Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Diplomarbeit Nr. 3616

Bootstrapping Provisioning Engines for On-demand Provisioning in Cloud Environments

Lukas Reinfurt

Studiengang: Informatik

Prüfer/in: Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Betreuer/in: Dipl.-Inf., Dipl.-Wirt. Ing.(FH) Karolina Vukojevic
Beginn am: 08.01.2014
Beendet am: 08.07.2014

CR-Nummer: 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. Making services available on-demand would be better suited in those situations. 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 designed to be generic and can be used together with other systems.

Table of Contents

List of Abbreviations

AMI	Amazon Machine Image, page 13
APIs	application programming interfaces, page 14
AWS	Amazon Web Services, page 16
BPEL	Business Process Execution Language, page 13
BPMN	Business Process Modeling Notation, page 13
CLI	command-line interface, page 72
CORBA	Common Object Request Broker Architecture, page 43
CSAR	Cloud Service Archive, page 14
EC2	Elastic Compute Cloud, page 16
FSM	Finite State Machine, page 71
IaaS	Infrastructure as a Service, page 15
JPF	Java Plugin Framework, page 91
JRE	Java Runtime Environment, page 91
JSPF	Java Simple Plugin Framework, page 91
NIST	National Institute of Standards and Technology, page 15
OASIS	Organization for the Advancement of Structured Information Standards, page 12
ODE-PGF	ODE Pluggable Framework, page 22
OMG	Object Management Group, page 43
OS	Operating System, page 15
OSGi	Open Service Gateway initiative, page 91

PaaS	Platform as a Service, page 15
PCI	Payment Card Industry, page 20
PubSub	publish-subscribe pattern, page 65
RDC	remote desktop connection, page 50
REST	Representational State Transfer, page 43
RMI	Remote Method Invocation, page 43
SaaS	Software as a Service, page 15
SCXML	State Chart XML, page 94
SDKs	Software Development Kits, page 17
SIMPL	Simulation Data Management System, page 22
SimTech	Simulation Technology, page 21
SimTech Modeler	SimTech Workflow Modeling & Monitoring Tool, page 22
SLA	Service Level Agreement, page 15
SMC	State Machine Compiler, page 94
SOA	Service Oriented Architecture, page 18
SOAP	Simple Object Access Protocol, page 43
SPI	Service Provider Interface, page 91
SSH	Secure Shell, page 11
SWfMSs	scientific workflow management systems, page 21
TOSCA	Topology and Orchestration Specification for Cloud Applications, page 12
UDDI	Universal Description, Discovery, and Integration, page 19
VMs	virtual machines, page 10
VPN	virtual private network, page 53
W3C	World Wide Web Consortium, page 94
WSDL	Web Service Description Language, page 19

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 are not reasonable anymore. One of these assumptions is that services should run continuously. This made sense in large enterprises where those services are used every day. 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. To provision those service, provisioning engines might be used, but these also have to be set up first. This creates the need for a bootstrapping mechanism that can deploy provisioning engines when 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 automatically and on-demand in cloud environments. 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] [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

We begin with an introduction to some fundamental topics in Chapter 2. First, we explain bootstrapping, followed by a general overview of provisioning with some details on TOSCA[3] [34] and OpenTOSCA. We explain the concept of cloud computing and describe Amazon's cloud platform. We also present the basics of service oriented architecture, workflows, and workflow management systems. Finally, we describe the SimTech project as well as the SimTech SWfMS.

In Chapter 3 we present previous work on the subject of this diploma thesis. First, we summarize the paper that build the foundation of this diploma thesis. Then, we discuss a previous diploma thesis that extended parts of this paper. In Chapter 4 we also present some related work. We list the requirements that were given for this diploma thesis in Chapter 5. We also explain some additional constraints that we introduced.

We present the design of the bootware in Chapter 6. First, we discuss component division, followed by the integration into existing modeler applications. Next, we select an external communication mechanism. We describe the extensibility mechanism, followed by the different kinds of plugins. We also discuss the event system, the context object, the web service interface, and the instance store. Then, we describe the execution flow and the use of finite state machines, before the final bootware architecture is presented. We also present a step by step description of the whole bootstrapping process in Chapter 7.

In Chapter 8 we present details on the implementation of the bootware. We describe the integration into the SimTech Modeler with an Eclipse plugin. We also explain the bootware core library. Then, we select the plugin framework, publish subscribe library, and state machine library that we will use for the implementation. We also describe the context object and the web service operations. Then, we give an overview over some plugins we implemented. In Chapter 9 we list some possibilities for future improvement. We summarize the previous chapters in Chapter 10, before presenting a conclusion.

[3] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`

2 Fundamentals

This chapter starts with a short description of bootstrapping, followed by an introduction into provisioning. Then, we provide a short overview of the cloud landscape, with focus on Amazon's cloud offerings because these are used in this diploma thesis. We also introduce service oriented architecture and explain workflows and workflow management systems. We finish with an overview of the SimTech project[1], of which this diploma thesis is a part of.

2.1 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 [42]. 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" [21]. 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].

[1] `http://www.iaas.uni-stuttgart.de/forschung/projects/simtech/`

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[2]. 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.

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.

2.2 Provisioning

This section provides an overview of provisioning in the context of computing. First, we present a general introduction and describe some of the provisioning solutions available today. Then, we focus in particular on TOSCA and OpenTOSCA because those are used in the prototypical implementation later on.

2.2.1 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, but their core functionality is 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]. Because we are working in a cloud environment, we will not have to deal with hardware directly, but rather with virtual machines (VMs). So for us, provisioning means the creation and deletion of VMs in a cloud environment, as well as the installation, configuration, monitoring, running and stopping of software on these VMs [23].

There are many benefits to using an automated provisioning solution instead of doing the provisioning by hand. The manual approach is limited by how much work a single person can

[2]`http://venturebeat.com/2008/11/20/the-art-of-the-bootstrap/`

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 time and money compared to a manual approach. As 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, because the human factor is largely replaced by automation. 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[3], 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[4], Chef[5], Puppet[6], and TOSCA[7], 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 Amazon's 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 Secure Shell (SSH) and therefore does not require additional software on the nodes that are configured. The solutions also differ in modularity and flexibility. While Ansible, Chef, Puppet, and TOSCA are highly flexible and can be used in a fine grained modular fashion, this also makes them more complex to use, for example compared to AWS CloudFormation.

[3]`http://aws.amazon.com/cloudformation`
[4]`http://www.ansible.com`
[5]`http://www.getchef.com/chef`
[6]`http://puppetlabs.com/`
[7]`https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`

2.2.2 TOSCA

Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standard created by the Organization for the Advancement of Structured Information Standards (OASIS)[8] [34]. Its development is also supported by various industry partners, which include IBM, Cisco, SAP, HP and others. Its aim is to provide a language that can describe service components and their relations in a cloud environment independent fashion. The following description is based on version 1.0 of the specification [34].

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 2.1 shows such a definitions document. Aside from the actual service template, shown on the left, it also contains a number of type definitions and some templates based on those definitions. These definitions and templates can also be imported from a separate definitions document.

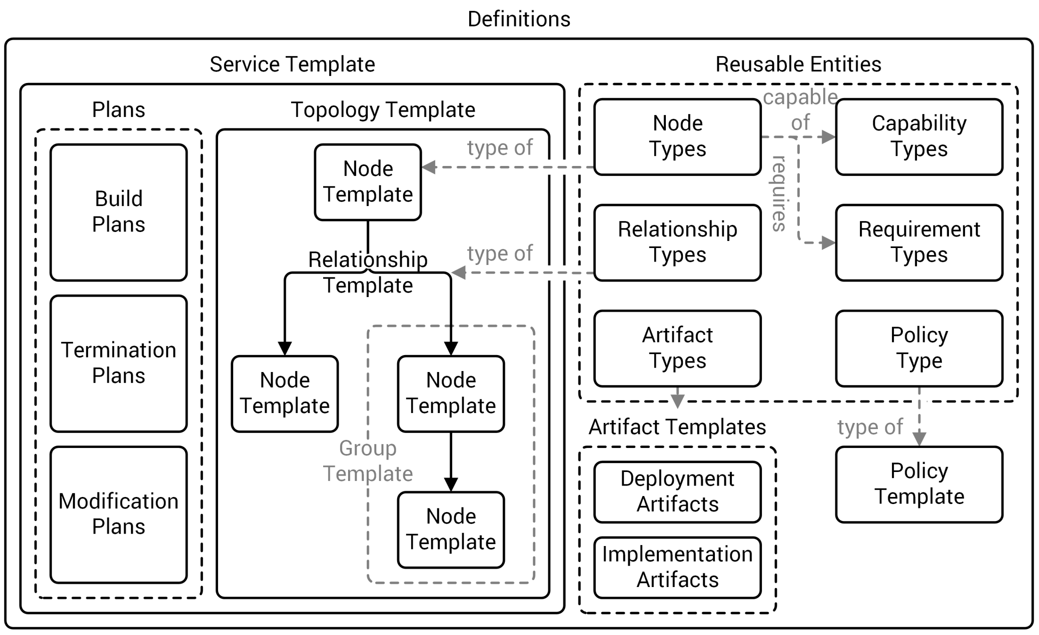

Figure 2.1: TOSCA definitions structure [based on 34].

The service template consists of two parts: A topology template and plans. Topology templates, as seen in the center of Figure 2.1, model the structure of a service and the middleware and infrastructure supporting it as a directed graph. The vertices of the graph represent

[8]`https://www.oasis-open.org/`

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 2.1. They define the properties of a component, as well as operations to manipulate a component, so called interfaces. Additionally, node types can be annotated with requirements and capabilities. These, in turn, are defined by requirement and capability types, which also belong to the group of reusable entities. This allows for requirement and capability matching between different components. The edges of the graph represent connections between nodes, which are defined by relationship templates that specify the properties of the relation. An example for such a connection would be a node A, representing a web service that 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 2.1, are used to manage the service that is defined by the service template. TOSCA distinguishes between three types of plans: Build plans, termination plans, and modification plans. Build plans describe how instances of a service are created. Termination plans describe how such a service is removed. Modification plans manage a service during its runtime. These plans consist of one or more tasks, i.e. an operation on a node (via an interface) or an external service call, and the order in which these tasks should be performed. They can be written in a process description language like Business Process Execution Language (BPEL)[9] or Business Process Modeling Notation (BPMN)[10].

The bottom right of Figure 2.1 shows artifact templates which represent artifacts. Artifacts are things that can be executed directly (e.g.: scripts, archives) or indirectly (e.g.: URL, ports). TOSCA further distinguishes between two types of artifacts, namely deployment and implementation artifacts. Deployment artifacts materialize instances of a node and are used by a build plan to create a service. An example for this is an Amazon Machine Image (AMI) which creates an Apache server once deployed in a VM. Implementation artifacts 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 2.1 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 a node type for an application server that expresses that it supports high availability by referencing a matching policy template.

[9] `http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html`
[10] `http://www.bpmn.org/`

One or more TOSCA definitions are packaged, together with some metadata 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.

2.2.3 OpenTOSCA

OpenTOSCA is a browser based open-source implementation of a TOSCA container, created at the IAAS at the University of Stuttgart, which supports the execution of TOSCA CSAR archives [2]. Figure 2.2 shows the architecture of OpenTOSCA. Its functionality is realized in three main components, which are the Controller, the Implementation Artifact Engine, and the Plan Engine. After a CSAR is uploaded to OpenTOSCA it can be deployed in three steps. In the first step, the CSAR file is unpacked and its content is stored for further use. The TOSCA XML files are then loaded and processed by the Controller. The Controller in turn calls the Implementation Artifact Engine and the Plan Engine. The Implementation Artifact Engine knows how to deploy and store the provided implementation artifacts via plugins. Plans are then run by the Plan Engine, which also uses plugins to support different plan formats. OpenTOSCA also offers two application programming interfaces (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].

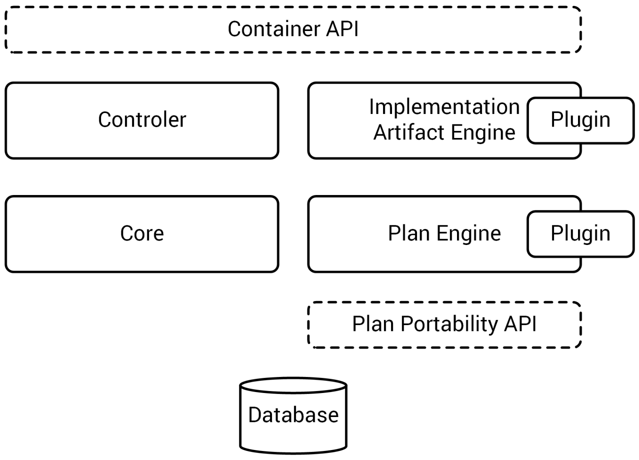

Figure 2.2: OpenTOSCA architecture [based on 2].

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. Because 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."[11] [35]

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." [25]

Cloud services can be categorized into different cloud service models, according to what exactly each service encompasses [28]. Figure 2.3 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 they want 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.

[11] 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." [32]

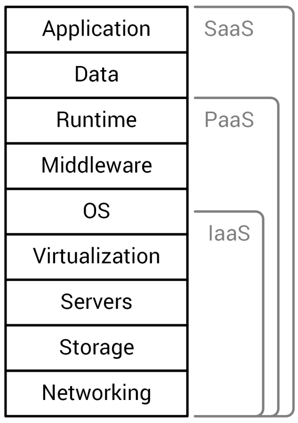

Figure 2.3: Cloud service models [based on 16].

Today, there are many different cloud providers offering a huge selection of services. The range of providers spans from large corporations like Amazon[12], Google[13], Microsoft[14], and IBM[15] to small, focused providers like Heroku[16] or Jelastic[17] and even solutions to build own clouds, like OpenStack[18]. The next section describes Amazon's cloud services in more detail because 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)[12]. 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 diploma thesis is Elastic Compute Cloud (EC2)[19], Amazon's 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

[12]`http://aws.amazon.com`
[13]`https://cloud.google.com`
[14]`http://azure.microsoft.com`
[15]`http://www.ibm.com/cloud-computing`
[16]`https://www.heroku.com`
[17]`http://jelastic.com`
[18]`https://www.openstack.org`
[19]`http://aws.amazon.com/ec2`

of specialized instances which are optimized for a specific purpose[20]. These include instances optimized for computation performance (C3), memory-intensive applications (R3), or high storage instances (I2). For this diploma thesis we will be using Amazon's low cost micro instances (T1).

Also of interest to this diploma thesis is Elastic Beanstalk[21], Amazon's 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[22], 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[23]. In this diploma thesis, we will use the AWS SDK for Java[24] to interact with Amazon's cloud resources programmatically.

2.4 Service Oriented Architecture

In highly dynamic markets, companies must be flexible and adapt their business processes quickly to changing environments. This often includes cooperating or merging with other businesses, business process optimization, or outsourcing. There have been distributed system technologies in the past that were created to support such dynamic processes on an IT level, but their tight coupling and lack of interoperability resulted in islands of middleware and corresponding application. The integration between those islands became a new problem that was solved with message oriented middleware [40].

Message oriented middleware enables integration of applications by wrapping them in adapters. These adapters are connected with channels which pass along messages. Channels can ensure a certain quality-of-service, such as exactly-once delivery. They also can change the messages in other ways, for example by transforming them between different formats. This allows for loosely coupled communication because format changes do not

[20]http://aws.amazon.com/ec2/instance-types/
[21]http://aws.amazon.com/elasticbeanstalk
[22]http://docs.aws.amazon.com/awsconsolehelpdocs/latest/gsg/getting-started.html
[23]https://aws.amazon.com/tools/
[24]https://aws.amazon.com/sdkforjava

affect the ability for two applications to integrate. The underlying integration middleware can also offer more advanced message exchange patterns, such as asynchronous send and receive or send-and-forget, which further helps with loosely coupled interaction [40].

Service Oriented Architecture (SOA) is an architecture paradigm that emerged as a result from the lessons learned from the failure of other distributed systems and the success of message-oriented middleware. It focuses on loose coupling and dynamic binding between services [40]. In this case, a service is “a logical representation of a repeatable business activity that has a specified outcome” [31]. Further characteristics of services are that they are self-contained, that they are composable, i.e. new services can be build by combining multiple other services, and that they are discoverable based on metadata that describes their various aspects. They also operate like black boxes to their consumers, i.e. no information of how they are implemented or provided is needed to use them [40].

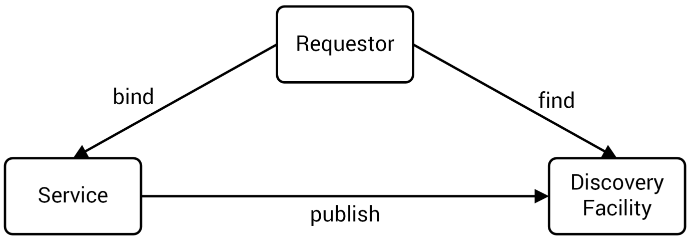

Figure 2.4: The SOA triangle [based on 40].

Figure 2.4 shows the basic principle behind SOA: The SOA triangle, made up of the bind/publish/find approach. First, a provider creates an abstract definition of a service that includes enough information to allow others to bind to this service. The provider then publishes metadata describing this service to a directory or registry. A requestor can then use the discovery facility associated with this registry to find services that fulfills his functional and non-functional requirements, based on the available metadata. After selecting a service, the requestor then retrieves the corresponding binding information, binds to this service, and starts sending requests to it [40].

To simplify this process for the requestor, a middleware called service bus is introduced, as shown in the middle of Figure 2.5. The requestor now sends the description of the service it intends to use and the data it intends to send to the service to the service bus. The service bus uses the description to find matching services with the discovery facility, selects one of them, retrieves the binding information and binds to the services. Then, if necessary, it transforms the data send by the requestor and sends a request to the service. The response it receives is passed back to the requestor, which now no longer has to deal with any of the above steps [40].

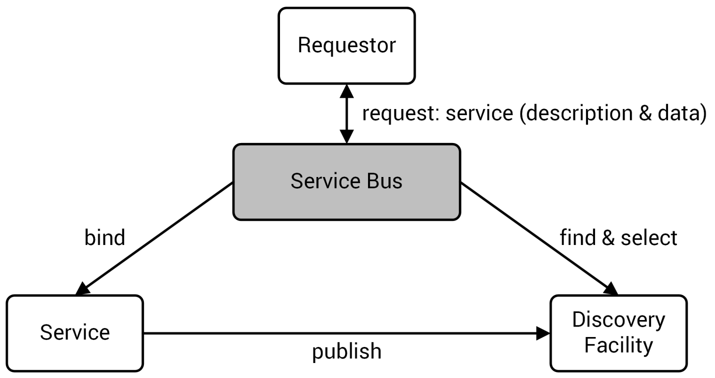

Figure 2.5: The SOA triangle including a service bus [based on 40].

Web service technology is one implementation of a service oriented architecture. It uses wrappers to hide implementation specific functionality and therefore allows applications with different programming models to interact with each other. To describe these wrappers, the standardized Web Service Description Language (WSDL) is used. WSDL describes the interfaces of the wrappers, which allows a requestor to use any wrapper implementing a particular interface, which creates technology abstraction. Additionally, quality-of-service descriptions and business-relevant data allow service selection based on business criteria, rather than IT criteria. This allows requestors to switch dynamically between providers offering identical services with little or no changes to the application, which creates provider abstraction. Universal Description, Discovery, and Integration (UDDI) can be used as a service registry where WSDLs of web services can be published and found [40].

A service bus is also at the center of this implementation. It combines a number of SOA capabilities, specified by numerous web service specifications, to offer the discovery, selection, and binding functionality described earlier. It can cope with various transport protocols and deal with both XML and non-XML messages. Quality of service is supported via policies and can include reliable messaging, security, and transaction capabilities. It also supports atomic services and composed services and provides features for service discovery and negotiation [40].

2.5 Workflows and Workflow Management Systems

Workflows and workflow management systems are another tool to increase the flexibility of businesses in times of change. Hollingsworth defines a workflow as “the computerised facilitation or automation of a business process, in whole or part” [19]. In other words, a workflow describes the tasks associated with a business process and the order in which these tasks are to be executed in such a way that they can be automated with the help of computers. Workflows are often visually represented as directed graphs, with vertices representing the tasks and edges defining the order of these tasks. Figure 2.6 shows such a graph that represents a simple business workflow. Note that the tasks can be a mixture of human and automated tasks. In Figure 2.6 for example, the *identify payment method*, *accept cash*, and *prepare package* tasks could be executed by humans, while the *process credit card* task would be handled by a computer program. Moreover, in combination with SOA, the *process credit card task* could be a call to an external service provided by an PCI[25] compliant business specialized in payment processing.

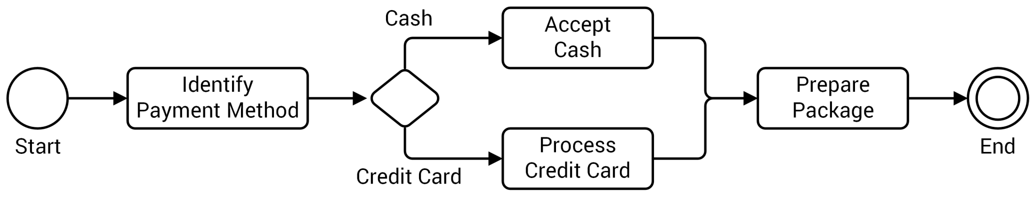

Figure 2.6: A simple workflow represented as a graph.

The automation of a workflow is handled by a workflow management system, which Hollingsworth defines as “a system that completely defines, manages and executes workflows through the execution of software whose order of execution is driven by a computer representation of the workflow logic” [19]. In other words, a workflow management system receives a workflow as input and then executes the actions associated with each workflow task in the particular order described by the workflow. For the example in Figure 2.6, this could mean that the workflow management system presents an employee with a graphical user interface, where they can select the payment method based on the choice of a customer. If the customer chooses to pay cash, the workflow management system would then show a dialog where the employee could enter this cash transaction. If the customer chooses to pay with a credit card, the workflow management system would call an external service with the credit card details to approve the transaction. In the final step, the workflow management system could assist the employee with preparing the package by automatically printing a label or displaying useful information.

In the past, workflows have been mainly applied in a business context, in particular for modeling and re-engineering of business processes. This lead to the development of business-

[25]Payment Card Industry (PCI): The PCI Security Standards Council developed a data security standard (PCI DSS)[26]to enhance the security of credit card information. Businesses handling credit card information are encouraged to comply with these requirements to prevent security breaches and improve trust. Because this can be a complicated process, outsourcing credit card handling can save resources.

[26]`https://www.pcisecuritystandards.org/documents/PCI_DSS_v3.pdf`

centered standards like BPEL or BPMN to describe workflows, and corresponding workflow management systems that can execute these workflows. One particular characteristic of these business workflows is that they are fairly static, i.e. they will not change during a workflow execution and only rarely in between due to business process re-engineering. This also means that the existing business workflow infrastructure is geared towards these static workflows [36].

In recent years however, new applications for the use of workflows have emerged, among them scientific workflows. These scientific workflows differ from business workflows in that they are much more dynamic and therefore require more flexibility in the tools supporting them. The reason for this is that the processes involved in scientific workflows are rarely completely know in advance. Exploration and trial and error often play a role, which can lead to unpredictable changes in the processes [36]. Existing workflow technology often cannot offer the required flexibility for scientific workflows, which is why modification of existing technology or creation of new technology is required to fully support scientific workflows. For this reason, scientific workflow management systems (SWfMSs) have been created, which offer scientist adequate support throughout the experimentation process.

2.6 SimTech

Since 2005, the German federal and state government have been running the Excellence Initiative[27], 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 projects 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 [15].

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*[28]. 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.

[27]`http://www.dfg.de/en/research_funding/programmes/excellence_initiative/index.html`
[28]`http://www.simtech.uni-stuttgart.de/forschung/pn/PN6/index.en.html`

2.6.1 SimTech SWfMS

As part of this project, the SimTech SWfMS was developed. It is a system that enables scientists to easily create, manage and execute simulation workflows which are a subcategory of scientific workflows [17]. The SimTech SWfMS introduces extensions to the BPEL language that add functionality to support the requirements of simulation workflows, such as passing data by reference to support larger amounts of data often found in science [also see 41], or shared context between workflows [26]. 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 [29]. 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 [17, 20].

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[29] and extends its functionality with various plugins. Figure 2.7 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, they click on a button to execute the workflow on the workflow middleware. The middleware consists of various components, some of which are shown in Figure 2.8. Most of them are executed by an application server, in this case Apache Tomcat[30]. The workflow is deployed on the workflow engine, in this case ODE Pluggable Framework (ODE-PGF)[31], which executes the workflow step by step. If a step involves the execution of a service, the ESB (Apache Service Mix[32]) is called, which resolves the services and passes along the request and the response. Further components include SimTech Auditing, which is used for auditing purposes. It is connected to the workflow engine via a messaging middleware (Apache ActiveMQ[33]). It is also connected to a database where it stores its data.

[29]`http://www.eclipse.org/ide/`
[30]`http://tomcat.apache.org/`
[31]`http://www.iaas.uni-stuttgart.de/forschung/projects/ODE-PGF/`
[32]`http://servicemix.apache.org/`
[33]`http://activemq.apache.org/`

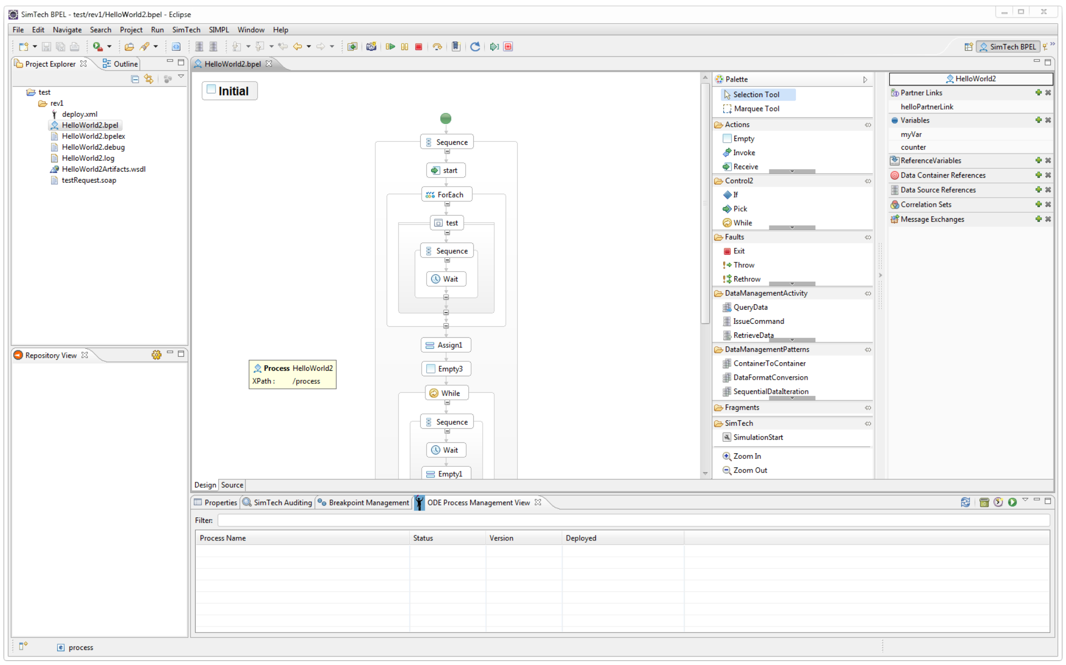

Figure 2.7: The SimTech Modeler user interface.

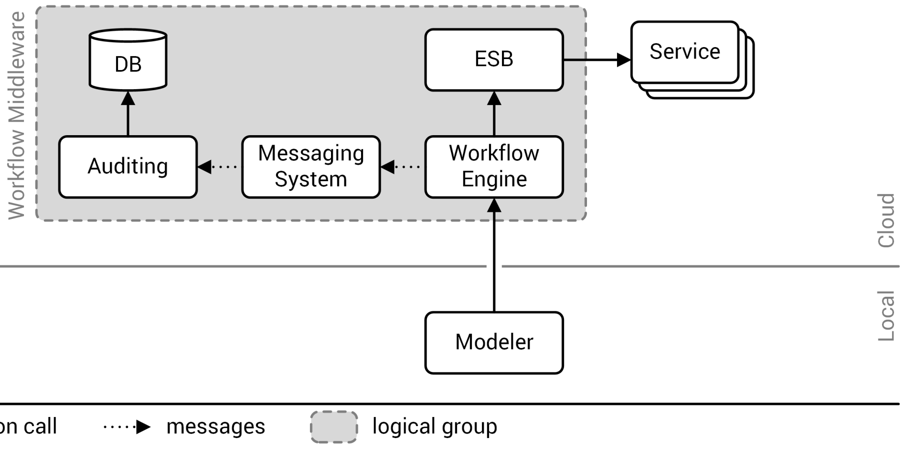

Figure 2.8: Some of the SimTech SWfMS components.

3 Previous Work

This chapter summarizes previous work on the subject of this diploma 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. identified requirements that need to be addressed to make the current approach used for scientific workflows more suitable for scientific simulation work [37]. 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 proposed 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 [37].

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

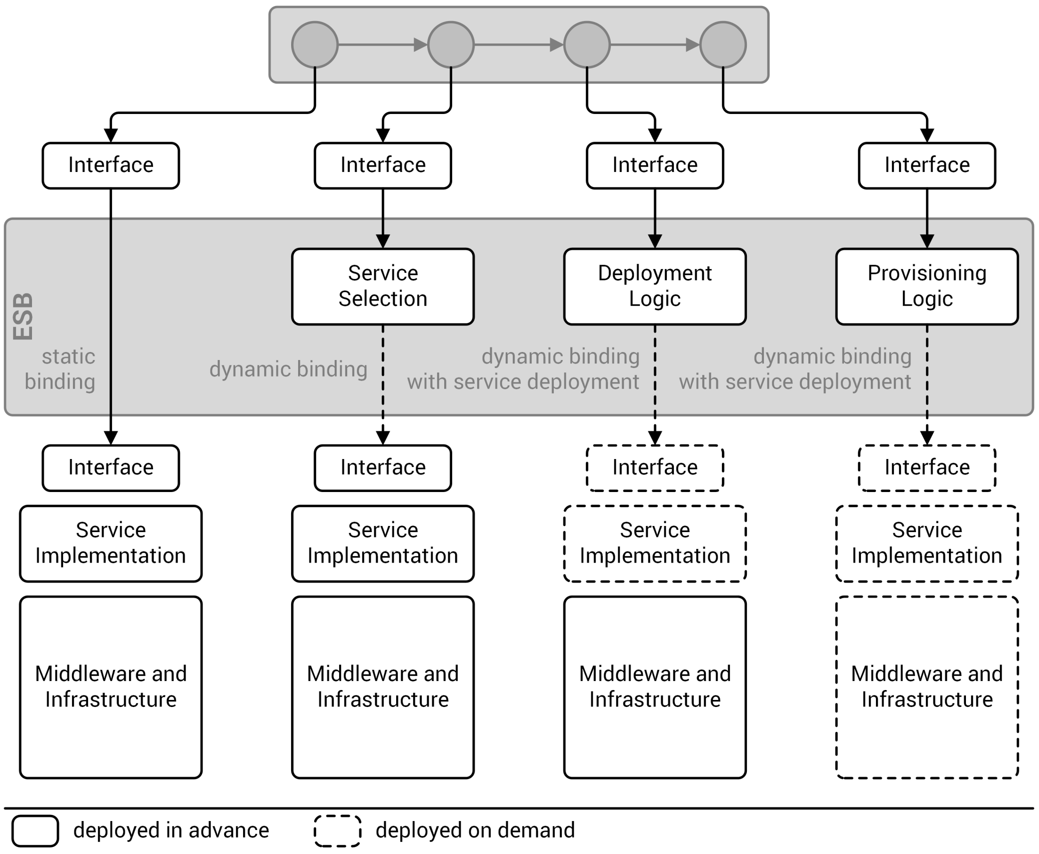

Figure 3.1: Simplified overview of service binding strategies [based on 37].

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 [37].

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 in 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 is not 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

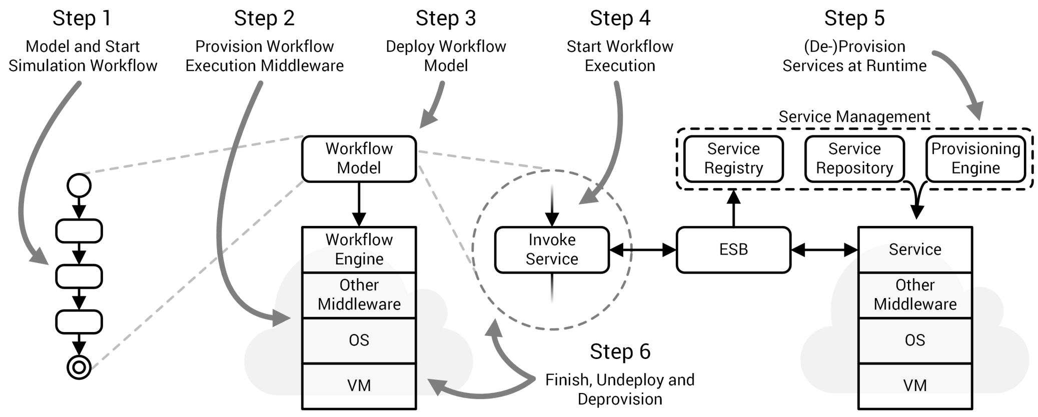

Figure 3.2: Steps during the on-demand provisioning of workflow execution middleware and simulation services [based on 37].

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 [37].

The architecture they present, shown in Figure 3.3, can be separated into a local part at the 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. 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 they start the execution of 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 (step 1), which in turn deploys the workflow middleware (step 2). 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 (step 3). This takes place in the third phase, the service runtime phase [37].

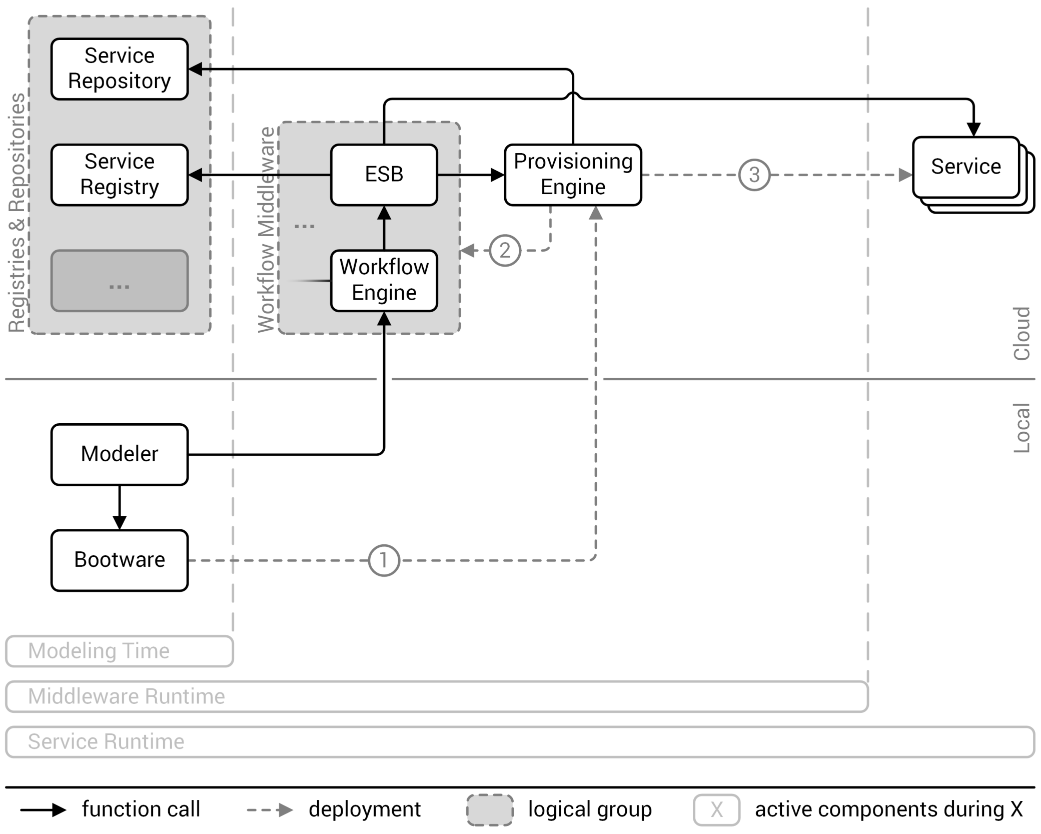

Figure 3.3: Proposed architecture [based on 37].

3.2 Dynamic Provisioning of Web Services for Simulation Workflows

Schneider found some problems with the architecture proposed in Section 3.1 [30]. 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 are not 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 [30].

Furthermore, it 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 is certainly not true for all of them.

This problem is further amplified because there are no standards defined for such a service repository [30].

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 is not the case [30].

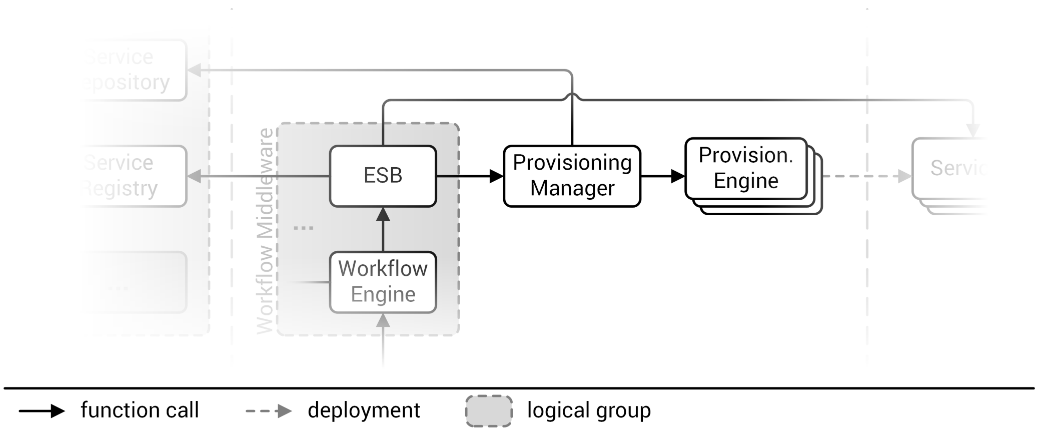

Figure 3.4: Extended architecture with added provisioning manager [based on 38].

Schneider further refines the previously shown middleware architecture by adding a provisioning manager as intermediary between the ESB and the provisioning engines [30]. 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. It 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 do not support [30].

4 Related Work

This chapter summarizes related work of other authors that is of interest to this diploma thesis. In general, there seems to be little work that is closely related to ours. Bootstrapping is a somewhat overloaded term that appears in many contexts, ranging from statistics to communication hardware, which are not related to our work. Several approaches for on-demand provisioning of services have been presented in [9], [13], and [23] that are somewhat related to our work. However, they do not rely on existing provisioning solutions and each only use one provisioning mechanism tailored to their specific situation. The provisioning mechanisms themselves are also not deployed on-demand, so they do not have the need for a bootstrapping procedure.

Chrysoulas et al. presented a dynamic service deployment (DSD) architecture for grid computing [9]. It handles service code retrieval, selects the installation location based on the result of a match making algorithm, and deploys the service at the selected location. Their approach relies on already existing resources in the grid and does not have to provision additional infrastructure or install middleware. It also does not use any existing provisioning solution for the service deployment.

Dörnemann et al. describe a solution for on-demand resource provisioning for BPEL workflow activities in Amazon's EC2 [13]. They introduce a load balancer component that is called by the workflow engine when a service call is made during a workflow execution. If there are not enough resources available to run the requested service, it can start new EC2 instances through an internal provisioner. The provisioner can also be extended to support other cloud provider via the use of external configuration files. Their approach is limited to starting and stopping preconfigured virtual machines that already contain all middleware necessary to run a service. It is not able to create arbitrary infrastructure topologies. They also do not handle the provisioning of the workflow middleware. They do not rely on already existing provisioning solutions and their provisioner is a fixed part of the load balancer.

Kirschnick et al. present an extensible architecture for automatic provisioning of cloud infrastructure and services at different cloud providers [23]. For this process they designed a so called service orchestrator which uses user defined service models, which describe the topology of a cloud service, to provision new cloud services and to trigger reconfiguration and topology changes of existing services. It has an abstraction layer that provides abstract methods to handle the management, installation, configuration, and starting of software via infrastructure, packages, applications, configuration, and VM connection managers. Similar to our work, their system is extensible to support different cloud providers, connection types, and application. They do not rely on any existing provisioning solutions but rather present a new one.

Regarding bootstrapping, Goehner et al. present the lightweight infrastructure-bootstrapping infrastructure (LIBI), an API specification and a reference implementation that can bootstrap processes in high-performance computing environments [18]. Here, it is necessary to start processes on many nodes and supply them with the initial information needed so that they can get into an execution ready state. LIBI delivers improved launch time over sequential or parent-creates-children approaches, which suffer from serialization bottlenecks. Their bootstrapping approach only has to work in an environment where all the infrastructure is already running, so they do not have to provision VMs or middleware.

Another diploma thesis that is worked on in parallel to this diploma thesis is designing the provisioning manager that was described in Section 3.2 [22]. It is the main user of the bootware system designed in this diploma thesis because it will deploy provisioning engines through the bootware on behalf of the workflow middleware. It also uses plugins to communicate with these provisioning engines. To avoid code duplication, libraries will be created that can be used by both the plugin manager plugins and the bootware plugins.

5 Requirements and Constraints

In this chapter we present 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 diploma thesis. Then, we describe additional constraints which we added to limit the scope of the work.

5.1 Requirements

The main goal of this diploma thesis is to lay a foundation by creating the core design of the bootware. 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 the bootware when needed. It is therefore a core requirement to keep the bootware 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 diploma 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 cannot be hidden completely, because the user has to specify a cloud provider and the corresponding log-in credentials somewhere. 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 frequent status updates.

A further requirement is that the bootware 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. It should also be easy to distribute and setup and it has to be able to run on an average personal computer.

5.2 Constraints

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 will not be finished with this diploma thesis. Other people will have to extend it in the future and because 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.

We also constrain the bootware usage to one bootware per user. We do not plan for multi-tenancy, i.e. multiple users using the same bootware. Additionally, we assume that a provisioning engine can be installed on a single computing resource, i.e. a single VM. If a particular provisioning engine requires a more complicated infrastructure topology, it cannot be provisioned by the bootware in its current form. 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.

6 Design

In this chapter we will develop the design of the bootware. This design is held intentionally abstract. Some specific implementation details will be described in Chapter 8. We will describe the component division, modeler integration, external communication, extensibility, and other aspects of the bootware system. We will also present a step by step description of the internal process during a bootstrapping operation, before presenting the final bootware architecture. But before we explain these details, we present a rough overview of what we want to accomplish with the bootware and how we plan to do it. Figure 6.1 shows an overview over the steps involved in the bootstrapping process.

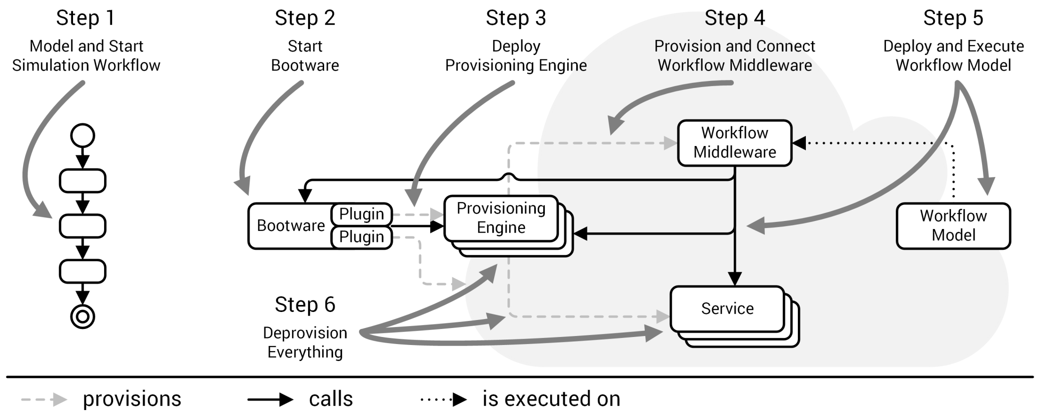

Figure 6.1: Overview over the steps involved in the bootstrapping process.

In the first step, a user creates a workflow in a modeler application. Now, they want to execute the workflow, for which they need some workflow middleware (i.e. a SWfMS), but at the moment, no workflow middleware is running. So first, the bootware is started to help with setting up this middleware, as shown in step two. The bootware can load various plugins that allow it to provision cloud resources and applications. In step three, it uses those plugins to create a cloud resource, for example a VM, and to deploy a provisioning engine on this resource. In the fourth step, the bootware tells this provisioning engine to provision the workflow middleware that is needed to execute the workflow. Then, it sets up the connection between the modeler and this middleware. Now that the workflow middleware is running and connected, the workflow can be deployed and executed on this middleware, which is shown in step five. During this workflow execution, various services might be called. These services might also not be available at this time, so the workflow middleware has to call provisioning engines to provision those services. These provisioning engines might also not exist, so the workflow middleware also calls the bootware to deploy the provisioning engines it needs to provision the services. When the workflow execution is finished, all services, the workflow middleware, the provisioning engines, the underlying cloud resources, and the bootware are deprovisioned in the sixth and final step.

To summarize, the bootware has to be able to provision cloud resources, provisioning engines, and the workflow middleware by using various plugins. It has to connect the local modeler to the workflow middleware and support the workflow middleware by deploying additional provisioning engines if needed. It also has to remove all resources once the workflow execution is finished. Now that we have a rough understanding of the bootware and the bootstrapping process, we can begin describing the various parts of its design in more detail.

6.1 Component Division

As described in Section 3.1, 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. 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 kick-starts the remote bootware, which then handles the actual provisioning of provisioning engines. The first question that has to be answered is whether this division is reasonable, or if another alternative makes more sense. We will now discuss the viability of four such alternatives.

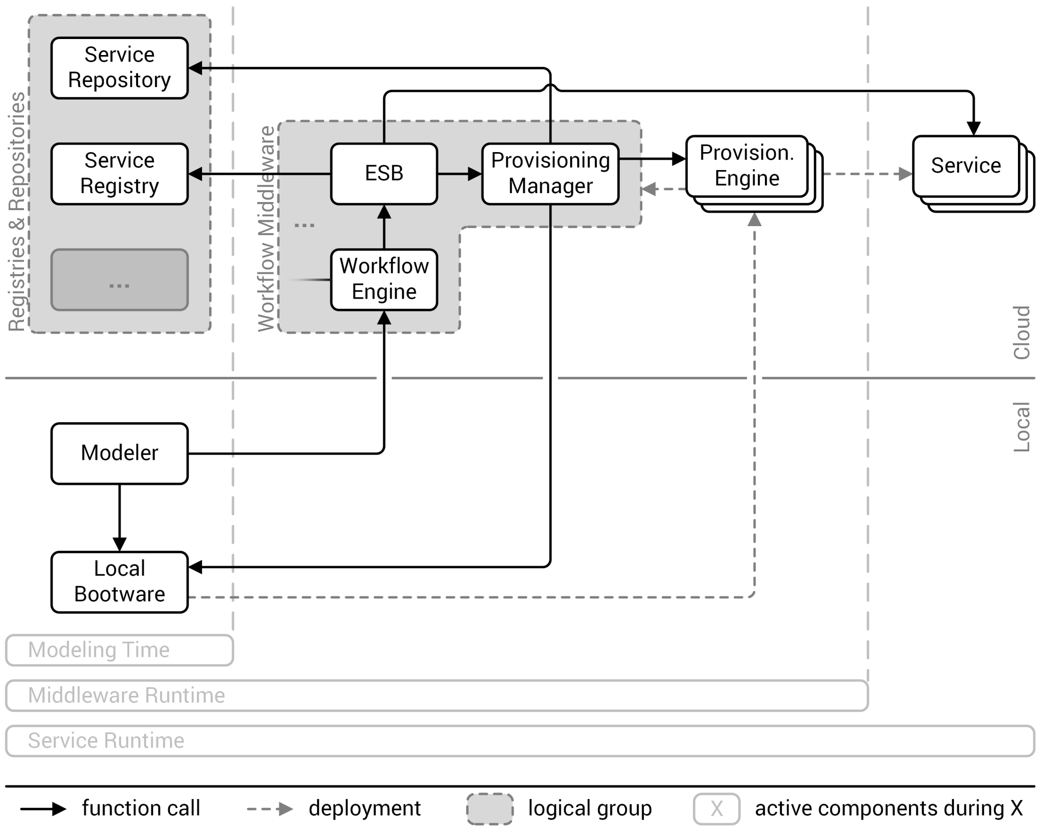

Figure 6.2: Simplified overview of the single local component architecture.

6.1.1 Single Local Component

First, we consider the simplest case: A single local bootware component as shown in Figure 6.2. In this scenario, all provisioning processes are initiated from a bootware installed locally on the users machine, alongside or as part of the workflow modeler.

The advantages of this architecture lie in its simplicity. Only one component has to be created and managed. We would not have to deal with bringing the bootware into a cloud environment and each user would have his own personal bootware instance, so multi-tenancy would not 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 does not have to be considered.

The disadvantages are caused by the component being local. Because 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 6.2 shows, all calls to the bootware from the provisioning manager would leave the remote environment. Also, all calls from the bootware 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. [24]. 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 [24].

6.1.2 Single Remote Component

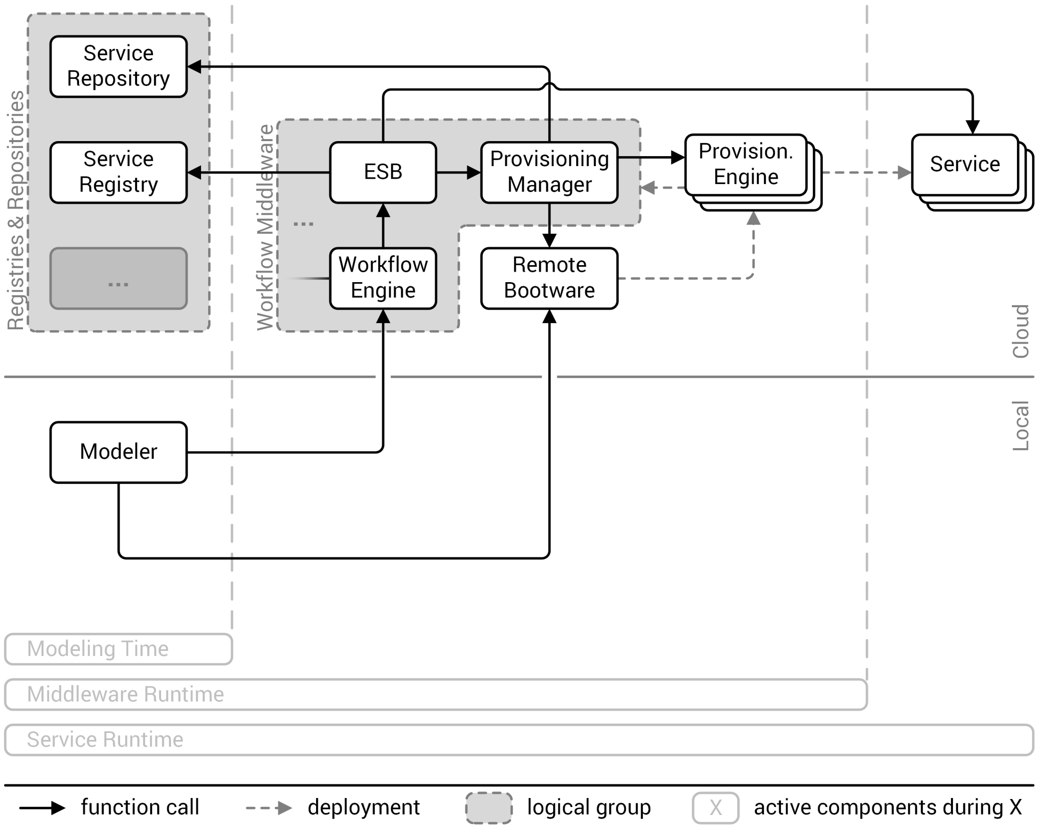

Figure 6.3: Simplified overview of the single remote component architecture.

The next obvious choice, as displayed in Figure 6.3, 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.

Because there are not 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 keeps this one instance running, or they provision the bootware once they need it and deprovisions it when they are done.

In the first case, the user would only have to provision the bootware once, but this creates a new problem: The user does not know where exactly to put the bootware. Because one requirement is that multiple cloud environments should be supported, it is possible that the bootware 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. While the other approaches presented here do not completely eliminate this problem, they at least have the option to move the bootware with each individual bootware execution, while in this first case, the bootware would stay in one place for multiple, possibly many bootware executions.

Another problem in this first case is that the bootware would be running all the time, even if the user does not need it, which would increase costs. This problem could be reduced 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 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 and possibly increase the running costs.

In the second case, the user would provision the bootware whenever they need it. Now the user would be able to pick a cloud environment that is close to the other components that they plan 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 they should not 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 6.4, 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.

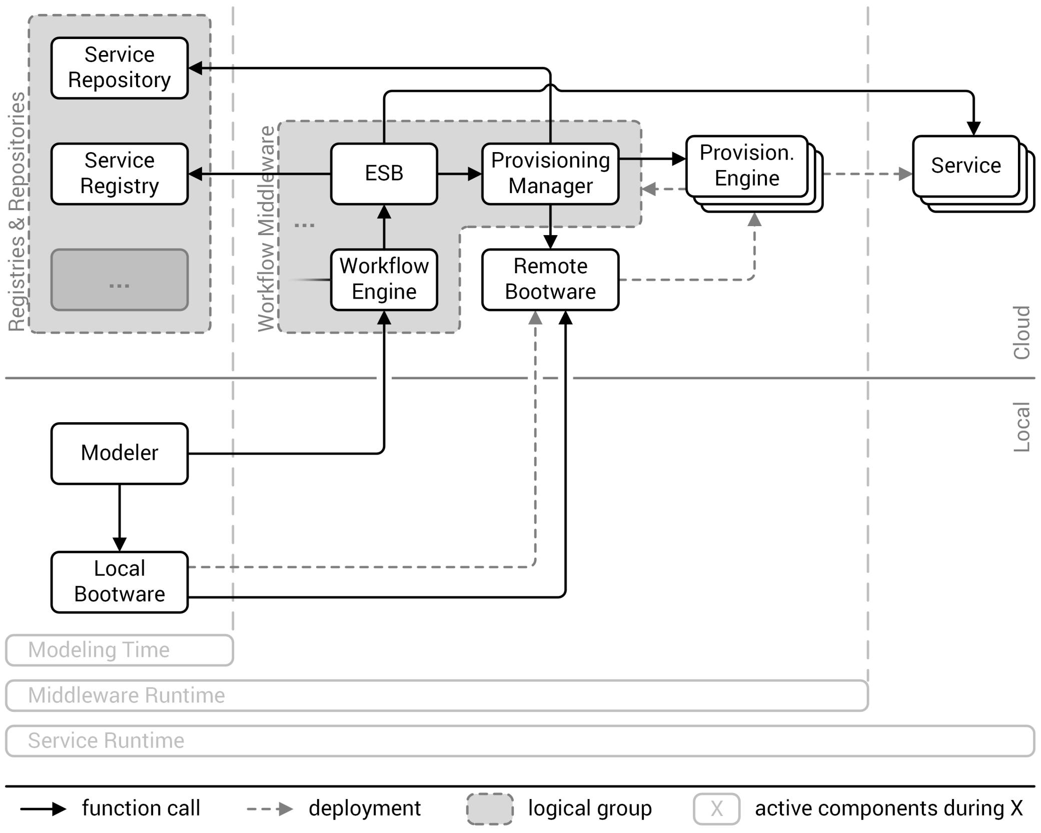

Figure 6.4: Simplified overview of the 2-tier architecture.

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, because the local bootware handles all that. Because 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.

But we also introduce new problems. For one, we now have duplicate functionality between the two components. Both have to know how to provision a component into multiple cloud environments. The local bootware has to be able to put its remote counterpart into any cloud environment. The remote bootware has to be able to provision other components into the same environment in which it runs (ideally, to minimize costs). Because it can be located in any cloud environment, it has to be able to do this in any cloud environment. Independent from this, it also has to be able to provision to any environment that the user or the 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 cannot 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

6.1.4 Cloning

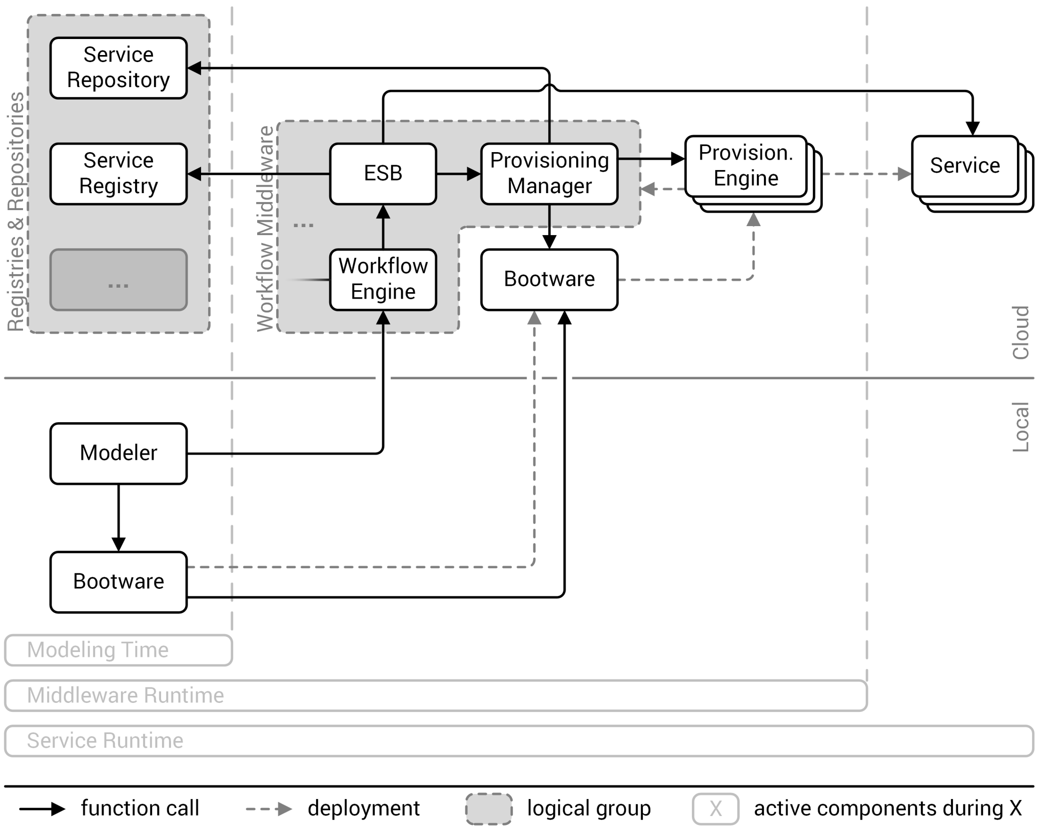

Figure 6.5: Simplified overview of the cloned component architecture.

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 6.5. 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 would therefore not 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 would not require for local operation and vice versa. However, because we want to keep the whole bootware, including the remote part, fairly lightweight, it is 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 cannot 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

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 are 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 not 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 6.4 is not a specific modeler and in theory it should be possible to use the bootware with any modeler (and any workflow middleware) without too much modification. So, by keeping the bootware as a separate generic component and only implementing a small, modeler specific adapter, we are able to support different environments without changing the core bootware components. We call this abstract concept the bootware adapter, as shown in Figure 6.6.

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 cannot say how exactly we can integrate in this process. Specific integration details for the modeler used in this diploma thesis, the SimTech Modeler, will be discussed in Section 8.1. We know however what needs to happen in the bootware adapter to get the bootstrapping process going.

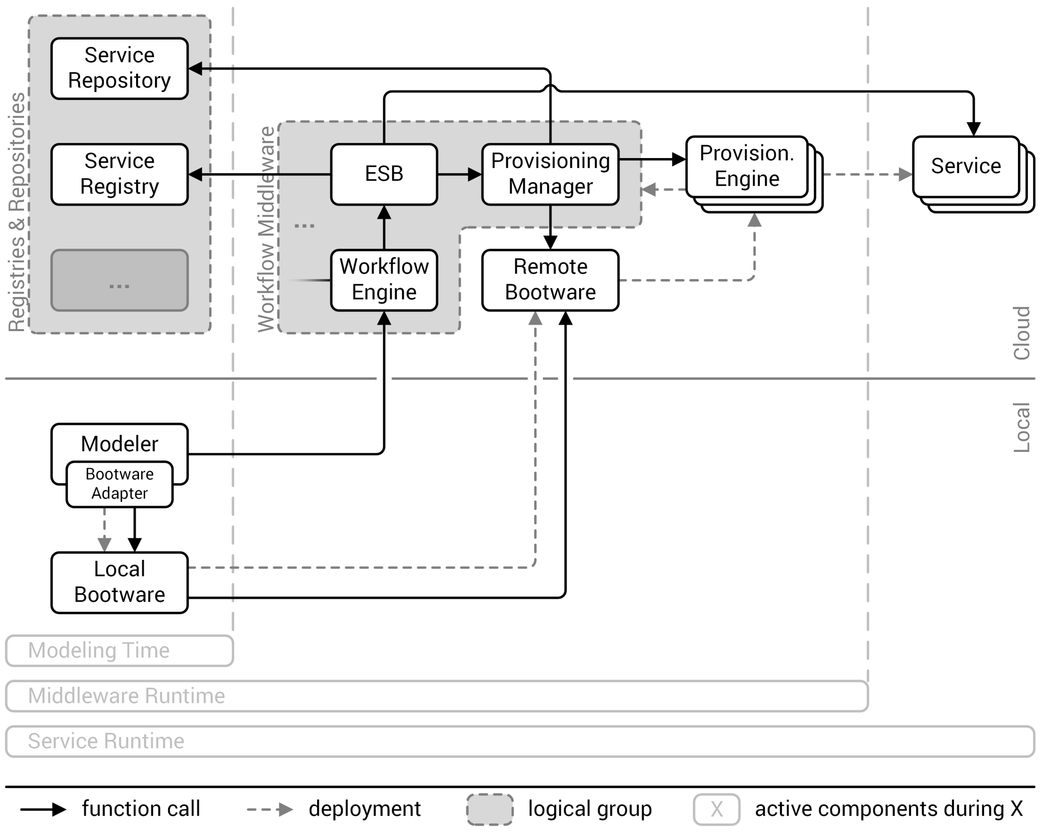

Figure 6.6: Modeler integration with a plugin.

First, the bootware adapter has to start the local bootware so that it will be in a state where it can receive and process requests. This is shown in Figure 6.6 as deployment operation from the bootware adapter to the local bootware and involves starting an executable and maybe passing along some sort of configuration file. Once the local bootware is running, the bootware adapter has to set up the context for the following requests. This includes telling the bootware configuration details, like the 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 information about the cloud provider and the provisioning engine that should be used, as well as the service package reference for the workflow middleware. This request is shown in Figure 6.6 as function call from the bootware adapter to the local bootware. The local bootware will take this information and provision the remote bootware, which in turn will deploy a provisioning engine in the specified cloud environment. This provisioning engine will then provision the workflow middleware. If successful, it returns a list with information concerning the workflow middleware, like endpoint references and other details, to the remote bootware, which passes it back to the local bootware. The local bootware passes it to the bootware adapter, which then has to set up the modeler with this information for the actual workflow deployment.

This is the minimal work the bootware adapter 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 from the bootware adapter to the local bootware in Figure 6.6 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 external communication mechanism will be implemented.

6.3 External Communication

In Section 6.2 we established that a bootware adapter 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 bootware adapter.

As the provisioning processes kicked off by the bootware can potentially take a long time to finish (in the range of minutes to hours), we face possible timeouts when using synchronous communication. As alternative we could use asynchronous communication with callbacks. This would avoid timeouts but also creates a new problem. The callback message send as response is separated from the original message and therefore appears as unsolicited message to the client. If the client rejects unsolicited messages, for example because it is located behind a firewall, the callback message might be blocked. This could be a problem because in the environment where the bootware will most likely be used, i.e. at universities, secure networks with firewalls are very common and asynchronous callbacks could therefore be problematic. Another solution is to use polling, i.e. after a request was sent to the bootware, the bootware is polled periodically for a response. This also avoids timeouts as well as the firewall problematic. Disadvantages of polling, for example when many clients poll a server at the same time and cause a bottleneck, will most likely not be a problem in our case, because we only have a very restricted number of clients for each bootware instance and no multi-tenancy.

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 diploma 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 objects in one Java VM to invoke methods on objects in another Java VM. Depending on the implementation, it can be used with polling or asynchronous callbacks. But because RMI is limited to Java and we might want to communicate with the bootware from a component written in another programming language, RMI does not 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 polling and asynchronous method invocation via callbacks [1], as well as transport layer encryption and other security features [10].

As a second alternative, we could communicate with messages by using message-oriented middleware. As explained earlier in Section 2.4, it supports communication between different components using adapters and channels. Asynchronous communication is supported by using message queues for temporary storage. The middleware can also provide additional persistent storage and backups for high availability [11]. It may also support security features like encryption. Another alternative are web services via Simple Object Access Protocol (SOAP) or Representational State Transfer (REST). Like CORBA, web services also support polling and asynchronous invocation, as well as security mechanisms [39].

As the whole SimTech SWfMS already uses SOAP based web services, it would make sense to also use SOAP based web services as external communication mechanism for the bootware. The technology and knowledge is already in place and introducing a second mechanism like CORBA would unnecessarily increase the complexity of the project, especially because CORBA does not offer any significant advantages over SOAP based web services. Using a message-oriented middleware would also be an option but introducing another component seems to complicated, especially because we do not need most of the features that it offers (e.g.: transactions, persistence, etc.). Figure 6.7 shows the addition of web service call and

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

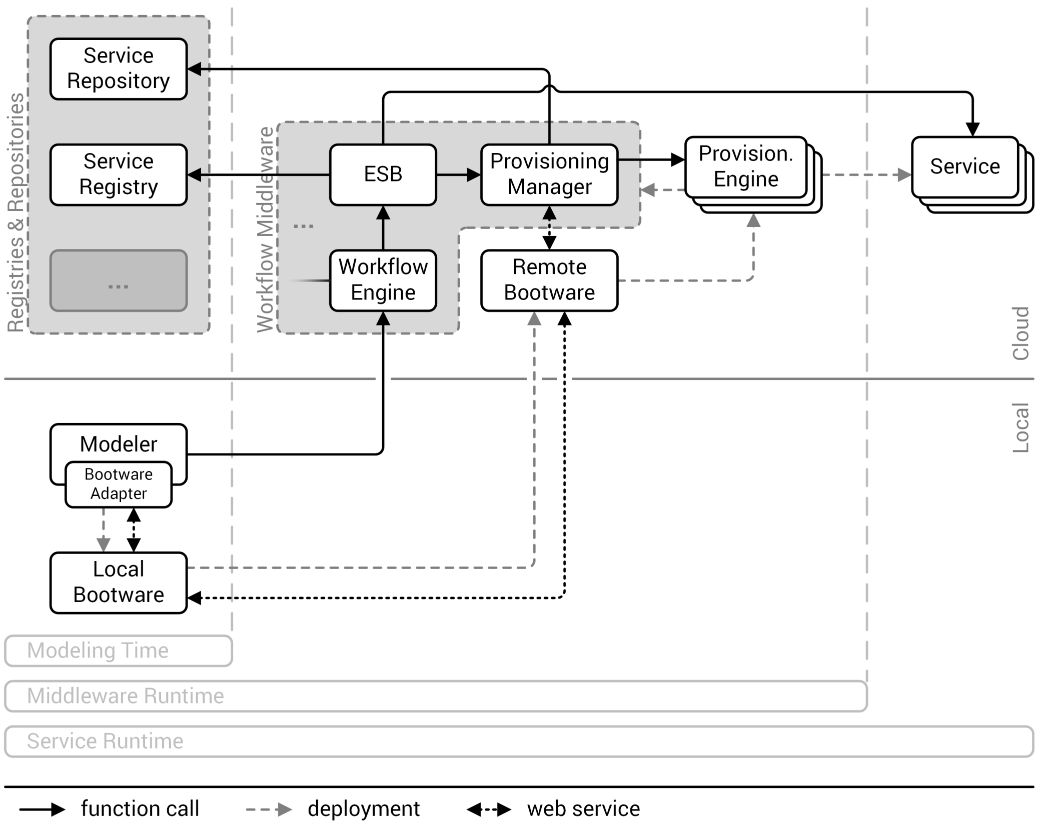

Figure 6.7: Simplified overview of the 2-tier architecture with web service communication.

return communication between the bootware adapter and the local bootware, and between the remote bootware and the local bootware, as well as the provisioning manager. With polling, long running provisioning processes will not pose a problem. We do however still need information during those long running processes to give the user some feedback. For this, a secondary communication mechanism which supports sending multiple feedback messages has to be used.

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 pushes messages to a message queue to which the local bootware (and other components if needs be) can subscribe to receive future messages. Figure 6.8 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. Because 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

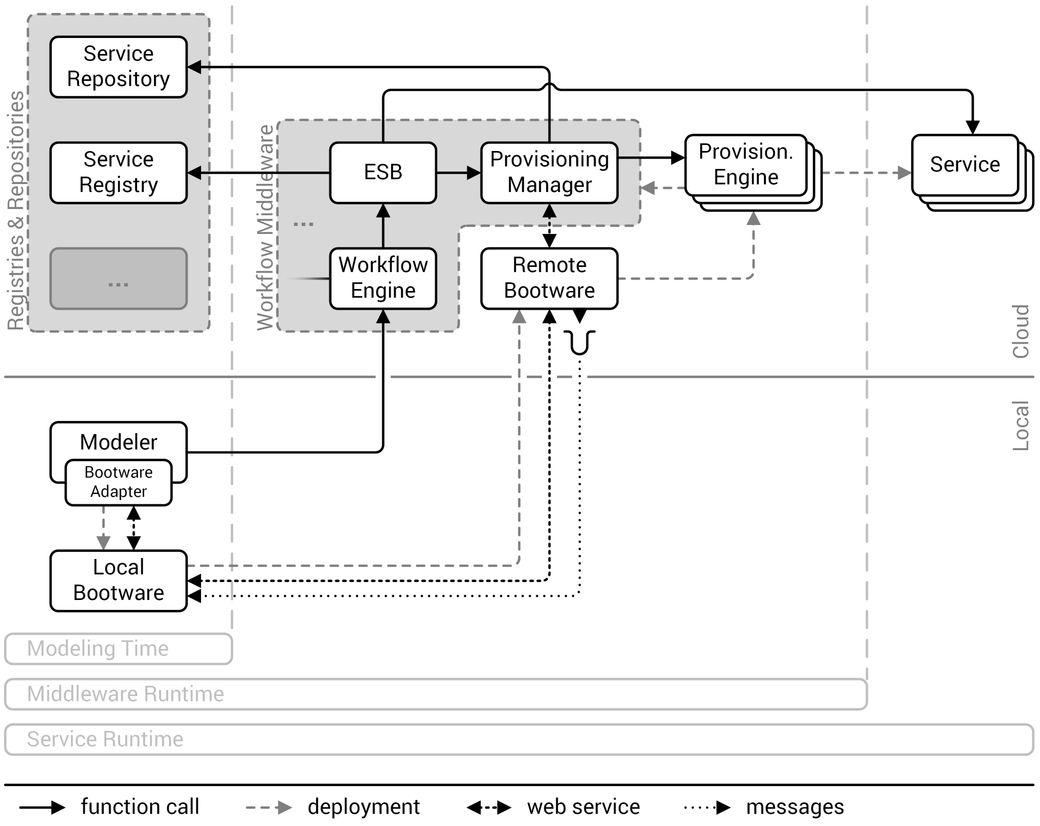

Figure 6.8: Simplified overview of the 2-tier architecture with web service and messaging queue communication.

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

The requirements for the bootware 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 interfaces 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 extensions 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 because 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 components 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 extensions are no longer part of the main bootware components. They are compiled separately into plugins that can be loaded into the main bootware components on the fly. Managing the plugins is the responsibility of a plugin manager, which encapsulates all functionality to load and unload plugins. It is described in more detail in the next section.

There are several possibilities to realize such a plugin 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 because multiple such frameworks already exist. It would also shift resources away from the actual goal of this diploma thesis, which is designing the bootware. 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 Manager

The plugin manager is a component of the bootware core that encapsulates all functionality for managing plugins, such as loading and unloading specific plugins. At the least, it should support two operations: The loadPlugin operation and the unloadPlugin operation. The loadPlugin operation would be called by the bootware when it needs to load a specific plugin. It would supply the path to the plugin as input parameter to the loadPlugin operation, which will use this information to load and return an instance of this specific plugin. This plugin instance can then be used by the bootware until it decides to unload this instance by calling the unloadPlugin operation of the plugin manager. The plugin manager might also support various other utility operations for convenience, such as an unloadAllPlugins operation, which would unload all loaded plugins at once. The plugin manager implementation will be described in more detail in Section 8.7, after we have selected a plugin framework.

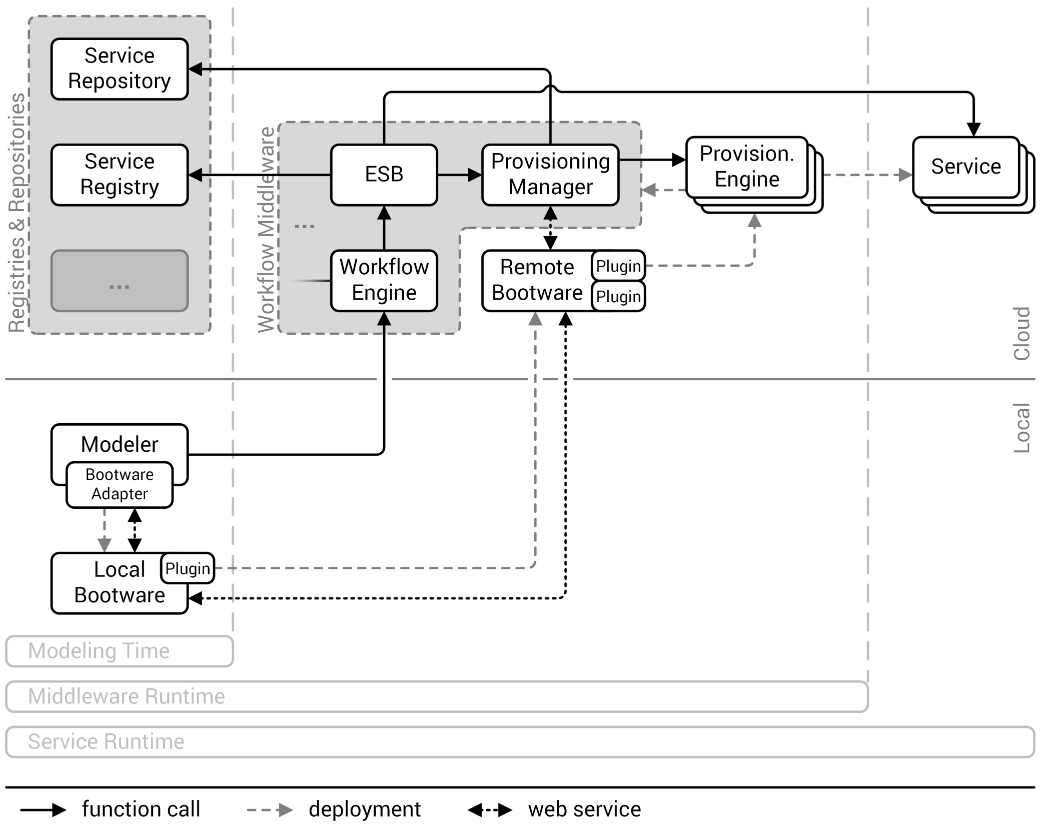

Figure 6.9: Simplified overview of the 2-tier architecture with plugins.

6.4.3 Plugin Repository

Now that we have introduced plugins we face new problems. Figure 6.9 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 cause errors if plugin versions get out of sync.

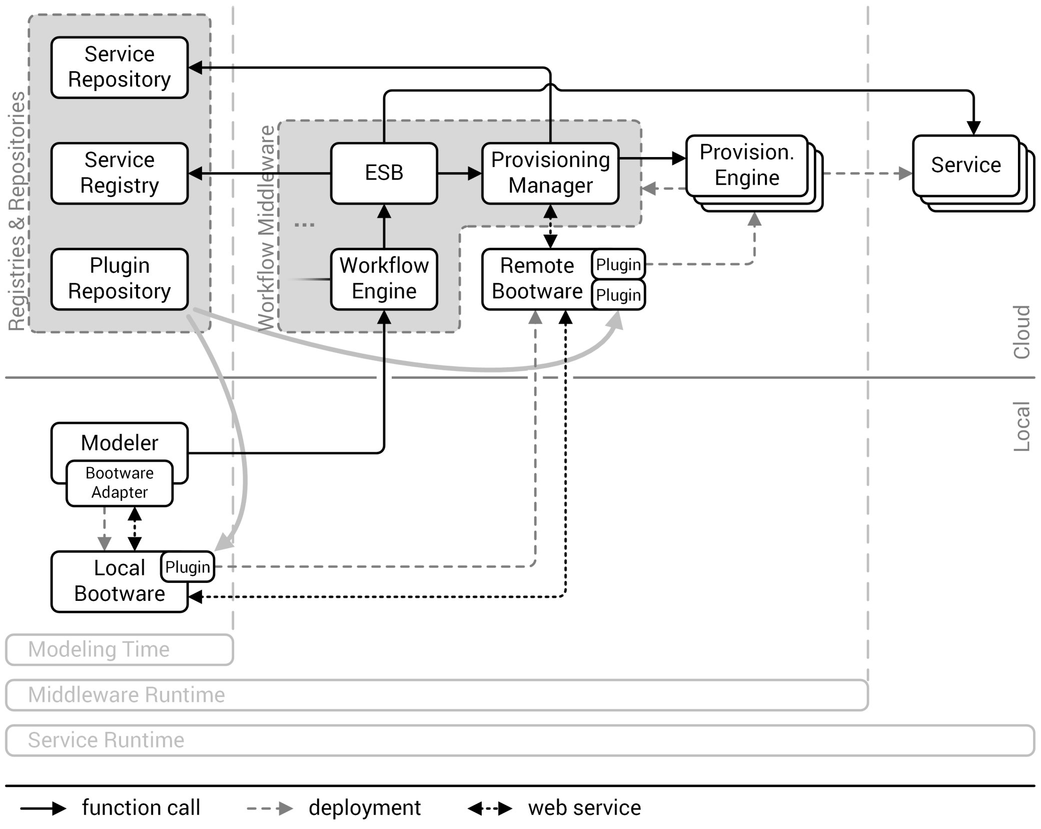

Figure 6.10: Simplified overview of the 2-tier architecture with a plugin repository.

To remedy this situation we introduce a central plugin repository, as shown in Figure 6.10. 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 diploma thesis. This work is left for the future and the plugin repository will not be mentioned in any other figures apart from Figure 6.10.

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). We do not even have to constrain these plugins to cloud resources and can make them more generic, as long as we can run the plugin and get an IP address to a computing resource that we can use. For example, we could also provide a plugin that starts and stops a VM on our local machine, which could be useful for quick and inexpensive local testing. So a better name for these plugins would be *resource 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 a given resource 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 *application plugins*.

Now that we have resource plugins and application plugins, we should be able to provision the resource we need and use application plugins to install and run any software on it. But there is a step in between provisioning the resource and installing the software that we are glancing over: We have to somehow communicate with the resource to be able to install something on it. The communication functionality could be part of either the resource plugins or the application plugins, or it could be separated into independent communication plugins. For the sake of efficiency and extensibility it would be best to use independent communication plugins. For example, if a user wanted to add a new communication type that should be used to install x applications in y environments, they could do so by writing one new communication plugin, instead of adding the functionality x-times to all application plugins, or y-times to all resource plugins. This would also reduce code duplication. Therefore, a third plugin type is necessary: The *communication 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. Because 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 *provision workflow middleware 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 than the four other types. They are not a part of the core bootware process but allow us to add functionality that reacts to (or creates) events inside the bootware for other purposes like, for example, logging. How the actual event system used by these plugins will be implemented will be discussed in Section 6.10.

With this fifth plugin type we have now covered all plugin types we will need. Next, we present some examples of how the different plugin types will work together. Then, we will describe each plugin type in more detail, starting with the common operations that all plugin types have to implement.

6.5.1 Examples

Figure 6.11 shows an overview over the basic process involving the plugins. The event plugins were omitted because they are not an essential part of this process. In step one, the bootware calls a resource plugin to create a new resource instance, for example a VM. Then, in step two, it creates a communication channel to this instance using a communication plugin. This communication channel is then used in step three by an application plugin to install an application on the instance. The fourth step is only executed when the remote bootware provisions the workflow middleware. Here, a provision workflow middleware plugin can call this application, in this case a provisioning engine, to provision a workflow middleware. How the bootware knows which plugins it should call is determined by the context which is explained in Section 6.6.

Figure 6.12 shows the same process but with an exemplary selection of specific plugin instances. In this case, an Amazon plugin is used in step one to create an EC2 instance. In step two, a SSH plugin creates a connection to this EC2 instance. Over this SSH connection, an OpenTOSCA plugin installs OpenTOSCA on the EC2 instance, as shown in step three. Finally, in step four, a call OpenTOSCA plugin calls the just deployed OpenTOSCA container to provision the workflow middleware. For another combination of plugins, the process might look like Figure 6.13. Here, a Chef server is installed on an Azure VM over a remote desktop connection (RDC). The Chef server is then used to provision the workflow middleware.

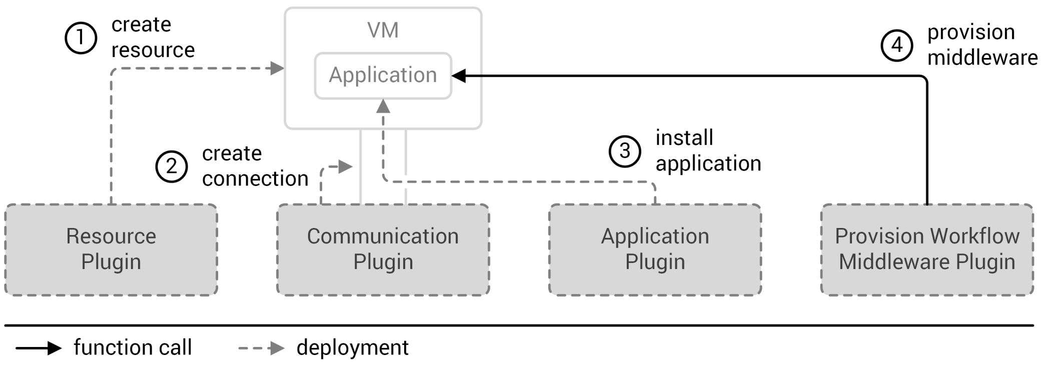

Figure 6.11: Simplified overview of the plugin process.

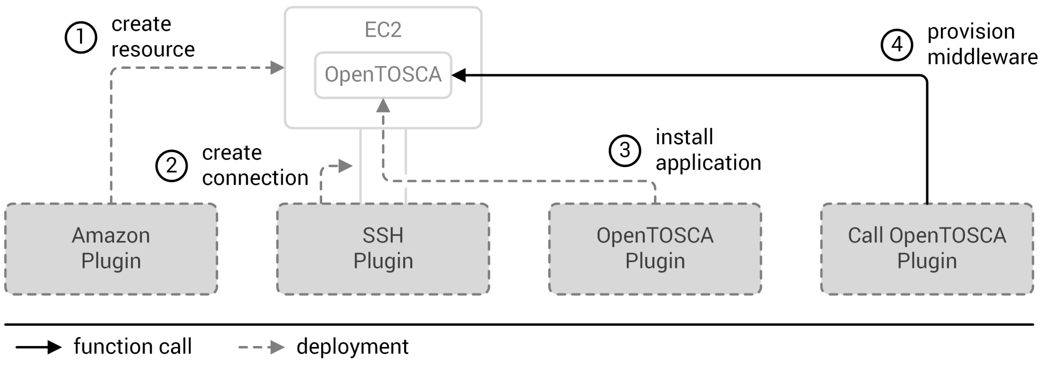

Figure 6.12: Exemplary plugin process for Amazon and OpenTOSCA.

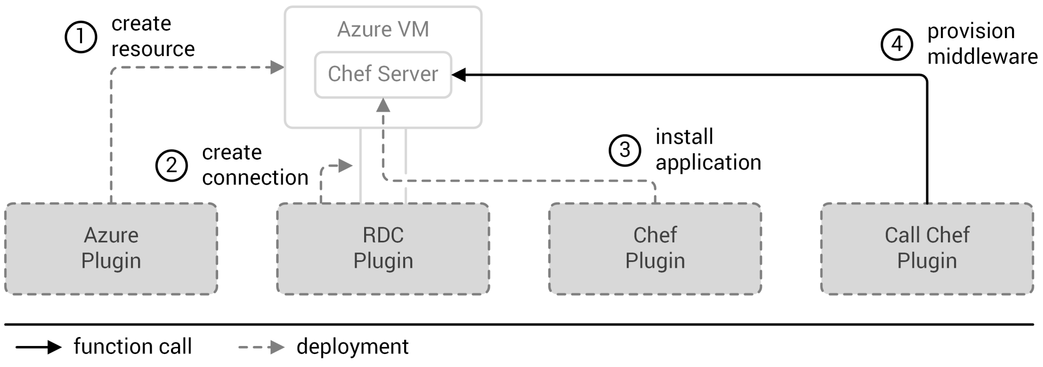

Figure 6.13: Exemplary plugin process for Azure and Chef.

6.5.2 Common Operations

Table 6.1 shows the two common operations that all plugin types must implement. The initialize operation is called by the plugin manager when it 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. It takes a configuration object as parameter, which is taken from the request context. This allows the plugins to be configured from the outside if necessary. The shutdown operation is called by the plugin manager when it unloads a plugin. It can be useful to clean up plugin resources before the plugin is removed, for example by deleting temporary files or closing a communication channel.

Operation	Input	Output	Description
initialize	Configuration	-	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.

6.5.3 Resource Plugins

Resource plugins are responsible for provisioning a computing resources that the user wants to use during the bootware process. This could be a VM on a local machine, or an IaaS or PaaS environment in the cloud. To be able to do this, a resource plugin has to implement a range of functions using some API or SDK provided by the virtualization software or cloud provider.

Operation	Input	Output	Description
deploy	-	Instance	Deploys a communication ready instance of some resource and returns an instance object
undeploy	Instance	-	Completely removes a given instance

Table 6.2: Interface to be implemented by resource plugins.

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 communication plugin. It takes no input parameters, but relies on the configuration passed to the initialize operation to get the configuration details it needs, like login credentials. 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.

6.5.4 Communication Plugins

Communication plugins are responsible for creating a communication channel to a previously deployed resource that can later be used by application plugins to execute their operations on the resource. The connection could be made by using SSH, RDC, virtual private network (VPN), Telnet, or other communication mechanisms supported by the resource. The communication plugins should be implemented generically, 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 connect operation. If the connection was established successfully, the operation returns a connection object that can be used later by application 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: Interface to be implemented by communication plugins.

6.5.5 Application Plugins

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

Table 6.4 shows the operations that plugins of this type should implement. The deploy operation installs an application on an instance. This can include uploading files from the local machine or downloading files from other machines. To execute this operation, a connection to the instance is necessary, which is supplied as input with the connection object. The undeploy operation removes an application from an instance. In most cases this will not be necessary, because the instance will be destroyed in the undeploy phase and with it all the application data (assuming it was not stored in some other persistent storage). This method is provided for completeness and for special cases. The start operation starts an application which previously was installed with the deploy operation. If the application was started successfully, it returns the URL to the running application. The stop operation stops the execution of a previously started application. In most cases this will not be necessary, because the application will be removed together with the instance in the undeploy phase. This method is provided for completeness and for special cases.

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

Table 6.4: Interface to be implemented by application plugins.

6.5.6 Provision Workflow Middleware Plugins

Provision workflow middleware plugins provide the bootware with a unified way to call provisioning engines and trigger provisioning and deprovisioning operations. Table 6.5 shows the operations that these plugins should implement. The provision operation calls a provisioning engine and triggers 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 workflow middleware package that the provisioning engine should provision. When completed successfully, the provisioning operation returns a list with information about the just provisioned workflow middleware. This list can contain arbitrary information, such as endpoint references pointing to the various components of the workflow middleware, or any other information that might be necessary to connect the modeler to the workflow middleware. 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 package reference.

Operation	Input	Output	Description
provision	Provisioning Engine Endpoint Reference, Service Package Reference	Information List	Tells the provisioning engine to provision the given workflow middleware package
deprovision	Provisioning Engine Endpoint Reference, Service Package Reference	-	Tells the provisioning engine to deprovision the given workflow middleware package

Table 6.5: Interface to be implemented by provision workflow middleware plugins.

In parallel to this diploma thesis, another diploma thesis is being written about the provisioning manager, which will also use plugins to call provisioning engines [22]. Because these plugins are similar in functionality, it makes sense to create libraries for particular provisioning engines that can then be used by both the provisioning manager plugins and the bootware plugins. This would reduce overall code duplication. We will not describe those libraries in more detail in this thesis. We assume that such libraries will exist and that we can use them for implementing our plugins.

6.5.7 Event Plugins

Apart from the initialize and shutdown operations described in Table 6.1, event plugins only implement the handle operation, as shown in Table 6.6. It takes an event type as input. Every time an event of this type is triggered, all handle operation associated with this event type will be called and can execute some code, for example logging the event. Note that each event plugin can declare more than one handle function to be able to react to multiple events.

Operation	Input	Output	Description
handle	Event Type	-	Is called every time an event of the given type is triggered

Table 6.6: Interface to be implemented by event plugins.

6.6 Context

During the bootstrapping process, the bootware has to know certain things to be able do its job. For example, it has to know which plugins it should use to process a request or which login credentials it should use to authenticate itself with a cloud provider. This information can be combined in one central object, which defines the nature of the current request: The *context*. The bootware will read the information provided in the context at various stages of the bootstrapping process to load plugins or to supply them with a particular configuration set required for a request. 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 8.4.

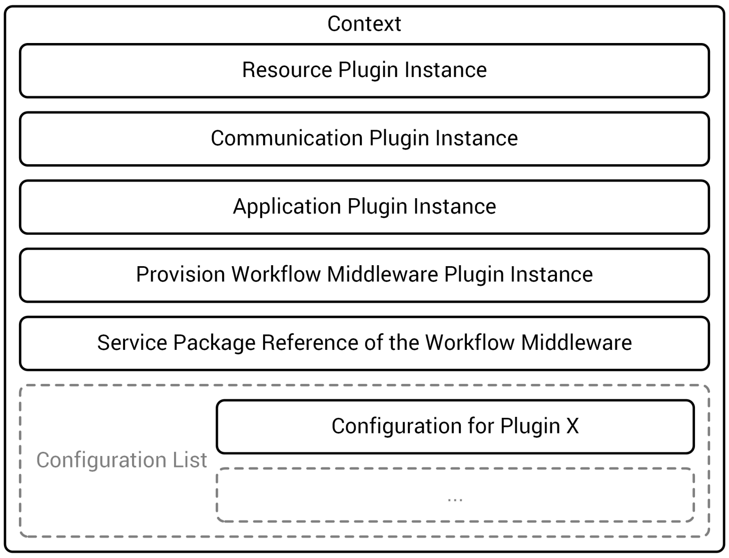
Context
Resource Plugin Instance
Communication Plugin Instance
Application Plugin Instance
Provision Workflow Middleware Plugin Instance
Service Package Reference of the Workflow Middleware
Configuration List
Configuration for Plugin X
...

Figure 6.14: Content of the context object.

Figure 6.14 shows the context object and its content. As we can see in the upper half, it defines the plugin instances to be used for the current request. The resource plugin instance defines, which resource plugin should be used to provision the requested resource. The communication plugin instance selects, which communication plugin the bootware should use to connect to this resource. The application plugin instance defines the application that should be provisioned on this resource, which will be a provisioning engine in our case. Finally, the provision workflow middleware plugin instance defines the plugin that should be used to call this provisioning engine to provision the workflow middleware. It will use the service package reference of the workflow middleware, which is also defined in the context, as input to start the provisioning of the workflow middleware.

In the bottom half of Figure 6.14 we can see that the context can also contain configuration for different plugins. This is necessary because various plugins might need to be configured properly to be able to fulfill their task. For example, most resource plugins will need some kind of login credentials to authenticate with the resource provider. As another example, when creating a EC2 instance in the Amazon cloud, the user also has to select in which region this instance should be created and which ports should be opened. These and other configuration details can be supplied from the outside with the context. In the future, the context might be extended to hold additional information, but for this diploma thesis, this context will be sufficient.

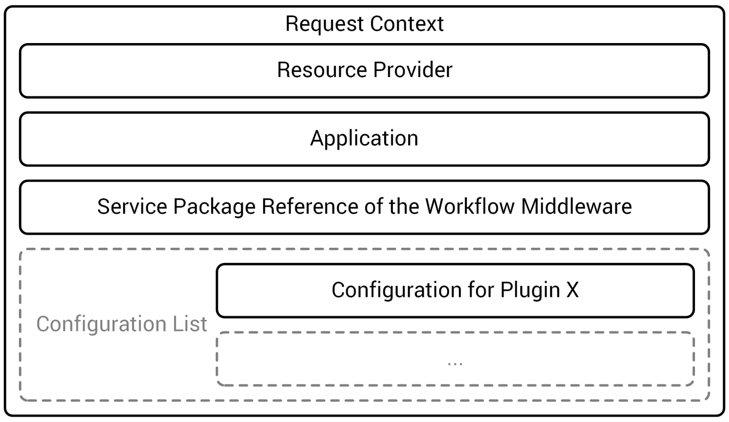
Request Context
Resource Provider
Application
Service Package Reference of the Workflow Middleware
Configuration List
Configuration for Plugin X
...

Figure 6.15: Information send with a request.

This context has to somehow be generated from the information the bootware receives with each request. However, components outside of the bootware should not know anything about the inner workings of the bootware, i.e. which plugins are used to fulfill a request. All information that might be supplied with a request can be seen in Figure 6.15. The resource provider parameter specifies, which resource provider should be used (e.g. a specific cloud provider). The application parameter specifies the application that should be deployed on this resource (e.g. a specific provisioning engine). The service package reference of the workflow middleware is used by the bootware to retrieve the service package to provision the workflow middleware. In the configuration list, configuration values that have to be supplied from the outside can be specified, such as login credentials for a cloud provider.

Based on this information the bootware has to decide which specific plugin instances should be used to process the request. In the future, this could be done by querying a registry with this information, which would then return a set of specific plugins and settings. Figure 6.16 shows an exemplary mapping of the parameters supplied in a request context to the specific plugins instances and configuration sets. In this example, the request wants OpenTOSCA in an Amazon cloud. The registry would then look up, if a registry entry exists for this set of parameters. In this case, the registry contains applications as primary keys and resources as secondary keys. The entry returned for a specific parameter combination contains the specific plugin instances and additional configuration parameters that are necessary to provision the requested application on the requested resource.

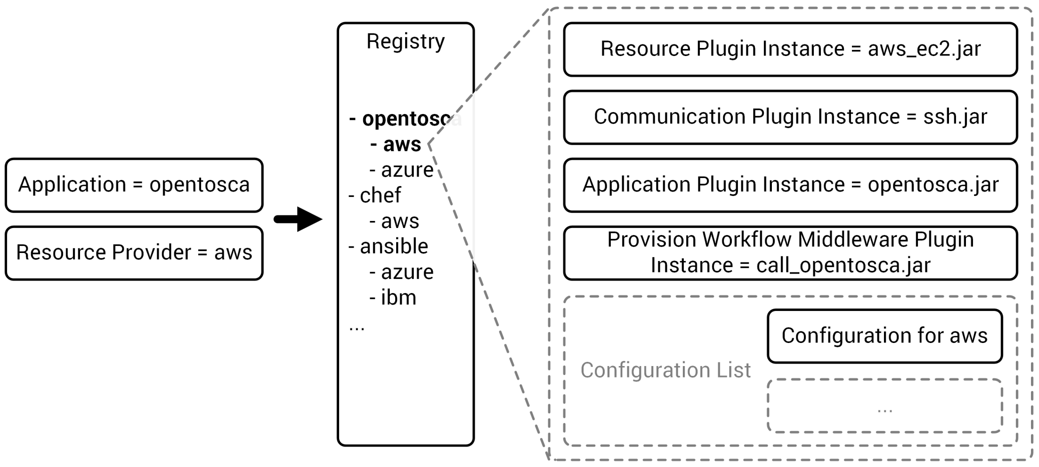
Application = opentosca
Resource Provider = aws
Registry
- opentosca
- aws
- azure
- chef
- aws
- ansible
- azure
- ibm
...
Resource Plugin Instance = aws_ec2.jar
Communication Plugin Instance = ssh.jar
Application Plugin Instance = opentosca.jar
Provision Workflow Middleware Plugin Instance = call_opentosca.jar
Configuration List
Configuration for aws
...

Figure 6.16: Mapping of the request parameters to specific plugins.

The bootware has to combine the information received in a request context with the information that it received from the registry by merging the two sets into the final context. However, there is one caveat in this scenario. Right now, other components making a request to the bootware would have to supply configuration parameters like login credentials with each request. Unlike the other request parameters, the configuration might not change between requests. Additionally, the other components calling the bootware, may not know (and maybe should not know) anything about some content of the configuration, like login credentials. While this might change in the future, it would make sense to be able to set the configuration once when starting the bootware, so that it does not have to be delivered with each request and so that other components can still use the bootware without also sending a configuration. It should however still be possible to override or update the configuration at a later point. Overriding would allow any request to temporarily use other configuration values if necessary. Updating the configuration 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 the configuration even during the bootstrapping process.

For setting the configuration at the beginning and for updating it later during the process, a *setConfiguration* method will be added to the bootware web service. The configuration set by this method will be treated as the default configuration by the bootware. It will be used during the process if no other configuration is provided. If however a request is sent with a configuration that also contains values already set in the default configuration, these values will override already existing default values temporarily for this request. This behavior could also be extended to other parts of the context in the future if necessary.

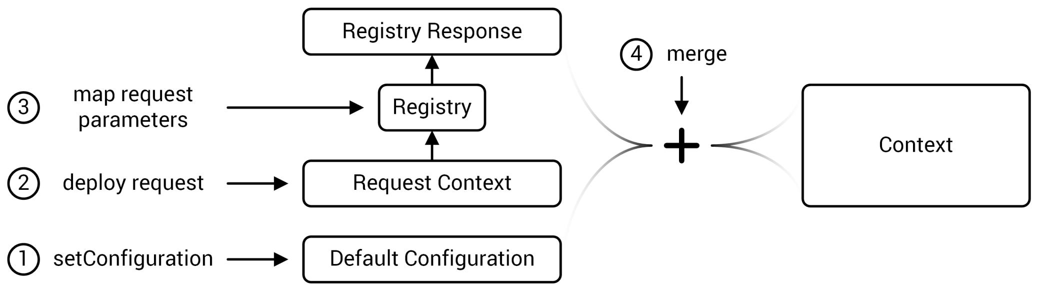

Figure 6.17: Building the final context through merging.

Figure 6.17 shows the merging process which now merges the default configuration, the request context, and the response from the registry. In step one, at the beginning of the bootware execution, the default configuration is set using the *setConfiguration* operation. A deploy request is received in step two, which contains the request context as shown in Figure 6.15. In step three, the parameters from the request context are mapped to specific plugin instances and additional configuration set using the registry, as shown in Figure 6.16. In step four, the default Configuration, the request context, and the registry response are merged into the final context object. This context object is then used to fulfill the request. While we have planned to use a registry for mapping request parameters to concrete values, due to time constrains we were not able to specify this registry in more detail or implement it in our prototype. For our implementation we use a local mechanism for mapping the request parameters. The registry is left for future work and will not be covered further in this diploma thesis.

6.7 Web Service Interface

By now, we know that we will use a web service interface for remote communication. Table 6.7 shows the web service operations provided by the local and remote bootware. 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 6.6 we also mentioned the *setConfiguration* operation for setting or updating configuration values. We add two additional operations that we also need, the *getActiveApplications* operation and the *shutdown* operation. Both the local and the remote bootware will have to implement all of these operations, except the *getActiveApplications* operation, which is only needed in the remote bootware.

Operation	Input	Success Response
deploy	Context	Information List
undeploy	Endpoint References	-
setConfiguration	Configuration List	-
getActiveApplications[2]	-	Application List
shutdown	-	Confirmation Message

Table 6.7: Web service operations provided by the local and remote bootware.

6.7.1 Deploy

The *deploy* operation is called whenever a new application (e.g.: a provisioning engine, or initially, the remote bootware) should be deployed. As input it takes a request context object as described in Section 6.6. If it was able to successfully deploy the requested application, it responds with a list of information concerning the application. This list can contain endpoint references, ports, or any other information that might be needed later. If the deployment failed, it responds with an error message.

[2]only in remote bootware

6.7.2 Undeploy

The *undeploy* operation is essentially the reversal of the *deploy* operation. As input it takes an endpoint reference to an application 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 does not take a context object as input, but the context is still needed for the *undeploy* operation because 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 6.8.

6.7.3 Set Configuration

The *setConfiguration* operation is used to transmit or update the default configuration used by plugins. As input it takes a list of configurations that should be saved. If the list provided is empty, the default configuration list saved in the bootware will be emptied. If the list provided is not empty, the default configuration list saved in the bootware will be overwritten by this list. The configuration can still be overwritten on a per request basis if the context send with the request also contains a configuration. If the configuration was updated successfully, it responds with a success message. If the configuration could not be updated, it responds with an error message.

6.7.4 Get Active Applications

The *getActiveApplications* operation is used by the provisioning manager to check if a provisioning engine it needs already exist. This operation just returns a list of all active application. There is no reason for this operation to be called on the local bootware, so this operation will be implemented in the remote bootware only.

6.7.5 Shutdown

This operation triggers the shutdown of sequence of the bootware. It behaves a little differently in the local and remote bootware. In the local bootware it first calls the *shutdown* operation of the remote bootware. When the confirmation response from the remote bootware is received, it deprovisions all active applications that the local bootware deployed (i.e. the remote bootware). In the remote bootware, the *shutdown* operation first calls a provisioning engine to deprovision the workflow middleware. Once this is done, it deprovisions all active applications that the remote deployed (i.e. the various provisioning engines), before returning a response.

6.8 Instance Store

The instance store stores information about applications that were deployed by the bootware in the past and are still active. In Section 6.7 we already mentioned that we need to store some information about active applications, but we did not explain why. There are several reasons why this is useful.

One big reason is that we cannot guarantee that an *undeploy* operation will be called for every application deployed by the bootware, because 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 is better to have a failsafe in place. In the worst case scenario, failing to call the *undeploy* operation for some applications could lead to rogue applications 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 applications 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 it deployed.

Another reason to store some information about deployed applications is to simplify the interaction with other components. If we would not store any information and make the bootware stateless, each component using the bootware (e.g.: the bootware adapter, the local bootware, and the provisioning manager) would be required to keep track of all applications it deployed using the bootware, so that this information can be supplied when it is time to undeploy. This places an extra burden on these components and scatters around the information about deployed applications. By storing this information in the bootware we simplify the usage of the bootware for other components and concentrate this information in one place. With the *getActiveApplications* operation introduced in Section 6.7 and offered by the remote bootware, other components can always get a list of all active applications if they need it. This operation also uses the information stored in the instance store.

We should also think about how such a storage mechanism might be different for the local and remote bootware. The local bootware only ever 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 provisioning engines 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. However, for this diploma thesis 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 applications, we need to discuss what exactly we need to store. We need to store enough information to be able to *undeploy* an active application without any further input. For this we need to know: The resource plugin that was used to provision the resource, the connection plugin that was used to connect to it, the application plugin that was used to deploy the active application, 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 application, so we will just store the whole context object. Because 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.

6.9 Shutdown Trigger

One thing that we have not mentioned yet is how the bootware will be shut down. The bootware can not just stop. It has to make sure that all applications and all the resources it has provisioned are removed before it shuts down itself. But how does the bootware know when it is time to start this procedure? After all, this depends on the workflow middleware. The shutdown process should start when the workflow middleware is finished with the workflow execution, so the bootware has to be informed of this somehow.

One possibility is to trigger the shutdown procedure from the bootware plugin in the modeler. If the bootware adapter can access this information through the modeler, it can call the *shutdown* operation of the local bootware, which will in turn call the *shutdown* operation of the remote bootware, which will eventually lead to the removal of all remote components. If this is possible using a particular modeler depends on the modeler and the integration possibilities for the bootware adapter.

There is a second method that can be used instead. We already introduced the event plugin type, which can also trigger events in the bootware, in particular the shutdown event. An event plugin could be created that somehow communicates with the workflow middleware to receive notice when the execution is finish. For example, in the SimTech SWfMS, the workflow engine publishes events into a message queue. An event plugin could be created that subscribes to this messages queue and reacts to a particular event by triggering the shutdown event inside the bootware. This plugin would then be loaded into the local bootware and would trigger the shutdown procedure, which would in turn call the *shutdown* operation of the remote bootware as before.

6.10 Event System

In Section 6.3 we mentioned that we need a way to provide the user with updates during the long running bootstrapping process. In order to do this, we introduced event plugins in Section 6.5 that would allow us to react to events in the bootware. Now, we will describe the event system that will be used to distribute these events. Note that this event system is not a core part of the bootware functionality and that the bootware can function completely without it. It just allows us to access state information of the bootware core and the plugins in a more fine grained way if we need it.

Ideally, we want a central location where we can access all events generated throughout the bootstrapping process. These events could include events generated by the bootware core, as well as by plugins. For example, we might like to receive an event when the process of deploying a new provisioning engine is started. This would be an event generated by the bootware core when it receives a corresponding deploy request. During this request execution, multiple plugins would be called to deploy the provisioning engine. We might also like to receive events during the execution of those plugins. For example, when a resource plugin creates a cloud resource, it could generate an event when it successfully authenticated with the cloud provider, another event when it started the creation of a VM, and a final event when the VM is ready for use. We could consume all these events with an event plugin and use them to inform the user of the bootstrapping progress, or log them to a text file.

These events could be triggered by the bootware core or by any plugin, but plugins should be completely independent from each other. Because an event plugin does not know about other plugins, it cannot listen for events at other plugins directly. The only known constant to an event plugin is the bootware core. Therefore, we need an event system 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 [14].

6.10.1 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. It 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 [14]. Using this pattern, 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.

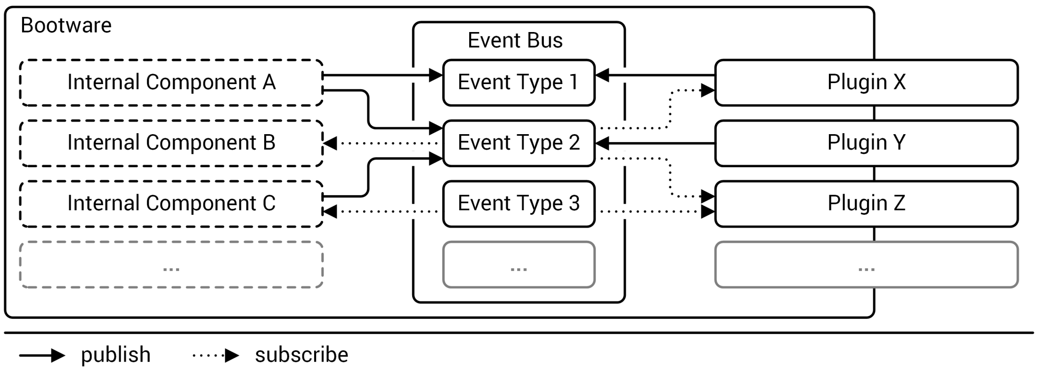

Figure 6.18: Bootware internal communication with PubSub pattern.

6.10.2 Event Types

When using PubSub and events to communicate, it is 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 event type. 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 because we can now just react to the parent event if we do not 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.

- **BaseEvent** on which all other events are based
  - **CoreEvent** published by the bootware core
    - **PluginManagerEvent** for loading and unloading plugins
      - **PluginLoadEvent** could be info, success, warning, or error
      - **PluginUnloadEvent** could be info, success, warning, or error
      - **...**
    - **...**
  - **PluginEvent** published by a plugin
    - **ResourcePluginEvent** contains further child events defined by plugin
    - **CommunicationPluginEvent** contains further child events defined by plugin
    - **ApplicationPluginEvent** contains further child events defined by plugin
    - **EventPluginEvent** contains further child events defined by plugin
    - **...**

Figure 6.19: Exemplary event hierarchy.

Figure 6.19 shows an exemplary event hierarchy for the bootware. As we can see, every event is based on the BaseEvent, shown at the top. Events can be further divided into core events that are published by the bootware core and plugin events that are published by plugins. Core events contain events from the various core components of the bootware, for example the plugin manager. The plugin manager events are further divided into events for certain operations that can also have different severity values (e.g.: info, success, warning, and error). Plugin events are divided by plugin types. Different plugins can also add further child events to these event types.

6.11 Execution Flow

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

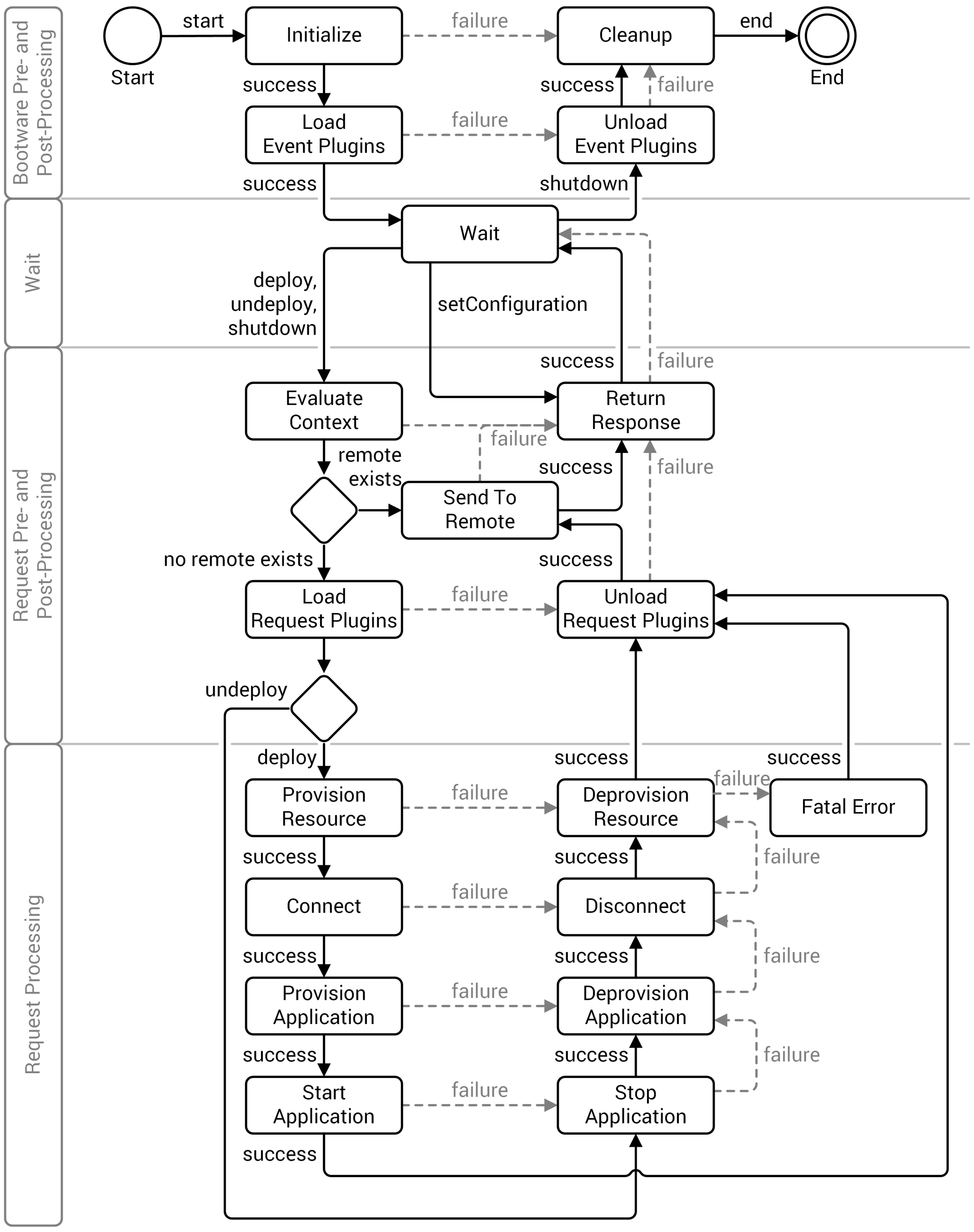

Figure 6.20: Execution flow in the local bootware.

Figure 6.20 shows a graph that represents the major steps during the bootware execution in the local bootware as flow diagram. The bootstrapping process is started when the bootware adapter starts the local bootware, which is represented by the *start* activity in the top left corner of Figure 6.20. From there, the bootware first does some initializations. If those 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 6.20. In most cases however, the initialization should succeed. Then, the local bootware will execute the next activity, where it tries to load the event plugins.

The event plugins are loaded once at the beginning of the local bootware execution because they will not change at a per request basis (like the other plugins). Any events generated by the bootware core or by plugins after this point can now be handled by the event plugins. If loading one of these plugins fails, the local bootware will try to unload already loaded plugins before continuing with the *cleanup* activity. If the plugins are loaded successfully, the local bootware transitions into the *wait* activity, shown in the top center of Figure 6.20.

The local bootware is now ready and waits for requests from the outside. A deploy request will be send to the local bootware when the bootware adapter calls it to provision the workflow middleware. A setConfiguration request might also be send by the bootware adapter to set the default configuration. The undeploy and shutdown request will be triggered by the local bootware itself, when it is time to deprovision the workflow middleware and the remote bootware.

If a shutdown event is received, the local bootware will first tell the remote bootware to undeploy all active applications. Next, the local bootware will undeploy the remote bootware by running through the undeploy process fragment shown on the bottom left with the appropriate plugins. Then, it will shut itself down by first unloading the event plugins and then running the cleanup code. This is the only normal way to shut down the bootware. We only hint at the setConfiguration request here, since it just replaces the saved configuration with the one sent in the request. Deploy and undeploy requests however are more complicated. If such a request is received, the local bootware transitions to the next activity, where it evaluates the request context.

In the *evaluate context* activity, the information send with the request is used to generate the full context, which contains all the information necessary to fulfill the request, as described in Section 6.6. If this cannot be done for some reason, the local bootware returns a response containing an error message before returning to the *wait* activity. If the context is created successfully, the local bootware tries to send the request on to the remote bootware, as shown in the middle of Figure 6.20. For this to work, the remote bootware has to exist in the requested remote environment, which will not 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* activity. In the *load request plugins* activity the plugins specified in the context are loaded. If this fails, the local bootware tries to unload already loaded plugins before returning an error response and returns to the *wait* activity. If the plugins are loaded successfully, the local bootware now starts either the deploy process fragment at the bottom left, or the undeploy process fragment shown at the bottom right of Figure 6.20, 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 6.20 one after another. In the *provision resource* activity, the deploy operation of the resource plugin will be called. Then, in the *connect* activity, a connection with this resource will be established by the communication plugin. Over this connection, the requested application is provisioned in the *provision application* activity, and started in the *start application* activity, using the application plugin. If one of these activities fails, the local bootware transitions over to the corresponding undeploy activities on the right and works its way backwards to undo all operations that where already executed. This process fragment is the same as the undeploy process, shown on the bottom right of Figure 6.20, which is triggered by an undeploy request.

If the *stop application*, *deprovision application*, or *disconnect* activities fail, the local bootware just continues with the next undeploy activity because these operations are not considered critical. However, if the *deprovision resource* activity fails, the local bootware transitions to a *fatal error* activity, shown at the right of Figure 6.20, because this step is considered critical. This activity 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* activity 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 with the *unload request plugins* activity, 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 6.20 with the *send to remote* activity. The local bootware will wait here until it receives a response from the remote bootware.

Now, we move our attention to the remote bootware, where the requests continues to be processed. Figure 6.21 shows the execution flow of the remote bootware. As we can see, it is largely identical to the local bootware. The *send to remote* activity is gone because it is not needed in the remote bootware. Instead, as the bottom of Figure 6.21 shows, the *provision middleware* and *deprovision middleware* activities were added. The remote bootware also supports the getActiveApplications request. Other than that, the local and remote processes are the same.

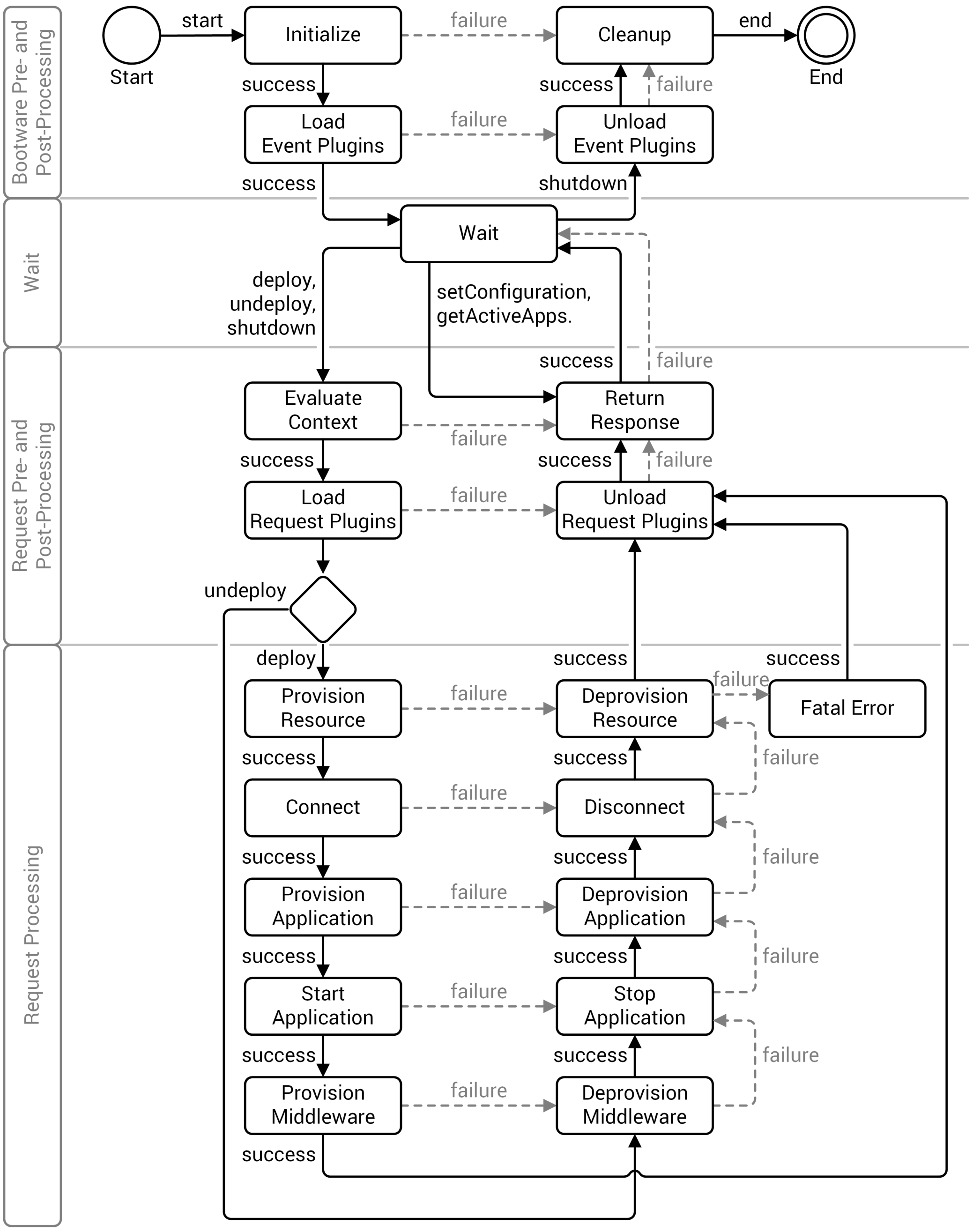

Figure 6.21: Execution flow in the remote bootware.

Like the local bootware, the remote bootware went through the initialization steps shown at the top of Figure 6.21 when it was started by the local bootware. It then waited in the *wait* activity for a request. Now, it receives the request from the local bootware, creates the context, loads the request plugins and executes the deploy operation. This should result in a provisioning engine being started by the application plugin. After that, the remote bootware executes the new *provision middleware* activity at the bottom left of Figure 6.21, which will use the just started provisioning engine to deploy the workflow middleware by executing the provision workflow middleware plugin. Once the middleware is running, the remote bootware is finished with this request and returns the information list containing the endpoint references of the middleware in the response to the local bootware, before returning to the *wait* activity.

This brings us back to Figure 6.20, where the local bootware has now received the answer from the remote bootware in the *send to remote* activity. Now, the local bootware can finish its request by sending back a response to the bootware adapter, before returning to the *wait* activity. The local bootware is now done until it is time to undeploy the remote bootware. Meanwhile, the bootware adapter 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 fragments shown at the bottom, or the getActiveApplications operation only hinted at in Figure 6.21.

As Figure 6.20, Figure 6.21 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 blocks. Therefore, it could be advantageous to use other methods that are more fitting for this process. As we already described the process as a directed graph, 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.

6.11.1 Finite State Machine

In theoretical computer science, a Finite State Machine (FSM) is a formal, abstract model of computation "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 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. The states map directly onto the activities described in Figure 6.20 and Figure 6.21. When The FSM enters a state, it executes a function associated with this state, which would be the implementation of said activities. 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.

6.12 Final Bootware Architecture

In Figure 6.22 we present the final architecture of the bootware in context with a SWfMS. New components are marked black and include the local and remote bootware, their plugins, and the bootware adapter. Old components that existed previously are shown in white.

Figure 6.23 and Figure 6.24 show the final architecture of the local and remote bootware. They only differ in some small details, but this might change in the future. At the bottom we can see some exemplary event plugins. These are loaded at the beginning of the bootware execution by the plugin manager, shown on the left of both figures. For demonstrations purposes, both figures show a wider range of possible event plugins. All these plugins provide some sort of input and/or output mechanism for the bootware. A command-line interface (CLI) plugin, as shown in Figure 6.23, could be used to make the local bootware operations accessible via a command-line interface. An 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 at the remote bootware, so that they can be

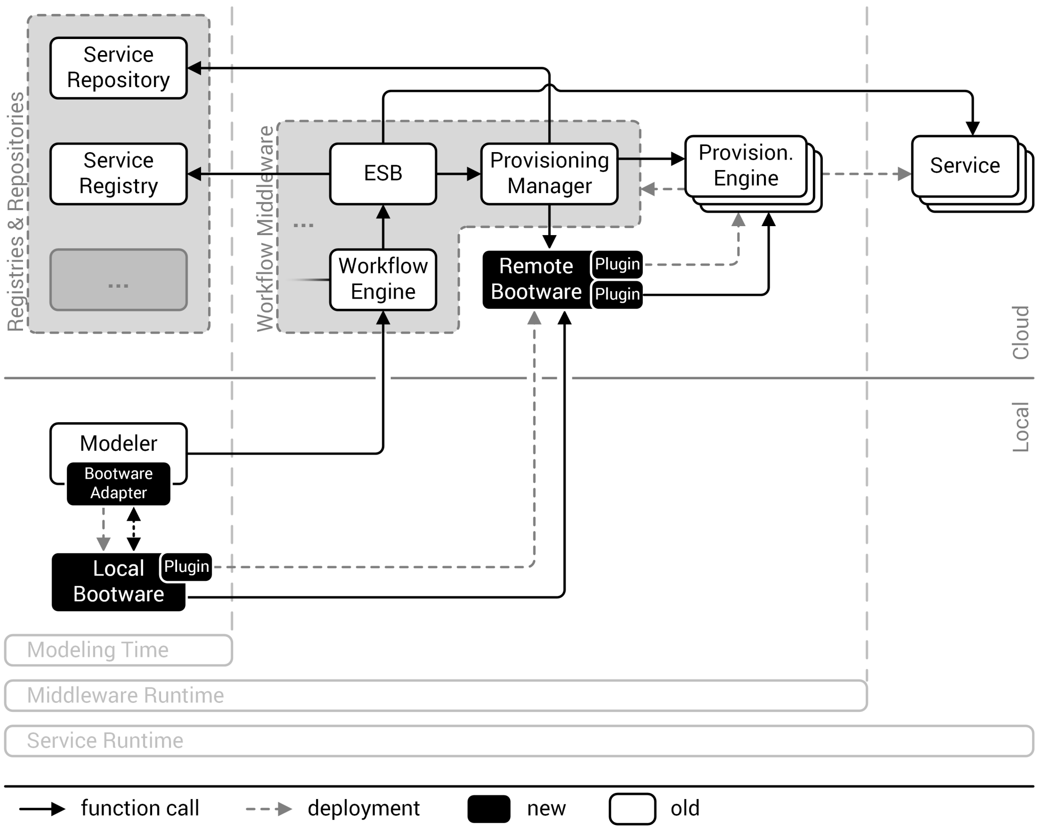

Figure 6.22: The final architecture of the bootware.

consumed by other components, like the local bootware. Finally, an undeploy trigger plugin in the local bootware, as shown in Figure 6.23, could trigger the undeployment of the bootware and all running applications by listening for a specific message at the workflow middleware. Besides the event plugins there is always the web service interface, shown at the bottom right of both figures, which provides the standard way to interact with the bootware.

All event plugins work by implementing event handlers for certain events published at the event bus, or by publishing events to the event bus themselves. As we can see in the center of both figures, 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 6.11. At certain points during the bootstrapping process, operations are delegated to the plugin manager to load plugins, and to the resource, communication, application, and provision workflow middleware plugins, shown at the top of both figures.

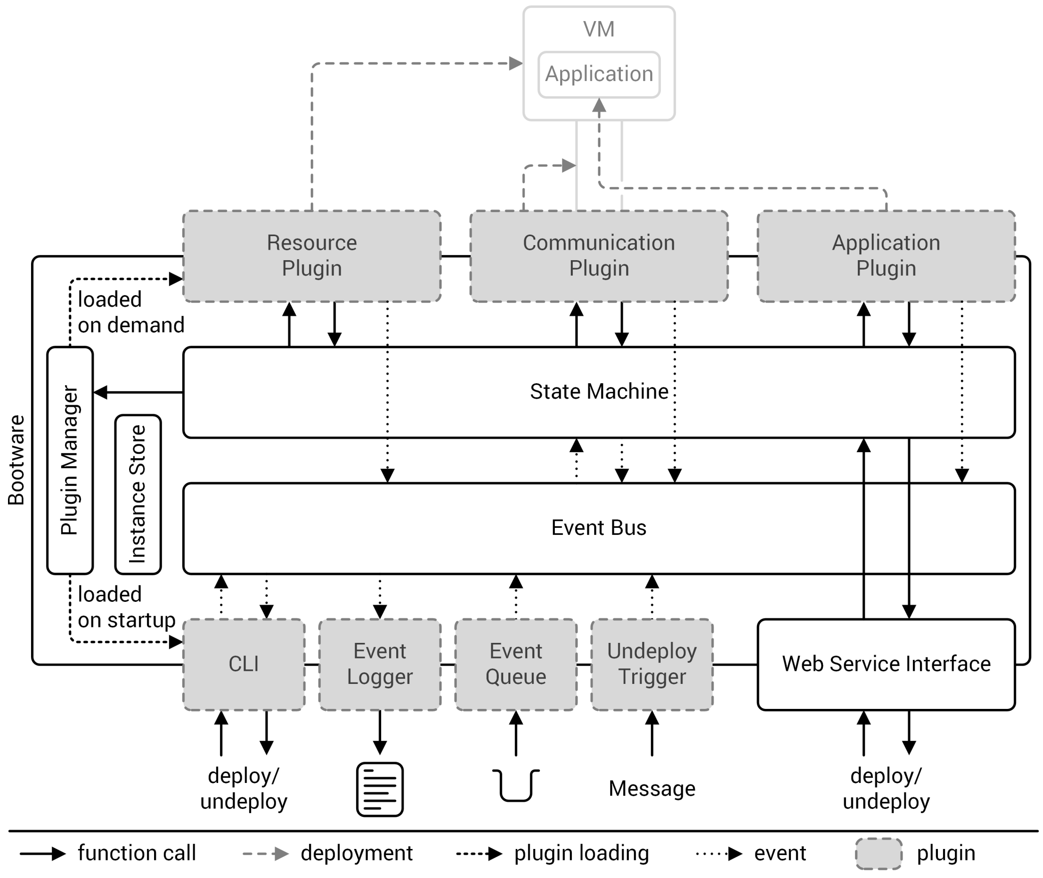

Figure 6.23: The final architecture of the local bootware component.

The resource, communication, and application plugins implement the actual bootstrapping operations. At the top, both figures show an exemplary result of these bootstrapping operations. In this particular case, the resource plugin started a VM, to which the communication plugin set up a communication channel. The application plugin then used this communication channel to provision the application 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 6.24 as an additional function call from the provision workflow middleware plugin to the previously deployed application. During the bootstrapping procedure, events are sent from all these plugins back to the event bus to be delivered to the loaded event plugins. As we can now see, the local and the remote bootware are quite similar, but differ in enough ways that a cloned architecture, as described in Section 6.1, might not be the best choice, especially because both components might drift further apart in their functionality in the future. Therefore, we decide to not alter our original decision to got with a 2-tiered architecture.

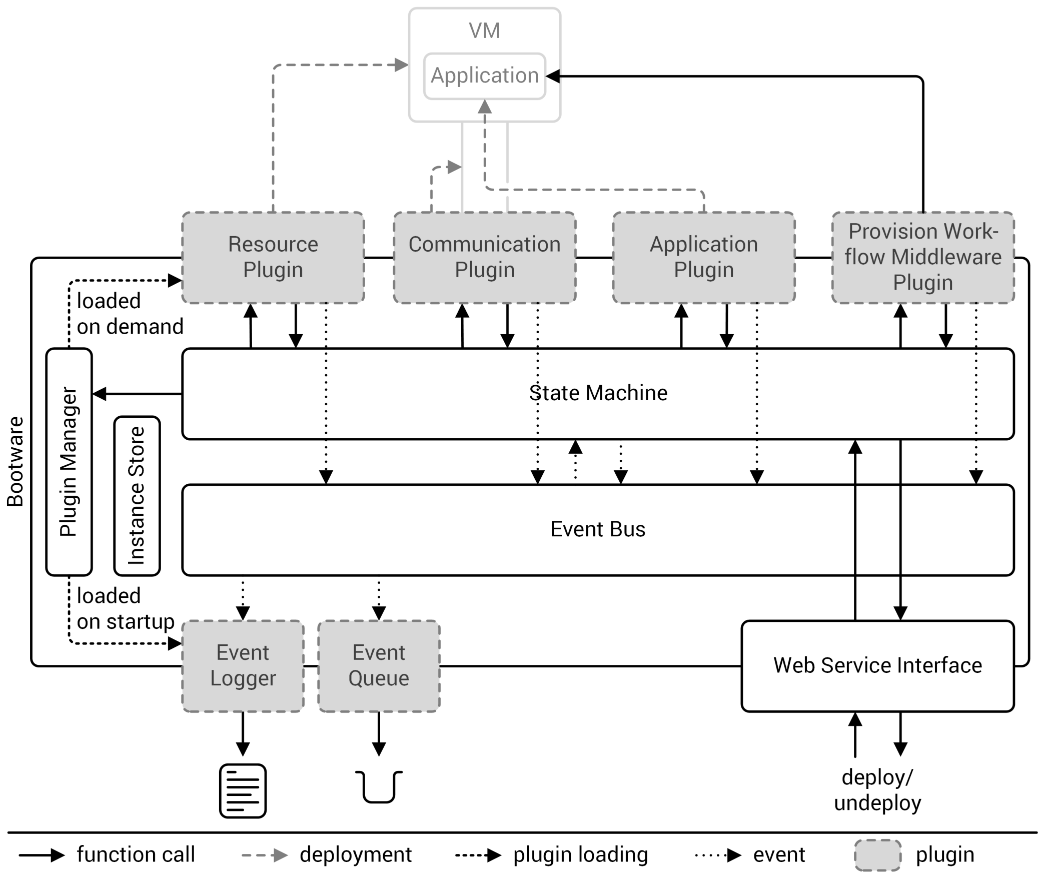

Figure 6.24: The final architecture of the remote bootware component.

7 Bootstrapping Process

This chapter describes the bootstrapping process in its entirety. The entire process can be divided into three phases. During the bootstrapping phase, the local and remote bootware components are started and deploy a provisioning engine, which in turn provisions the workflow middleware. Once the workflow middleware is ready, the second phase starts, which is the workflow execution phase. During this phase, the remote bootware might be called multiple times to deploy or undeploy new provisioning engines. The third and final phase, the shutdown phase, begins when the workflow execution is finished. In this phase, all remaining services, provisioning engines, the workflow middleware, the remote and the local bootware, as well as all the underlying resources are deprovisioned. 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.

At the beginning, a user starts the Modeler, which includes the bootware adapter, as seen on the bottom left of Figure 7.1. If they have not done so already, they configure the bootware adapter with their cloud login credentials to be used during the bootware process and other parameters that might be needed. They might also configure other aspects of the bootware through graphical user interfaces provided by the bootware adapter. They then use the Modeler to create a workflow as usual. Once the workflow is finished and ready to be executed, they click the start button as usual. This marks the beginning of the bootstrapping phase. The bootware adapter 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 adapter 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

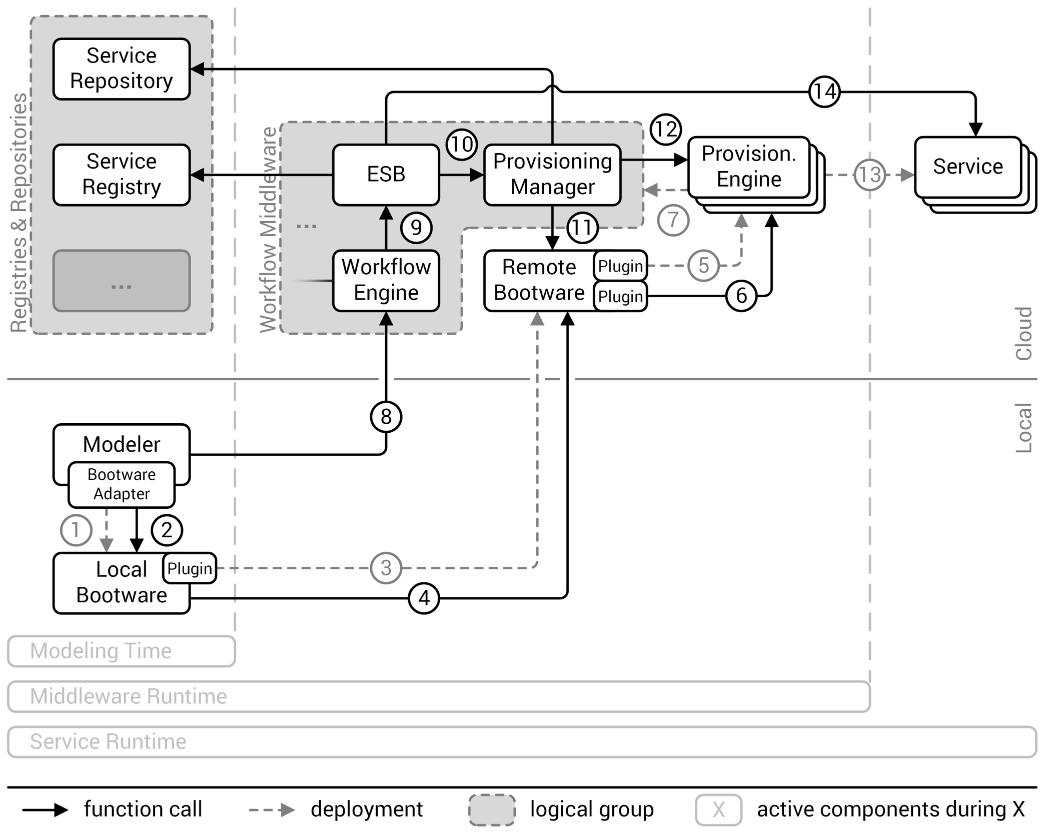

Figure 7.1: The step-by-step bootware process.

middleware is up and running, the provisioning engine returns the information about the workflow middleware, such as endpoint references, to the remote bootware, which in turn returns it to the local bootware, which returns it to the bootware adapter. The bootware adapter uses this information to link the modeler to the workflow middleware. This is the end of the bootstrapping phase. Now begins the workflow execution phase.

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 is not, 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 information about the provisioning engine, such as endpoint references, 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, which 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 9, 10, 11, 5, 12, 13 and 14). At some point, the workflow will be finished. This marks the end of the workflow execution phase and the start of the shutdown phase.

If it has not 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 adapter. At this point, no remote components should be running anymore. The local bootware now shuts down itself, which completes the whole process.

Figure 7.2 shows the bootstrapping phase as sequence diagram, which displays the interaction between the components arranged by time from top to bottom. The lifetime of a particular component is represented by a dashed line running from the top to the bottom. If an activity box is displayed over the line, the component is active at this moment in time. Activity is usually triggered by receiving a call from another component and ended by returning a response to this call. Calls are represented by arrows between activity boxes. The can be further distinguished between synchronous and asynchronous calls, depending on the form of the arrow head. A response is displayed as a dashed arrow between activity boxes. The end of the lifetime of a component (i.e. when it is shutdown) is marked by a cross that ends the lifetime line.

In Figure 7.2 we can clearly see how one component triggers the next one during the bootstrapping phase. Starting at the top left, the deploy action starts an escalating process, where one component starts the next, beginning with the modeler starting the bootware adapter. The bootware adapter then starts the local bootware and sends a deploy requests. The local

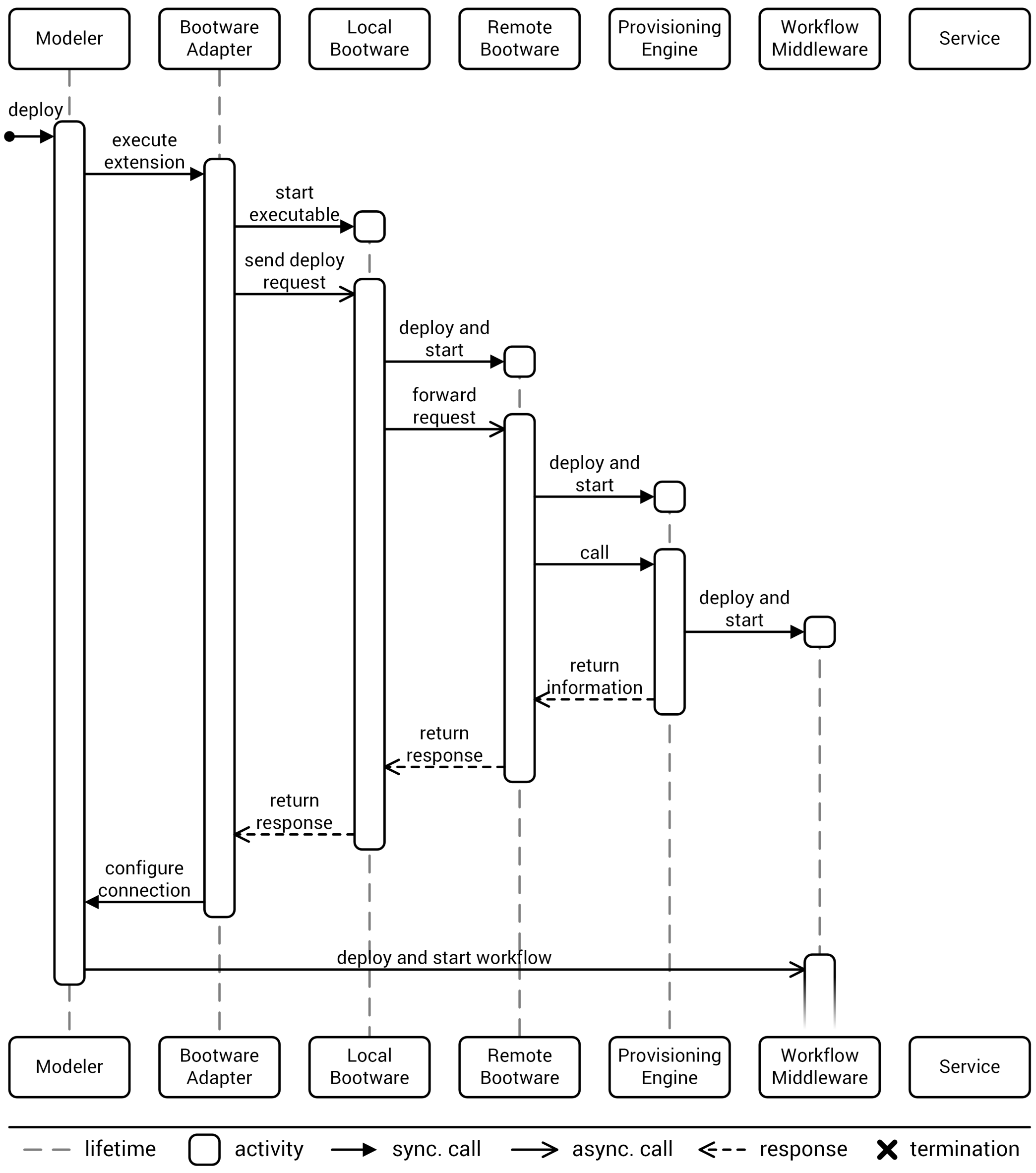

Figure 7.2: Sequence diagram of the bootstrapping phase.

bootware deploys and starts the remote bootware and forwards the deploy request. The remote bootware deploys and starts a provisioning engine, which it then calls to provision the workflow middleware. Once the workflow middleware is running, every component returns a response to the component which called it, which ends when the bootware adapter receives a response from the local bootware. This response contains endpoint references and other information about the workflow middleware, which the bootware adapter uses to configure the connection between the modeler and the workflow middleware. The modeler can now deploy and start the workflow on the workflow middleware, which concludes the bootstrapping phase.

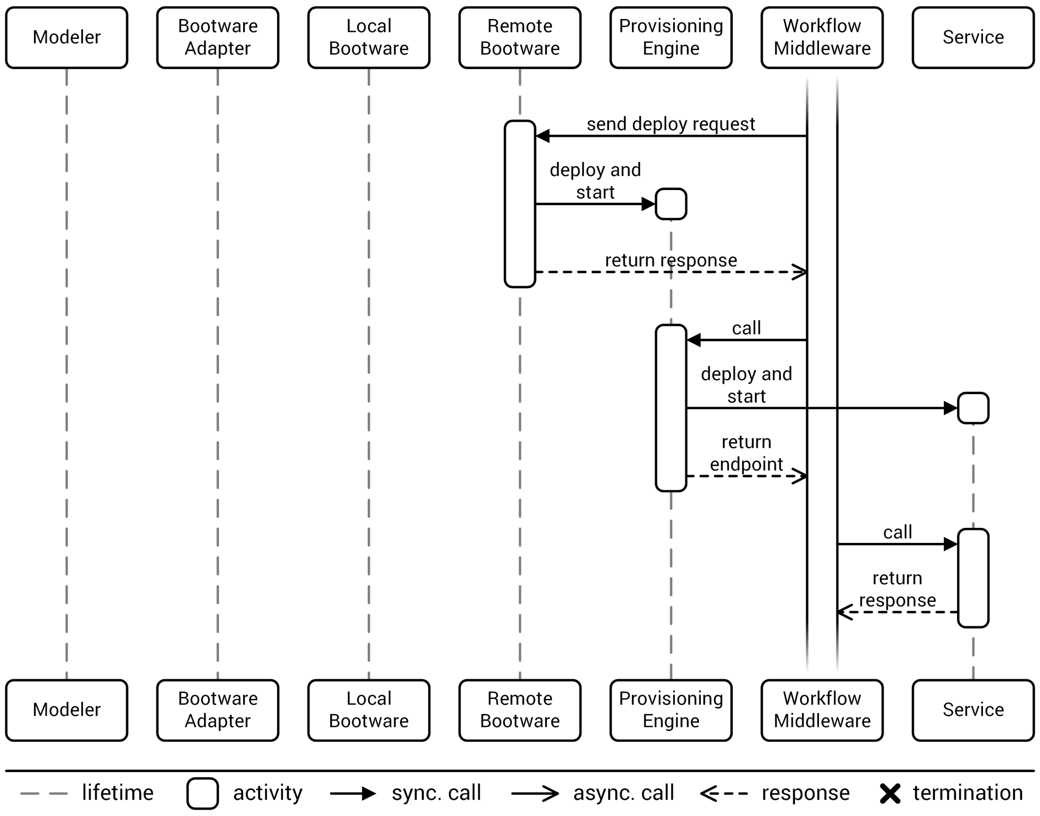

Figure 7.3: Sequence diagram of the workflow execution phase.

In the workflow execution phase, which is also depicted as a sequence diagram in Figure 7.3, the workflow middleware might now call external services. If these services do not exist already, the workflow middleware has to provision them first (via the provisioning manager). To provision a service, a particular provisioning engine is needed, which also might not exist yet. In this case, a deploy request is sent from the workflow middleware (i.e. the provisioning manager) to the remote bootware, which then deploys and starts the requested provisioning engine. Once the particular provisioning engine exists, the workflow middleware calls it to deploy the service it wants to execute. This process might be repeated multiple times during the whole workflow execution.

The shutdown phase begins once the workflow execution is finished. As we can see in Figure 7.4, the workflow engine might need to stop and undeploy remaining services by

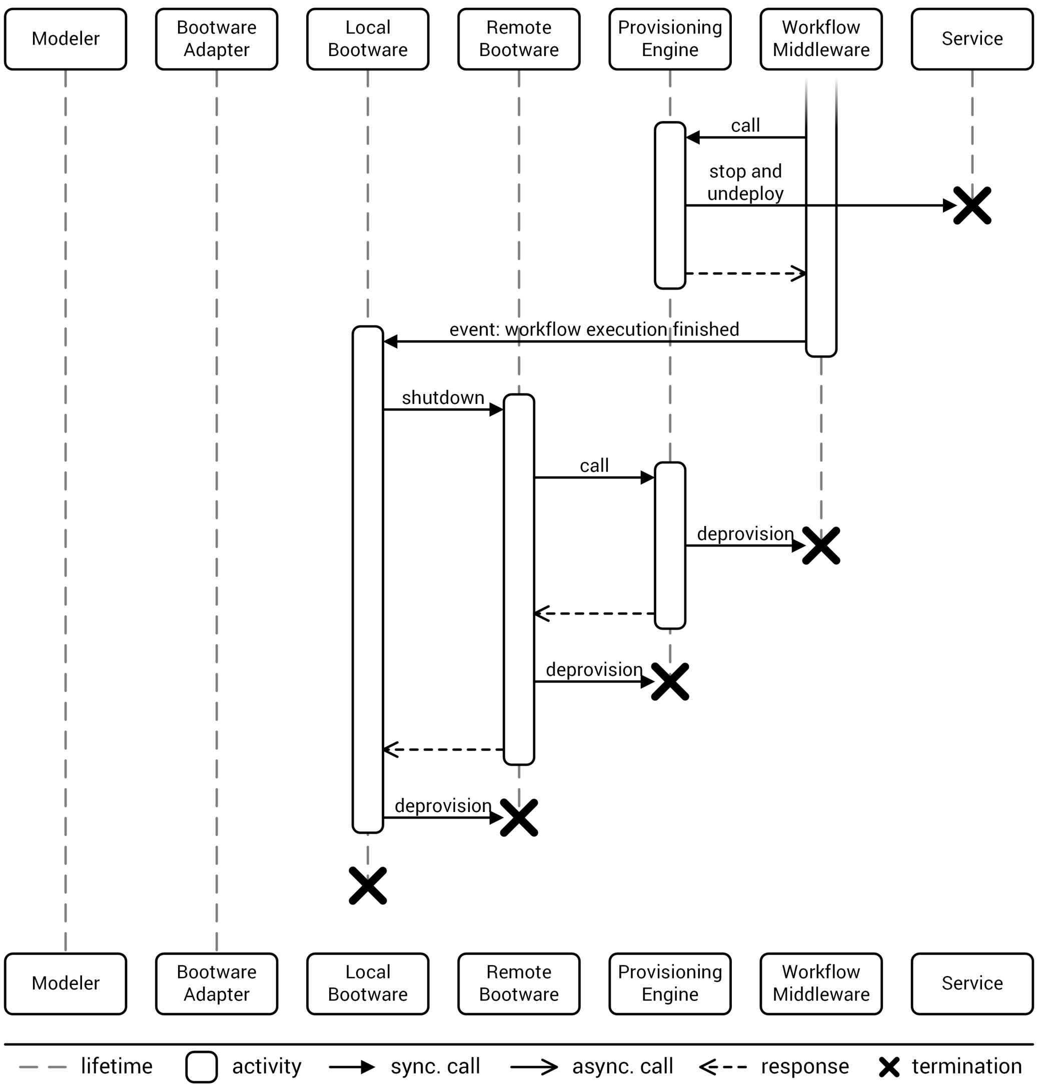

Figure 7.4: Sequence diagram of the shutdown phase.

calling the particular provisioning engines via the provisioning manager. Once all services are removed, the workflow execution is truly finished and an event marking this state is emitted. The local bootware has been listening for this event through one of its event plugins and triggers the removal of the remaining components by first calling the shutdown operation of the remote bootware. The remote bootware calls a provisioning engine to deprovision the workflow middleware, before deprovisioning all remaining provisioning engines itself. Once this is done, a response is sent to the local bootware, which can now deprovision the remote bootware, before shutting down itself. This concludes the shutdown phase.

8 Implementation

Until now we have described the bootware in a generic context because it should work with various different SWfMSs. 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.

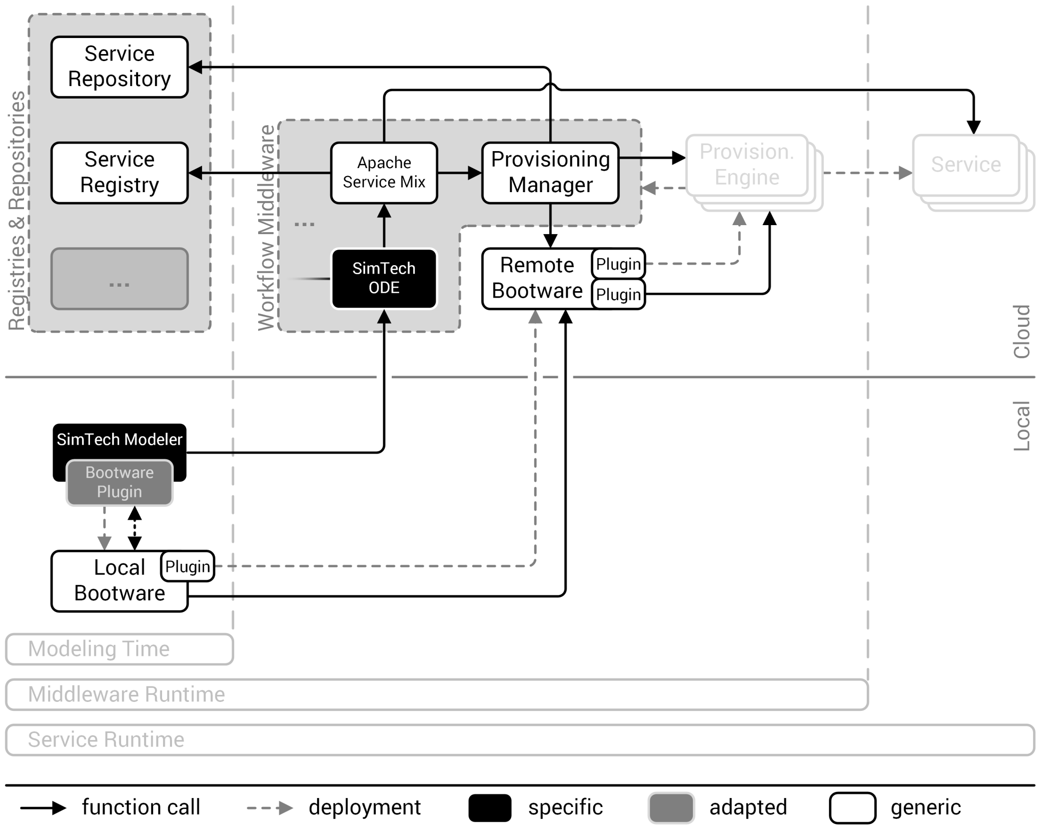

Figure 8.1: Specific and generic components and adapters.

The specific components, shown in black, are those components that belong to a specific SWfMS. 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. 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 either one 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, an implementation of the bootware adapter described in Section 6.2, 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 components. 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.

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 ready and should be executed, the user clicks on a button in the SimTech Modeler and the workflow is deployed and executed on the already running SimTech SWfMS. The bootware has to be integrated into this process. We described this as a generic bootware adapter in Section 6.2, but now we need an actual implementation of this adapter, which will be specific to the SimTech Modeler. The 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. We call it the *bootware plugin*. 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 do not 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 does not need the bootware functionality, they do not have to load the bootware plugin and the SimTech plugin will continue to function as before.

plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.0"?>
<plugin>

  <extension-point id="hoverHelpers"
                   name="%HOVERHELP_HELPER_NAME"
                   schema="schema/hoverHelpers.exsd"/>
  <extension-point id="expressionEditors"
                   name="%EXPRESSION_LANGUAGE_EDITORS"
                   schema="schemas/expressionEditors.exsd"/>
  <extension-point id="actions"
                   name="%ACTIONS_NAME"
                   schema="schemas/actions.exsd"/>
  <extension-point id="modelListener"
                   name="Model Listener"
                   schema="schemas/modelListener.exsd"/>
  <extension-point id="uiObjectFactories"
                   name="UIObjectFactories"
                   schema="schemas/uiObjectFactories.exsd"/>
  <extension-point id="bootware"
                   name="Bootware"
                   schema="schema/bootware.exsd"/>

...
```

Listing 8.1: Extension points defined by the org.eclipse.bpel.ui plugin.

The second scenario looks preferable to the first one, so this is what we are going to do. We 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 SimTech SWfMS before the

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

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.

The existing Eclipse plugin that has to be modified is the *org.eclipse.bpel.ui* plugin. In its *plugin.xml*, it has already defined some extension points, as can be seen in Listing 8.1 in line 5-19. We add another extension point for the bootware, as shown in line 20-22. Now, we have to integrate this extension point into the already existing deploy process that is executed when the user click the start button in the SimTech Modeler. This button and the class that implements its functionality are defined further down in the *plugin.xml*, as shown in Listing 8.2.

plugin.xml

```
1185 ...
1186
1187 <action
1188   class="org.eclipse.bpel.ui.agora.actions.StartAction"
1189   disabledIcon="icons/elcl16/resume_co.gif"
1190   enablesFor="*"
1191   icon="icons/elcl16/resume_co.gif"
1192   id="org.eclipse.bpel.ui.agora.start.action"
1193   label="Start"
1194   menubarPath="simTech/process/action"
1195   style="push"
1196   toolbarPath="simTech/process/action"
1197   tooltip="Starts the process instance execution">
1198 </action>
1199
1200 ...
```

Listing 8.2: Definition of the action button.

As we can see in line 1188, the class that implements the button functionality is the *StartAction* class. We modify its run method to load and execute any plugin that implements the bootware extension point, before the original deploy code continues. As shown in Listing 8.3, we have to get all extensions that implement the bootware extension point from the extension registry (line 50-52) and create an object of the *IBootwarePlugin* type (line 55-56). Now, we are able to call any method defined by this object, in this case the *execute* method (line 57). After this method has finished, the original code continues its execution (line 60).

```
StartAction.java
...

  public void run(IAction arg0) {

    fEditor.refreshEditor();

    IExtensionRegistry reg = Platform.getExtensionRegistry();
    IConfigurationElement[] extensions =
      reg.getConfigurationElementsFor("org.eclipse.bpel.ui.bootware");
    for (int i = 0; i < extensions.length; i++) {
      IConfigurationElement element = extensions[i];
      IBootwarePlugin plugin =
        (IBootwarePlugin) element.createExecutableExtension("class");
      plugin.execute(); // Can be any method defined in IBootwarePlugin
    }

    // continue with original code
...
```

Listing 8.3: The modified run method in the *StartAction* class.[2]

Now that we have all code in place to execute the bootware extension, all we have to do is to create a bootware plugin that implements the bootware extension point and the *execute* method. Like the Eclipse plugin we just modified, the bootware plugin has a *plugin.xml*, shown in Listing 8.4. Here, we just declare an extension in line 8-13 that implements the bootware extension point (line 9) with the *org.simtech.bootware.eclipse.BootwarePlugin* class (line 11). This is also the place where other integration functionality could be implemented in the future, for example by adding new menus for configuring the context object, or new views that show the bootstrapping process.

The *BootwarePlugin* class, shown in Listing 8.5, implements the execute method called by the extension point. In this method we do everything we need to do to integrate the bootware into the start process. Due to limited space we cannot present the actual code here, but the process is roughly as follows: First, the local bootware has to be started by calling the executable. Once it is running, a deploy request is sent to it, containing a context object with all necessary configuration parameters. Now, the bootware plugin has to wait for the deploy request to be executed. Once the request is finished, the endpoint references to

[2]Note: The code shown here was shortened for presentation and is not complete. The main elements are however present.

plugin.xml
```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin
  name="SimTech Bootware Eclipse Plugin"
  id="org.simtech.bootware.eclipse"
  version="1.0.0">

  <extension
    point="org.eclipse.bpel.ui.bootware">
    <execute
        class="org.simtech.bootware.eclipse.BootwarePlugin">
    </execute>
  </extension>

</plugin>
```

Listing 8.4: The bootware plugin plugin.xml.

various workflow middleware components and other information returned in the response message are used to set up the connection from the SimTech modeler to the middleware. The bootstrapping process is now finished and the original deploy code continues.

BootwarePlugin.java
```java
public class BootwarePlugin implements IBootwarePlugin {

  public final void execute() {
    // Start local bootware.
    // Get context.
    // Send deploy request with context.
    // Wait for request to finish.
    // Set up URLs to workflow middleware.
  }

}
```

Listing 8.5: The bootware plugin implementation.

8.2 Bootware Core Library

We have seen in Section 6.11 and Section 6.12 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 this 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.

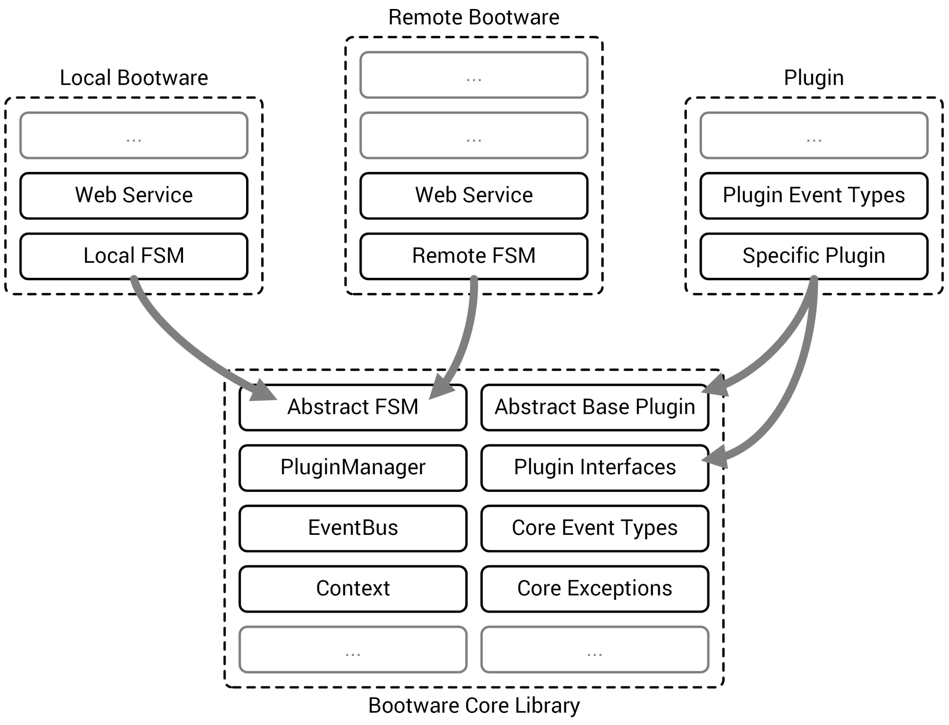

Figure 8.2: The bootware core library and exemplary usage.

Because 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 FSM 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 activities shown in Figure 6.20 and Figure 6.21. This way, the local and remote bootware can import shared activities from the library and only have to define their custom activities (e.g.

the *provision middleware activity* in the remote bootware or the *send to remote* activity in the local bootware) and the transitions. They can also overwrite the activities 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 inherit this common functionality. They also have to implement one of the plugin interfaces defined in the bootware core library, so for example, a resource plugin has to implement the resource 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 frameworks and libraries we will use to implement the requested functionality. In this section we present the frameworks and libraries we chose and the reasoning behind it. We begin with plugin frameworks, followed by PubSub and FSM libraries.

8.3.1 Plugin Frameworks

All 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 are then 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 because 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 just load every plugin at startup, switch between them internally when necessary, and never unload them. However, 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 would not 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. For example, we can imagine 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 diploma thesis, but it is 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[3]	JSPF[4]	JPF[5]	OSGi[6]
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.

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

Table 8.1 shows a comparison of four Java plugin frameworks, the first of which is the Service Provider Interface (SPI)[7]. It is an extension mechanism integrated in Java which is a little more advanced than 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, does not depend on any external libraries, is well documented, and mature because it is used in the Java Runtime Environment (JRE). One could also say that it is somewhat standardized because it is a 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.

The next contender is the Java Simple Plugin Framework (JSPF)[4], an open-source plugin framework build for small to medium sized projects. Its focus is simplicity and the author explicitly states that it is not intended to replace JPF or OSGi[8]. 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)[5] is another open-source plugin framework. Compared to JSPF it is 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)[6], a plugin framework standard developed by the OSGi Alliance. It provides a general-purpose Java framework that supports the deployment of extensible bundles [27]. 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.

As 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[9], Eclipse Equinox[10], and Knopflerfish[11]. All of

[7]`http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html`
[8]`https://code.google.com/p/jspf/wiki/FAQ`
[9]`http://felix.apache.org`
[10]`http://eclipse.org/equinox`
[11]`http://www.knopflerfish.org`

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 because 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 because they all implement the same specification and should therefore be - at least in theory - completely interchangeable.

8.3.2 PubSub Libraries

Many of the well know messaging middlewares offer support for PubSub, for example ActiveMQ[12], RabbitMQ[13], and ZeroMQ[14]. But, because we are looking for an internal communication mechanism only, all of these solutions are somewhat overpowered. We do not have to worry about network problems, so we do not need guaranteed delivery or message queuing capabilities. We also do not need persistence or transactional capabilities We do not have to handle millions of subscribers or events, so high scalability is not a concern. We do not even necessarily need asynchronous communication. Instead, we need a lightweight in-memory solution. Therefor we will ignore the middleware heavyweights and look for smaller 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 because 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 do not 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 are finished. Only then should the program continue execution. But asynchronous invocation might still be useful in some cases, so we also add it here.

[12]`http://activemq.apache.org`
[13]`http://www.rabbitmq.com`
[14]`http://zeromq.org`

		PubSub Libraries				
		EventBus[15]	Guava Event Bus[16]	Simple Java Event Bus[17]	MBassador[18]	Mycila PubSub[19]
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. However, outdated coding practices and many features also make it fairly heavyweight. Guava Event Bus on the other hand is a rather 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 does not support weak references and filtering, which does not 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 easier to use. Compared to the other libraries it is not that popular and lacks in documentation. MBassador is a light-weight and performance minded PubSub library. As we can see in Table 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

[15]`http://eventbus.org/` (Site was offline when last checked.)
[16]`https://code.google.com/p/guava-libraries/wiki/EventBusExplained`
[17]`https://code.google.com/p/simpleeventbus/`
[18]`https://github.com/bennidi/mbassador`
[19]`https://github.com/mycila/pubsub`

enough, but because it is relatively new, it is not very popular yet and may lack in maturity. From the alternatives presented here, MBassador seems to be the only one that offers all the functionality we need combined with relative maturity and good documentation. We will therefore use it for our implementation.

8.3.3 State Machine Libraries

Because we want to implement the bootware process with a FSM, 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 diploma 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 do not want the FSM to be overly complex to use and want to avoid introducing additional conversion or compilation steps. Table 8.3 shows six state machine libraries available for Java.

Apache Commons SCXML[21] aims to be a java state machine engine that is capable of executing state machines defined in 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) [33]. Apache Commons SCXML looks like a good match for our needs because it is event-based and also uses a (soon to be) standard. But the current state of the implementation seems to be lacking because the SCXML specification has changed a lot. The most recent release is version 0.9, which was released in late 2008. It is about 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[20].

EasyFlow[22] 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. There also is State Machine Compiler (SMC)[23], a state machine compiler that targets fifteen different 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 seem to be to complicated for our needs.

[20]`http://commons.apache.org/proper/commons-scxml/roadmap.html`

		State Machine Libraries					
		Commons SCXML[21]	EasyFlow[22]	SMC[23]	stateless4j[24]	squirrel-foundation[25]	Unimod[26]
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.

Stateless4j[24] is a lightweight library for creating FSMs directly in Java code. Compared to the other alternatives, it lacks in documentation and does bot seem to be very popular. Squirrel-foundation[25] 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. Finally, there is Unimod[26], 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 too complex for our needs.

From the alternatives presented here, Apache Commons SCXML would be our first choice if the standard and the implementation were more mature. However, at this point in time this is not the case. For this diploma thesis we will use squirrel-foundation to implement the state machine. If Apache Commons SCXML becomes a viable option in the future, replacing squirrel-foundation could be considered. As it supports exporting the state machine as SCXML, this could be used to ease a possible transition.

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

8.4 Context

context.xml

```xml
<context>
  <resourcePlugin>aws-ec2.jar</resourcePlugin>
  <communicationPlugin>ssh.jar</communicationPlugin>
  <applicationPlugin>opentosca.jar</applicationPlugin>
  <!--Optional:-->
  <provisionWorkflowMiddlewarePlugin>
    call_opentosca.jar
  </provisionWorkflowMiddlewarePlugin>
  <!--Optional:-->
  <servicePackageReference>
    ../opentosca.csar
  </servicePackageReference>
  <!--Optional:-->
  <configurationList>
    <entry>
      <key>aws</key>
      <value>
        <configuration>
          <entry>
            <key>secretKey</key>
            <value>874w5zhpswe98tzhg0w87ser049tadsiph</value>
          </entry>
          <entry>
            <key>accessKey</key>
            <value>g9w276og9746gw5</value>
          </entry>
          <entry>
            <key>instanceType</key>
            <value>t2.medium</value>
          </entry>
          ...
        </configuration>
      </value>
    </entry>
  </configurationList>
</context>
```

Listing 8.6: Sample context represented in XML.

Listing 8.6 shows an exemplary context generated by the bootware in XML form. As we can see in line 2-4, it defines the resource, connection, and application plugins that should be used during the bootstrapping process by supplying the name of the plugin *.jar*. It can also contain a provision workflow middleware plugin, as can be seen in line 6-8. This is optional and will only be used on the first request, when the remote bootware will also call a provisioning engine to provision the workflow middleware. This is also where the service package reference in line 10-12 will be used, which points to the workflow middleware package that should be provisioned by the provisioning engine called by the provision workflow middleware plugin. In line 14-35 we can also see the optional configuration list. It contains configuration values that are passed to plugins if required. In this case, it contains login credentials for Amazon's cloud, shown in line 19-26, which are used by the *aws-ec2.jar* plugin to authenticate its requests made to Amazon. In line 27-30 we can also see an instance type parameter. This and other parameters will also be read by the *aws-ec2.jar* plugin.

8.5 Web Service Interface

In Section 6.3 we decided to use web service calls and returns as external communication mechanism. 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 6.6 we also described the *setConfiguration* operation that has to be supported. Additionally, the *getActiveApplications* and *shutdown* operations are needed.

8.5.1 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 8.7 shows an exemplary deploy request as SOAP message. In line 6 we can see that the deploy method is called with a request context provided as argument in line 7-11, which will be used by the bootware to generate a full context like the one shown in Listing 8.6. In this particular example, only the *resourceProvider* and *application* parameters are specified, which could be a call from the provisioning manager.

`deploy-request.xml`

```xml
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:rem="http://remote.bootware.simtech.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <rem:deploy>
      <context>
        <resourceProvider>aws</resourceProvider>
        <application>opentosca</application>
      </context>
    </rem:deploy>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 8.7: Sample *deploy* request in a SOAP message.

The response that is returned once the request has been executed successfully is shown in Listing 8.8. It contains an information list in line 5-10, which contains a reference to the application that was deployed during the request, in this case OpenTOSCA.

`deploy-response.xml`

```xml
1  <S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
2    <S:Body>
3      <ns2:deployResponse xmlns:ns2="http://remote.bootware.simtech.org/">
4        <return>
5          <informationList>
6            <entry>
7              <key>opentosca</key>
8              <value>http://aws.com:8080/</value>
9            </entry>
10         </informationList>
11       </return>
12     </ns2:deployResponse>
13   </S:Body>
14 </S:Envelope>
```

Listing 8.8: Sample *deploy* response in a SOAP message.

If the deploy request somehow failed, a SOAP message containing a SOAP fault will be returned, which is shown in Listing 8.9. 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-error.xml

```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>resourceProvider cannot be empty</faultstring>
6        <detail>
7          <ns2:DeployException
8            xmlns:ns2="http://remote.bootware.simtech.org/">
9            <message>resourceProvider cannot be empty</message>
10         </ns2:DeployException>
11       </detail>
12     </S:Fault>
13   </S:Body>
14 </S:Envelope>
```

Listing 8.9: Sample *deploy* error in a SOAP message.

8.5.2 Undeploy

The *undeploy* operation will be called by multiple components to reverse the actions that where previously made by *deploy* operations. Listing 8.11 shows an exemplary undeploy request in a SOAP message. As argument it contains one or more endpoint references to already deployed applications, as can be seen in line 7-12.

undeploy-response.xml

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

Listing 8.10: Sample *undeploy* response in a SOAP message.

undeploy-request.xml

```xml
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:rem="http://remote.bootware.simtech.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <rem:undeploy>
      <endpoints>
        <entry>
          <key>opentosca</key>
          <value>http://aws.com:8080/</value>
        </entry>
      </endpoints>
    </rem:undeploy>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 8.11: Sample *undeploy* request in a SOAP message.

When all applications have been undeployed successfully, a response will be send, as shown in Listing 8.10. The response is empty because there is nothing interesting to return.

undeploy-error.xml

```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>Undeploy operation failed</faultstring>
      <detail>
        <ns2:UndeployException
          xmlns:ns2="http://remote.bootware.simtech.org/">
          <message>Undeploy operation failed</message>
        </ns2:UndeployException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

Listing 8.12: Sample *undeploy* error in a SOAP message.

In case of a failure, an error will be return. As can be seen in Listing 8.12, 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.

8.5.3 Set Configuration

In addition to the main *deploy* and *undeploy* operations, the bootware web service also supports the *setConfiguration* operation. Using this operation, the configuration can be set independently from deploy requests if necessary. Listing 8.14 shows an exemplary *setConfiguration* request. In line 7-23, it contains a configuration list, which can contain one or more configuration sets. Each configuration set is made up of one or more configuration entries, which are key value pairs, where the key describes the configuration type and the value the actual configuration value. What content a particular key has to contain depends on what the plugins are looking for when they read the configuration. In the example code in line 9, we send one configuration set for AWS, which consists of two credentials, a *secretKey* in line 12-15 and an *accessKey* in line 16-19. Configuration content like this is the reason why the communication with the bootware should be encrypted.

setConfiguration-error.xml

```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>Configuration could not be set</faultstring>
      <detail>
        <ns2:SetConfigurationException
          xmlns:ns2="http://remote.bootware.simtech.org/">
          <message>Configuration could not be set</message>
        </ns2:SetConfigurationException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

Listing 8.13: Sample *setConfiguration* error in a SOAP message.

setConfiguration-request.xml

```xml
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:rem="http://remote.bootware.simtech.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <rem:setConfiguration>
      <configurationList>
        <entry>
          <key>aws</key>
          <value>
            <configuration>
              <entry>
                <key>secretKey</key>
                <value>874w5zhpswe98tzhg0w87ser049tadsiph</value>
              </entry>
              <entry>
                <key>accessKey</key>
                <value>g9w276og9746gw5</value>
              </entry>
            </configuration>
          </value>
        </entry>
      </configurationList>
    </rem:setConfiguration>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 8.14: Sample *setConfiguration* request in a SOAP message.

setConfiguration-response.xml

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

Listing 8.15: Sample *setConfiguration* response in a SOAP message.

If the *setConfiguration* operation was successful, the response in Listing 8.15 will be returned. Again, it is empty because there is nothing interesting to return. Like the *deploy* and *undeploy* operations, the *setConfiguration* operation also returns an error message if the operation failed. As can be seen in Listing 8.13, it also contains a SOAP fault string in line 5 and the original *SetConfigurationException* thrown by the *setConfiguration* operation in line 7-10.

8.5.4 Get Active Applications

The *getActiveApplications* operation is called by the provisioning manager to retrieve already deployed provisioning engines. If a provisioning engine it needs is already active, it does not have to call the bootware to provision a new one. As already explained in Section 6.7 this is only needed in the remote bootware and therefore we only implement it there. Listing 8.16 shows a *getActiveApplications* request in a SOAP message. No parameters are required.

getActiveApplications-request.xml

```xml
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:rem="http://remote.bootware.simtech.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <rem:getActiveApplications/>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 8.16: Sample *getActiveApplications* request in a SOAP message.

The response that is returned contains a list of all applications that where active when the request was made. As we can see in Listing 8.17 lines 6-11, it contains an applications list with zero or more entries. Each entry consists of a key value pair, where the key identifies the application and the value contains a URL to the application. In this example, the entry points to an OpenTOSCA container instance. If the *getActiveApplications* request failed for some reason, an error message is returned. As can be seen in Listing 8.18, it contains a SOAP fault string in line 5 and the original *GetActiveApplicationsException* thrown by the *getActiveApplications* operation in line 7-10.

getActiveApplications-response.xml

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

Listing 8.17: Sample *getActiveApplications* response in a SOAP message.

getActiveApplications-error.xml

```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>Error retrieving active applications</faultstring>
      <detail>
        <ns2:GetActiveApplicationsException
          xmlns:ns2="http://remote.bootware.simtech.org/">
          <message>Error retrieving active applications</message>
        </ns2:GetActiveApplicationsException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

Listing 8.18: Sample *getActiveApplications* error in a SOAP message.

8.5.5 Shutdown

The *shutdown* operation triggers the shutdown procedure. During this procedure, all active applications will be undeployed. The local bootware will also forward this request to the remote bootware and wait for a response so that it can deprovision the remote bootware before shutting down itself. Listing 8.19 shows a shutdown request in a SOAP message. No parameters are required.

shutdown-request.xml

```xml
<soapenv:Envelope
  xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:rem="http://remote.bootware.simtech.org/">
  <soapenv:Header/>
  <soapenv:Body>
    <rem:shutdown/>
  </soapenv:Body>
</soapenv:Envelope>
```

Listing 8.19: Sample *shutdown* request in a SOAP message.

If the additional processes executed during shutdown (i.e. undeploy applications or middleware) were successful, the response in Listing 8.20 will be returned.

shutdown-response.xml

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

Listing 8.20: Sample *shutdown* response in a SOAP message.

If the additional processes failed for some reason, an error response like the one showed in Listing 8.21 will be returned. It contains a SOAP fault string in line 5 and the original *ShutdownException* thrown by the *shutdown* operation in line 7-10.

```
shutdown-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>Shutdown operation failed</faultstring>
      <detail>
        <ns2:ShutdownException
          xmlns:ns2="http://remote.bootware.simtech.org/">
          <message>Shutdown operation failed</message>
        </ns2:ShutdownException>
      </detail>
    </S:Fault>
  </S:Body>
</S:Envelope>
```

Listing 8.21: Sample *shutdown* error in a SOAP message.

8.6 State Machine

The state machine we use to implement the bootware execution flow is divided into two parts. We have a generic part that is shared by both the local and remote bootware. This part is defined in the *AbstractStateMachine* class that is part of the bootware core library. The second part, which is specific to either the local or remote bootware, is defined in their respective implementations.

The *AbstractStateMachine* class defines some utility functions for starting and stopping the state machine. It also contains the *buildDefaultTransition* method, which simplifies the definition of most of the transitions in Figure 6.20 and Figure 6.21. As we can see when looking at these two figures, many of the activities have a *success* and a *failure* transition. With the *buildDefaultTransition* method, states and transitions associated with these activities can be defined with less code.

However, the most important part of the *AbstractStateMachine* is the abstract class *Abstract-Machine*. This class defines all the functions that are called in states that are shared by the local and remote bootware, so that we avoid code duplication. For example, as we can see when looking at Figure 6.20 and Figure 6.21, both bootwares share the *connect* activity at the bottom left. Listing 8.22 shows how the function associated with this activity is defined in the *AbstractStateMachine* class. We can see in line 198 that the connect method of a communication plugin is called and the resulting connection is stored in a variable for later use. If this succeeds, a *success* event is fired in the state machine (line 203), which would trigger a transition to the next state, which in this case is the *provision application* state. However, if for some reason no connection can be established by the communication plugin's connect method, a *ConnectConnectionException* is thrown. This would trigger a failure event in the state machine (line 201), which would lead to a transition to the *disconnect* state. We can see how the result of some code execution can influence the transitions in the state machine. The functions for other shared states are implemented in a similar fashion.

AbstractMachine.java

```
192 ...
193
194   protected void connect(final String from,
195                          final String to,
196                          final String fsmEvent) {
197     try {
198       connection = communicationPlugin.connect(instance);
199     }
200     catch (ConnectConnectionException e) {
201       stateMachine.fire(StateMachineEvents.FAILURE);
202     }
203     stateMachine.fire(StateMachineEvents.SUCCESS);
204   }
205
206 ...
```

Listing 8.22: An excerpt showing the connect function in the *AbstractMachine* class.

Both the local and the remote bootware extend the *AbstractStateMachine* and *AbstractMachine* classes in their implementations. If they define other states for which no functions are defined in the *AbstractMachine* class, they can just add these new functions. They can also override existing functions if they need to. For example, the local bootware adds the *sendToRemote* function to its implementation of the *AbstractMachine* class because the send remote activity is unique to the local bootware. To complete the implementation of their particular state machines, the local and remote bootware also have to define their states and transitions. They can use the already mentioned *buildDefaultTransition* function defined in the *AbstractStateMachine* for the common success-failure transitions, or the original syntax for other transitions.

8.7 Plugin Manager

The plugin manager is a thin wrapper class around the Apache Felix OSGi framework. It encapsulates all the functionality required for loading and unloading OSGi plugins. Table 8.4 lists all operations offered by the plugin manager.

Operation	Input	Output	Description
PluginManager	-	PluginManager Instance	Initializes, configures, and starts the OSGi framework
registerShared-Object	Object	-	Register an object that should be shared with plugins
loadPlugin	Path	Plugin Instance	Loads the plugin at the given path
unloadPlugin	Path	-	Unloads the plugin at the given path
unloadAllPlugins	-	-	Unloads all loaded plugins
stop	-	-	Unloads all plugins and stops the OSGi framework

Table 8.4: Operations offered by the plugin manager.

The constructor (*PluginManager*) creates a new plugin manager instance. In the background, it initializes the OSGi framework. Part of this initialization is telling the framework which extra packages it should export. This is necessary so that plugins can resolve their dependencies on packages that are part of the *bootware core library*. Listing 8.23 shows an excerpt of the plugin manager class where we can see the extra packages that are exported in line 42-48. Plugins have dependencies on various bootware core packages shown here, such as the exceptions and plugins package. They also depend on some packages from the PubSub library we use, MBassador. All these dependencies are resolved by configuring the OSGi framework to export these packages.

The *registerSharedObjects* operation allows us to register any object that is part of the bootware core with the OSGi framework, so that plugins are also able to access it. We use this to share the EventBus instance with all plugins, so that they are able to subscribe to, and also publish, events. The *loadPlugin* operation loads the plugin at the given path into the OSGi framework and returns an instance of this plugin. The *unloadPlugin* operation unloads an already loaded plugin. The plugin manager also offers an *unloadAllPlugins* operation that unloads all loaded plugins at once. This operation is also called during the plugin manager's *stop* operation, which stops the OSGi framework, which is necessary for an orderly shutdown.

PluginManager.java

```java
...

final String extraPackages =
    "org.simtech.bootware.core;version=1.0.0,"
  + "org.simtech.bootware.core.events;version=1.0.0,"
  + "org.simtech.bootware.core.exceptions;version=1.0.0,"
  + "org.simtech.bootware.core.filters;version=1.0.0,"
  + "org.simtech.bootware.core.plugins;version=1.0.0,"
  + "net.engio.mbassy.listener;version=1.1.2,"
  + "net.engio.mbassy.common;version=1.1.2";
config.put(Constants.FRAMEWORK_SYSTEMPACKAGES_EXTRA, extraPackages);

...
```

Listing 8.23: Extra packages exported by the plugin manager.

All these operations are called at specific points during the state machine execution to load and unload the needed plugins. For example, once the state machine enters the *unload event plugins* state, associated with the activities shown in the top right of Figure 6.20 and Figure 6.21, it executes the *unloadEventPlugins* method, shown in Listing 8.24. As we can see in line 293, it just calls the plugin manager's *unloadAllPlugins* operation, which will unload all remaining plugins (which should be only event plugins at this point). We can also see that exceptions are used to control the state machine transitions. If all plugins are unloaded successfully and no exception is thrown, a *success* event is fired (line 298), which will cause a transition in the state machine, in this case to the *cleanup* state. However, if somehow the *unloadAllPlugins* operation fails, it throws a *UnloadPluginsException*, which is caught and triggers a *failure* event (line 296). In this way, the result of the plugin manager operations can influence the execution flow of the bootware.

```
AbstractStateMachine.java
...

protected void unloadEventPlugins(final String from,
                                  final String to,
                                  final String fsmEvent) {
  try {
    pluginManager.unloadAllPlugins();
  }
  catch (UnloadPluginException e) {
    stateMachine.fire(StateMachineEvents.FAILURE);
  }
  stateMachine.fire(StateMachineEvents.SUCCESS);
}

...
```

Listing 8.24: The *unloadEventPlugins* method defined in the *AbstractStateMachine* class.

8.8 Plugins

Now, we will describe the implementation of a few plugins. We implemented a resource plugin that can create and remove EC2 instances in Amazon's cloud. We created a communication plugin that allows the bootware to connect to a remote system via SSH and then execute commands on, or upload files to this system. We also implemented two application plugin, one for the remote bootware itself and one for OpenTOSCA. Additionally, we created event plugins, for example a file logger plugin that logs bootware events into a text file.

8.8.1 AWS EC2 Plugin

This resource plugin allows the bootware to create and remove EC2 instances in Amazon's cloud. It uses the AWS SDK for Java[27] to implement this functionality. This SDK specifies a specific set of action that have to be taken to start an EC2 instance, which we map onto the operations defined by each resource plugin (i.e.: *initialize*, *shutdown*, *deploy*, and *undeploy*, as described in Section 6.5). Figure 8.3 shows a simplified overview of these actions and how they map onto the resource plugin operations.

[27] `http://aws.amazon.com/sdkforjava/`

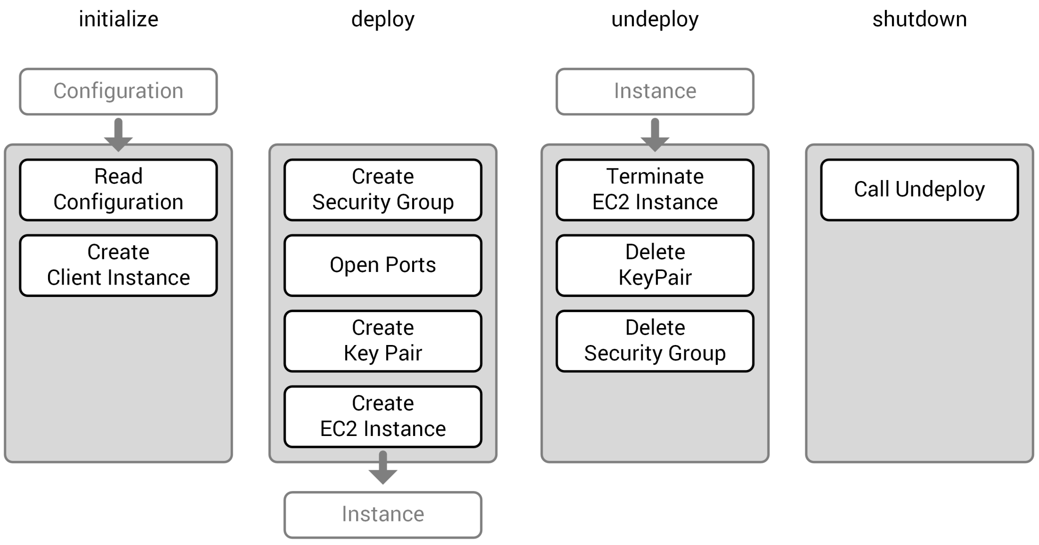

Figure 8.3: The operations implemented by the AWS EC2 plugin.

The *initialize* operation, shown on the left of Figure 8.3, which is called once when the plugin is loaded, creates a client instance, which is an object on which all the following actions will be called. The client instance is bound to a specific AWS region, which is read from the configuration object that is passed into the *initialize* operation. As we can see in the *deploy* operation in Figure 8.3, we first have to create a security group[28]. Security groups are essentially virtual firewalls that allow or deny traffic to and from all EC2 instances associated with it. EC2 instances have to be associated with a security group, so we have to create one. In the next step we open all ports in this security group that we later want to use for communication. Which ports we open is determined by reading the configuration object. We also have to create a SSH key pair and retrieve the private key, which we later use when we connect to this EC2 instance via SSH. In the last step we create the actual EC2 instance. Once it is up and running, the *deploy* operation is finished and returns an instance object which contains the URL where the EC2 instance can be reached, as well as the private key for SSH access. The *undeploy* operation reverses the *deploy* operation. First, it terminates the EC2 instance. Once the instance is stopped, the key pair and the security group that were created earlier are removed. We do not have to close the ports we opened, because they are part of the security group and do not exist anymore once the security group is removed. After this, the EC2 instance created earlier is successfully removed. There are no further actions necessary during the *shutdown* operation, but for safety we call the *undeploy* operation, in case it was not called earlier.

[28] `http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-network-security.html`

8.8.2 SSH Plugin

This communication plugin allows the bootware to connect to a remote system via SSH. It uses the Ganymed SSH-2 library[29], which implements the SSH-2 protocol in Java. Figure 8.4 shows a simplified overview of the actions necessary to create a SSH connection and how they map onto the communication plugin operations.

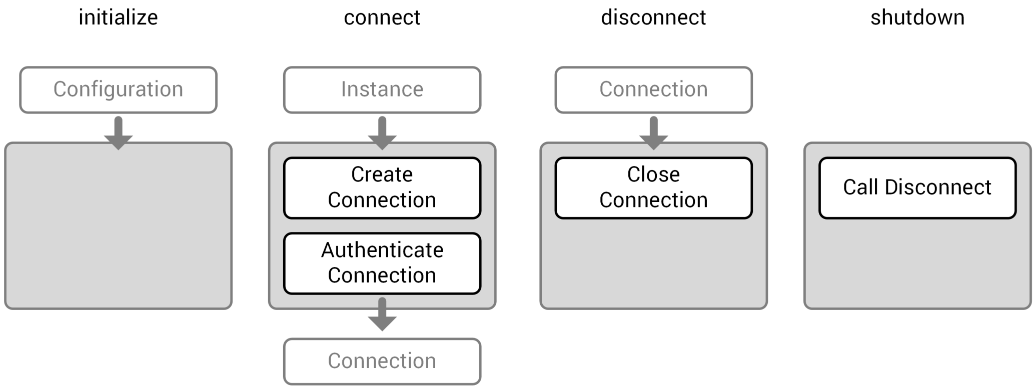

Figure 8.4: The operations implemented by the SSH plugin.

No actions are taken in the *initialize* operation. During the *connect* operation, we first have to create a connection object, which is bound to a certain host name, i.e. the IP address of the remote system that we want to connect to. We get this address from the instance object passed into the connect operation. Then, we have to authenticate this connection. Multiple authentication methods are supported by SSH-2 protocol, including password and public key authentication. The necessary values for these authentication methods are read from the instance object passed into the *connect* operation. Once the connection is authenticated, a connection object is returned, which supports the execute and upload operation that other components can use.

The *disconnect* operation simply closes the connection associated with the connection object that is passed into it. The *disconnect* operation is also called by the *shutdown* operation at the end of the plugin life cycle to close any connection that might still be open.

[29] `https://code.google.com/p/ganymed-ssh-2/`

8.8.3 Remote Bootware Plugin

This application plugin allows the local bootware to install the remote bootware on a remote system. Figure 8.5 shows a simplified overview of the steps involved in the installation of the remote bootware and how they map onto the application plugin operations. The *undeploy* and *stop* operations where omitted because they are not really required in this case.

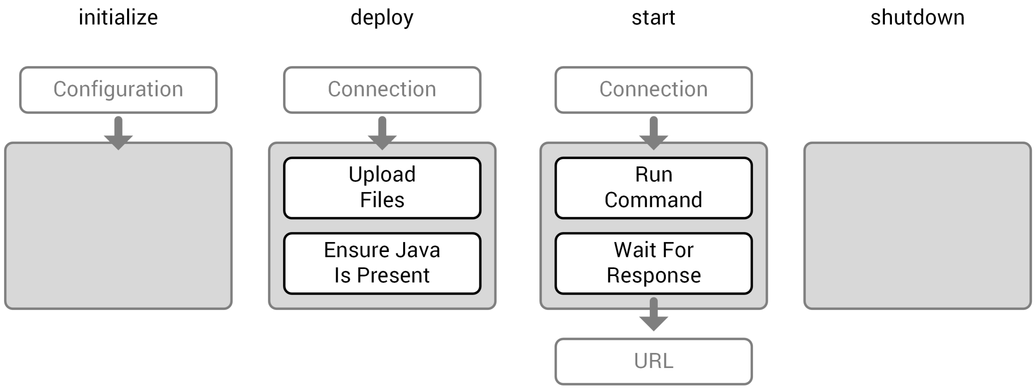

Figure 8.5: The operations implemented by the remote bootware plugin.

In this plugin, the *initialize* operation does not take any actions. The *deploy* operation first uses the operations provided by the connection object it receives as input to upload the remote bootware files from the local to the remote machine. Then, it checks if the Java version required to execute the remote bootware is present. If not, it installs the required Java version. The remote bootware should now be ready to start. In the *start* operation a command to execute the remote bootware is sent to the remote machine. Then, the port for the remote bootware web interface is polled until a response is received, which means that the remote bootware should now be ready. Finally, the URL to the remote bootware is returned.

8.8.4 OpenTOSCA Plugin

This application plugin allows the bootware to install an OpenTOSCA container on an EC2 instance. It executes the installation steps described in the OpenTOSCA manual over a connection provided by a communication plugin. Figure 8.6 shows a simplified overview of the steps involved in the installation of OpenTOSCA and how they map onto the application plugin operations. The *undeploy* and *stop* operations where omitted because they are not really required in this case.

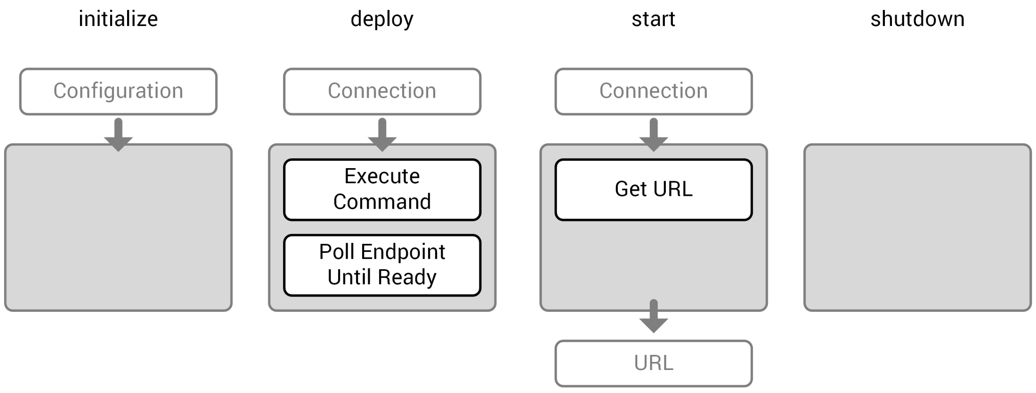

Figure 8.6: The operations implemented by the OpenTOSCA plugin.

The setup procedure for OpenTOSCA is very simply. Only one command has to be executed over SSH, which will automatically download and install all necessary components. After that, port 8080 on the EC2 instance is polled periodically until a connection is possible, which means that the installation process is finished. The *start* operation only has to return the URL pointing to the OpenTOSCA instance because OpenTOSCA was already started by the installation script.

8.8.5 OpenTOSCA Workflow Middleware Plugin

This provision workflow middleware plugin allows the bootware to provision a workflow middleware using the OpenTOSCA container. Figure 8.7 shows a simplified overview of the steps involved in provisioning and deprovisioning the workflow middleware with OpenTOSCA and how they map onto the provision workflow middleware plugin operations.

The *initialization* and *shutdown* operations are not used in this plugin. The *provision* operation first has to get the actual CSAR URL from the service package repository, for which it uses the service package reference that was passed in as parameter. The CSAR URL is then used to upload the CSAR to the OpenTOSCA container. Once the CSAR is uploaded, the build plan contained inside it can be executed. The information it returns after its completion is passed back as Map<String, String> (i.e. the implementation of the information list). The deprovision operation just executes the termination plan contained in the CSAR.

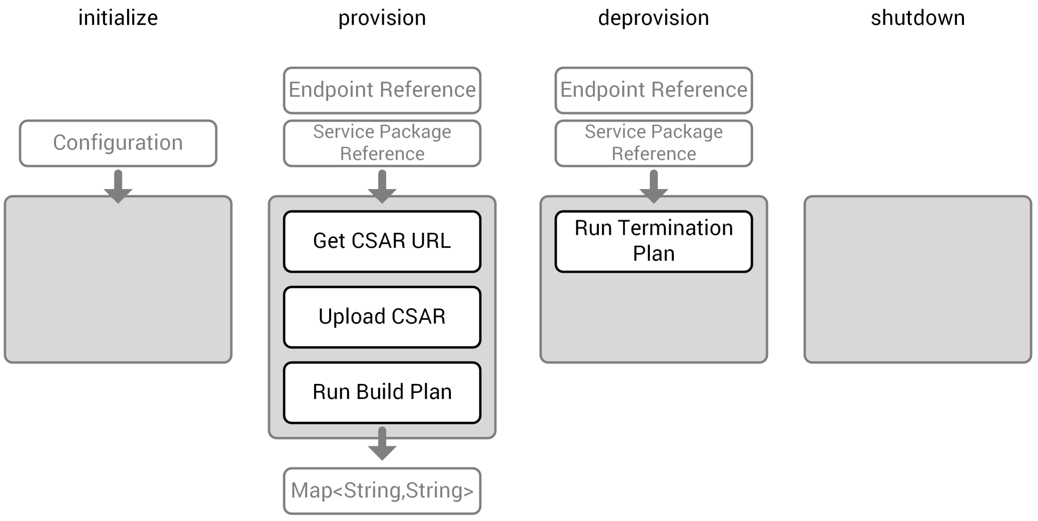

Figure 8.7: The operations implemented by the OpenTOSCA workflow middleware plugin.

8.8.6 File Logger Plugin

This event plugin logs all events generated by the bootware to a text file. Figure 8.8 shows a simplified overview of the implementation of this plugin. The *initialize* operation creates a writer object. The two event handlers shown in the middle use it to write the events they receive into a text file. The event handler shown on the left reacts to all events of the type *BaseEvent*, which is the parent event of all events generated by the bootware. Therefore, it logs any event generated by the bootware into the text file. The event handler shown on the right reacts to a special *DeadMessage* event type generated by the PubSub library we use, MBassador. This event is generated each time an event is published to the event bus to which no one subscribed. Those events are not received by any listener and are therefore dead. We log them here for debugging purposes. The *shutdown* operation just closes the write object that was created by the *initialize* operation.

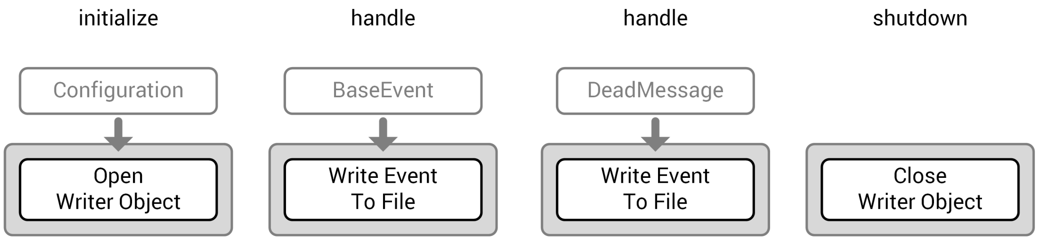

Figure 8.8: The operations implemented by the file logger plugin.

9 Future Work

With this diploma thesis we created a foundation which, while usable 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. This list is by no means exhaustive and other possibilities for improvements might become evident in the future.

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 resource types), communication mechanisms, and applications. Along with a greater variety of plugins, a plugin repository, as described in Subsection 6.4.3 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 does not necessarily need to be changed for this.

9.2 Secure Communication and Secure Plugins

As we already mentioned in Section 6.3, it is necessary to secure the communication with the bootware because 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[1] SOAP extension for the web service communication. In Subsection 8.3.1 we also mentioned that security for plugins could be a problem. OSGi provides an optional security

[1] `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss`

layer based on Java permissions[2] that can be used to apply permission based security. For example, it should be possible to only allow plugins to access specific files or folders with Java file permissions[3]. As part of this work, it could also make sense to investigate other possible security enhancements to the bootware.

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 and only supports 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 configuration values.

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 cannot be established with a cloud provider, the bootware will abort and undeploy already provisioned applications. 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 for this one service 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.

9.5 Crash Recovery

In Section 6.8 we mentioned that we store active instances in-memory. Right now, if the bootware crashes during the bootstrapping process with instances still active, there is no way to continue the process after a restart or at least undeploy remaining instances. These remaining instances will have to be removed by hand, which is not ideal. This could be improved by storing active instances in some sort of persistent storage, so that they can still be retrieved after a crash for recovery purposes.

[2] `http://docs.oracle.com/javase/7/docs/api/java/security/Permission.html`
[3] `http://docs.oracle.com/javase/7/docs/api/java/io/FilePermission.html`

10 Summary and Conclusion

In this diploma thesis we presented a design for a 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 consisting of generic local and remote bootware components. We also introduced the notion of a bootware adapter to connect the local bootware component to a specific modeler. We described a web service interface to allow various components to communicate with the bootware. We made this architecture extensible via plugins and described five different plugin types. We also added an event bus to allow plugin to create and react to events. We described the execution flow that was implemented with a finite state machine.

Then, we presented details of the implementation of the bootware components and the integration into the SimTech SWfMS. We described a specific implementation of the bootware adapter, the bootware plugin, an Eclipse plugin that integrates the bootware into the existing SimTech Modeler environment. We explained the bootware core library that is used as foundation for both the local and remote bootware implementation. We also selected Apache Felix to implement the plugins, MBassador for the internal event bus, and squirrel-foundation for the state machine implementation. We described the content of the context object and the various web service requests and responses. Finally, we gave an overview over various plugins, including a resource plugin for Amazon EC2 instances, a SSH communication plugin, an application plugin for the remote bootware, and an event plugin for file logging.

There were some aspects that we did not further elaborate on. A plugin repository has to be created for the bootware to reach its full potential. Communication with the bootware has to be made secure before it can be used in a real life environment. Other improvements like better modeler integration and failure management should be considered. These tasks are left for future work to explore. In conclusion, there is still work to be done, but the work we presented here should have build a foundation for a part of a system that allows the SimTech SWfMS and other simulation workflow management systems to be used in a fashion that is more in line with scientific work principles.

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.

[9] Christos Chrysoulas et al. "Applying a Web-Service-Based Model to Dynamic Service-Deployment". In: International Conference on Computational Intelligence for Modelling, Control and Automation, and International Conference on Intelligent Agents, Web Technologies and Internet Commerce. 2005.

[10] *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/`.

[11] Edward Curry. “Message-Oriented Middleware”. In: *Middleware for Communications*. Wiley, 2004, pp. 1–28. URL: http://www.edwardcurry.org/publications/curry_MfC_MOM_04.pdf.

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

[13] Tim Dörnemann, Ernst Juhnke, and Bernd Freisleben. “On-Demand Resource Provisioning for BPEL Workflows Using Amazon’s Elastic Compute Cloud”. In: 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID. 2009.

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

[15] *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.

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

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

[18] Joshua D. Goehner et al. “LIBI: A framework for bootstrapping extreme scale software systems”. In: *Parallel Computing* 39.3 (2013), pp. 167–176.

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

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

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

[22] Nedim Karaoguz. “On-Demand Provisioning of Services”. Diploma Thesis 3614. IAAS, University of Stuttgart, 2014.

[23] 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.

[24] 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.

[25] 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`.

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

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

[28] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. "A Taxonomy, Survey, and Issues of Cloud Computing Ecosystems". In: *Cloud Computing: Principles, Systems and Applications*. Springer, 2010, pp. 21–46.

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

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

[31] *Service Oriented Architecture : What Is SOA?* The Open Group. URL: `http://www.opengroup.org/soa/source-book/soa/soa.htm`.

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

[33] *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/`.

[34] *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`.

[35] 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`.

[36] Gottfried Vossen and Mathias Weske. "The WASA Approach to Workflow Management for Scientific Applications". In: *Workflow Management Systems and Interoperability*. Springer, 1998, pp. 145–164. URL: `https://bpt.hpi.uni-potsdam.de/pub/Public/PaperArchive/asi.pdf`.

[37] 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: `ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/INPROC-2013-58/INPROC-2013-58.pdf`.

[38] 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.

[39] *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`.

[40] Sanjiva Weerawarana et al. *Web Services Platform Architecture*. Prentice Hall PTR, 2005.

[41] 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`.

[42] 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 July 7, 2014.

Declaration of Authorship

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

Stuttgart, July 8, 2014, ...

(Lukas Reinfurt)