootware first does some

initializations. If these fail for some reason, the cleanup code will be executed before the bootware execution is ended, as can be seen on the top right corner of Figure 4.10. In most cases however, the initialization should succeed. Then, the bootware will transition to the next state, where it tries to load the event plugins.

The event plugins are loaded once at the beginning of the bootware execution, since they will not change at a per request basis (like the other plugins). If loading these plugins fails, the bootware will try to unload them before continuing to the cleanup state. If the plugins are loaded successfully, the bootware transitions into the wait state, shown in the top center of Figure 4.10.

Once the bootware is in the wait state it is ready to receive requests from the outside. If a shutdown event is received in this state, the bootware will start the shutdown procedure by first unloading the event plugins and then running the cleanup code. This is the only normal way to shutdown the bootware. If a request is received in the wait state, the bootware transitions to the next state, where it reads the request context.

The request context contains all the information necessary to fulfill the request. This includes, among others, the type of the request(deploy or undeploy), the login credentials for a cloud provider, and the names of one of each of the infrastructure, connection, and payload plugin to be used during the bootstraping process. If the context can not be read, the bootware returns a response containing an error message before returning into the wait state. If the context is read successfully the bootware transitions to the load request plugins state.

In the load request plugins state the three plugins specified in the context are loaded. If this fails, the bootware tries to unload them before return an error response and returning to the wait state. If the plugins are loaded successfully, the bootware now starts either the deploy process or the undeploy process, shown at the bottom of Figure 4.10, depending on the type of the request.

If the request was a deploy request, the bootware will now execute the deploy, connect, and start operations of the infrastructure, connection, and payload plugins, one after another. If one of those operations fails the bootware transitions over to the corresponding undeploy operation and works its way backwards to undo all operations that where already executed. This process is the same as the undeploy process trigger by an undeploy request.

If the stop payload, deprovision payload, or disconnect states fail, the bootware just continues with the next undeploy state, since these operations are not considered critical. However, if the deprovision infrastructure state fails, the bootware transitions to a fatal error state, show at the right of Figure 4.10, since this step is considered critical. This state failing could mean that resources are still active in the cloud and human interaction is necessary to remove them

to stop further costs from incurring. The fatal error state is responsible for informing the user that he has to interfere.

The successful, as well as the unsuccessful execution of either the deploy or the undeploy process all finish in the unload request plugin state, where the plugins that where needed for this particular request are unloaded, before a suitable response is returned and the bootware returns to the wait state.

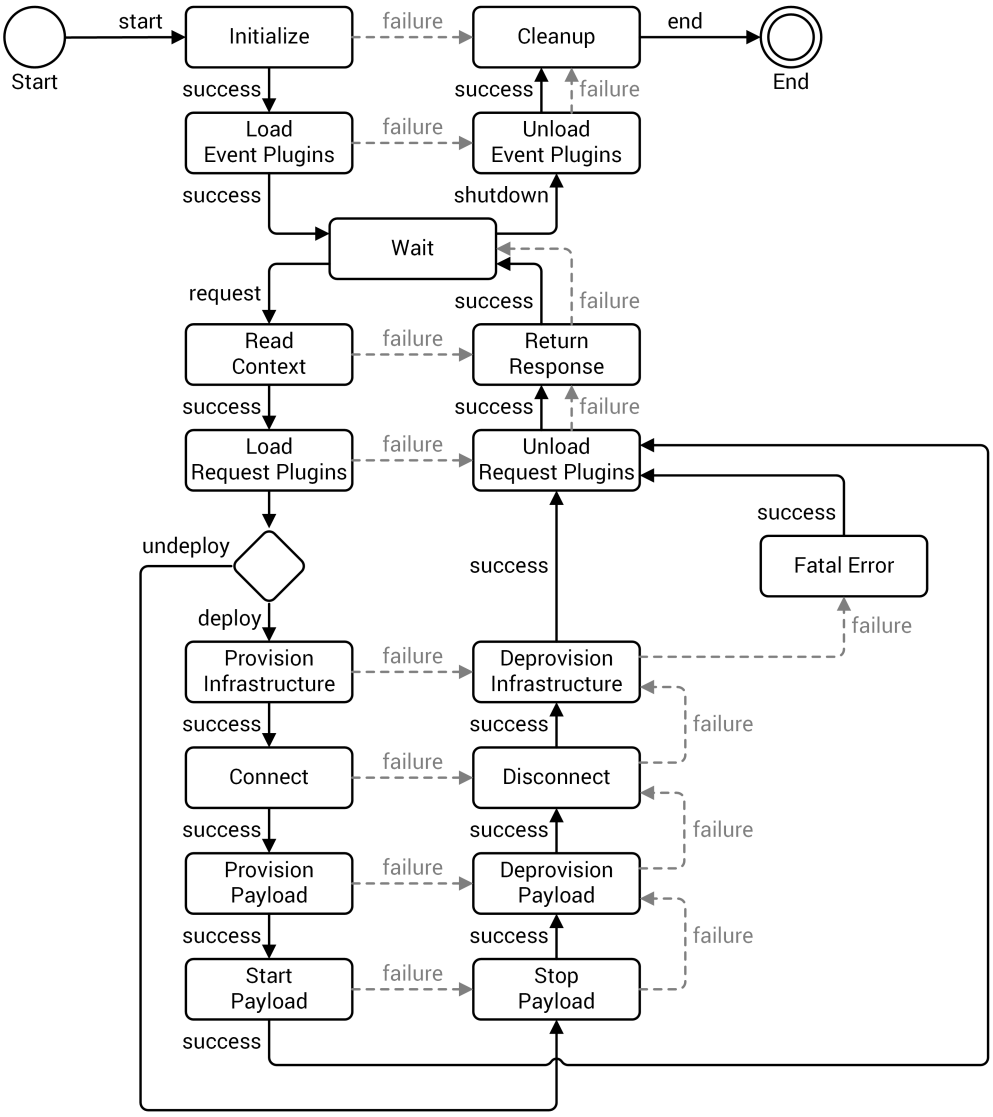

Figure 4.10: Execution flow during the bootstrapping process.

As Figure 4.10 and the description above show, this is quite a complicated process with many conditional transition. Using traditional programming methods like if/else blocks to implement this process would lead to a rather unwieldy and complicated construct with lots of nested if/else block. Therefore it could be advantageous to use other methods that are more fitting for this process. Since we already describes the process as a directed graph with states and transition, it would be ideal if we could take this whole graph and use it in the bootware. Fortunately this is possible by implementing the process using a finite state machine (FSM).

4.4 Final Bootware Architecture

Figure 4.11 shows the final architecture of the bootware component. At the bottom we can see six exemplary event plugins. These are loaded at the beginning of the bootware execution by the plugin manager, shown on the left of Figure 4.11. For demonstrations purposes, Figure 4.11 shows a wider range of possible event plugins. All these plugins provide some sort of input andd/or output mechanism for the bootware component. The web service plugins could provide a web service interface. This would allow other components to start the deploy and undeploy operations with web service calls. Another possibility for interaction would be a command-line interface (CLI) plugin that would make these operations accessible via a command-line interface. A event logger plugin could be used to write all bootware events to a log file. We can also imagine a event queue plugin that pushes all bootware events into some message queue so that they can be consumed by other components. Finally, an undeploy trigger plugin could trigger the undeployment of the bootware and all running payloads when it receives a message from the ODE event queue.

All event plugins work by implementing event handlers for certain events published at the event bus, or by publishing events to the event bus themselves. As we can see in the center of Figure 4.11, the event bus and the state machine form the core of the bootware. The event bus is responsible for distributing events between the various plugins and the state machine. The state machine implements the entire bootstrapping process, as described earlier in subsection 4.3.6. At certain points during the bootstrapping process, operations are delegated to the plugin manager to load plugins, and to the infrastructure, connection, and payload plugins, shown at the top of Figure 4.11.

The infrastructure, connection, and payload plugins implement the actual bootstrapping operations. At the top, Figure 4.11 shows an exemplary result of these bootstrapping operations. In this particular case, the infrastructure plugin started a VM, to which the connection plugin set up a communication channel. The payload plugin then used this communication channel

to provision the payload inside the VM. During the bootstrapping procedure, events are sent from those plugins back to the event bus to be delivered to the loaded event plugins.

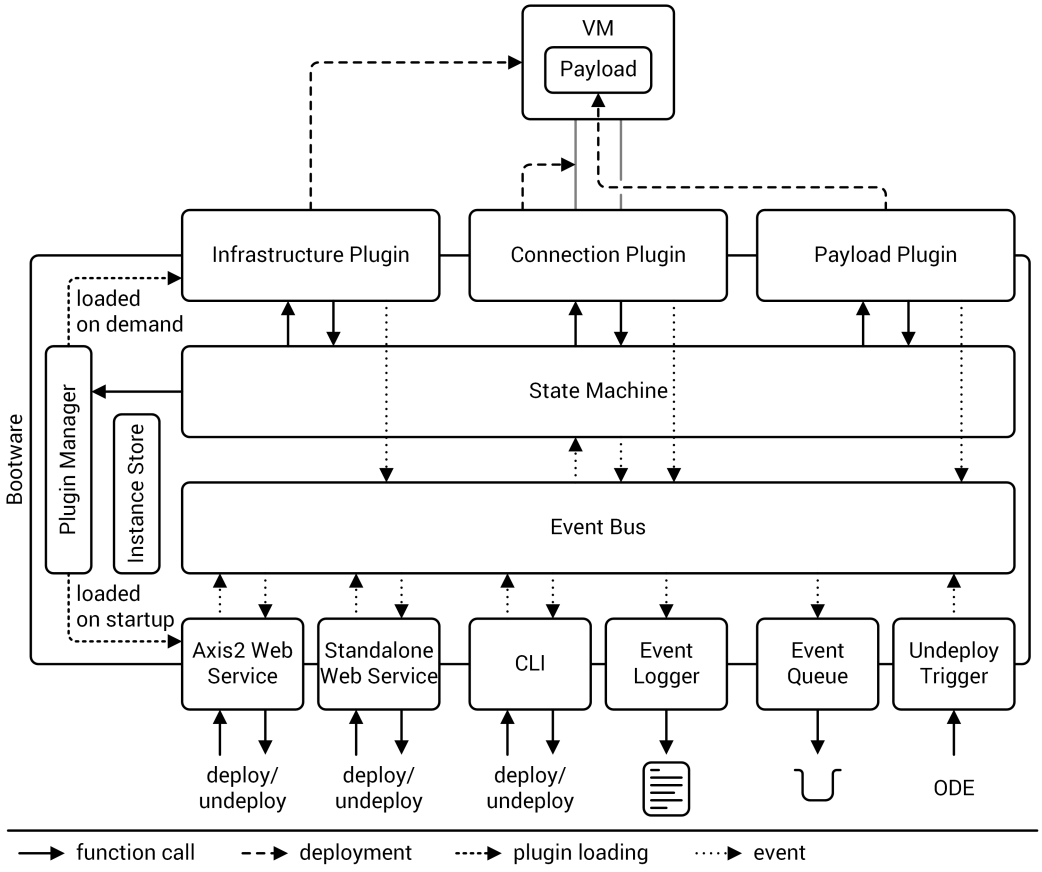

Figure 4.11: The final architecture of the bootware component.

5 Bootstrapping Process

This chapter describes the bootstrapping process in its entirety.

6 Implementation

In this chapter we present details on the implementation of the bootware component. First we select specific frameworks and libraries that allow us to implement the architecture we developed in section 4.3. Then we present detailed descriptions of the implementation of some parts of the bootware.

6.1 Selecting Frameworks and Libraries

Before we can begin with the actual implementation, we have to decide on which framework and libraries we will use to implement the requested functionality. In this section we present the frameworks and libraries we chose and the reasoning behind it. We begin with plugin frameworks, followed by web service and PubSub libraries.

6.1.1 Plugin Frameworks

All of the frameworks that we compare here offer the basic functionality that we need to extend the core bootloader component, i.e. the developer defines interfaces that then are implemented by one or more plugins. These plugins are compiled separately from the main component and are then packaged in *.jar* files for distribution. These packages are loaded during runtime and provide the implementation for the specific interface they implement. There are however some advanced functional differences and some non-functional differences that will be considered here.

Dynamic loading allows us to load and replace plugins during runtime, without completely restarting the application. This is an important feature, since it is possible that the bootware has to use many different plugins during its lifetime. For example, this would be the case when several services have to be provisioned, each with different provisioning engines. In this case, the bootware has to load the appropriate plugins for every provisioning engine to

be able to fulfill its task. We could of course just load every plugin at startup, switch between them internally when necessary, and never unload them. This could become a problem if the number of available plugins increases in the future. Then, loading all plugins could take some time and slow down the entire bootware process. In many cases, some or most of the plugins would never be used and loading them wouldn't be necessary at all. Therefore, it seems far more reasonable to load and unload plugins dynamically when needed.

Security is also a must have feature. Consider the following scenario: The bootware component is used by multiple separate users who can share plugins using a plugin repository. A malicious user could create a new plugin and upload it to the repository. This plugin can contain virtually any code. For example, it could erase all files or open a back door in the system when it is executed. Other users might trust the plugin author and try the plugin without checking its code first. Proper security feature might be able to prevent harm in such situations. Due to time restrictions, plugin security will not be discussed further in this thesis, but it's still vital to select the right framework now, so that security features can be implemented in the future.

We also consider some non-functional features that might influence the selection. There is already a plugin framework in use in the SimTech project, so it could be beneficial to choose the same framework, because the necessary knowledge and experience already exists. The requirements section also mentioned that using software based on open standards is encouraged. If possible, the complexity should be low while still providing all the necessary functional properties. Frameworks with high popularity and an active development community might be more mature or provide more documentation and support.

		Plugin Frameworks			
		SPI[1]	JSPF[2]	JPF[3]	OSGi[4]
functional	Dynamic Loading	✗	✗	✓	✓
	Security	✗	✗	✗	✓
non-functional	Used in SimTech	✗	✗	✗	✓
	Standard	(✓)	✗	✗	✓
	Complexity	low	low	medium	high
	Popularity	medium	low	medium	high
	Active Development	✓	✗	✗	✓

[1] http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html
[2] https://code.google.com/p/jspf
[3] http://jpf.sourceforge.net
[4] http://www.osgi.org

Table 6.1: Feature comparison of Java plugin frameworks

Table 6.1 shows a comparison of four Java plugin frameworks, the first of which is the Service Provider Interface (SPI). It is an extension mechanism integrated in Java which is a little more advanced then the manual extension mechanism described in subsection 4.3.2. It is also based on a set of interfaces and abstract classes that have to be implemented by an extension. In the case of SPI, these interfaces and abstract classes are called services and a specific implementation of such a service is called service provider. However, unlike in the manual approach, specific implementations are loaded from .jar files in specific directories or in the class path. These .jar files also include metadata to identify the different service providers [25].

SPI is easy to use, doesn't depend on any external libraries, is well documented, and mature, since it is used in the Java Runtime Environment (JRE). One could also say that it is somewhat standardized, since it is part of Java. But as we can see in Table 6.1 on the left, it neither supports dynamic loading, nor security features and is therefore not a good fit for our needs.

The next contender is the Java Simple Plugin Framework (JSPF)[2], an open-source plugin framework build for small to medium sized projects. Its main focus is simplicity and the author explicitly states that it is not intended to replace JPF or OSGi [6]. As a result it is lightweight and easy to use but does not support advanced features like dynamic loading or security.

Java Plugin Framework (JPF)[3] is another open-source plugin framework. Compared to JSPF it's a little more complex and popular. As we can see in Table 6.1, it also supports dynamic loading. However, the last version was released in 2007 and development seems to have stopped. This is not necessarily bad but might show that there will be no future development of this framework.

This leaves us with the final contender, which is Open Service Gateway initiative (OSGi)[4], a plugin framework standard developed by the OSGi Alliance. It provides a general-purpose Java framework that supports the deployment of extensible bundles [21]. The right column of Table 6.1 shows, that it supports dynamic loading, as well as security. OSGi is under active development, fairly popular, and has also been used in the SimTech project. Compared to the other alternatives it is pretty complex, but considering the other factors, it is the only real alternative. Therefore we will use OSGi to provide the extensibility required for the bootware.

Since OSGi by itself is only a standard, we still have to select an OSGI implementation. As with all other libraries and frameworks we use, we are looking for an open-source implementation, so we will ignore commercial OSGi implementations. There are three open-source OSGi

implementations to choose from: Apache Felix[5], Eclipse Equinox[6], and Knopflerfish[7]. All of them are under active development and implement the OSGi core framework specification, as well as the OSGi security specification (among others). We will be using Apache Felix, since it is already being used in the SimTech project. But it should be straight forward to change to another implementation in the future if necessary, since they all implement the same specification and should therefore in theory be completely interchangeable.

6.1.2 Web Service

6.1.3 PubSub Libraries

Many of the well know Messaging Middlewares offer support for PubSub, for example ActiveMQ[8], RabbitMQ[9], and ZeroMQ[10]. But, since we are looking for an internal communication mechanism only, all of these solutions are somewhat overpowered. We don't have to worry about network problems, so we don't need guaranteed delivery or message queuing capabilities. We also don't need persistence or transactional capabilities We don't have to handle millions of subscribers or events, so high scalability isn't a concern. We don't even necessarily need asynchronous communication. Instead, we need a lightweight in-memory solution. Therefor we will ignore the middleware heavyweights and look for smaller, more light weight PubSub libraries.

We have a few functional requirements that a library has to support for our use case. These can be seen on the left-hand side of Table 6.2. Weak references are an important feature, since we have a lot of plugins that will register as listeners to the event bus. These plugins can be removed at any time and weak references allow us to remove them without explicitly unregistering them from the event bus. Instead of crashing, the event bus will just ignore references to listeners that don't exist anymore. Even if we explicitly unregister all our plugins, weak references give us a safety net if we forget it at some point.

We also need support for an event hierarchy. This allows us to model our events in a very fine grained modular fashion and organize them into logical groups. It also allows listeners to react to a whole group of specific events or only to a small subset of such a group. A filtering feature gives us even more control over what events a listener will react to. It allows us to

[5] http://felix.apache.org
[6] http://eclipse.org/equinox
[7] http://www.knopflerfish.org
[8] http://activemq.apache.org
[9] http://www.rabbitmq.com
[10] http://zeromq.org

filter out specific events, for example by their content, to handle them differently, or to ignore them.

We also want event handlers to be invoked synchronously. If an event is published, all event handlers for this event should be executed one after another until they finished execution. Only then should the program continue execution. But asynchronous invocation might still be useful in some cases, so we also add it here.

We will also consider some non-functional qualities like popularity, maturity and documentation that can give an indication of the usefulness of a library.

		PubSub Libraries				
		EventBus[11]	Guava Event Bus[12]	Simple Java Event Bus[13]	MBassador[14]	Mycila PubSub[15]
functional	Weak References	✓	✗	✓	✓	✓
	Event Hierarchy	?	✓	?	✓	✓
	Filtering	✓	✗	✓	✓	✗
	Sync. Invocation	✓	✓	✓	✓	✓
	Async. Invocation	✓	✓	✓	✓	✓
non-functional	Popularity	high	medium	low	medium	low
	Maturity	high	medium	medium	medium	medium
	Documentation	high	low	low	medium	medium

Table 6.2: Feature comparison of Java PubSub libraries

The first library we look at is EventBus. As can be seen in Table 6.2 on the left, EventBus supports most of the functionality we need. From the libraries presented here it is also the oldest one, so it is mature, fairly popular and well documented. But its age is also is weakness.

Guava Event Bus is a fairly simple PubSub library. It is part of the Google core libraries for

[11] http://eventbus.org/
[12] https://code.google.com/p/guava-libraries/wiki/EventBusExplained
[13] https://code.google.com/p/simpleeventbus/
[14] https://github.com/bennidi/mbassador
[15] https://github.com/mycila/pubsub

Java 1.6+ and is therefore fairly popular, but it lacks in documentation. It also doesn't support weak references and filtering, which doesn't make it a good fit for our use case.

Simple Java Event Bus is a simpler alternative to EventBus. It lacks some of the advanced features of EventBus but is also simpler to use. Compared to the other libraries it is not that popular and lacks in documentation.

MBassador is a light-weight and performance minded PubSub Library. As we can see in Table 6.2, it supports all functional features that we need (and some more). It is also relatively mature, has good enough documentation and is somewhat popular.

Finally, we have Mycila PubSub, a modern replacement for EventBus. It supports all the functional features we need, except filtering. Its documentation is good enough, but since it is relatively new, it's not very popular (yet) and may lack in maturity.

7 Summary and Conclusion

Bibliography

[1] *About AWS*. Amazon Web Services, Inc. URL: http://aws.amazon.com/about-aws/.

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

[3] *Amazon EC2 Instances*. Amazon Web Services, Inc. URL: http://aws.amazon.com/ec2/instance-types/.

[4] *Asynchronous Method Invocation for CORBA Component Model, Version 1.0*. Tech. rep. Object Management Group, Inc., Apr. 2013. URL: http://www.omg.org/spec/AMI4CCM/1.0/PDF/.

[5] *AWS Elastic Beanstalk (Beta)*. Amazon Web Services, Inc. URL: http://aws.amazon.com/elasticbeanstalk/.

[6] Ralf Biedert. *Java Simple Plugin Framework F.A.Q.* URL: https://code.google.com/p/jspf/wiki/FAQ.

[7] Tobias Binz et al. "OpenTOSCA -- A Runtime for TOSCA-based Cloud Applications". In: International Conference on Service-Oriented Computing. LNCS. Springer, 2013. URL: http://www.iaas.uni-stuttgart.de/RUS-data/INPROC-2013-45%20-%20OpenTOSCA_A_Runtime_for_TOSCA-based_Cloud_Applications.pdf.

[8] Werner Buchholz. "The System Design of the IBM Type 701 Computer". In: Institute of Radio Engineers (IRE). Vol. 41. 10. 1953, pp. 1262–1275. URL: http://ftp.cs.duke.edu/~xwy/publications/cloudcmp-imc10.pdf.

[9] *Common Object Request Broker Architecture (CORBA) Specification, Version 3.3 - Part 2: CORBA Interoperability*. Tech. rep. Object Management Group, Inc., Nov. 2012. URL: http://www.omg.org/spec/CORBA/3.3/Interoperability/PDF/.

[10] *Cyber-Infrastructures and beyond (PN 6)*. SRC Simulation Technology. URL: http://www.simtech.uni-stuttgart.de/forschung/pn/PN6/index.en.html.

[11] *Definition of Provisioning*. ATIS Telecom Glossary. URL: http://www.atis.org/glossary/definition.aspx?id=2474.

[12] Patrick Th. Eugster et al. "The many faces of publish/subscribe". In: *ACM Computing Surveys (CSUR)* 35.2 (June 2003), pp. 114–131. URL: core.kmi.open.ac.uk/download/pdf/12642066.pdf.

[13] *Excellence Initiative at a Glance*. Tech. rep. German Research Foundation, Nov. 2013. URL: http://www.dfg.de/download/pdf/dfg_im_profil/geschaeftsstelle/publikationen/exin_broschuere_en.pdf.

[14] *Flexibility of Simulation Workflows*. SRC Simulation Technology. URL: http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/project_flexibility.php.

[15] *Getting Started with the AWS Management Console*. Amazon Web Services, Inc. URL: http://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/getting-started.html.

[16] James Joyce. *Ulysses*. Project Gutenberg. URL: http://www.gutenberg.org/files/4300/4300-h/4300-h.htm.

[17] Johannes Kirschnick et al. "Toward an Architecture for the Automated Provisioning of Cloud Services". In: *Communications Magazine* 48.12 (2010), pp. 124–131. URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5673082.

[18] Johannes Kirschnickand et al. "Toward an architecture for the automated provisioning of cloud services". In: *Communications Magazine, IEEE* 48.12 (Dec. 2010), pp. 124–131. URL: http://jmalcaraz.com/wp-content/uploads/papers/AlcarazCalero-2010-CommMag-Preprint.pdf.

[19] Ang Li et al. "CloudCmp: Comparing Public Cloud Providers". In: 10th ACM SIGCOMM conference on Internet measurement. 2010, pp. 1–14. URL: http://ftp.cs.duke.edu/~xwy/publications/cloudcmp-imc10.pdf.

[20] Peter Mell and Timothy Grance. *The NIST Definition of Cloud Computing*. Tech. rep. National Institute of Standards and Technology, Sept. 2011. URL: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[21] *OSGi Service Platform core Specification*. Tech. rep. OSGi Alliance, Apr. 2011. URL: http://www.osgi.org/download/r4v43/osgi.core-4.3.0.pdf.

[22] *Package java.rmi*. Oracle Corporation. URL: http://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html#package_description.

[23] Javier Rojas. *The art of the bootstrap*. URL: http://venturebeat.com/2008/11/20/the-art-of-the-bootstrap/.

[24] Valeri Schneider. "Dynamische Provisionierung von Web Services für Simulationsworkflows". Diploma. University Stuttgart, 2013. URL: ftp://ftp.informatik.uni-stuttgart.de/pub/library/medoc.ustuttgart_fi/DIP-3473/DIP-3473.pdf.

[25] *ServiceLoader*. Oracle Corporation. URL: http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html.

[26] *SLA (Service-level Agreement)*. Gartner, Inc. URL: http://www.gartner.com/it-glossary/sla-service-level-agreement/.

[27] *Tools for Amazon Web Services*. Amazon Web Services, Inc. URL: https://aws.amazon.com/tools/.

[28] *Topology and Orchestration Specification for Cloud Apllications*. Committee Specification 01. OASIS, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf.

[29] Luis M. Vaquero et al. "A Break in the Clouds: Towards a Cloud Definition". In: *ACM SIGCOMM Computer Communication Review* 39.1 (Jan. 2009), pp. 50–55. URL: http://www.sigcomm.org/sites/default/files/ccr/papers/2009/January/1496091-1496100.pdf.

[30] Karolina Vukojevic-Haupt, Dimka Karastoyanova, and Frank Leymann. "On-demand Provisioning of Infrastructure, Middleware and Services for Simulation Workflows". In: Proceedings of the 6th IEEE International Conference on Service Oriented Computing & Applications (SOCA 2013). 2013. URL: ???.

[31] *Web Services Security: SOAP Message Security 1.1*. Tech. rep. OASIS, Feb. 2006. URL: https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[32] Benjamin Zimmer. *figurative 'bootstraps' (1834)*. The American Dialect Society Mailing List. URL: http://listserv.linguistlist.org/cgi-bin/wa?A2=ind0508B&L=ADS-L&D=0&P=14972.

All links were last visited on May 16, 2014.

Declaration of Authorship

I hereby certify that the diploma thesis entitled

Bootstrapping Provisioning Engines for On-demand Provisioning in Cloud Environments

is entirely my own work except where otherwise indicated. Passages and ideas from other sources have been clearly indicated. To date, neither this diploma thesis nor essential parts thereof were subject of an examination procedure. Until now, I don't have published this diploma thesis or parts thereof. The electronic copy is identical to the submitted copy.

Stuttgart, May 16, 2014, ...

(Lukas Reinfurt)