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: 08.01.2014
beendet am: ??.??.2014

CR-Klassifikation: C.2.4, D.2.2, D.2.13, H.4.1, I.6.7

Abstract

The assumption that services should run continuously is no longer reasonable in science oriented environments, where dynamic working approaches lead to fluctuating service utilization. For on-demand provisioning of services in cloud environments, suitable provisioning engines have to be set up first. This diploma thesis presents the design for a 2-tiered bootware component that deploys provisioning engines into remote environments that can then be used to provision services on-demand. The bootware can be called by other components via a web service interface and supports multiple provisioning engines and cloud environment via plugins. The integration of the bootware into the SimTech SWfMS with an Eclipse plugin is also described, the bootware however is a generic component that can be used in other systems.

Table of Contents

List of Abbreviations

AMI	Amazon Machine Image, page 18
API	Application Programming Interface, page 46
APT	Advanced Packaging Tool, page 27
AWS	Amazon Web Services, page 14
BPEL	Business Process Execution Language, page 10
BPMN	Business Process Modeling Notation, page 10
CLI	command-line interface, page 62
CORBA	Common Object Request Broker Architecture, page 38
CSAR	Cloud Service Archive, page 19
EC2	Elastic Compute Cloud, page 14
ESB	Enterprise Service Bus, page 11
FSM	Finite State Machine, page 61
IaaS	Infrastructure as a Service, page 13
JPF	Java Plugin Framework, page 76
JRE	Java Runtime Environment, page 75
JSPF	Java Simple Plugin Framework, page 76
NIST	National Institute of Standards and Technology, page 13
OASIS	Organization for the Advancement of Structured Information Standards, page 17
ODE-PGF	ODE Pluggable Framework, page 10
OMG	Object Management Group, page 38

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 small, independent bootstrapping system that can deploy provisioning engines on-demand in cloud environments in an automatic fashion. It should be able to provision various provisioning engines in different 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[1] as cloud environment and OpenTOSCA[2] as provisioning engine should be implemented.

[1] http://aws.amazon.com/
[2] http://www.iaas.uni-stuttgart.de/OpenTOSCA/indexE.php

1.2 Structure of this Document

chapter 2 introduces some fundamental topics. It describes the SimTech project and the SimTech SWfMS, which initiated this diploma thesis. It also gives a short introduction into bootstrapping. Then, cloud computing in general is explained, followed by a closer look at the Amazon Web Services, which are used in this diploma thesis. Finally, provisioning in general, as well as with TOSCA and OpenTOSCA is described.

chapter 3 presents previous work on the subject diploma thesis. First, the paper that builds the foundation of this diploma thesis is discussed, followed by a previous diploma thesis that extended parts of this paper. chapter 4 also presents some related work, for example another architecture for automatic provisioning of cloud services.

chapter 5 lists the requirements that were given for this diploma thesis. It also introduces some additional constrains that were introduced during the writing of this diploma thesis.

chapter 6 presents the design of the bootware. First, the component division is discussed, followed by the integration into existing modeler applications. Then, an external communication mechanism is selected. The extensibility mechanism is described next, followed by the different kinds of plugins. Internal communication, the context object, the web service interface, and the instance store are also discussed. The execution flow and the use of finite state machines is described, before the final bootware architecture is presented.

chapter 7 describes the whole bootstrapping process step by step.

chapter 8 presents details on the implementation of the bootware. The reasons for the selection of the plugin framework and the PubSub and state machine libraries are discussed. Next, the integration into the SimTech Modeler using an Eclipse plugin is described. Then, the implementation of the context object and the web service interface are presented.

chapter 10 summarizes the previous chapters. It describes the work that has been done and also the work that still needs to be done, areas that could be improved, and future work that might be useful. Finally, a conclusion is presented.

2 Fundamentals

This chapter starts with a short overview of the SimTech project[1], 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[2], 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. 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 [14].

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 [11].

[1] `http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/`
[2] `http://www.dfg.de/en/research_funding/programmes/excellence_initiative/index.html`

2.1.1 SimTech SWfMS

As part of this project, the SimTech Scientific Workflow Management System (SWfMS) was developed. It is a system that enables scientists to easily create, manage and execute simulation workflows [16].

A workflow is a "computerised facilitation or automation of a business process, in whole or part" [17]. It is defined by a process model that describes tasks and the order in which these tasks should be executed [17]. Until recently, workflows have been mostly used to describe business processes, which lead to the development of business-centered standards like the Business Process Execution Language (BPEL)[3] or the Business Process Modeling Notation (BPMN)[4]. However, the SimTech SWfMS is used for simulation workflows, a subcategory of scientific workflows, which came more and more into use in recent years. These workflows have requirements that cannot be fulfilled by BPEL or BPMN. The SimTech SWfMS therefore introduces extensions to the BPEL language that add functionality to support these requirements, such as passing data by reference to support larger amounts of data often found in science [also see 35], or shared context between workflows [23]. Other extension introduces by the SimTech SWfMS to support simulation workflows include a service bus that supports late binding, rebinding, and legacy simulation software, as well as the Simulation Data Management System (SIMPL) that provides unified access methods for arbitrary external data [26]. Additionally extension where also made in the areas of flexibility, to support a "model as you go" approach, and in human user involvement, to support human tasks for decision making, data manipulation, or workflow repair [16, 18].

The SimTech SWfMS consists of the SimTech Workflow Modeling & Monitoring Tool (SimTech Modeler) and the workflow middleware. The SimTech Modeler is based on Eclipse JEE[5] and extends its functionality with various plugins. Figure 2.1 shows the SimTech Modeler user interface. It allows the user to create simulation workflows using a graph as visual representation, where vertices represent simulation tasks and edges describe the progression between those tasks.

Once the user is done modeling the simulation workflow, he clicks on a button to execute the workflow on the workflow middleware. The middleware consists of various components that are executed by an application server, in this case Apache Tomcat[6]. The workflow is send to the workflow engine (ODE Pluggable Framework (ODE-PGF)[7]), which executes the workflow

[3] http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html
[4] http://www.bpmn.org/
[5] http://www.eclipse.org/ide/
[6] http://tomcat.apache.org/
[7] http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/

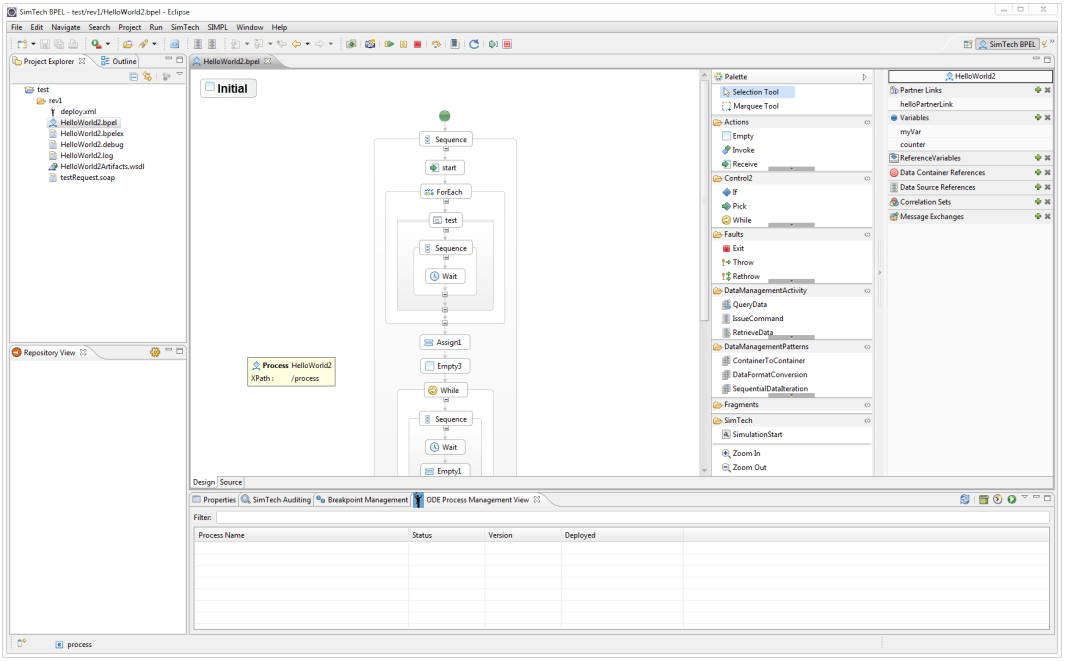

Figure 2.1: The SimTech Modeler user interface.

step by step. If a step involves the execution of a service, the Enterprise Service Bus (ESB) is called, who resolves the services and passes along the request and the response.

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 [36]. 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” [19]. 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].

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 [25].

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 programs 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 resources, such as servers, databases, communication services, 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."[8] [31]

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." [22]

Cloud services can be categorized into different cloud service models, according to what exactly each service encompasses. Figure 2.2 shows the three most common service models. 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 OS, nor any middleware, application, or data.

Today, there are many different cloud providers offering a huge selection of services. The range of providers spans from large corporations like Amazon[9], Google[10], Microsoft[11], and

[8] 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." [28]

[9] `http://aws.amazon.com`

[10] `https://cloud.google.com`

[11] `http://azure.microsoft.com`

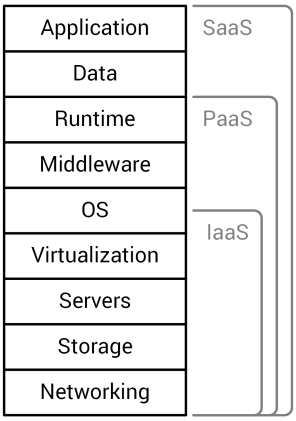

Figure 2.2: Cloud service models [based on 15].

IBM[12] to small, focused providers like Heroku[13] or Jelastic[14] and even solutions to build own clouds, like OpenStack[15]. 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)[9]. 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.

The most relevant cloud offering for this thesis is Elastic Compute Cloud (EC2)[16], 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. In addition to general purpose instances (M3), Amazon offers a wide selection of specialized instances, which are optimized for a specific purpose[17]. 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).

[12] http://www.ibm.com/cloud-computing
[13] https://www.heroku.com
[14] http://jelastic.com
[15] https://www.openstack.org
[16] http://aws.amazon.com/ec2
[17] http://aws.amazon.com/ec2/instance-types/

Also of interest to this thesis is Elastic Beanstalk[18], Amazons PaaS offering. Customers can upload an application and Elastic Beanstalk takes care of deployment and scaling. 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[19], 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. Additionally, Amazon provides a command line interface, tools for Eclipse and Visual Studio, and Software Development Kits (SDKs) for several programming languages, including Java, .Net, Python, Ruby, and the Android and iOS platforms[20]. In this thesis, we will use the AWS Java SDK[21] 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 solutions available today. It focuses in particular on TOSCA and OpenTOSCA, 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. This core functionality can be stated more precisely 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" [12]. 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

[18] http://aws.amazon.com/elasticbeanstalk
[19] http://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/getting-started.html
[20] https://aws.amazon.com/tools/
[21] https://aws.amazon.com/sdkforjava

cloud environment, as well as the installation, configuration, monitoring, running and stopping of software on these VMs [20].

There 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 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[22], 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[23], Chef[24], Puppet[25], 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 CloudFormation 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

[22] http://aws.amazon.com/cloudformation
[23] http://www.ansible.com
[24] http://www.getchef.com/chef
[25] http://puppetlabs.com/

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 CloudFormation.

2.4.2 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard created by the Organization for the Advancement of Structured Information Standards (OASIS)[26]. 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 [30].

TOSCA defines an XML syntax, which describes services and their relations in a so called service template. All elements needed to define such a service template are provided in the TOSCA definitions document. Figure 2.3 shows such a definitions document. Aside from the actual service template, shown on the left, it also contains a number type definitions and some templates based on those definitions. These definitions and templates can also be imported from a separate definitions document.

The service template consists of two parts: A topology template and plans. Topology templates, as seen in the center of Figure 2.3, model the structure of a service and the middleware and infrastructure supporting it 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 right of Figure 2.3. 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.

Plans, shown on the left of Figure 2.3, 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.

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

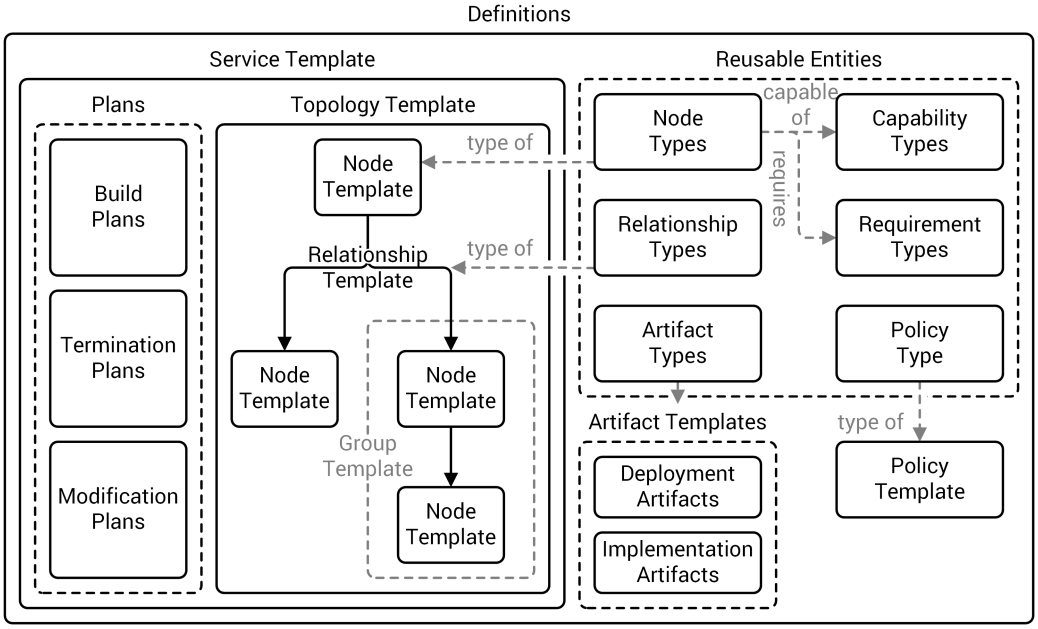

Figure 2.3: TOSCA definitions structure [based on 30].

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 a process description language like BPMN or BPEL.

The bottom right of Figure 2.3 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 implement 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.

The bottom right of Figure 2.3 also shows policy templates that refer to specific policy types. A policy template can define concrete values for a policy specified in a policy type. A node template can then reference a policy template to declare that it supports some non-functional properties or a certain kind of quality-of-service. An example would be an node type for an application server that expresses that it supports high availability by referencing a matching

policy template.

One or more TOSCA definitions are packaged, together with some meta data and possibly other files, 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 [2]. Figure 2.4 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 [2].

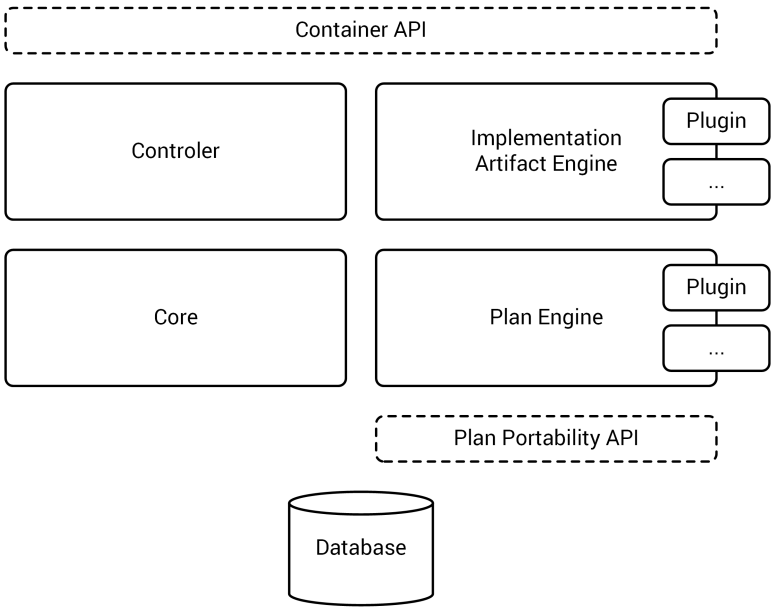

Figure 2.4: OpenTOSCA architecture [based on 2].

3 Previous Work

This chapter summarizes previous work on the subject of 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.

3.1 On-demand Provisioning for Simulation Workflows

Vukojevic-Haupt et al. identify requirements that need to be addressed to make the current approach used for scientific workflows more suitable for scientific simulation work [32]. The current approach used in the SimTech SWfMS is based on the assumption of service-oriented computing that services are always running. This can make sense for business applications with a large, steady stream of transactions. Scientific workflows however are executed infrequently, but when they are executed they need a lot of resources. Keeping all those resources running all the time is not efficient, so a more flexible way to allocate and use those resources is needed. The following requirements where identified to be able to improve this situation: 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 [32].

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

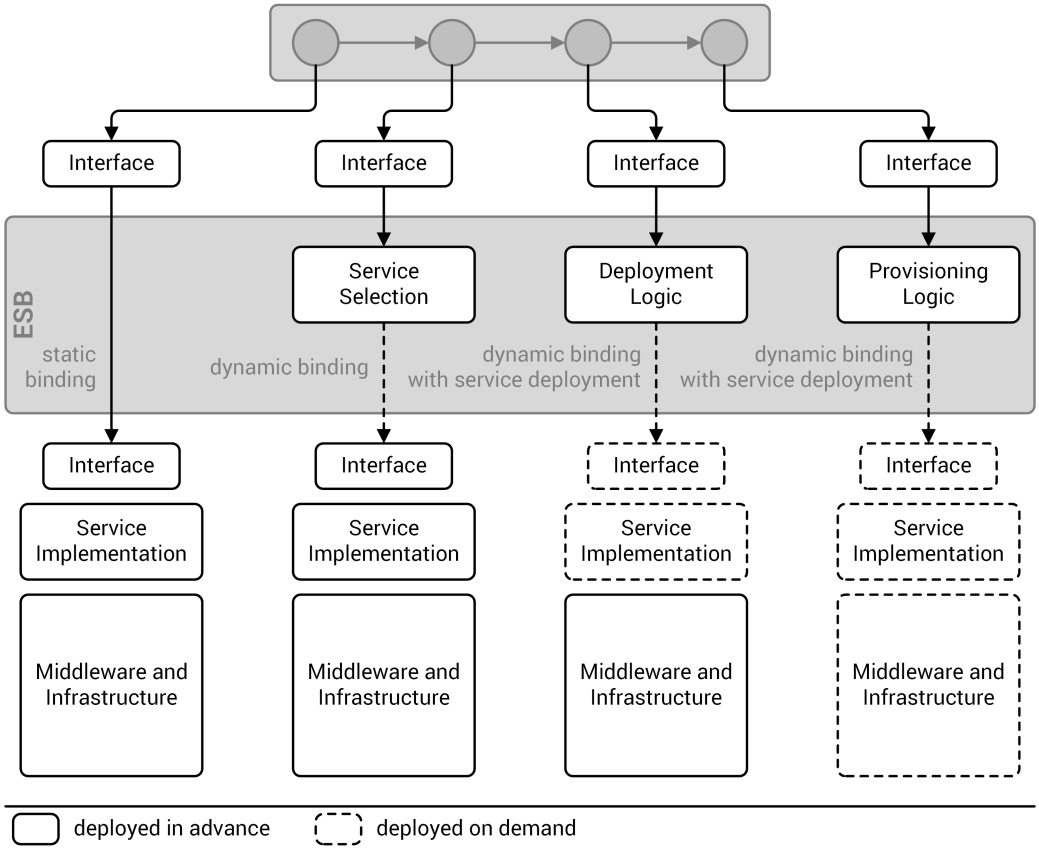

Figure 3.1: Simplified overview of service binding strategies [based on 32].

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 [32].

Their approach for dynamic provisioning and deprovisioning of workflow middleware and simulation services is separated into six steps, as can be seen in Figure 3.2. The first step is to model and start the execution of a simulation workflow using a local modeling tool like the SimTech Modeler. In the second step, the middleware for executing the workflow, e.g. the SimTech SWfMS, and its underlying infrastructure are provisioned to a cloud environment. 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, a task might invoke some external service that is not yet available. The ESB determines this by checking the service registry, which stores information about available services. 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, during which the provisioning engine retrieves the artifacts needed

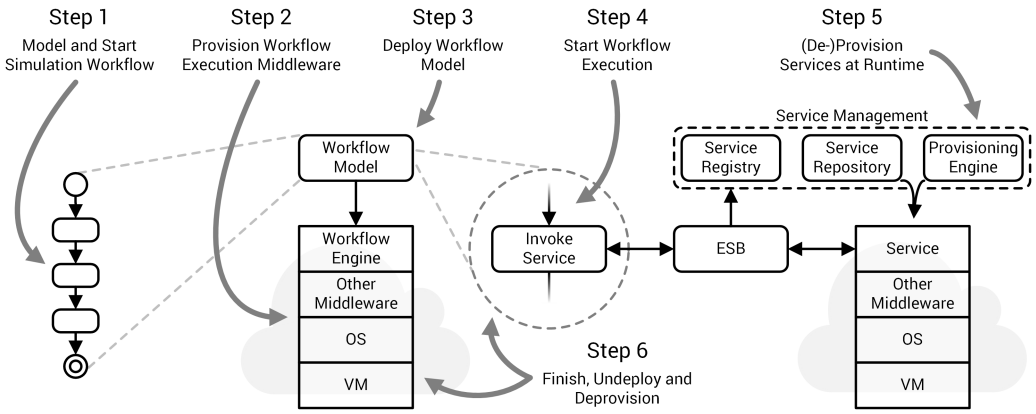

Figure 3.2: Steps during the on-demand provisioning of workflow execution middleware and simulation services [based on 32].

to provision the requested service from the service repository. The ESB then routes service calls and responses between the invoking workflow activity and the service. The service is also deprovisioned by the provisioning engine 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 [32].

The architecture they present, shown in Figure 3.3, can be separated into a local part at bottom and a cloud part at the top, as well as different phases. The bars at the bottom of Figure 3.3 show, which components are active during which phase [32]. Figure 3.3 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 starts the execution of the the workflow, the local bootware component kicks of the on demand provisioning process and therefore the second phase, called middleware runtime phase. In this phase, the bootware 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.

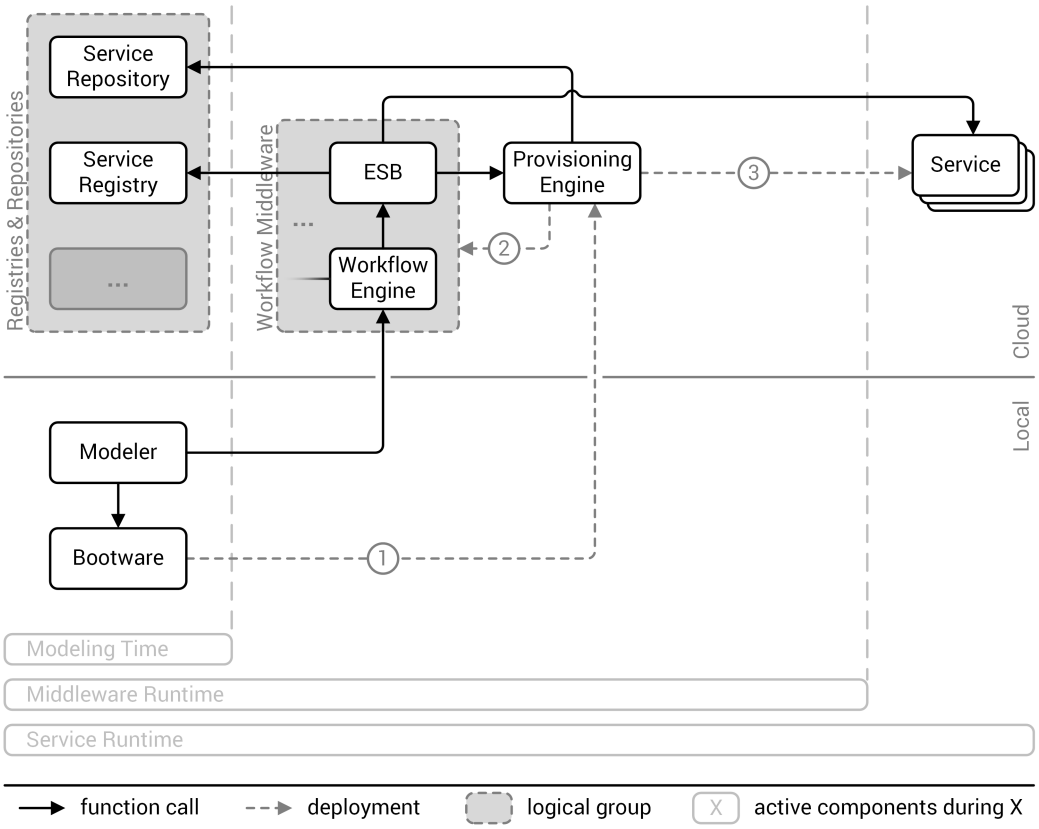

Figure 3.3: Proposed architecture [based on 32].

3.2 Dynamic Provisioning of Web Services for Simulation Workflows

Schneider found some problems with the architecture proposed in section 3.1 [27]. 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 [27].

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 [27].

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 [27].

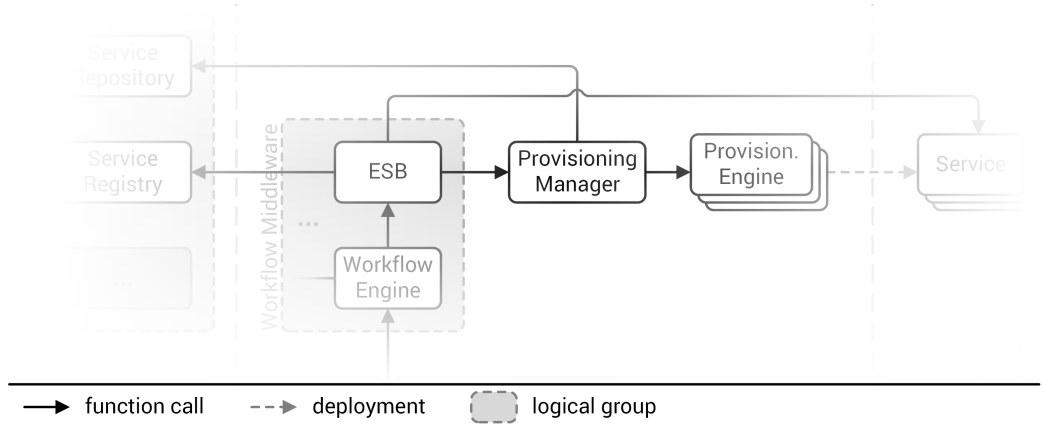

Figure 3.4: Extended architecture with added provisioning manager [based on 33]

Schneider further refines the previously shown middleware architecture by adding a provisioning manager as intermediary between the ESB and the provisioning engines [27]. Figure 3.4 shows an excerpt of the extended architecture with the additional provisioning manager at the center. 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 [27].

4 Related Work

This chapter summarizes related work of other authors that is of interest to this thesis. We take a look at a paper that presents work similar to this diploma thesis.

Another diploma thesis that is worked on in parallel to this diploma thesis is designing the provisioning manager that was described by Schneider. more ...titel, ref, summary

4.1 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 [20].

Figure 4.1 shows a simplified overview of the service orchestrator architecture. Users can submit service models via a service API, shown at the bottom of Figure 4.1. 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 and to trigger reconfiguration and topology changes of existing services [20].

The service orchestrator is build in two layers. The orchestration layer, shown at the bottom of Figure 4.1, 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 4.1, 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 [20].

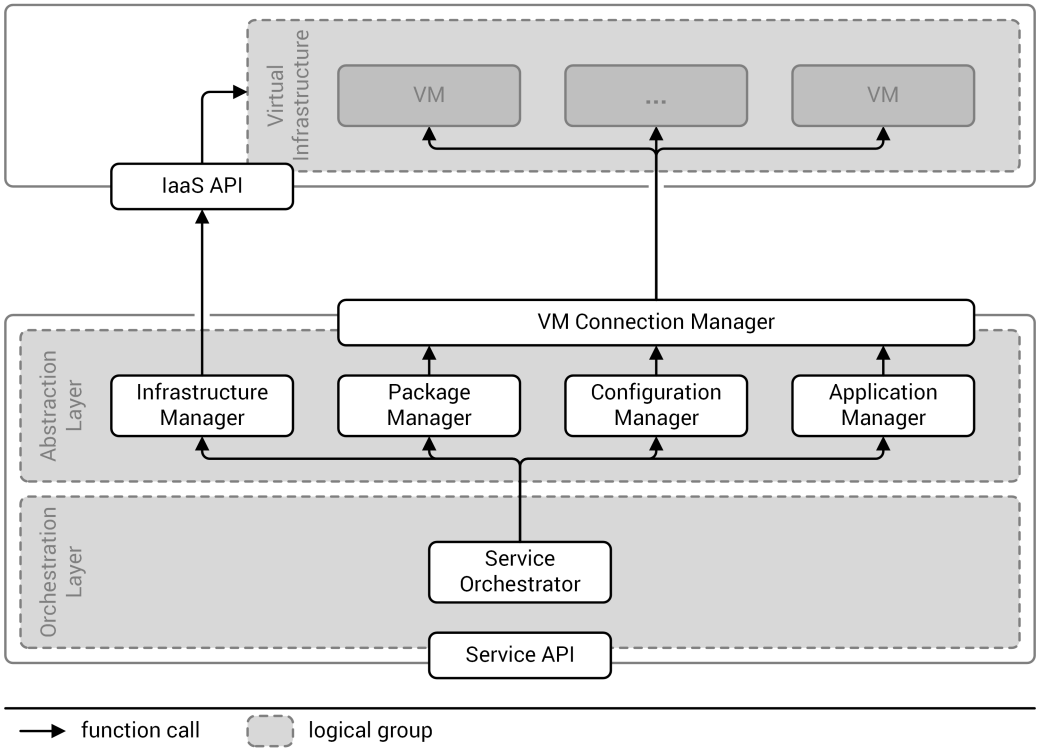

Figure 4.1: Simplified architecture for automatic provisioning of cloud services [based on 20]

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 4.1. 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 [20].

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 [20].

5 Requirements and Constraints

In this chapter we will talk about the requirements and constraints that shape the development of the bootware. We begin with the requirements, which were explicitly given at the beginning of this thesis. Then, we describe additional constraints which we added to limit the scope of the work.

5.1 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 executing 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. In this case, lightweight means that the bootware should be small, independent program that does not require a huge supporting infrastructure to be executed.

5.2 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 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. v

6 Design

In this chapter we will develop the design of the bootware. This design is held intentionally abstract. Specific implementation details will be selected and described in section 8.3.

6.1 Component Division

As described in section 3.1 on page 21, 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 24. 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.

6.1.1 Single Local Component

First we consider the simplest case: A single local component as shown in Figure 6.1. 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.

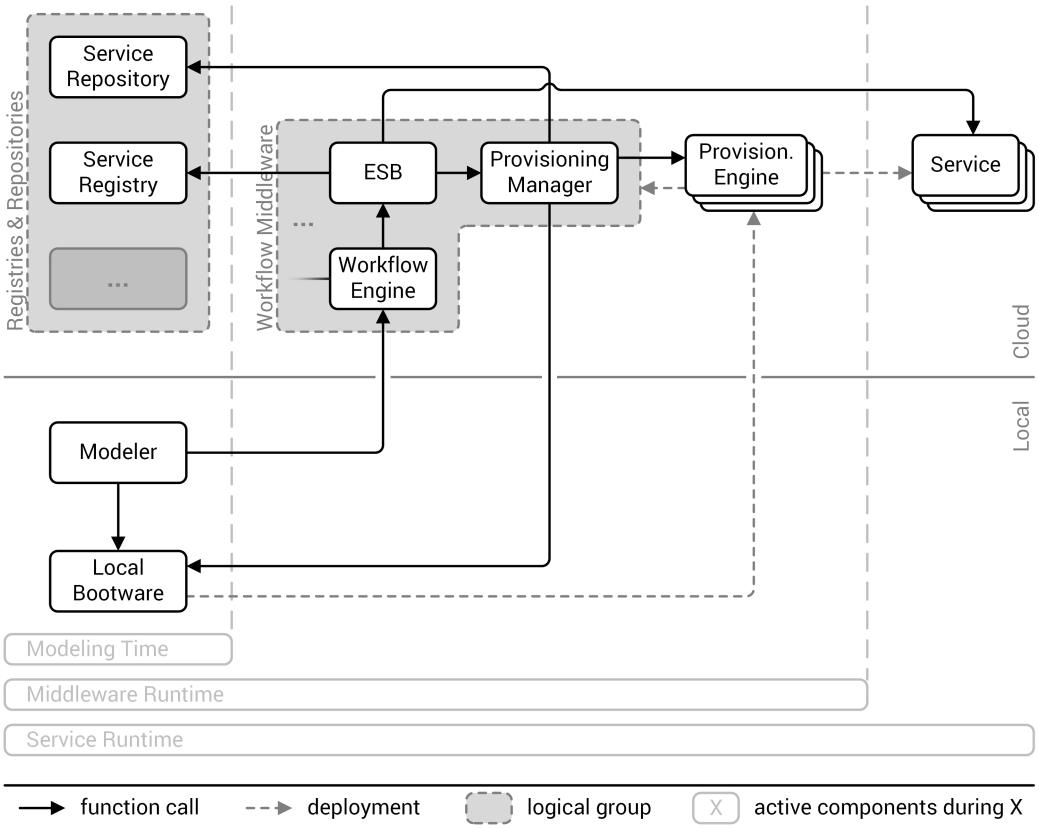

Figure 6.1: Simplified overview of the single local component architecture

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 bringing the bootware into a cloud environment and each user would have his own personal bootware 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 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. As Figure 6.1 shows, all calls to the bootware component from the provisioning manager 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. [21]. 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 [21].

6.1.2 Single Remote Component

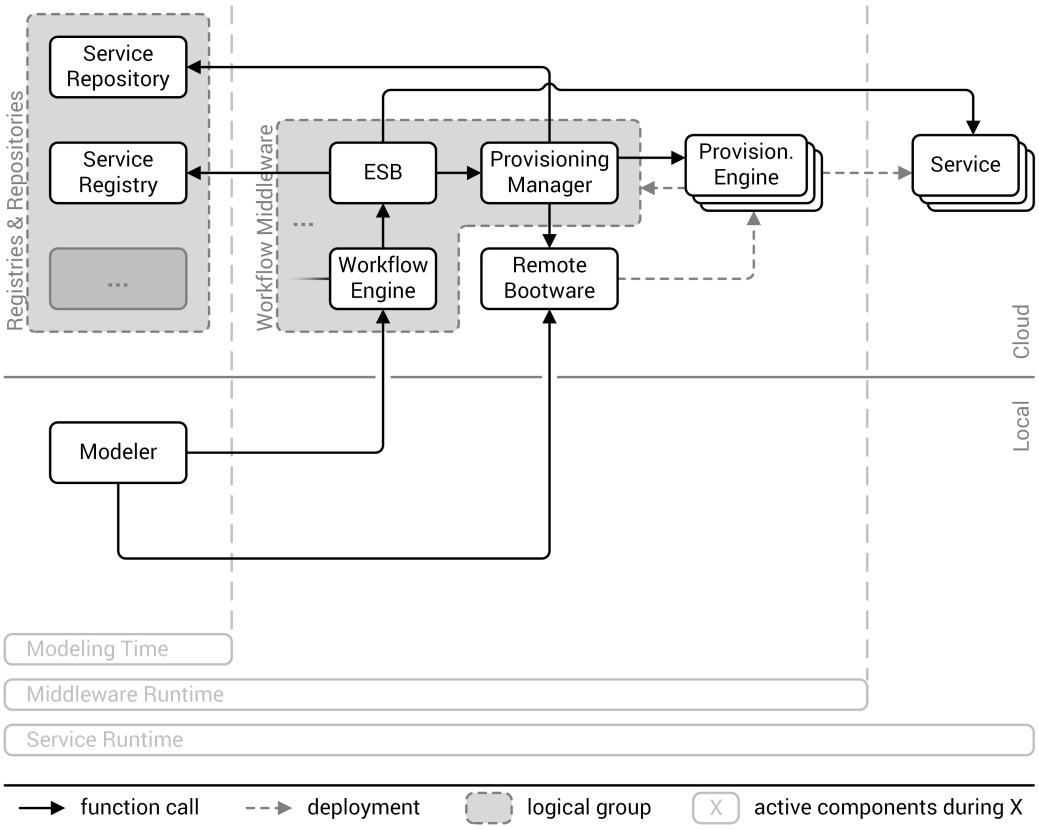

Figure 6.2: Simplified overview of the single remote component architecture

The next obvious choice, as displayed in Figure 6.2, 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 provisions 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.

6.1.3 2-Tier Architecture

Next, we take a look at a 2-tier architecture, as shown in Figure 6.3, 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 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

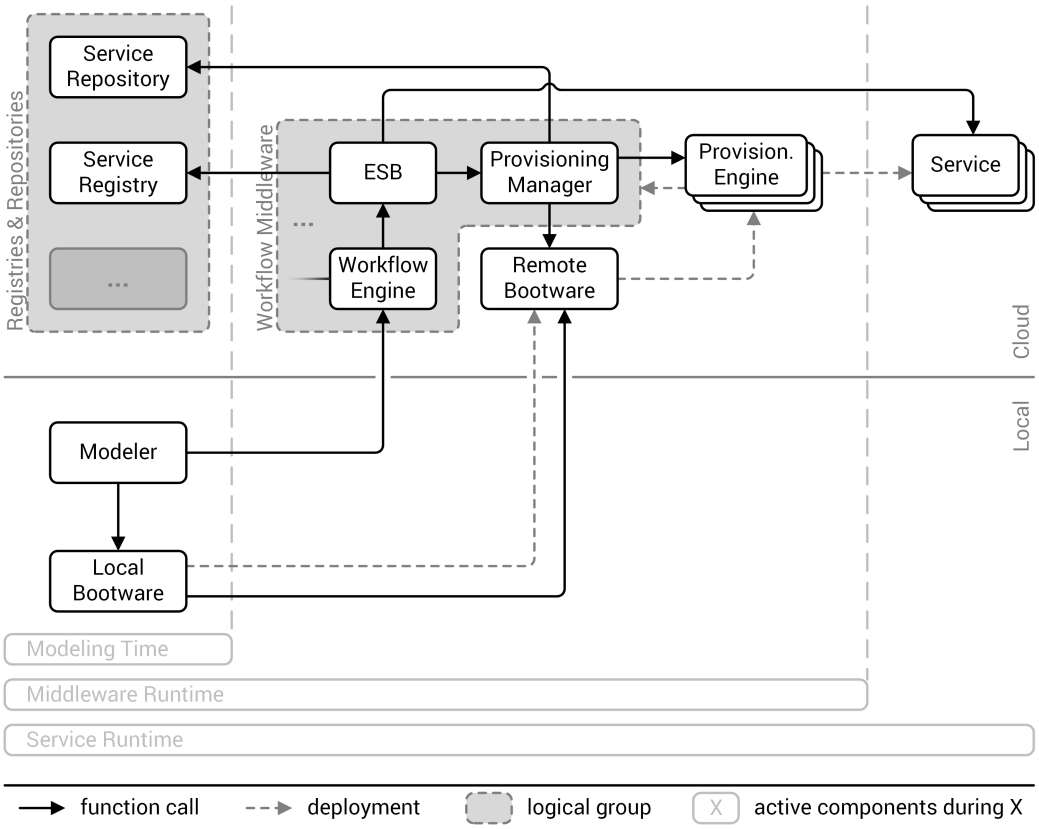

Figure 6.3: Simplified overview of the 2-tier architecture

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 section 6.4. 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 section 6.3

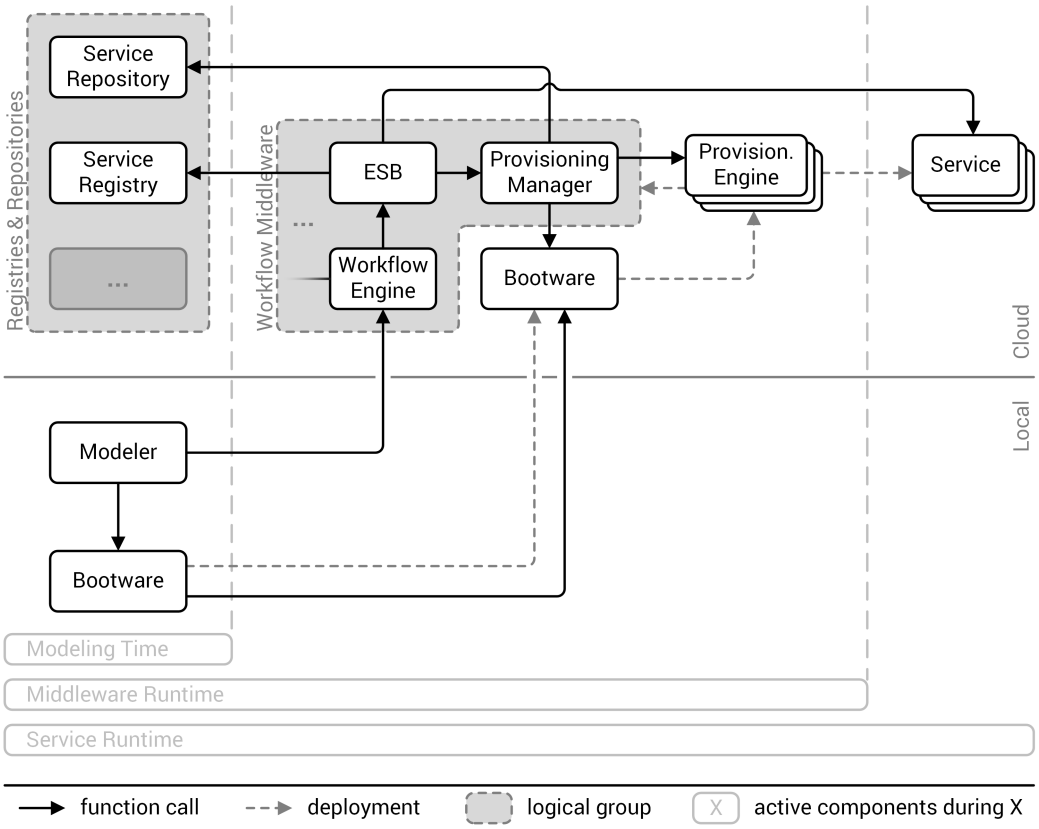

Figure 6.4: Simplified overview of the cloned component architecture

6.1.4 Cloning

This architecture can be seen as an alternative form of the 2-tier architecture described in subsection 6.1.3. 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, as shown in Figure 6.4. 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. Duplication of any functionality wouldn't therefore be an issue. The disadvantage would be that the local bootware would be exactly as complex as the remote bootware and might contain functionality that it wouldn't require for local operation and vice versa. 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.

6.1.5 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.

6.2 Modeler Integration

Looking at Figure 6.3, we can see that the first interaction with the bootware is the call from the Modeler to the local bootware, which starts the bootstrapping process. So in this section we're going to take a look at the integration between modeler and bootware in more detail. The first question we face is: Why even divide the modeler and the local bootware? Why don't we integrate the local bootware functionality into the modeler? We go this route, because we want the bootware to be as generic as possible. The modeler in Figure 6.3 is not a specific modeler and in theory it should be possible to use the bootware with any modeler (and any workflow middleware) without to much modification. So by keeping the bootware as a separate generic component and only implementing a small modeler specific connector, we are able to support different environments without changing the core bootware components.

In chapter 5 we mentioned that the bootware should hook into the already existing deploy process in the modeler. How this deployment process works depends on the actual modeler that is used, so at the moment, we can't say how exactly we can integrate in this process. Specific integration details for our modeler, the SimTech SWfMS, will be discussed in section 8.1. We know however what needs to happen in this connector to get the bootstrapping process going.

First, the connector has to start the local bootware component so that it will be in a state where it can receive and process requests. This is shown in Figure 6.5 as deployment operation from the modeler bootware plugin to the local bootware and involves starting an executable and maybe passing along some sort of configuration file. Once the local bootware component is

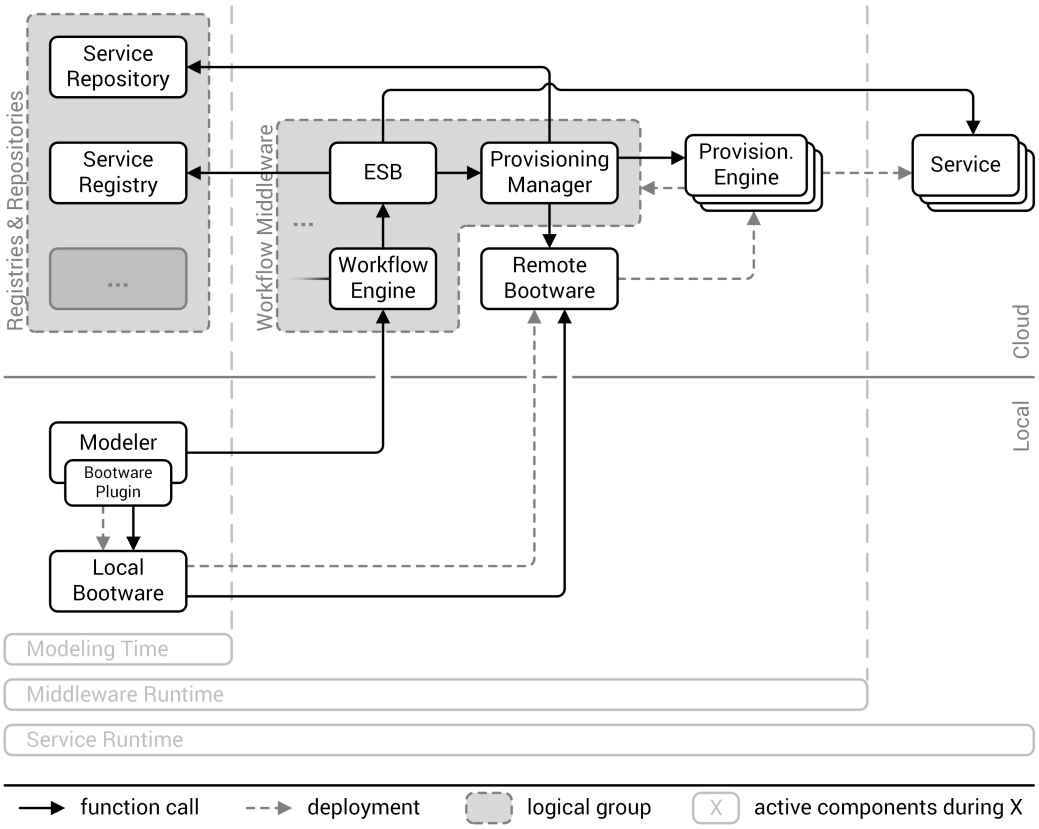

Figure 6.5: Modeler integration with a plugin.

running, the connector has to set up the context for the following requests. This includes telling the bootware the login credentials for all cloud providers that will be used. Once this is done, the modeler has to make one request to the local bootware, containing the cloud provider, the provisioning engine, and the package reference for the workflow middleware, which is shown in Figure 6.5 as function call from the modeler bootware plugin to the local bootware. The local bootware will take this information and provision the remote bootware, a provisioning engine, and the workflow middleware in the specified cloud environment. If successful, it returns the endpoint references of the workflow middleware to the connector, who then has to set up the modeler to use these references for the actual workflow deployment.

This is the minimal work the connector has to do to kick off the bootstrapping process. Additional functionality can be implemented if desired, but is not necessary for the core bootstrapping process. This additional functionality could include user interface integration, additional bootware management functionality, etc. The function call in Figure 6.5 assumes that there exists some interface in the local bootware that is accessible from the outside.

In the next section we will discuss how this remote communication mechanism will be implemented.

6.3 External Communication

In section 6.2 we established that a bootware plugin in the modeler has to call the local bootware. From section 6.1 we also know that both the local bootware and the provisioning manager have to call the remote bootware. We now have to decide, how this 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 provisioning manager and the modeler plugin.

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 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)[1], which allows object in one Java VM to invoke methods on objects in another Java VM. 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 [1] and transport layer encryption and other security features [9]. Another alternative are web services via Simple

[1] `http://docs.oracle.com/javase/7/docs/api/java/rmi/package-summary.html#package_description`

Object Access Protocol (SOAP) or Representational State Transfer (REST). Like CORBA, web services also support asynchronous invocation, as well as security mechanisms [34].

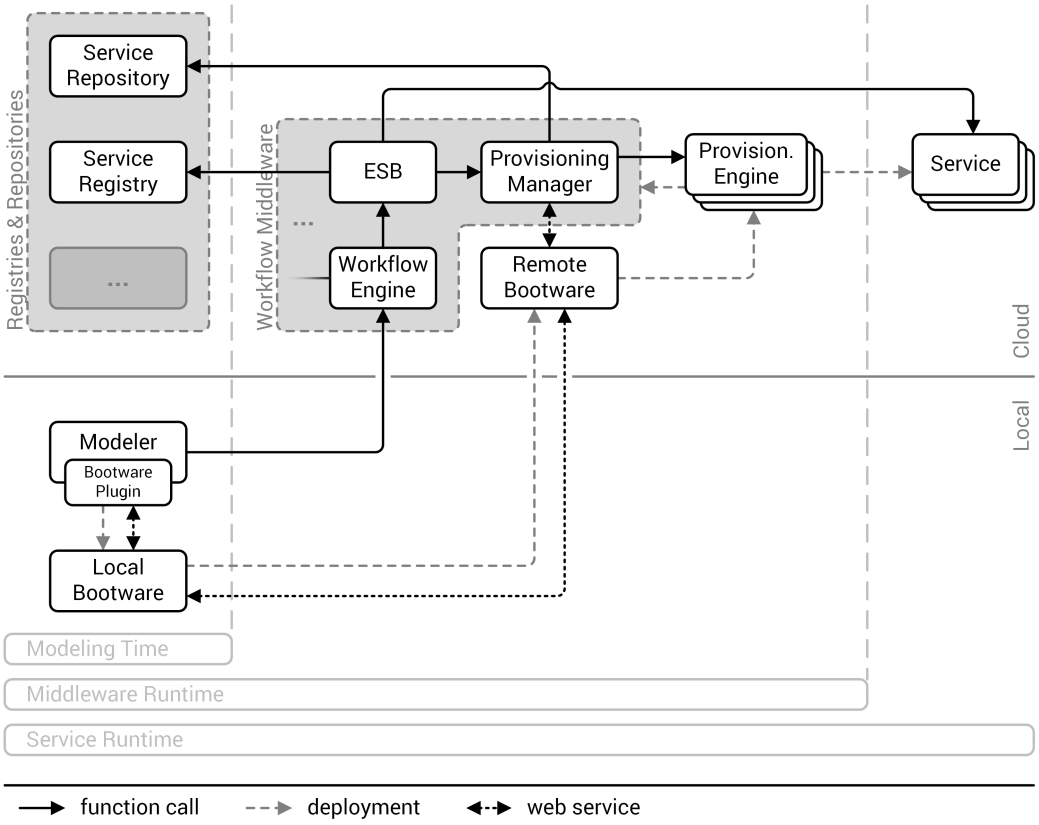

Figure 6.6: Simplified overview of the 2-tier architecture with asynchronous web service communication

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 6.6 shows the addition of asynchronous web service call and return communication between the modeler bootware plugin and the local bootware, and between the remote bootware and the local bootware, as well as the provisioning manager.

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.

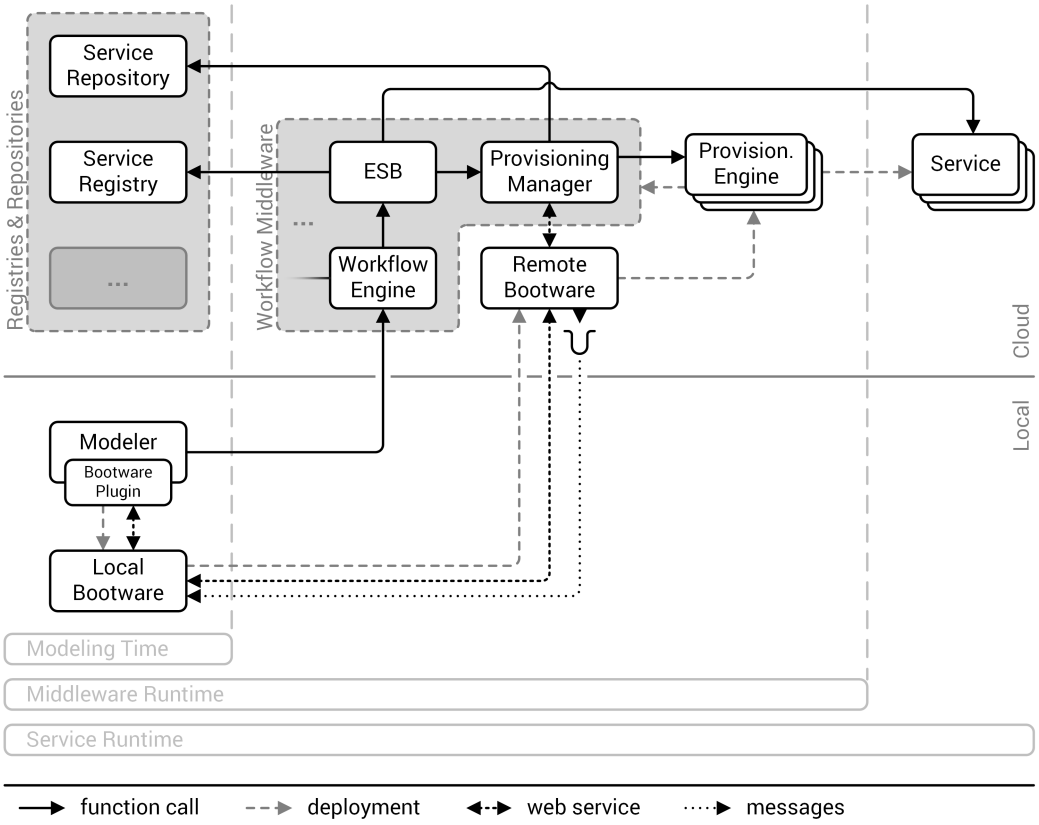

Figure 6.7: Simplified overview of the 2-tier architecture with asynchronous web service and messaging queue communication

This secondary communication channel 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 6.7 shows the proposed architecture with an additional (and optional) message queue that allows the local bootware or other components to listen to status updates from the remote bootware. Since it is not necessary for the successful use of the bootware, it would make sense to implement this secondary communication mechanism as an extension to the bootware. This extension would not be part of the core bootware, but rather an additional component that could be used when needed. This would allow us to add arbitrary communication extensions to the bootware depending on future needs. How this can be done will be discussed in the next section.

6.4 Extensibility

In section 6.3 we mentioned a secondary communication mechanism that would be best implemented in form of an extension to the bootware. The requirements for the bootware component also state that support for different cloud environments and provisioning engines should be achieved through means of software engineering. These requirements are intentionally vague to allow for the selection of 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.

6.4.1 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. 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 8.3.1.

6.4.2 Plugin Repository

Now that we have introduced plugins we face new problems. Figure 6.8 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 6.9. 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. Its design and implementation are left for future work and the plugin repository will not be mentioned in any other figures apart from Figure 6.9.

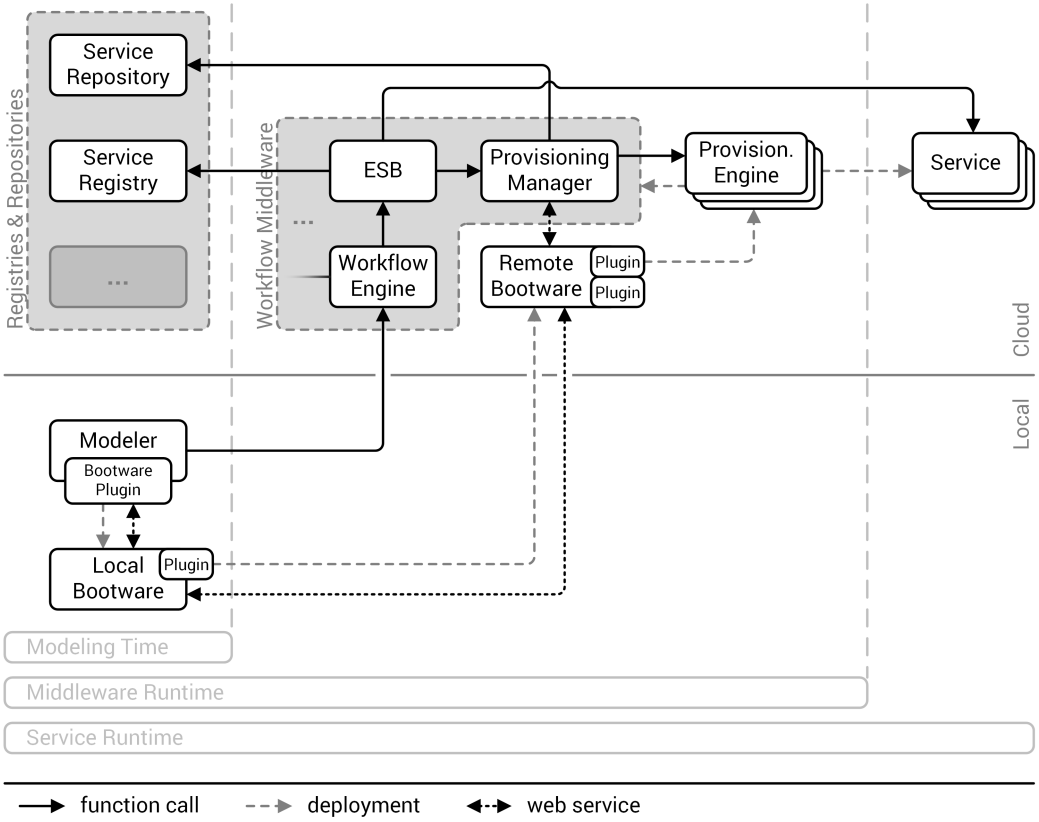

Figure 6.8: Simplified overview of the 2-tier architecture with plugins

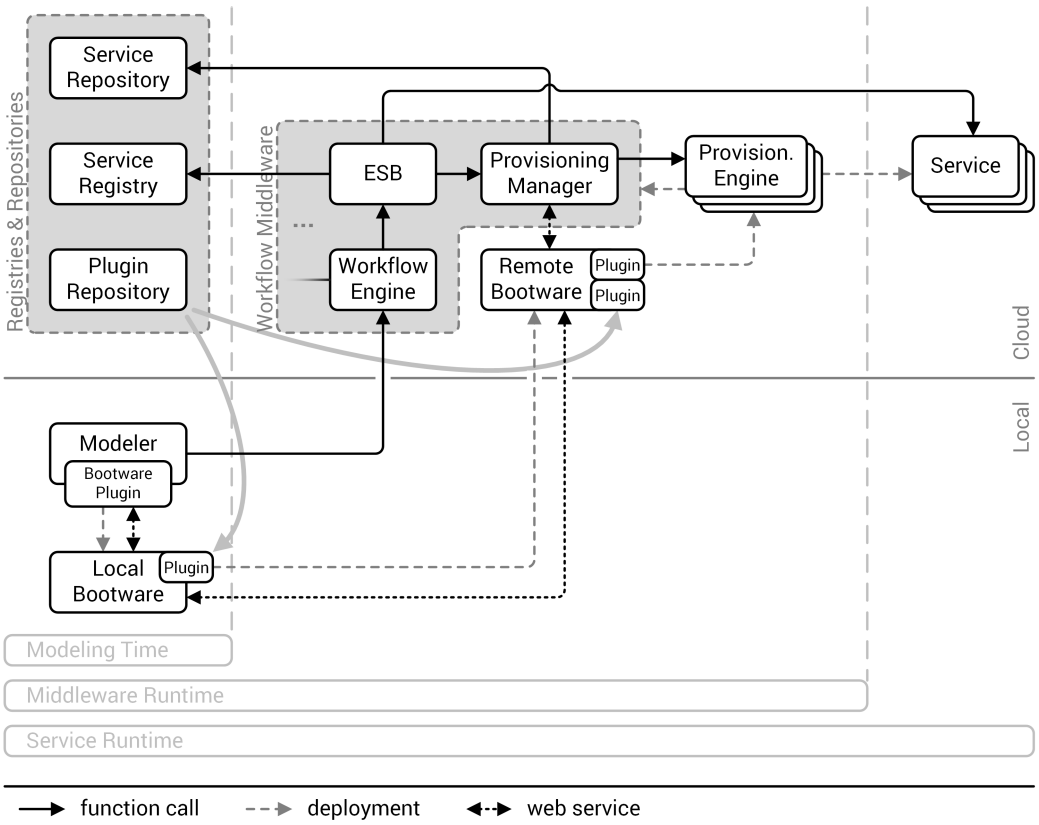

Figure 6.9: Simplified overview of the 2-tier architecture with a plugin repository

6.5 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 bare bone 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*.

The remote bootware also has to handle the initial provisioning of the workflow middleware, which involves calling a provisioning engine to tell it to start the provisioning process. Since this has to be done differently for all provisioning engines, it would make sense to also package this functionality into plugins that can be interchanged. We therefore introduce a fourth plugin type: The *provisioning engine plugins*.

In section 6.3 we also introduced the notion of secondary communication channels realized by plugins. We can generalize this into a more versatile fifth plugin type: The *event plugins*. These plugins are a bit less specific then the four other types. They allow users to add functionality that reacts to (or creates) events inside the bootware. How the actual event system will be implemented will be discussed in section 6.6. With this fifth 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 6.1: Common operations to be implemented by all plugin types

Table 6.1 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.

6.5.1 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 6.2 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 6.2: Interfaces to be implemented by infrastructure plugins

6.5.2 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 6.3 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 6.3: Interfaces to be implemented by connection plugins

6.5.3 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 commands on a resource.

Table 6.4 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 6.4: Interfaces to be implemented by payload plugins

6.5.4 Provisioning Engine Plugins

Provisioning engine plugins provide the bootware with a unified way to call provisioning engines an trigger provisioning and deprovisioning operations. Table 6.5 shows the operations that these plugins should implement. The provision operation calls a provisioning engine and trigger the provisioning process. It takes two inputs: An endpoint reference, which points to the provisioning engine that should be used, and a service package reference, which points to the service package that the provisioning engine should provision. When completed successfully, the provisioning operation returns an endpoint reference to the just provisioned service. The deprovision operation calls a provisioning engine and triggers the deprovisioning process. It takes the same inputs as the provisioning operation, an endpoint reference to the

provisioning engine and a service package reference.

Operation	Input	Output	Description
provision	Endpoint Reference, Service Package	Endpoint Reference	Tells the provisioning engine to provision the given service package
deprovision	Endpoint Reference, Service Package	-	Tells the provisioning engine to deprovision the given service package

Table 6.5: Interfaces to be implemented by provisioning engine plugins

6.5.5 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 6.1. 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.

6.6 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 [13].

6.6.1 Publish Subscribe Pattern

The publish-subscribe pattern (PubSub) is a messaging pattern that consists of three types of participant: An 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 [13].

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.

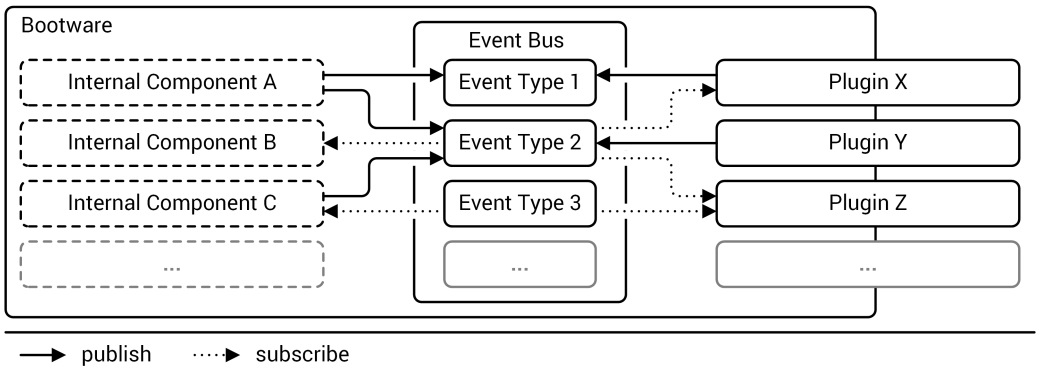

Figure 6.10: Bootware internal communication with PubSub pattern.

6.6.2 Event Types

When using PubSub and events to communicate, its usually a good idea to not only use one type of event, but many different types. Using different kinds of event allows us to subscribe only to specific events or react differently based on the type of an event. But what if we want to react to each event type in the same way, for example for logging purposes? Now, many different event types complicate things more. This is where event hierarchies become useful. At the core of an event hierarchy is a single base event By extending and refining this base event, other, more specific event types can be created, which again can be used as base type for even more specific events. This allows us to create a fine grained hierarchy of events and also enables us to subscribe to particular sub sets of this hierarchy. This makes event handling much easier, since we can now just react to the parent event if we don't need to distinguish between different event types for a particular task.

A second mechanism to differentiate between events is some sort of severity value that each event contains. Many events will be published in an event system, but not all of them might be of the same importance. The majority might be of low value while a few events might be very important. For example, for logging purposes we might not be interested in every event, but only warnings and errors. By adding a severity attribute to the base event type, all events could be categorized in different severity groups and filtered accordingly if needed.

As we can see, we might benefit from a well thought-out event hierarchy.

base event plugin event plugin loading event plugin execution event state machine event transition event start event stop event

6.7 Context

During the bootstrapping process, the bootware has to know certain things to be able do its job. This information can be combined in one central object, which defines the nature of the current request: The context. In this section we will take a closer look at this context object and its content. How exactly the context is implemented is shown in section 8.5.

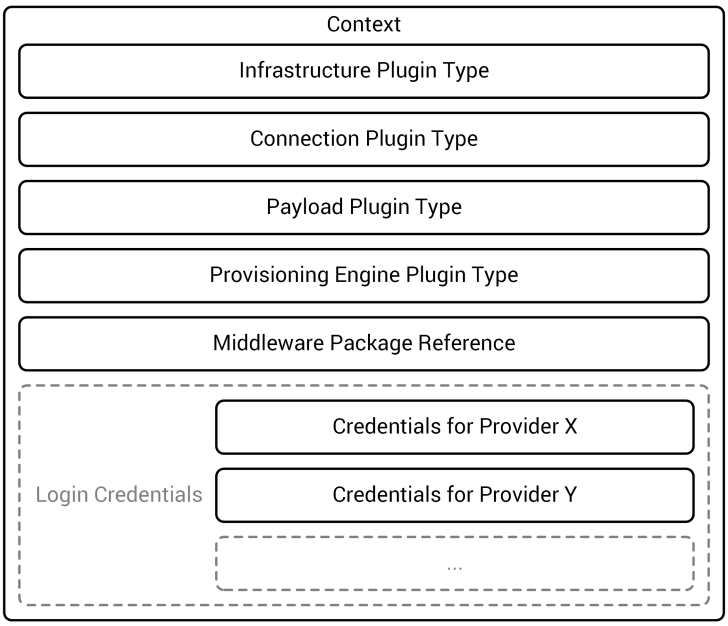

Figure 6.11: Content of the context object.

Figure 6.11 shows the context object and its content. As we can see in the upper half of Figure 6.11, it defines the plugin types to be used for the current request. The infrastructure plugin type defines, which infrastructure plugin should be used to provision the requested infrastructure. The connection plugin type selects how the bootware should connect to to this infrastructure. The payload plugin type defines the payload that should be provisioned on this infrastructure, which will be a provisioning engine in most cases. Finally, the provisioning engine plugin type defines the plugin that should be used to call the provisioning engine. It will use the middleware package reference, which is also defined in the context, as input to start the provisioning of the workflow middleware. In the bottom half of Figure 6.11 we can see that the context can also contain login credentials for different infrastructure providers (i.e. cloud providers). These will be used by the infrastructure plugins to connect to a specific account at their provider. In the future, the context might be extended to hold additional information, but for this thesis, this context will be sufficient.

Now that we have defined how the context will look like, we need to find a way to actually get it to the bootware. There are a few things that we have to keep in mind when doing this. First, these values have to be changeable by the user, so it doesn't make sense to hard-code them into the bootware. Furthermore, the lifetime of the information carried in the context varies quite a bit. The plugin types are only useful for one request execution and are likely to change from request to request, for example if the provisioning manager wants to provision multiple services with different provisioning engines. Therefore, we must provide the plugin types on a per request basis. Since we already decided that we'll be using web services as external communication method, we can send the context containing the plugin types with each request as part of the soap message body.

The cloud credentials on the other hand aren't likely to change between requests. Additionally, we have other components calling the bootware who don't know (and maybe shouldn't know) anything about login credentials, for example the provisioning manager. While this might change in the future, it would make sense to be able to set the login credentials once when starting the bootware, so that they don't have to be delivered with each request and so that other components, who don't know them, can still use the bootware. It should however still be possible to override or update the login credentials at a later point. Overriding would allow any request to temporarily use other login credentials if necessary. Updating the login credentials at a later point could be useful, for example if the user accidentally provided the wrong credentials at the beginning. Without this functionality, the whole bootware process could fail (even while provisioning the very last service) and would have to be started again from the beginning. This could be avoided by providing the functionality to change login credentials even during the bootstrapping process.

For setting the login credentials at the beginning and for updating them later during the

process, a *setCredentials* method will be added to the bootware web service. The credentials set by this method will be treated as the default credentials by the Bootware. If no other credentials are provided, these will be used during the process. If however a request is send with a context that also contains login credentials, these credentials will override already existing default credentials temporarily for this request. This behavior could also be extended to other parts of the context in future, if necessary.

6.8 Web Service Interface

By now, we know that we will use a web service interface for remote communication. To trigger the basic functionality of the bootware, two operations have to be made public via the web service interface: The deploy and the undeploy operation. In section 6.7 we also mentioned the *setCredentials* operation for setting or updating the login credentials.

Operation	Input	Success Response
deploy	Context	endpoint references
undeploy	endpoint references	-
setCredentials	credentials list	-

Table 6.6: Web service operations provided by the local and remote bootware.

6.8.1 deploy

The deploy operation is called whenever a new payload (e.g. a provisioning engine, or initially, the remote bootware) should be deployed. As input it takes a context object as described in section 6.7. If it was able to successfully deploy the requested payload, it responds with an endpoint reference to the payload. If the deployment failed, it responds with an error message.

6.8.2 undeploy

The undeploy operation is essentially the reversal of the deploy operation. As input it takes an endpoint reference to a payload that should be undeploy. If the undeployment succeeds, it responds with a success message. If it fails, it responds with an error message.

Unlike the deploy operation it doesn't take a context object as input but the context is still needed for the undeploy operation, since it contains the information about which plugins have to be used. This means that we have to store the context object used during each deploy operation so that we can retrieve it later during the corresponding undeploy operation. This design is intentional and will be described in more detail in section 6.9.

6.8.3 setCredentials

The setCredentials operation is used to transmit or update the default login credentials used by the infrastructure plugins to authenticate with the infrastructure provider. As input it takes a list of credentials that should be saved. If the list provided is empty, the default credentials list saved in the bootware will be emptied. If the list provided is not empty, the default credentials list saved in the bootware will be overwritten by this list. The credentials can still be overwritten on a per request basis if the context send with the request also contains credentials. If the credentials were updated successfully, it responds with a success message. If the credentials couldn't be updated, it responds with an error message.

6.9 Instance Store

The instance store stores information about payloads that were deployed by the bootware in the past and are still active. In section 6.8 we already mentioned that we need to store some information about active payloads, but we didn't explain why. There are several reasons why this is useful.

One big reason is that we can't guarantee that an undeploy operation will be called for every payload deployed by the bootware, since we might not have control over all components that ultimately call the bootware. We could require that for each deploy call there must eventually be an undeploy call so that everything will be cleaned up in the end, but errors can be made and it's better to have a failsafe in place. In the worst case scenario, failing to call undeploy for some payloads could lead to rogue services remaining active after a bootware execution has

stopped without the user realizing it, which could get expensive. Storing enough information allows us to undeploy remaining payloads before shutting down the bootware even if they were never explicitly undeployed. Additionally, a warning could be return by the bootware to inform the user that some non-bootware component should be modified to explicitly undeploy all services he deployed.

Another reason to store some information about deployed payloads is to simplify the interaction with other components. If we wouldn't store any information and make the bootware stateless, each component using the bootware (e.g. the bootware modeler plugin, the local bootware, and the provisioning manager) would be required to keep track of all payloads he deployed using the bootware, so that this information can be supplied when it's time to undeploy. This places an extra burden on these components and scatters around the information about deployed payloads. By storing this information in the bootware we simplify the usage of the bootware for other components and concentrate this information.

We should also think about how such a storage mechanism might be different for the local and remote bootware. The local bootware only every deploys the remote bootware, so here we have to keep track of only one thing. The remote bootware on the other hand might deploy many payloads during an execution. For the local bootware it might be sufficient to store this information in a text file on the local machine where it is executed, whereas the remote bootware might use some sort of persistent storage in the cloud. This would allow it to retrieve this information even after a crash. For this thesis however we will be using simple in memory storage for both the local and remote bootware. Changing that to a more sophisticated storage solution is a possible option for future improvement.

Now that we know why it makes sense to store information about active payloads, we need to discuss what exactly we need to store. We need to store enough information to be able to undeploy an active payload without any further input. For this we need to know: The infrastructure plugin that was used to provision the infrastructure, the connection plugin that was used to connect to it, the payload plugin that was used to deploy the active payload, and login credentials for the remote environment if necessary. This is all contained in the context object that we used in the first place to deploy the payload, so we will just store the whole context object. Since we also use this storage for the undeploy operation, where we get an endpoint reference as input, we have to store it in such a way that we can map a particular context object to the provided endpoint reference.

6.10 Execution Flow

Until now we have established how the bootware can be called from outside components using a web service interface and a context to start the bootstrapping process. We also established that big parts of this process will be implemented as plugins. Now it's time to take a look at the actual internal structure of the bootware. What follows is a step by step description of the whole bootstrapping process.

Figure 6.12 shows a graph that represents the major steps during the bootware execution in the local bootware as flow diagram. The bootstrapping process is started by executing the local bootware, which is represented by the start state in the top left corner of Figure 6.12. From there, the bootware first does some initializations. If these fail for some reason, the cleanup code will be executed before the local bootware execution is ended, as can be seen on the top right corner of Figure 6.12. In most cases however, the initialization should succeed. Then, the local 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 local bootware execution, since they will not change at a per request basis (like the other plugins). If loading these plugins fails, the local bootware will try to unload them before continuing to the cleanup state. If the plugins are loaded successfully, the local bootware transitions into the wait state, shown in the top center of Figure 6.12.

Once the local 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 local 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 local bootware. If a request is received in the wait state, the local bootware transitions to the next state, where it reads the request context.

The request context contains all the information necessary to fulfill the request, as described id section 6.7. If the context can not be read, the local bootware returns a response containing an error message before returning into the wait state. If the context is read successfully the local bootware tries to send the request on to the remote bootware, as shown in the middle of Figure 6.12. For this to work, the remote bootware has already has to exist in the requested remote environment, which won't be the case during the first execution. Therefore, the local bootware first has to provision the remote bootware in the requested remote environment and so it transitions to the load request plugins state.

In the load request plugins state the plugins specified in the context are loaded. If this fails, the local bootware tries to unload them before return an error response and returning to the wait state. If the plugins are loaded successfully, the local bootware now starts either the

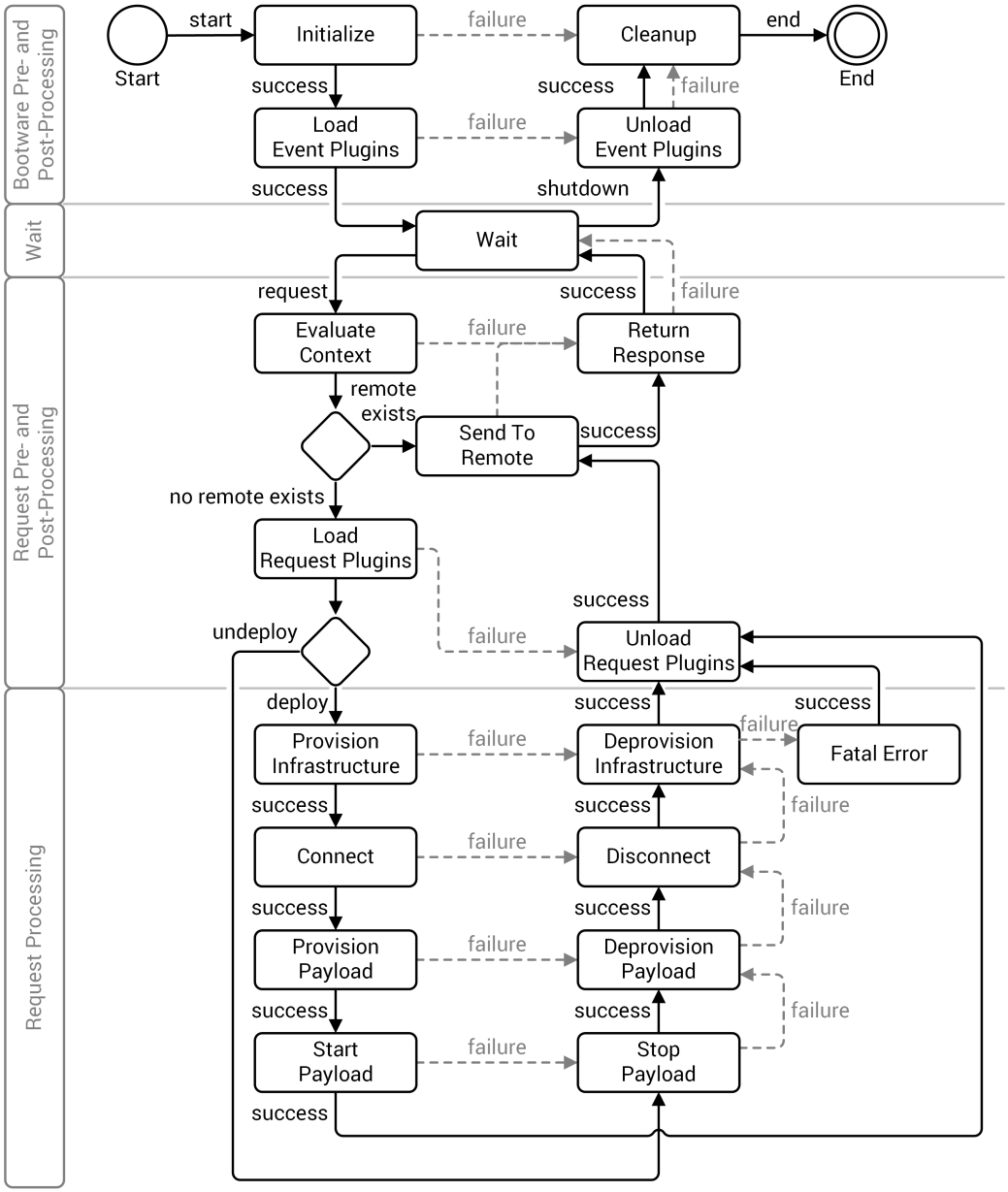

Figure 6.12: Execution flow in the local bootware.

deploy process or the undeploy process, shown at the bottom of Figure 6.12, depending on the type of the request.

If the request was a deploy request, the local bootware will now execute the steps shown in the bottom left of Figure 6.12 one after another, which include the deploy, connect, and start operations of the infrastructure, connection, and payload plugins. If one of those operations fails the local 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, shown on the bottom right of Figure 6.12, which is triggered by an undeploy request.

If the stop payload, deprovision payload, or disconnect states fail, the local bootware just continues with the next undeploy state, since these operations are not considered critical. However, if the deprovision infrastructure state fails, the local bootware transitions to a fatal error state, show at the right of Figure 6.12, 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 taking special actions to remedy this situation.

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. If everything went as planned, a remote bootware should now be running in the desired cloud environment and the local bootware can now pass on the request to this remote bootware, as shown in the center of Figure 6.12. The local bootware will wait in this state until she receives a response from the remote bootware.

Now, we move our attention to the remote bootware, where the requests continues to be processed. Figure 6.13 shows the execution flow of the remote bootware. As we can see, it is largely identical to the local bootware, at least at the moment. The send to remote state is gone, since we don't need this in the remote bootware. Instead, as the bottom of Figure 6.13 shows, the provision and deprovision middleware steps were added. Other than that, the local and remote processes are the same.

Like the local bootware, the remote bootware went through the initialization steps shown at the top of Figure 6.13 when it was started by the local bootware. It then waited in the wait state for any request. Now, it receives the request from the local bootware, reads the context, loads the request plugins and executes the deploy operation. This should result in a provisioning engine being started by the payload plugin. After that, the remote bootware enters the new provision middleware state at the bottom left of Figure 6.13, which will use the just started provisioning engine to deploy the workflow middleware. Once the middleware is

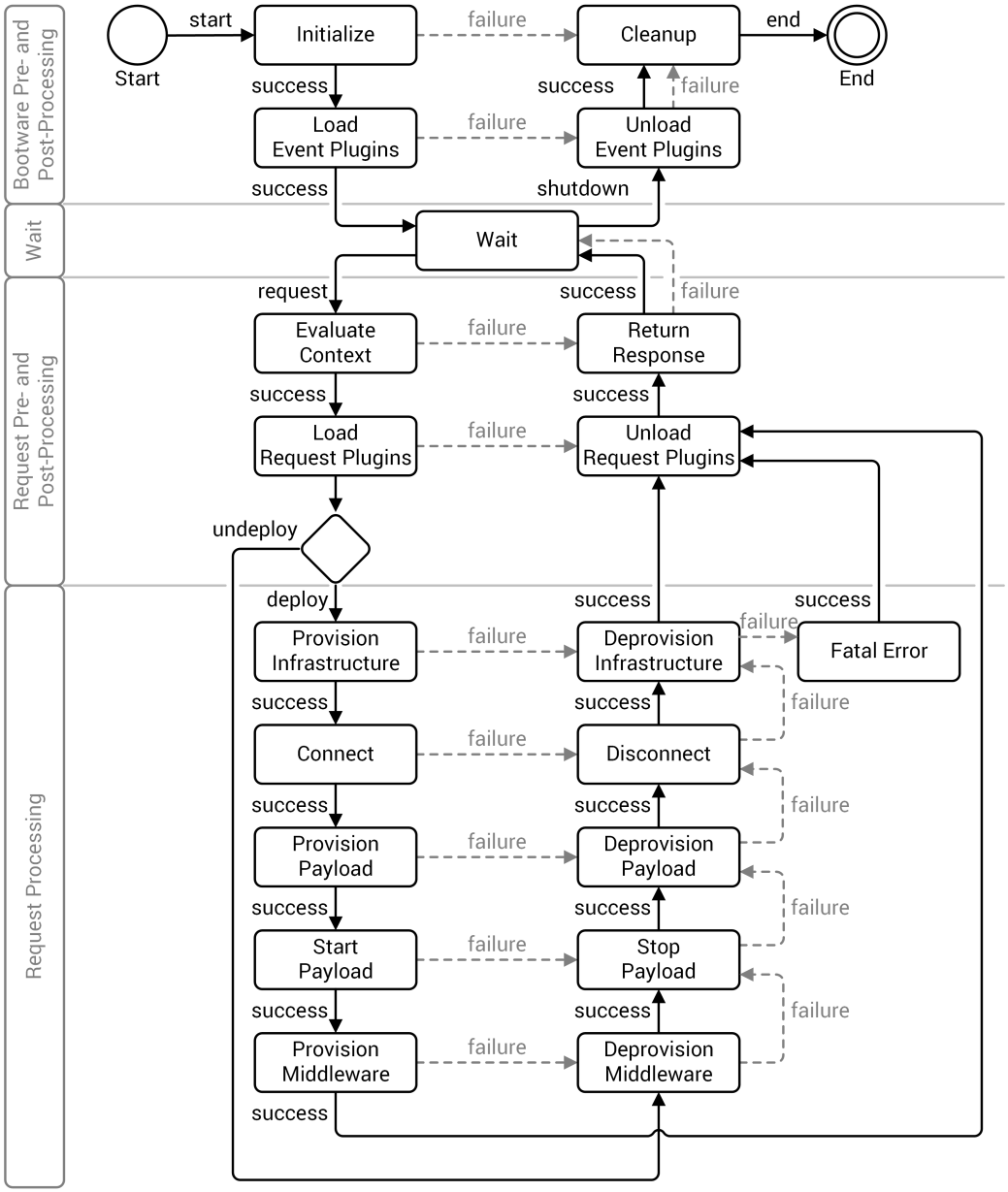

Figure 6.13: Execution flow in the remote bootware.

running, the remote bootware is finished with this request and returns the endpoint references of the middleware in the response to the local bootware, before returning into the wait state.

This brings us back to Figure 6.12, where the local bootware has now received the answer from the remote bootware in the send to remote state. Now, the local bootware can finish its request by sending back a response to the modeler bootware plugin, before returning to the wait state. The local bootware is now done until it's time to undeploy the remote bootware. Meanwhile, the modeler bootware plugin starts the workflow execution on the middleware, during which multiple calls from the provisioning manager to the remote bootware will occur, which will each time trigger the deploy or undeploy process shown at the bottom of Figure 6.13.

As Figure 6.12, Figure 6.13 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.

6.10.1 Finite State Machine

In theoretical computer science, a Finite State Machine (FSM) is formal, abstract model of computation that "consisting of a set of states, a start state, an input alphabet, and a transition function that maps input symbols and current states to a next state. Computation begins in the start state with an input string. It changes to new states depending on the transition function" [5]. In this context, a state is the "condition of a finite state machine [...] at a certain time. Informally, the content of memory" [6]. The start state is therefore the initial condition of a FSM. The alphabet is a "set of all possible symbols in an application. For instance, input characters used by a finite state machine, letters making up strings in a language, or symbols in a pattern element. In some cases, an alphabet may be infinite" [3]. The transition function is a "function of the current state and input giving the next state of a finite state machine" [7]. FSMs can further be distinguished in deterministic and non-deterministic FSMs. A deterministic FSM has at most one transition for each symbol and state, whereas a non-deterministic FSM can have non, one, or more transitions per symbol and state [4].

Aside from its uses in theoretical computer science, FSMs also have practical applications in digital circuits, software applications, or as lexers in programming language compilers. We are only interested in the use of FSMs for building software, so we can redefine what a FSM

means for our case. We want to use a FSM as an abstract machine that is defined by a finite list of states and and some conditions that trigger transitions between those states. Unlike a traditional FSM, we will not consume symbols from a set alphabet that will trigger state transitions. We want the state transitions to be triggered by events that we can emit at any time, so we want an event-driven FSM. The machine is in only one state at a time, its current state. At the start of the machine execution, it will be in the start state. From there, it can transition from one state to another when certain events are triggered, until it finally reaches an end state. When it enters a state, it executes a function associated with this state. The result of the execution of this function determines to which state the FSM will transition next. We will talk more about the actual implementation with FSMs in chapter 8.

6.11 Final Bootware Architecture

Figure 6.14 and Figure 6.15 show the final architecture of the local and remote bootware component. Since the only difference between them is the provisioning engine plugin shown in the top right corner of Figure 6.15, we will describe this figure only. At the bottom we can see four exemplary event plugins. These are loaded at the beginning of the bootware execution by the plugin manager, shown on the left of Figure 6.15. For demonstrations purposes, Figure 6.15 shows a wider range of possible event plugins. All these plugins provide some sort of input and/or output mechanism for the bootware component. A command-line interface (CLI) plugin could be used to make the bootware 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 an 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. Besides the event plugins there is always the web service interface, shown at the bottom right of Figure 6.15, which provides the standard way to interact with the bootware.

All event plugins and the web service interface 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 6.15, 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 section 6.10. At certain points during the bootstrapping process, operations are delegated to the plugin manager to load plugins, and to the infrastructure, connection, payload, and provisioning engine plugins, shown at the top of Figure 6.15.

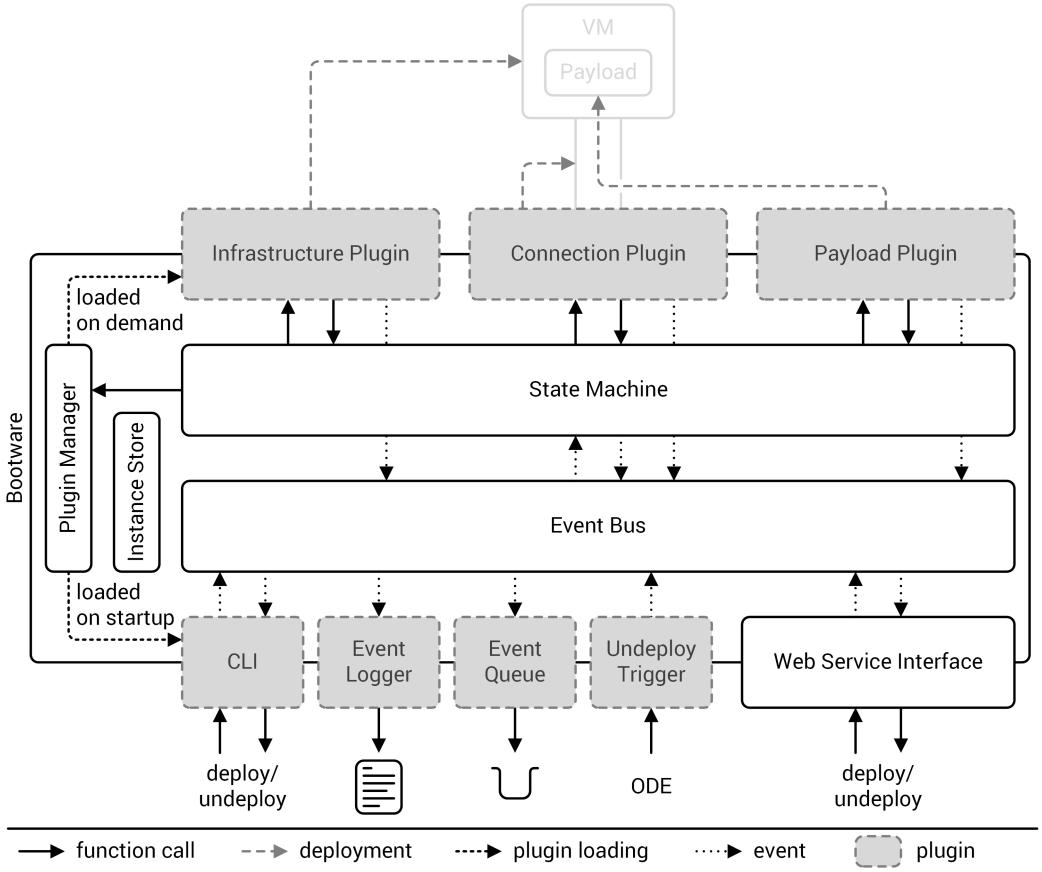

Figure 6.14: The final architecture of the local bootware component.

The infrastructure, connection, and payload plugins implement the actual bootstrapping operations. At the top, Figure 6.15 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. The provisioning engine plugin is only available in the remote bootware and allows it to call a provisioning engine with the details necessary to provision the workflow middleware. This is shown in Figure 6.15 as an additional function call from the provisioning engine plugin to the previously deployed payload. During the bootstrapping procedure, events are sent from all these plugins back to the event bus to be delivered to the loaded event plugins.

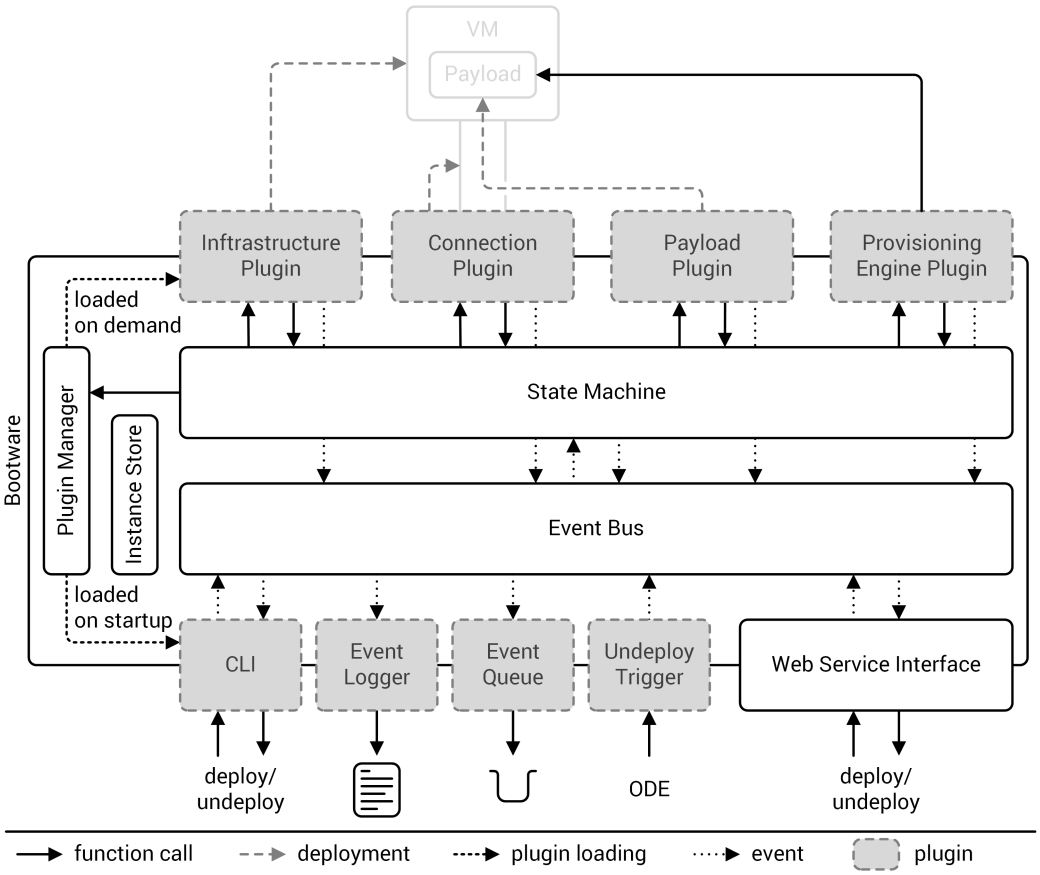

Figure 6.15: The final architecture of the remote bootware component.

7 Bootstrapping Process

This chapter describes the bootstrapping process in its entirety. In Figure 7.1, we can see the whole process with numerated steps. We will go through Figure 7.1 step by step in the following paragraphs to get a better understanding of the whole bootstrapping process.

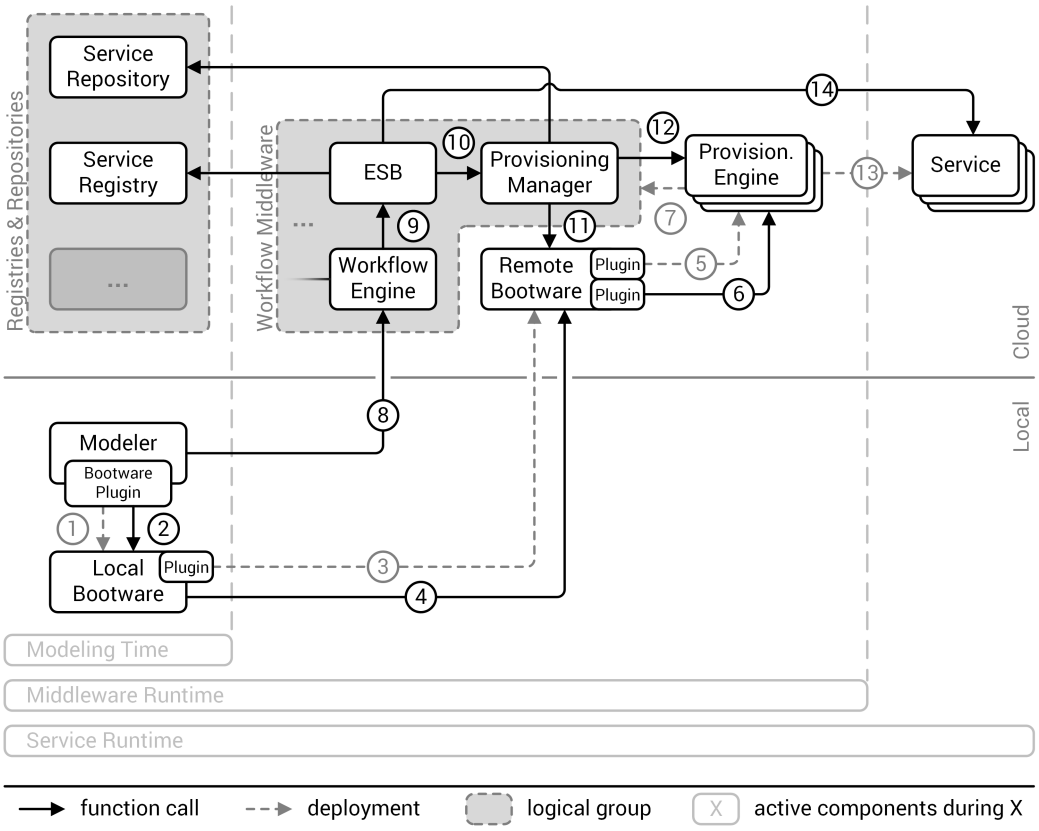

Figure 7.1: The step-by-step bootware process.

At the beginning, a user starts the Modeler, which includes the bootware plugin, as seen on the bottom left of Figure 7.1. If he hasn't done so already, he tells the bootware plugin his cloud login credentials to be used during the bootware process and which event plugins the

bootware should load. He then uses the Modeler to create a workflow as usual. Once the workflow is finished and ready to be executed, he clicks the start button as usual. The bootware plugin has hooked into the start process and takes over by starting the local bootware (step 1).

Once the local bootware is up and running, the bootware plugin calls it with the context the user provided (step 2). The local bootware first checks, if a remote bootware already exists in the requested remote environment. If not, the local bootware provisions a remote bootware using the information provided in the context (step 3). Once the remote bootware is deployed, it is called by the local bootware with a deploy request for the provisioning engine that will be used to deploy the workflow middleware (step 4). The remote bootware deploys the requested provisioning engine using the information provided in the context (step 5). Once the provisioning engine is up and running, the remote bootware calls the provisioning engine (step 6) and tells it to deploy the workflow middleware (step 7). Once the workflow middleware is up and running, the provisioning engine returns the endpoint references of the workflow middleware to the remote bootware, which in turn returns it to the local bootware, which returns it to the bootware plugin. The bootware plugin uses these end point references to link the modeler to the workflow middleware.

Once linked, the modeler deploys the workflow on the workflow middleware as usual and starts its execution (step 8). The workflow middleware now executes the workflow, during which it might encounter a point where it has to call a remote service. The remote service call is passed on to the ESB (step 9), which checks if the service is already reachable. If it is, execution continues as usual. If not, the ESB tells the provisioning manager to provision the requested service (step 10). The provisioning manager checks, if the provisioning engine needed to provision the requested service is already available. If it isn't, the provisioning manager calls the remote bootware with a request to provision the required provisioning engine (step 11). The remote bootware provisions the provisioning engine using the information from the request and the user context (step 5). Once the provisioning engine is up and running, the remote bootware returns an endpoint reference to the provisioning manager. The provisioning manager now calls the provisioning engine (step 12) and tells it to provision the required service (step 13). Once the service is available, the provisioning engine returns its endpoint reference to the provisioning manager, who in turn returns it to the ESB. The ESB can now call the service (step 14) and use the service response to continue with the workflow execution.

The workflow execution now continues in this fashion, spawning new provisioning engines and services through the provisioning manager and the remote bootware along the way (repeating steps 10, 11, 5, 12, 13 and 14). At some point, the workflow will be finished. If it hasn't done so already, the provisioning manager calls all relevant provisioning engines to undeploy any services that might still be running (step 12, 13). Once all services are undeployed, the work

of the workflow middleware is finished. The bootware is listening at the workflow middleware for this event and triggers the undeploy process once it happens. First, the remote bootware calls the provisioning engine that was used to provision the workflow middleware (step 6) and tells it to undeploy the workflow middleware (step 7). The provisioning engine returns the success to the remote bootware. Next, the remote bootware undeploys all provisioning engines that might still be running (step 5). Once all provisioning engines are gone, the remote bootware returns the success to the local bootware. The local bootware removes the remote bootware (step 3) and returns the success to the bootware plugin. At this point, no remote components should be running anymore. The bootware plugin now tells the local bootware to shutdown (step 1), which completes the whole process.

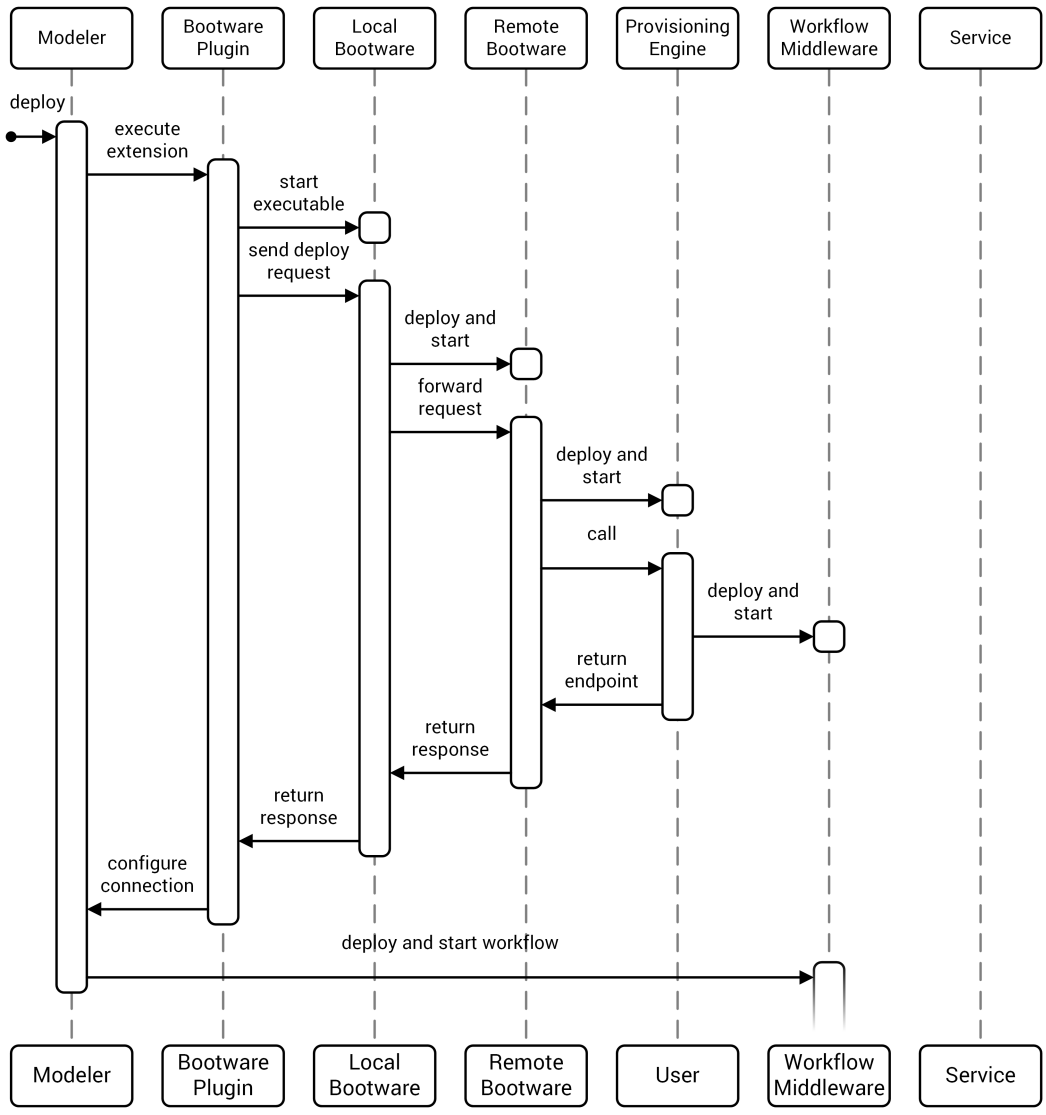

Figure 7.2: Sequence diagram of the bootstrapping phase.

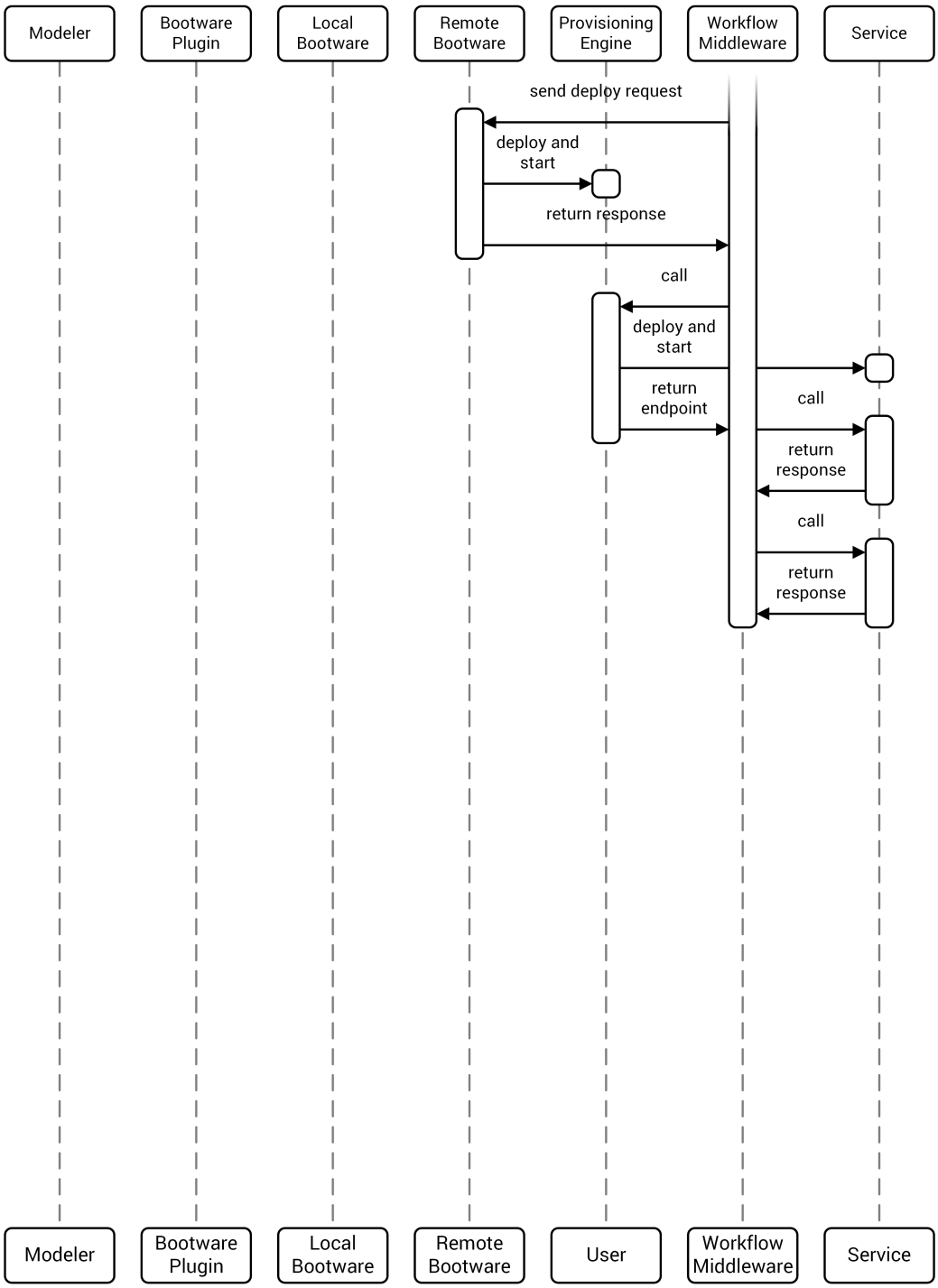

Figure 7.3: Sequence diagram of the workflow execution phase.

8 Implementation

Until now we have described the bootware in a generic context, since it should work with various different scientific workflow systems. But for the implementation we will have to work with a specific system, which in our case is the SimTech SWfMS. Figure 8.1 shows the bootware being used together with the SimTech SWfMS. It also shows the components as one of three types: specific, generic, and adapted. The specific components, shown in black in Figure 8.1, are those components that belong to a specific scientific workflow system. In our case these are the SimTech Modeler at the bottom left and the SimTech ODE (and other components that were omitted in this figure) in the center. On the other hand we have the generic components, shown in white in Figure 8.1. These are components that are build to be generic and can be used in all kinds of environments. In our case these are the local and remote bootwares and their plugins, as well as the provisioning manager and the ESB (here Apache Service Mix) and various repositories and registries. The specific and the generic components have to work together, but there should be no need to make huge modifications to one or the other to do so. Therefore, we need adapter components in some places, which are shown in gray in Figure 8.1. They are responsible for gluing together specific and generic components where necessary and should be the only components that have to be modified or created from scratch to fit to a specific environment. In our case this is the bootware plugin loaded in the SimTech Modeler on the bottom left. There also is an adapter component between the SimTech ODE and Apache Service Mix, which is not shown here.

For the implementation of this diploma thesis we will have to create the generic local and remote bootware components and their plugins, as well as the bootware plugin, which will be specific to the SimTech Modeler. In the rest of the chapter we present details on the implementation of the bootware component. First we describe the implementation of the bootware plugin. Next, we select specific frameworks and libraries that allow us to implement the architecture we developed in chapter 6. Then we present detailed descriptions of the implementation of some parts of the local and remote bootware and some plugins.

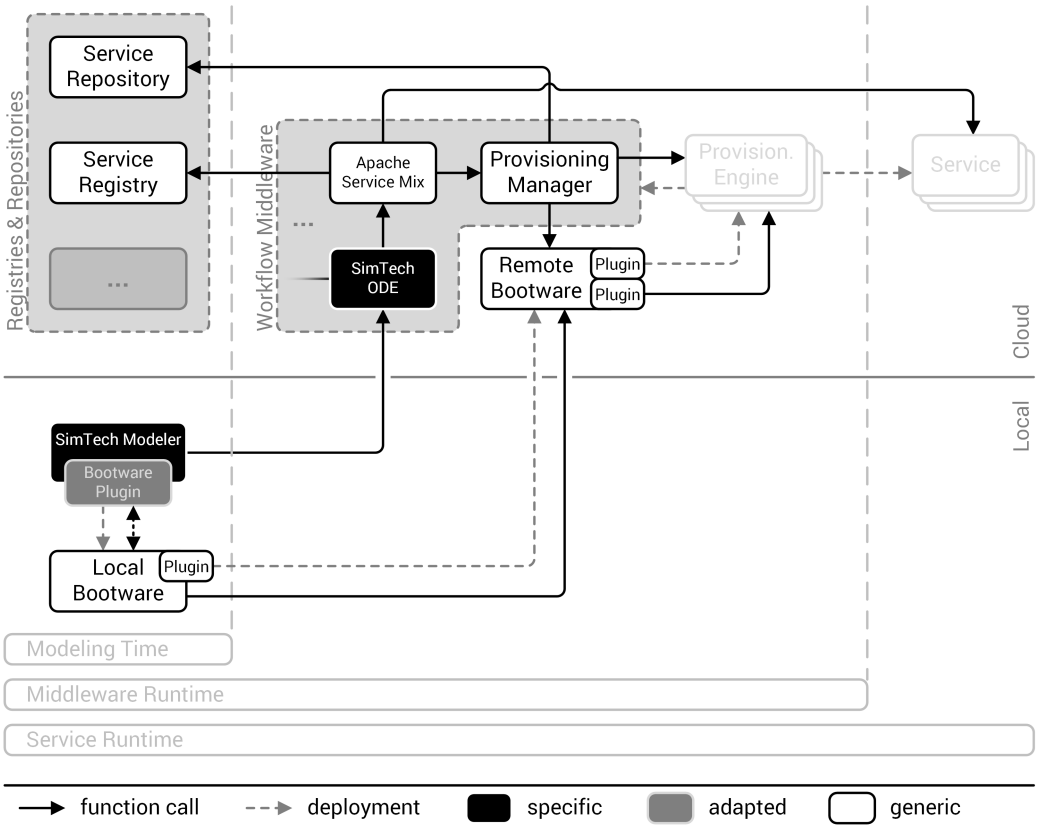

Figure 8.1: Specific and generic components and adapters.

8.1 Modeler Integration

In this section we describe the integration between the SimTech SWfMS and the bootware. Currently what happens is that, if a workflow is is ready and should be executed, the user clicks on a deploy button and the workflow is deployed on the SimTech SWfMS. Now we have to find a way to integrate the bootware into this process. The deploy button is realized by an Eclipse plugin that adds SimTech specific functionality to the Modeler (which is based on Eclipse). We therefore also have to create some kind of Eclipse plugin to hook into this process. There are two scenarios how we could go about this.

We could extend the existing plugin with the functionality that we need for the bootware. In this case we would always load the bootware extensions in the Modeler, even if we don't use the bootware at all.

We could also use a feature called extension points. Eclipse plugins can declare extensions

points, which allow other plugins to extend or customize parts of the plugin[1]. We could define an extension point in the already existing eclipse plugin and create a second plugin which implements this extension point. This way we can separate the bootware functionality from the other SimTech extensions and keep the changes to the existing plugin to a minimum. If a user doesn't need the bootware functionality, he doesn't have to load the bootware plugin and the SimTech plugin will continue to function as before.

The second scenario looks preferable to the first one, so this is what we are going to do. We will modify the already existing Eclipse plugin with an extension point that is triggered at the beginning of the existing deployment process. If the bootware plugin is loaded into the Modeler, it will implement this extension point and set up the SWfMS before the already existing deployment code continues. If it is not loaded, nothing new will happen and the existing deployment code will be executed like before. The bootware plugin can also add additional extension to the modeler, for example a configuration dialog for setting up the context or a view that shows progress messages from the bootstrapping process.

describe packaging. folder drop in to eclipse plugin dir. containing local bootware. remote bootware as payload

8.2 Bootware Core Library

We have seen in section 6.10 and section 6.11 that the local and remote bootware have some common functionality. It would make sense to implement these components in such a way that they can share common functionality. This would avoid code duplication and make changes to common functionality easier. Therefore we introduce the bootware core library, which will encapsulate the common functionality of both bootware components.

Since we are using Java for the implementation, the core library will be a *.jar* file containing common classes that will be imported by the local and remote bootware implementations and also by plugin implementations. Figure 8.2 shows a schematic view of the bootware core library and how its classes are used by various components. We can see that the library includes an abstract finite state machine class, which is used by both the local and the remote bootware implementation. The abstract FSM class defines common state machine functionality that is used by both bootware components. This includes function definitions for the shared states shown in Figure 6.12 and Figure 6.13. This way, the local and remote bootware can import shared states from the library and only have to define their custom

[1] http://wiki.eclipse.org/FAQ_What_are_extensions_and_extension_points%3F

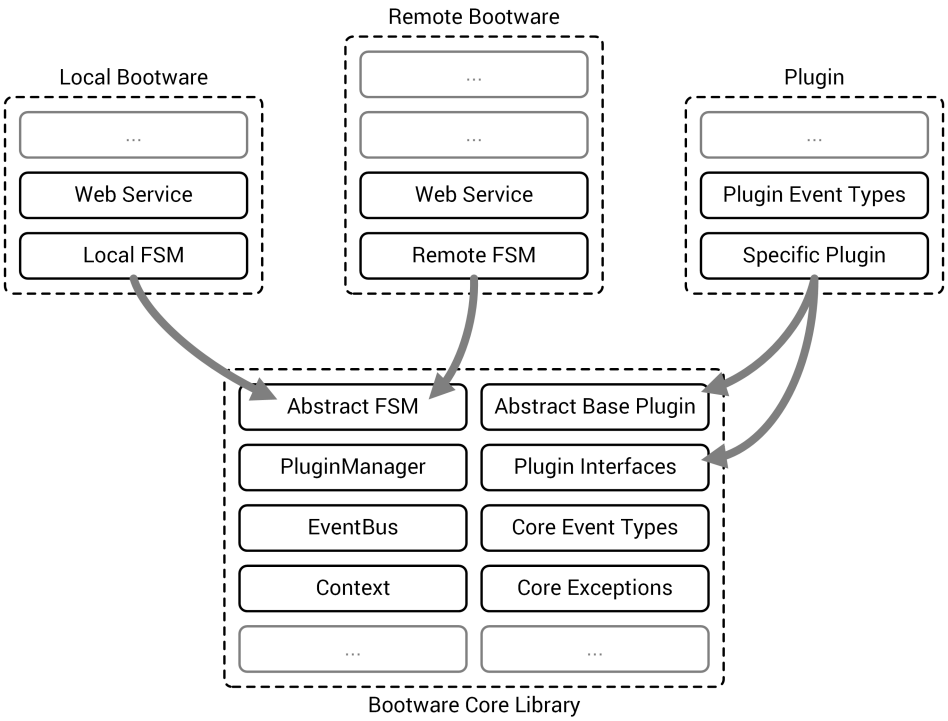

Figure 8.2: The bootware core library and exemplary usage.

states (e.g. the provision middleware states in the remote bootware) and the state transitions. They can also overwrite the states imported from the library if this is necessary.

The library also includes an abstract base plugin class, which implements some functionality that is common to all plugin. Actual plugin implementations can extend this base plugin class to get this common functionality. They also have to implement one of the plugin interfaces defined in the bootware core library, so for example, an infrastructure plugin has to implement the infrastructure plugin interface.

Aside from the code imported from the bootware core library, the components using the library are free to add various other code to their implementation. This way, the remote bootware could implement some extra functionality not needed in the local bootware, or a plugin could define its own event types.

8.3 Selecting Frameworks and Libraries

Before we can begin with the actual implementation of the local and remote bootware, 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 PubSub and FSM libraries.

8.3.1 Plugin Frameworks

All of the frameworks that we compare here offer the basic functionality that we need to extend the core bootware components, 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[2]	JSPF[3]	JPF[4]	OSGi[5]
functional	Dynamic Loading	✗	✗	✓	✓
	Security	✗	✗	✗	✓
non-functional	Used in SimTech	✗	✗	✗	✓
	Standard	(✓)	✗	✗	✓
	Complexity	low	low	medium	high
	Popularity	medium	low	medium	high
	Active Development	✓	✗	✗	✓

Table 8.1: Feature comparison of Java plugin frameworks

Table 8.1 shows a comparison of four Java plugin frameworks, the first of which is the Service Provider Interface (SPI)[6]. It is an extension mechanism integrated in Java which is a little more advanced then the manual extension mechanism described in section 6.4. 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.

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 8.1 on the left, it neither supports dynamic loading, nor security features and is therefore not a good fit for our needs.

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

The next contender is the Java Simple Plugin Framework (JSPF)[3], 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[7]. 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)[4] is another open-source plugin framework. Compared to JSPF it's a little more complex and popular. As we can see in Table 8.1, 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)[5], a plugin framework standard developed by the OSGi Alliance. It provides a general-purpose Java framework that supports the deployment of extensible bundles [24]. The right column of Table 8.1 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[8], Eclipse Equinox[9], and Knopflerfish[10]. 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.

8.3.2 PubSub Libraries

Many of the well know messaging middlewares offer support for PubSub, for example ActiveMQ[11], RabbitMQ[12], and ZeroMQ[13]. But, since we are looking for an internal communication

[7] https://code.google.com/p/jspf/wiki/FAQ
[8] http://felix.apache.org
[9] http://eclipse.org/equinox
[10] http://www.knopflerfish.org
[11] http://activemq.apache.org
[12] http://www.rabbitmq.com
[13] http://zeromq.org

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 8.2. 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 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 documenta-

tion that can give an indication of the usefulness of a library.

		PubSub Libraries				
		EventBus[14]	Guava Event Bus[15]	Simple Java Event Bus[16]	MBassador[17]	Mycila PubSub[18]
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 8.2: Feature comparison of Java PubSub libraries

The first library we look at is EventBus. As can be seen in Table 8.2 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 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

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

Table 8.2, 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.

8.4 State Machine Libraries

Since we want to implement the bootware process as a finite state machine, we must now decide how we will do it. It would certainly be possible to go with a hand made state machine implementation, but the time for this thesis is limited and we should use it for the actual design of the bootware. Therefore, it would be better to use an existing state machine library. In general, we are looking for an event-driven FSM, which allows us to define a set of states and transition between those states when specific events occur. Ideally we would prefer a standardized way to define the FSM and avoid proprietary formats. But we also don't want the FSM to be overly complex to use. Table 8.3 shows six state machine libraries available

for Java.

		State Machine Libraries					
		Commons SCXML[19]	EasyFlow[20]	SMC[21]	stateless4j[22]	squirrel-foundation[23]	Unimod[24]
functional	**Event Driven**	✓	✓	✓	✓	✓	✓
	Description Language	SCXML	Java	.sm	Java	Java, SCXML	UML, XML
non-func.	**Complexity**	med.	low	med.	low	low	high
	Popularity	med.	low	med.	low	med.	med.
	Maturity	low	med.	high	med.	med.	high
	Documentation	med.	low	high	low	high	high

Table 8.3: Feature comparison of Java state machine libraries.

Apache Commons SCXML[19] aims to be an java state machine engine that is capable of executing state machines defined as State Chart XML (SCXML). SCXML is a working draft specification for a general-purpose event-based state machine language that is currently being developed by the World Wide Web Consortium (W3C)[29]. Apache Commons SCXML looks like a good match for our needs, since it is event-based and also using a (soon to be) standard. But the current state of the implementation seems to be lacking since the SCXML specification has changed a lot. The most recent release is version 0.9, which was released in late 2008. It is to be replaced by version 2.0 that is currently being worked on and includes major changes, but a release date is not yet in sight [10].

EasyFlow[20] is a simple and lightweight FSM for Java. It is event-driven, but only supports describing the FSM directly in Java code. Compared to the other alternatives, it is not very well documented and not very popular. It would however be able to do the job.

State Machine Compiler (SMC)[21] is a state machine compiler that targets fifteen different

[19] http://commons.apache.org/proper/commons-scxml/
[20] https://github.com/Beh01der/EasyFlow
[21] http://smc.sourceforge.net/
[22] https://github.com/oxo42/stateless4j/
[23] https://github.com/hekailiang/squirrel
[24] http://unimod.sourceforge.net/

programming languages, including Java. It generates FSMs from a definition in *.sm* files. SMC is mature and has good documentation, but the use of an extra definition language and the extra step of compiling it into a Java representation seems to be to complicated for our needs.

Stateless4j[22] is a lightweight library for creating FSMs directly in Java code. Compared to the other alternatives, it lacks in documentation and doesn't seem to be very popular.

Squirrel-foundation[23] is a lightweight, flexible, and extensible FSM library for Java. Although relatively new, it is feature rich, well documented and relatively popular. It also supports some advanced features that might be useful. For example, it supports SCXML import and export.

Unimod[24] is a project that can create FSMs from UML descriptions created by an eclipse plugin. Unlike the other alternative, Unimod aims to create a unified methodology for application development and not just a library. This seems to be to complex for our needs.

8.5 Context

Listing 8.1 shows an exemplary context in XML form. As we can see in line 2-4, it is required to define the infrastructure, connection, and payload plugins that should be used during the bootstrapping process by supplying the name of the plugin *.jar*. It is also possible to specify a call payload plugin, as can be seen in line 6. This is optional and will only be used on the first request, when the remote bootware will also call OpenTOSCA to provision the workflow middleware. This is also where the service package reference in line 8 will be used, which points to the service package that should be provisioned by the provisioning engine called by the call payload plugin. In line 10-26 we can also see the optional credential list. If it is supplied in the context it will override credentials by the same name in the default credentials list that can be set with the setCredentials operation. If it is not supplied, the default credentials will be used.

context.xml

```
<context>
  <infrastructurePlugin>aws-ec2.jar</infrastructurePlugin>
  <connectionPlugin>ssh.jar</connectionPlugin>
  <payloadPlugin>opentosca.jar</payloadPlugin>
  <!--Optional:-->
  <callPayloadPlugin>callopentosca.jar</callPayloadPlugin>
  <!--Optional:-->
  <servicePackageReference>opentosca.csar</servicePackageReference>
  <!--Optional:-->
  <credentialsList>
    <entry>
      <key>AwsCredentials</key>
      <value>
        <credentials>
          <entry>
            <key>secretKey</key>
            <value>874w5zhpswe98tzhg0w87ser049tadsiph</value>
          </entry>
          <entry>
            <key>accessKey</key>
            <value>g9w276og9746gw5</value>
          </entry>
        </credentials>
      </value>
    </entry>
  </credentialsList>
</context>
```

Listing 8.1: Sample context represented in XML.

8.6 Web Service Interface

In section 6.3 we decided to use web service calls and returns as external communication mechanism and in section 6.7 we decided to pass along a context. Now, we need to the define the interface that will be made available by the web service to the outside. We obviously need the two main operations, deploy and undeploy, to be available from the outside. In section 6.7 we also described the *setCredentials* operation that has to be supported.

8.6.1 Deploy

The *deploy* operation will be called by at least two different components. Once by the bootware modeler plugin to deploy the remote bootware and the workflow middleware, and then each time the provisioning manager needs to provision a new service during a workflow execution. Listing 8.2 shows an exemplary deploy request as soap message. In line 6 we can see that the deploy method is called with the context provided as argument in line 7-11. In this particular example, only the required plugins are specified, which could be a call from the provisioning manager.

deploy-request.xml

```
1  <soapenv:Envelope
2    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3    xmlns:rem="http://remote.bootware.simtech.org/">
4    <soapenv:Header/>
5    <soapenv:Body>
6      <rem:deploy>
7        <context>
8          <infrastructurePlugin>aws-ec2.jar</infrastructurePlugin>
9          <connectionPlugin>ssh.jar</connectionPlugin>
10         <payloadPlugin>opentosca.jar</payloadPlugin>
11       </context>
12     </rem:deploy>
13   </soapenv:Body>
14 </soapenv:Envelope>
```

Listing 8.2: Sample deploy request in a soap message.

The response that is return once the request has been executed successfully is shown in Listing 8.3. It contains a list of endpoint references in line 5-10, which contains a reference to the payload that was deployed during the request, in this case OpenTOSCA.

If the deploy request somehow failed, a soap message containing a soap fault will be returned, which is shown in Listing 8.4. It contains a fault string with an error description in line 5, as well as the original DeployException that was thrown by the deploy operation in line 7-10.

deploy-response.xml

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:deployResponse xmlns:ns2="http://remote.bootware.simtech.org/">
      <return>
        <endpoints>
          <entry>
            <key>opentosca</key>
            <value>http://aws.com:8080/</value>
          </entry>
        </endpoints>
      </return>
    </ns2:deployResponse>
  </S:Body>
</S:Envelope>
```

Listing 8.3: Sample deploy response in a soap message.

deploy-error.xml

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
      <faultcode>S:Server</faultcode>
      <faultstring>infrastructureType cannot be empty</faultstring>
      <detail>
        <ns2:DeployException
          xmlns:ns2="http://remote.bootware.simtech.org/">
          <message>infrastructureType cannot be empty</message>
        </ns2:DeployException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

Listing 8.4: Sample deploy error in a soap message.

8.6.2 Undeploy

Like the *deploy* operation, the *undeploy* operation will be called by multiple components to reverse the actions that where previously made by deploy operations. Listing 8.5 shows an exemplary undeploy request in a soap message. As argument it contains one or more endpoint references to already deployed payloads, as can be seen in line 7-12.

undeploy-request.xml

```
1  <soapenv:Envelope
2    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
3    xmlns:rem="http://remote.bootware.simtech.org/">
4    <soapenv:Header/>
5    <soapenv:Body>
6      <rem:undeploy>
7        <endpoints>
8          <entry>
9            <key>opentosca</key>
10           <value>http://aws.com:8080/</value>
11         </entry>
12       </endpoints>
13     </rem:undeploy>
14   </soapenv:Body>
15 </soapenv:Envelope>
```

Listing 8.5: Sample undeploy request in a soap message.

When all payloads have been undeployed successfully, a response will be send, as shown in Listing 8.6. The response is empty since there is nothing interesting to return.

undeploy-response.xml

```
1 <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
2   <S:Body>
3     <ns2:undeployResponse
4       xmlns:ns2="http://remote.bootware.simtech.org/"/>
5   </S:Body>
6 </S:Envelope>
```

Listing 8.6: Sample undeploy response in a soap message.

In case of a failure, an error will be return. As can be seen in Listing 8.7, it has the same layout

as the error returned by the deploy operation. It contains a soap fault string in line 5 and the original UndeployException thrown by the undeploy operation in line 7-10.

undeploy-error.xml

```
1  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
2    <S:Body>
3      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
4        <faultcode>S:Server</faultcode>
5        <faultstring>Undeploy operation failed</faultstring>
6        <detail>
7          <ns2:UndeployException
8            xmlns:ns2="http://remote.bootware.simtech.org/">
9            <message>Undeploy operation failed</message>
10         </ns2:UndeployException>
11       </detail>
12     </S:Fault>
13   </S:Body>
14 </S:Envelope>
```

Listing 8.7: Sample undeploy error in a soap message.

8.6.3 SetCredentials

In addition to the main deploy and undeploy operations, the bootware web service also supports the *setCredentials* operation. Using this operation, login credentials can be set independently from deploy requests if necessary. Listing 8.8 shows an exemplary set credentials request. In line 7-23, it contains a credentials list, which can contain one or more credentials set. Each credentials set is made up of one or more credentials, which are key value pairs, where the key describes the credential type and the value the actual credential. In the example code in line 9, we send one credentials set for AWS, which consists of two credentials, a secret key in line 12-15 and an accessKey in line 16-19.

If the setCredentials operation was successful, the response in Listing 8.9 will be returned. Again, it is empty, since there is nothing interesting to return.

Like the deploy and undeploy operations, the setCredentials operation also returns an error message if the operation failed. As can be seen in Listing 8.10, it also contains a soap fault

setCredentials-request.xml

```
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:rem="http://remote.bootware.simtech.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <rem:setCredentials>
      <credentialsList>
        <entry>
          <key>AwsCredentials</key>
          <value>
            <credentials>
              <entry>
                <key>secretKey</key>
                <value>874w5zhpswe98tzhg0w87ser049tadsiph</value>
              </entry>
              <entry>
                <key>accessKey</key>
                <value>g9w276og9746gw5</value>
              </entry>
            </credentials>
          </value>
        </entry>
      </credentialsList>
    </rem:setCredentials>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 8.8: Sample setCredentials request in a soap message.

setCredentials-response.xml

```
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Body>
    <ns2:setCredentialsResponse
      xmlns:ns2="http://remote.bootware.simtech.org/"/>
  </S:Body>
</S:Envelope>
```

Listing 8.9: Sample setCredentials response in a soap message.

string in line 5 and the original SetCredentialsException thrown by the setCredentials operation in line 7-10.

setCredentials-error.xml

```
1  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
2    <S:Body>
3      <S:Fault xmlns:ns4="http://www.w3.org/2003/05/soap-envelope">
4        <faultcode>S:Server</faultcode>
5        <faultstring>Credentials could not be set</faultstring>
6        <detail>
7          <ns2:SetCredentialsException
8            xmlns:ns2="http://remote.bootware.simtech.org/">
9            <message>Credentials could not be set</message>
10         </ns2:SetCredentialsException>
11       </detail>
12     </S:Fault>
13   </S:Body>
14 </S:Envelope>
```

Listing 8.10: Sample setCredentials error in a soap message.

9 Future Work

With this diploma thesis we created a foundation which, while functional right now, might still need more work to become fully functional and useful in real working environment. In this chapter we present some opportunities for future improvements.

9.1 More Plugins and a Plugin Repository

For this diploma thesis we only implemented a few plugins. The plugin selection will certainly have to be extended in the future to cover a wider range of cloud providers (or other infrastructure types), connection mechanisms, and payloads. Along with a greater variety of plugins, a plugin repository, as described in subsection 6.4.2 would be beneficial. It would further decrease code duplication and facilitate plugin sharing. For this, a fitting repository format would have to be found and various other questions, such as security, need to be answered. On the implementation side, the integration of a plugin repository should be fairly straight forward. A mechanism to synchronize the local plugin directory with the repository has to be implemented and executed before the plugins are loaded. The code for loading plugins that is in place now doesn't necessarily need to be changed for this.

9.2 Secure Communication

As we already mentioned in section 6.3, it is necessary to secure the communication with the bootware, since it contains sensitive login information that should not be publicly accessible. For this, the communication has to be encrypted, which can be done by using the WS-Security SOAP extension for the web service communication. As part of this work, it could also make sense to investigate other possible security enhancements to the bootware components.

9.3 Better SimTech Modeler Integration

The integration of the bootware with the SimTech Modeler using the Bootware Plugin can also be extended in the future. The current integration is fairly minimal to support the most basic functionality. Improvements could be made to give the user more feedback on the provisioning progress. Additionally, a more intuitive way to configure the bootware could be implemented, for example with a graphical configuration interface that allows for the selection of plugins and login credentials.

9.4 Better Failure Management

Currently, the bootware will fail in many cases where it could continue, if the user could influence error recovery. For example, if for some reason a connection can't be established with a cloud provider, the bootware will abort and undeploy already provisioned payloads. This could happen in the middle of a workflow execution, where multiple services are deployed in different clouds. In this scenario, the ability for the user to select an alternative cloud provider could enable the bootware to continue instead of aborting, which would in turn allow the workflow execution to finish, instead of failing. Failure management mechanisms such as this would improve the usability of the bootware.

10 Summary and Conclusion

In this diploma thesis we presented a design for bootware system that is able to deploy various provisioning engines as well as a workflow middleware into remote environments, on-demand and fully automatic. Starting from previous work, we compared possible architecture alternatives and selected a 2-tiered architecture. We made this architecture extensible via plugins and integrated a web service interface.

List of Figures

List of Tables

List of Listings

Bibliography

[1] *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/.

[2] 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.

[3] Paul E. Black. *Dictionary of Algorithms and Data Structures: alphabet*. NIST. Dec. 2004. URL: http://xlinux.nist.gov/dads//HTML/alphabet.html.

[4] Paul E. Black. *Dictionary of Algorithms and Data Structures: deterministic finite state machine*. NIST. Dec. 2005. URL: http : / / xlinux . nist . gov / dads / /HTML / determFinitStateMach.html.

[5] Paul E. Black. *Dictionary of Algorithms and Data Structures: finite state machine*. NIST. Aug. 2013. URL: http://xlinux.nist.gov/dads//HTML/finiteStateMachine.html.

[6] Paul E. Black. *Dictionary of Algorithms and Data Structures: state*. NIST. Dec. 2004. URL: http://xlinux.nist.gov/dads//HTML/state.html.

[7] Paul E. Black. *Dictionary of Algorithms and Data Structures: transition function*. NIST. Dec. 2004. URL: http://xlinux.nist.gov/dads//HTML/transitionfn.html.

[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] *Commons SCXML 2.0 Roadmap*. The Apache Software Foundation. URL: http://commons.apache.org/proper/commons-scxml/roadmap.html.

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

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

[13] 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.

[14] *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.

[15] Christoph Fehling and Frank Leymann. *Cloud Computing*. Version 5. Gabler Wirtschaftslexikon. URL: http://wirtschaftslexikon.gabler.de/Archiv/1020864/cloud-computing-v5.html.

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

[17] David Hollingsworth. *The Workflow Reference Model*. Tech. rep. Workflow Management Coalition, Jan. 1995. URL: http://www.wfmc.org/standards/docs/tc003v11.pdf.

[18] *Human Users in Simulation Workflows*. SRC Simulation Technology. URL: http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/project_humanusers.php.

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

[20] Johannes Kirschnick 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.

[21] 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.

[22] 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.

[23] *Modelling of Simulation Workflows*. SRC Simulation Technology. URL: http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/project_modeling.php.

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

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

[26] *Runtime for Simulation Workflows*. SRC Simulation Technology. URL: http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/project_execution.php.

[27] 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.

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

[29] *State Chart XML (SCXML): State Machine Notation for Control Abstraction*. Tech. rep. W3C, May 2014. URL: http://www.w3.org/TR/2014/WD-scxml-20140529/.

[30] *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Tech. rep. OASIS, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf.

[31] 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.

[32] 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: ???.

[33] Karolina Vukojevic-Haupt et al. “Service Selection for On-demand Provisioned Services”. In: 18th IEEE International Enterprise Distributed Object Computing Conference, EDOC. 2014, accepted for publication.

[34] *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.

[35] Matthias Wieland et al. “Towards Reference Passing in Web Service and Workflow-Based Applications”. In: 13th IEEE International Enterprise Distributed Object Computing Conference, EDOC. 2009, pp. 109–118. URL: http://www.iaas.uni-stuttgart.de/institut/ehemalige/schumm/INPROC-2009-52%20-%20Towards%20Reference%20Passing%20in%20Web%20Service%20and%20Workflow-based%20Applications%20-%20Authors%20Postprint.pdf.

[36] 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 June 19, 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, June 19, 2014, ...

(Lukas Reinfurt)