Technische Universität München

Department of Informatics

Master's Thesis in Informatics

Providing a Remotely Manageable Runtime Environment for IoT Services

Fabian Benedikt Ohlenforst

TECHNISCHE UNIVERSITÄT MÜNCHEN

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Providing a Remotely Manageable Runtime Environment for IoT Services

Bereitstellung einer Fernadministrierbaren Laufzeitumgebung für IoT Dienste

Author	Fabian Benedikt Ohlenforst
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr.-Ing. Marc-Oliver Pahl, M. Sc. Stefan Liebald
Date	August 15, 2018

Informatik VIII
Chair for Network Architectures and Services

Abstract

The core of the Internet of Things is the ability of a growing number of devices equipped with communication capability to participate in a network (Smart Devices). The effort to limit their access and control their data results in the creation of Smart Spaces. Their main characteristic is the observe-control interaction between sensors and actuators to run automated services. The emerging problem in this context is the demand for management of these services through software where Smart Space Orchestration (S2O) comes into play.

The application of a middleware such as the Virtual State Layer is the approach pursued by the distributed S2O system (DS2OS). However, the communication between DS2OS and the services including the management of their lifecycles has to be enabled via a hierarchical service management. This is split into a bottom-up and a top-down part.

This thesis proposes a runtime environment to improve the availability of services and covers the bottom-up service management part including the lifecycle management of services. Therefore, system resources of the nodes are continuously monitored. Considering the capacities of the nodes, services are migrated before downtimes occur. Thus, the system can recover from service failures or Service Level Agreement violations. Furthermore, the services run autonomously after being installed once even in case of disconnection from the middleware. Domain specific challenges are identified and compared to related systems. After designing a runtime environment fitting specific service management requirements in terms of service availability a prototype is evaluated. The results show how the integration of a runtime environment ameliorates the disposability of services and contributes to an autonomous local service management.

Zusammenfassung

Der Grundstein des Internets der Dinge ist die Fähigkeit einer steigenden Anzahl von Geräten, die imstande sind zu kommunizieren, Teil eines Netzwerks zu sein (Intelligente Geräte). Das Bestreben ihren Zugriff zu beschränken und ihre Daten zu kontrollieren bringt die Entstehung von intelligenten Räumen mit sich. Deren wesentliche Eigenschaft liegt in der Wechselwirkung zwischen Sensoren und Aktoren, um automatisierte Dienste laufen zu lassen. Die dabei auftretende Problemstellung einer benötigten Software-gestützten Verwaltung dieser Dienste bringt die Orchestrierung dieser intelligenten Räume (Smart Space Orchestration - S2O) ins Spiel.

Der Einsatz einer Middleware wie die Virtual State Layer ist der Ansatz, der vom verteilten S2O System (DS2OS) verfolgt wird. Die Kommunikation zwischen diesem System und den einzelnen Diensten einschließlich der Verwaltung deren Lebenszyklen muss jedoch über ein hierarchisches Service-Management gewährleistet werden. Dieses ist in einen bottom-up und einen top-down Teil unterteilt.

Diese Arbeit untersucht den Einsatz einer Laufzeitumgebung auf die Verbesserung der Verfügbarkeit von Diensten und deckt den bottom-up Service-Management Teil einschließlich der Lebenszyklus-Verwaltung der Dienste ab. Daher werden die Systemressourcen der Knoten kontinuierlich überwacht. In Anbetracht der jeweiligen Knotenkapazität werden Dienste migriert bevor deren Ausfall eintritt. Somit kann das System sich auch von Service-Ausfällen und Service Level Agreement Verletzungen erholen. Überdies laufen die Dienste selbstständig nachdem sie erst einmal installiert wurden, sogar bei Verbindungsausfällen zwischen der Middleware. Herausforderungen werden identifiziert und mit ähnlichen Systemen verglichen. Nach dem Entwurf einer Laufzeitumgebung, die den speziellen Anforderungen im Hinblick auf die Verfügbarkeit der Dienste gerecht wird, wird diese anhand eines Prototyps evaluiert. Die Ergebnisse zeigen wie die Integration einer Laufzeitumgebung die Verfügbarkeit der Dienste verbessert und zu einem autonomen lokalen Service-Management beiträgt.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The extension of the existing Internet to the Internet of Things (IoT) represents an emerging new paradigm in our connected society because a growing amount of basically simple physical entities is equipped with communication capability like light bulbs or heaters, thus becoming part of a network with the ability to be remotely controlled via WLAN, Bluetooth or ZigBee [19–21]. This concept proceeds on the assumption that these so-called smart devices are connected to the Internet on a large scale. In order to hide the data of Smart Devices from unauthorized access from outside, there is a stronger focus on Smart Spaces as a subunit of the IoT. These are physical environments like a hotel suite offering automated services where sensors receive contextual information and actuators execute context-aware responses [22, 23]. The observe-control interaction between sensors and actuators enables devices to provide an environment to suit individual preferences through services (e.g. to provide a constant indoor temperature) as realized by the HomeKit framework from Apple [24].

This requires, however, an interoperability of the diverse devices to integrate also appliances with heterogeneous protocols in contrast to Apple's homogeneous HomeKit Accessory Protocol. Smart Space Orchestration (S2O), an extended management of Smart Devices through software [23], represents a promising solution to overcome this heterogeneity of the devices by managing the contexts provided by the devices such as the current state of a lamp (on/off). An example for such an S2O approach is the Distributed Smart Space Orchestration System (DS2OS) by [1] with a distributed middleware - the so-called Virtual State Layer (VSL) - as the core of the system.

However, the services for enabling interaction between DS2OS and the several devices have first to be deployed, installed, controlled and continuously monitored so that their availability can always be guaranteed even in case of node failures or node isolation after disconnection from the middleware. This is the task of the service management whose hierarchical structure is outlined in figure 2.1 and which will be explained in some detail in the following chapter. In the context of DS2OS, the service management is split into a top-down and a bottom-up approach. While the site-local service management (top-down) is responsible for resource allocation, load balancing and the deployment of services to the nodes of the distributed peer-to-peer system, the node-local service management (bottom-up) manages services (start, stop, migrate, update). It is on this level where a runtime environment creates the basis for a reliably working Smart Space Orchestration System such that services are highly available at any time and external user intervention is limited to a minimum even after service or node failures.

1.1 Goals of the thesis

The goal of this thesis is to investigate the properties of an optimal runtime environment for bottom-up service management to guarantee high availability of services even in case of disconnection from the middleware and to define what ’optimal’ means in this context. This includes the implementation of the Service Hosting Environment (SHE) which is “a fundamental component for autonomous management of services" [1].
Due to the multiple interfaces and interdependencies this thesis is to be considered in the context of the work of [25] and [5]. While the former provides an autonomous certificate management for the services, the latter is concerned with the top-down service management approach and controls the load balancing of the nodes. Thus, the design of the prototype incorporates these aspects and the evaluation of specific runtime environment properties uses their infrastructure components.
Different approaches how runtime environments for smart spaces meet the requirements for a seamlessly working and resilient peer-to-peer ecosystem are presented and their advantages and disadvantages in terms of required features are considered. The most crucial ones are availability, dynamic updates, monitoring and migration capabilities.

1.2 Outline

The thesis is structured as follows:
In Chapter 2 we will analyze the requirements for providing a remotely manageable runtime environment for IoT services with focus on the bottom-up service management approach.
In Chapter 3 we will have a look at subproblems derived from chapter 2 which are already addressed in other works.
A runtime environment solution is designed in chapter 4 where the questions defined at the end of chapter 2 are addressed to guarantee that the requirements are met.
After the design phase, the implementation specific details of the runtime environment are described in chapter 5.
The evaluation of the prototype implementation is conducted in chapter 6.

Finally, the thesis is concluded in chapter 7 where we summarize our findings followed by a short outlook on potential future work in this area in chapter 8.

Chapter 2

Analysis

The focus of this thesis is on improving the availability of services by providing autonomous local service management for a distributed Smart Space Orchestration (S2O) software through a runtime environment. Different runtime environment approaches are introduced and their strengths and weaknesses are presented.

A short introduction of Smart Spaces and S2O for a better understanding of the existing Distributed Smart Space Orchestration System (DS2OS) is given. DS2OS consists of three main components. The Smart Space Store (S2Store) is a global App store and is therefore responsible for the distribution of the services plus supports crowdsourced software development. The Smart Space Service management is in charge of managing the services and is described in detail afterwards. Finally, the core of DS2OS should not be forgotten: the peer-to-peer based knowledge management middleware called Virtual State Layer (VSL) [1, p. 275]. It consists of several Knowledge Agents (KAs) being connected to each other. Services can connect to the VSL overlay via the KAs and store or access content. Thus, the VSL is responsible for storing and providing the state of the S2O services [26].

The chapter is structured as follows:
Section 2.1 motivates and introduces runtime environments in the context of the hierarchical service management.
Section 2.2 analyzes and compares different promising runtime environment approaches.
Finally, section 2.3 lists the research questions this thesis aims to address.

2.1 Runtime Environment

This section gives an overview of various runtime environment requirements which are necessary for enabling the node-local service management (bottom-up) by providing a seamlessly and reliably working environment for microservices in a distributed peer-to-peer system even after disconnection from the middleware. Different approaches are considered and evaluated by weighing their pros and cons.

2.1.1 Terminology

Before elaborating the requirements and comparing the diverse concepts, the definition of a runtime environment is given and the intersections with other domains is explained.

According to Blom [27], a runtime environment is defined as the set of components that are required to run applications. Runtime environments allow a user to execute a computation, communicate and store data [28] . Local implementations might range from simple Unix accounts to sandboxing technologies to virtual machines.

[29] emphasizes the execution environment provision of a runtime environment to an application or software and the possibility to access system resources like RAM or CPU.

Combining the definitions of [27] and [29] a runtime environment can be summarized as follows:

A runtime environment is the set of components that are required to support the execution of an application or software by giving them the access to system resources.

Implicitly, the need for reconfiguration can be read in this definition if the underlying system changes. Especially for heterogeneous devices this can be a big problem. Therefore, this thesis emphasizes the dynamic character of runtime environments and its capability of reconfiguration and reallocation of resources.

A runtime environment is the reconfigurable set of components that are required to support the execution of applications or software by giving them access to and if applicable dynamically reallocating the system resources.

The concept of runtime environments has to be distinguished from that of operating systems. Their job is to “provide for an orderly and controlled allocation of the processor, memories, and I/O devices among the various programs wanting them” [30, p. 5].

A Smart Device is hardware like a sensor or actuator being capable of interacting with its environment and being connected to larger networks [31, p. 15].
The amount of all Smart Devices combined in a common network and usually restricted to a particular range represents a Smart Space [31, p. 15].

Smart Space Orchestration is the approach to manage Smart Devices within a Smart Space through software [31, p. 16].

In general, a service is a self-contained unit of software performing a specific task composed of an interface, a contract and implementation. The way how a provider of a service performs request from a consumer of the service is defined by the interface and the interaction between the provider and the consumer of a service is determined in the contract [32].
Throughout this thesis, software being capable of interaction with a DS2OS Smart Space to support Smart Space Orchestration is called service [31, p. 18].

Service Management is responsible for enabling the use of services to authorized users including configuration, access to the service and the management of continuous communications [33]. This includes the provision of high disposability of services as well as their deployment, installation, controlling and monitoring.

Besides merely preventing services from failures or at least ensuring that their downtimes are limited to a minimum, statements about their qualitative properties should not be ignored. A Service Level Agreement (SLA) is defined as a contract between the provider and consumer of a service in which the Quality of Service (QoS) terms are specified [34].

With respect to DS2OS, service management is split into two parts: a top-down and a bottom-up approach. While the site-local service management (top-down) is responsible for resource allocation, load balancing and the deployment of services to the nodes of the distributed peer-to-peer system, the node-local service management (bottom-up) manages services (start, stop, migrate, update).

2.1.2 Hierarchical Service Management of DS2OS

As already mentioned in the introductory part, the main characteristic of DS2OS is its VSL μ-middleware which, however, achieves its full potential only in conjunction with a sophisticated service management being able to exploit the advanced functionalities and capabilities of this middleware. Thus, before diving into the motivation for the elaboration of a suitable runtime environment for the node-local service management, a description of the overall hierarchy is given for better illustration.
As shown in figure 2.1, there are the following architecture components [1, p. 276]:

Service Hosting Environment (SHE) / Runtime Environment (RTE)
The SHE and the runtime environment merge smoothly so that they can be considered as a unit. Its role is on the one hand to provide the execution environment for running services. On the other hand, it is supposed to monitor and control running services in order to manage them on a low level.

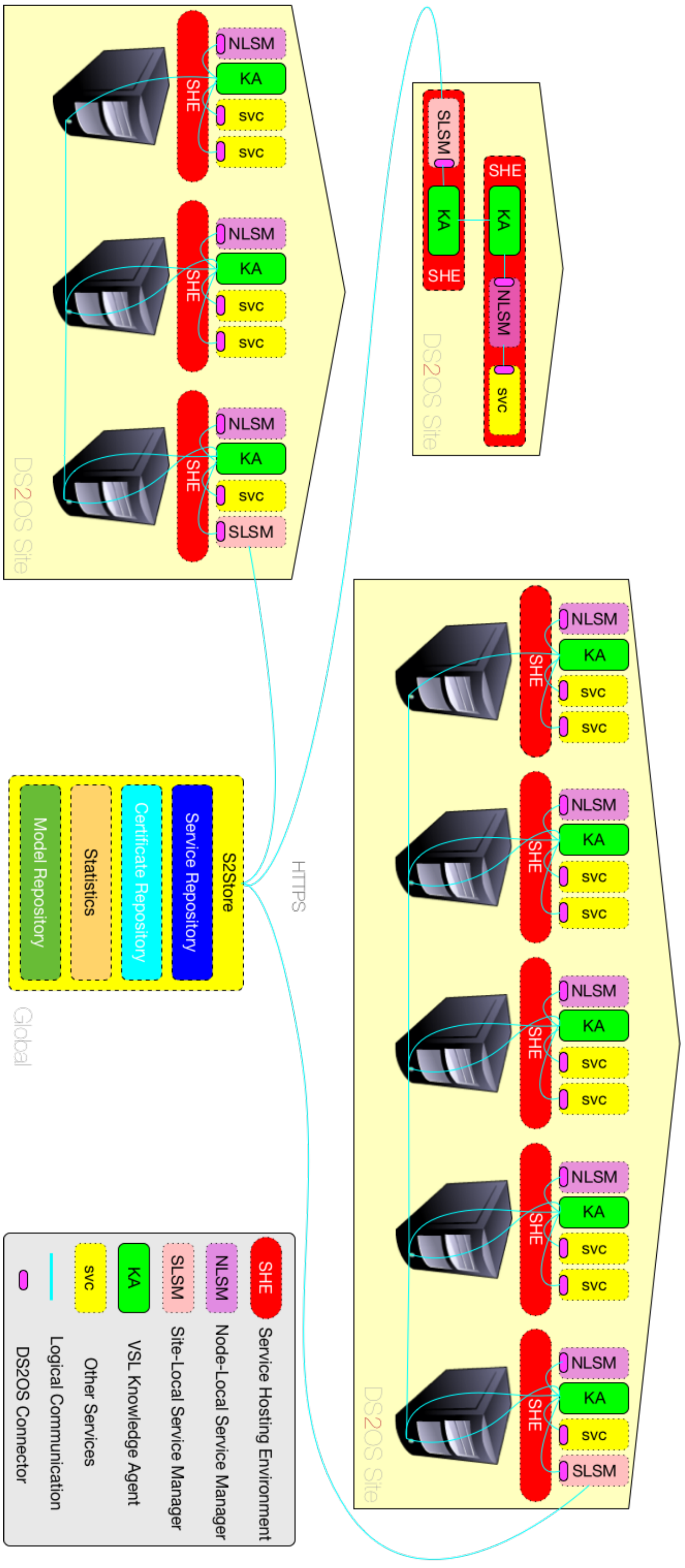


Figure 2.1: Hierarchical Service Management adopted from [1, p. 277]

Node Local Service Manager (NLSM)
: The NLSM is the first service running on the SHE and establishes the connection to the SLSM via the DS2OS connector. Thus, it collects all necessary information about all running services on its node and forwards them to the SLSM.

Site Local Service Manager (SLSM)
: The SLSM is responsible for the resource allocation on DS2OS site such that the capacities of all participating nodes are balanced. It requests resource metrics about all running services on every node from the respective NLSM to run optimization algorithms.

Smart Space Store (S2Store)
: The S2Store contains all services available to be downloaded by the SLSM. It is the central entity to which all SLSMs are connected. It contains the Context Model Repository (CMR) over which context models as abstract service interfaces are shared.

On the top left of figure 2.1, the service management hierarchy is depicted. The NLSM manages the computing node including the running service while the SLSM is responsible for the management on DS2OS site. The communication between NLSM and SLSM is done via the knowledge agents of the VSL where data about locally managed services is sent to and service executables and commands are received from the SLSM. The DS2OS connector enables services to be started, stopped and paused and provides the basis for transferring their context.
Another view on the hierarchical structure is shown in figure 2.2. The blue arrow indicates the service update propagation initiated by the SLSM. Thus, a new service package is downloaded from the S2Store and transferred via the SLSM to all NLSMs where the service is finally locally managed on the distributed node and executed on the SHE/RTE.
In the opposite direction, the NLSM collects usage statistics and failure reports which are aggregated by the SLSM and sent to the S2Store.

2.1.3 Motivation

Peer-to-peer (P2P) systems are distributed systems based on the concept of resource sharing by direct exchange between peer nodes, i.e. nodes having the same role and responsibility [35]. This decentralization results in a high amount of flexibility and scalability on the one hand, but represents on the other hand a risk in case of node failures.

DS2OS is built on a P2P overlay of so-called Knowledge Agents (KA) spanning the Virtual State Layer (VSL) [26]. In case of failures of such peer nodes services residing on these nodes fail due to the lack of communication with the node managing middleware.

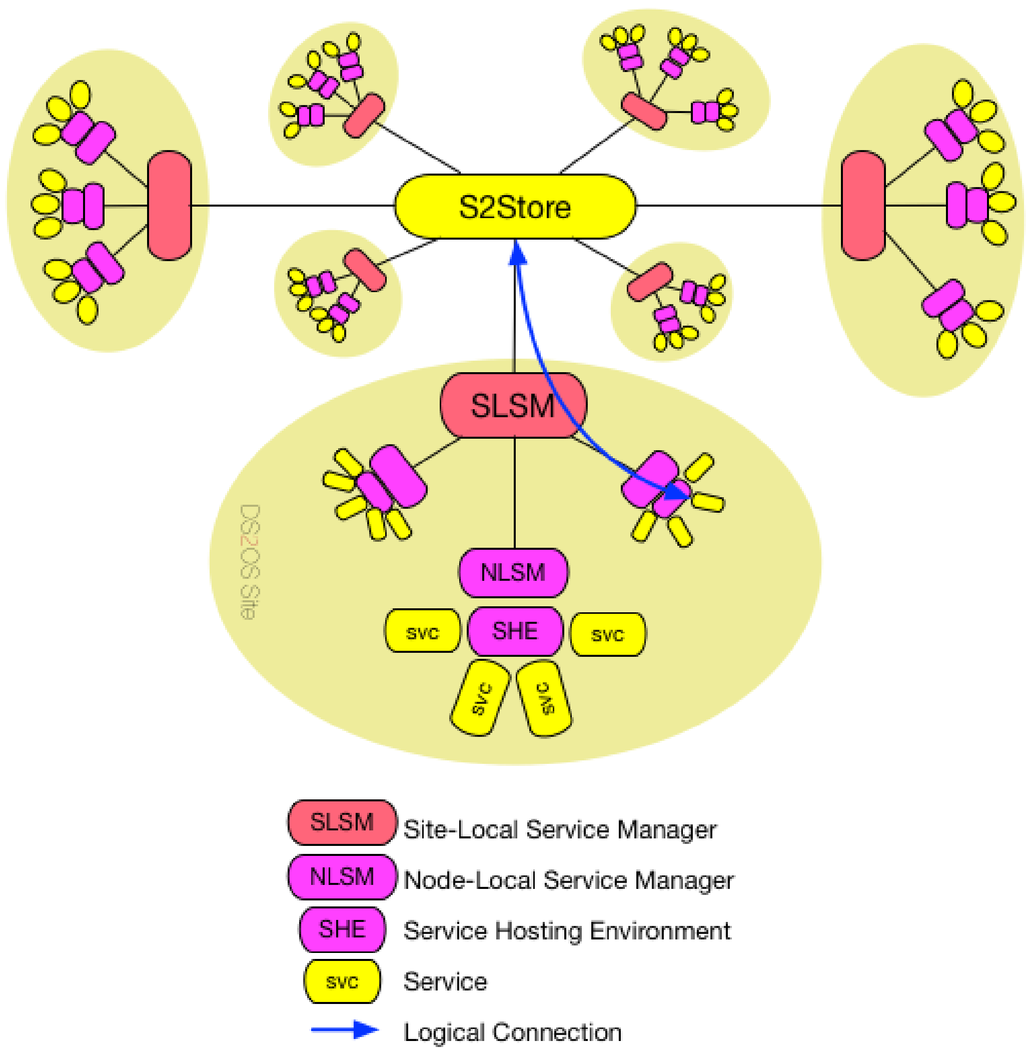

S2Store
SLSM
NLSM
SHE
svc
svc
svc
svc
DS2OS Site
SLSM Site-Local Service Manager
NLSM Node-Local Service Manager
SHE Service Hosting Environment
svc Service
Logical Connection

Figure 2.2: Service Management Overview adopted from [1, p. 286]

The provision of a runtime environment could contribute to an improved availability of services by providing an autonomous local service management. Techniques to achieve this are presented in the next section.
Apart from the scenario of middleware disconnection, the degree of user intervention is increased by ordinary tasks in connection with service management decisions. But the users of Smart Spaces should not be bothered with service management issues such as to decide to which node a service has to be deployed or be prompted to restart or stop a service after its failure. The use of a suitable runtime environment promises to reduce this degree because of its self-healing capacities to autonomously manage the lifecycles of the services under the hood.

Finally, the ability of a runtime environment to recover from and the timely reaction to possible SLA violations is of paramount importance for a qualitative service management and minimization of user intervention.

2.1.4 Autonomous Computing

2.1.4.1 MAPE-K Control Loop

The MAPE-K autonomic management control loop is the basis for many autonomic systems. It consists of the following operations: Monitoring (M), Analysis (A), Planning (P) and Execution (E). The K indicates the shared knowledge base supporting these operations. To provide a better description each MAPE-K component is explained in the following [36, p. 4].

Monitoring
The monitoring component is in charge of managing the several sensors providing information in terms of the system performance. In the context of DS2OS, this means that sensor services can retrieve the current resource consumption like CPU or memory and other performance metrics such as the request process latency.

Analysis
The analysis component takes charge of processing the information received by the monitoring component and generating high level events. It can, for instance, combine the CPU and memory usage metrics to indicate an overload condition.

Planning
The planning component's task is to select the actions to be executed to correct deviations of the original operational behavior. The planning component relies on a high level policy describing an adaptation schedule for the system. These policies may be described using Event Condition Action (ECA) rules defined by a high level language. Such an ECA rule describes what actions to be executed for a specific event and a given condition.

Execution
The execution component directs the actions selected by the planning component to the target components.

Knowledge Base
The knowledge base component stores information to support the other components.

Applied to DS2OS and its hierarchical service management, this means that the monitoring component is essential to decide, for instance, if nodes are overloaded. The metrics are generated by the SHE and are transferred to the SLSM that is able to plan further actions. Finally, the decisions made by the SLSM are executed by the SHE. Important information such as the running and stopped services are saved in the context model where they can be continuously accessed by the SLSM. Thus, the system is capable to react to other circumstances like the overloading of some nodes.

Thus, the MAPE-K control loop can be applied in the monitoring of resource consumption metrics in the context of an autonomous service management. Therefore, node congestions can be anticipated by migrating services in time.
Besides, SLA violations can be effectively triggered.
Finally, the MAPE-K control loop can be applied in the triggering and execution of the autonomous reconnection mechanism after the node disconnection from the middleware.

2.1.4.2 Migration

Spontaneous disconnection of IoT resources from a middleware makes the service management an error prone process. One key goal is therefore to guarantee a high degree of availability which is defined by [37] as follows:

Availability is the degree to which a service is operable; that is capable of producing responses to submitted requests.

High availability includes in this context constraints in terms of "the time window allowed for any response to arrive or the time window allowed for the system not to be operable" [37].

Migration is widely seen as a measure to ameliorate service disposability by evacuating services to new nodes so that they resume running with minimum downtime [37].
Service migration is about "moving an instance of a service that is currently interacting with a client to a new node in a non-disruptive manner" [38].

Based on this definition, it can be stated that DS2OS provides the best conditions for service migration since the service state is separated from the service logic by distributing the VSL on all hosts and due to its strict decoupling of entities [26]. This separation enables services to be restarted without losing their states by starting a copy of the migrated service on another computing node in paused state and transferring the current context (i.e. the service state) from the connector to the copy [1, p. 281].

The combination of the MAPE-K autonomic management control loop and migration can already achieve auspicious results in improving service availability. However, in case of migration possible delays in terms of the deployment process can lead to longer periods of service downtime. The previous provision of services that are supposed to be migrated on the destination nodes is capable of reducing this time.

2.1.4.3 Replication

Thus, Su et al. attempt to support failover in the sense of "deploy once, run forever" by proposing a decentralized fault tolerance mechanism for intelligent IoT middlewares being capable of detecting failures, recovering from them and reconfiguring the system autonomously [2]. The two major components of this mechanism are service replication and decentralized fault recovery. Instead of achieving redundancy through storing replicas on neighboring nodes, their dynamic replication approach is about random distribution of replicas within a predefined range. The deployment of a service onto several nodes is called service replication in this context. Each node tracks the host of the replicated service through a so-called strip being a list headed by the service and followed by the hosted nodes. Each node keeps a set of local strips and a set of monitored ones. Whereas local strips represent the services hosted locally and replicated on other nodes, monitored strips are the local strips replicated from the node which is monitored by the actual node.

Figure 2.3 illustrates an example for local and monitored strips on the nodes $\mathbf{N_1}$, $\mathbf{N_2}$ and $\mathbf{N_3}$. The arrows indicate how each node monitors the functionality of another one through heartbeat signals. Each service is replicated on three nodes, thus the redundancy level is 3. $\mathbf{S_A}$ is replicated on $\mathbf{N_1}$, $\mathbf{N_2}$ and $\mathbf{N_3}$ and active on $\mathbf{N_1}$ while inactive on the other nodes. The services on $\mathbf{N_2}$ and $\mathbf{N_3}$ will be sequentially activated if the service on $\mathbf{N_1}$ fails. Monitored strips contain the local strips of monitored nodes to recover from node failure. Here, $\mathbf{N_1}$ monitors $\mathbf{N_2}$ duplicating its local strip to the monitored strips. If $\mathbf{N_2}$ fails, $\mathbf{N_1}$ will know that $\mathbf{S_B}$ was active on $\mathbf{N_2}$ and is to be reactivated on $\mathbf{N_6}$. The hosts in the monitored strips are notified by $\mathbf{N_1}$ that $\mathbf{N_2}$ failed and the strips are to be updated. Since $\mathbf{N_2}$ was responsible for monitoring $\mathbf{N_3}$, this task is now carried out by $\mathbf{N_1}$.

The decentralized fault recovery is used in case of broken communication links. It is split into the decentralized failure detection and its recovery.
Decentralized failure detection is supposed to avoid single point of failures such that the failure of one system component does not cause the whole failure detection system to collapse. Failures are detected through heartbeats being periodically transmitted messages such that nodes are presumed to have failed when other nodes no longer receive messages from it after a prolonged period of time. The heartbeat protocol applied by [2] makes each node transmit a ping to the previous node as indicated by the arrows in figure 2.3.
If a failure is detected, the recovery process runs through two phases: Initially, the system has to ensure that all nodes carrying the strips of the failed node have consistent view of the strips to pick a replacement node and execute the recovery algorithm. Then, the changes have to be propagated to reconfigure other nodes.

However, storing strips with the replication chain and the service copies creates a lot

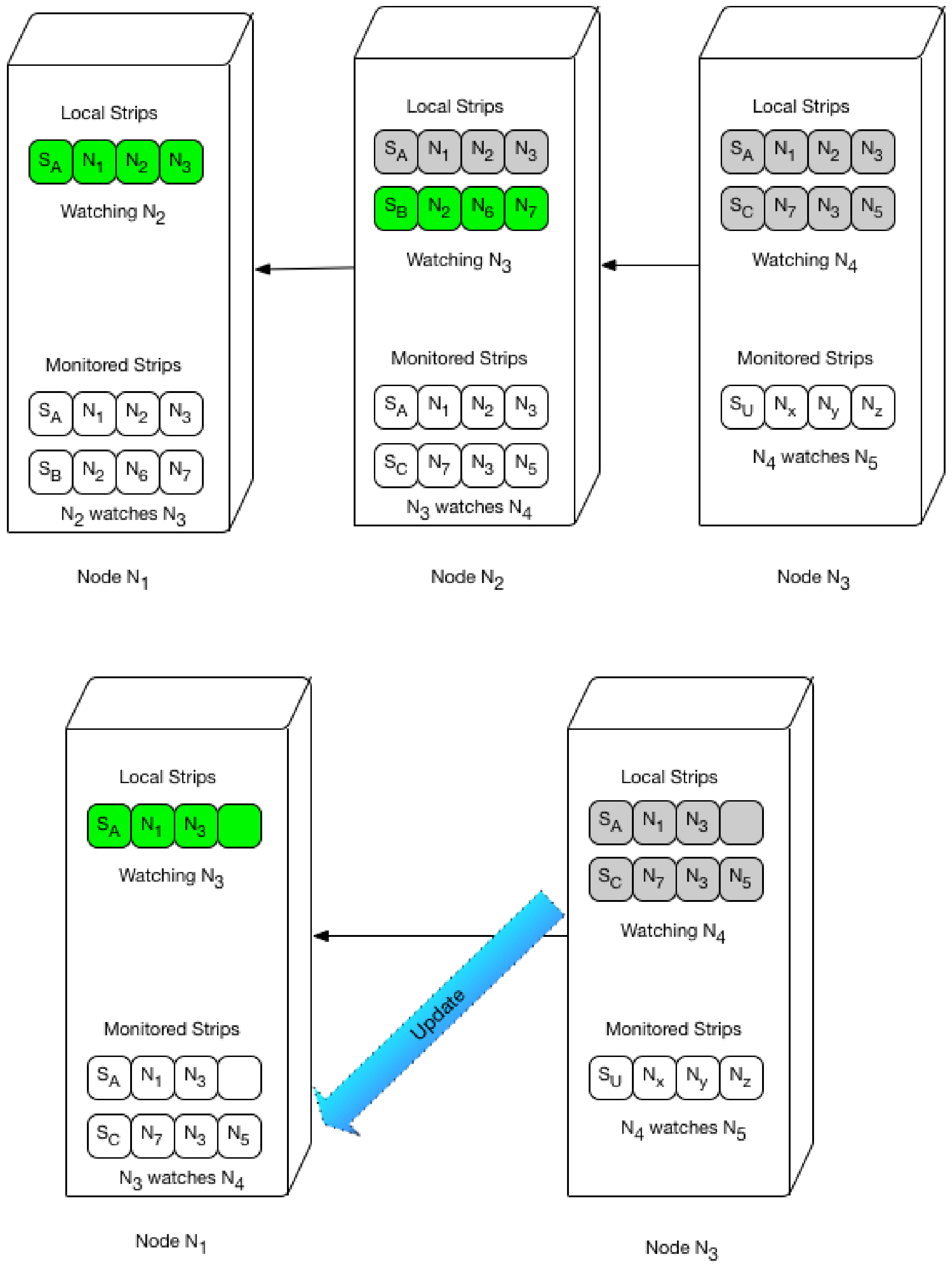


Figure 2.3: Service Replication adopted from [2]

of overhead. Nonetheless, the idea of heartbeats to actively monitor the functionality of nodes is a promising approach to trigger the deployment of replicas.

2.1.5 Security

The idea of DS2OS as an architecture for community based software development for Smart Spaces to take Smart Space users into the development loop and to create a foundation for the user community by providing code and uploading service applications to an App Store implies malicious contributers and malware trying to corrupt the benefits of open source Smart Space services.
Thus, it is a matter of trust to download provided service apps. However, a multi-dimensional rating system can at least give some conclusions what secure applications are by providing metrics like the amount of service downloads or crashes leading to natural selection [39].

Nonetheless, the peril of cyber attacks is omnipresent and even after a successful installation of non-malicious service software the update process of this application can lead to a gateway for attacks which raises the question how the integrity of the service binaries can be guaranteed. In this context, the thesis of [25] comes into play and provides the basis for a secure service deployment.

2.1.6 Requirements

Since the advantage of using a runtime environment is obvious, the question remains which features it should combine and which prerequisites it has to meet in order to enable autonomous computing and provide service availability. The following points are elaborated to create a basis for the most important aspects.

Security
: Runtime environments have to protect the underlying system resources from unauthorized access. Since open source applications can easily be downloaded, their execution in the runtime environment can result in unwanted behaviors if the underlying operating system is not secure enough to detect malicious attacks. Therefore, the validation of binaries can at least guarantee that the downloaded application or service is acknowledged or registered. Thus, the integrity of these binaries has to be periodically checked.

Migration
: Runtime environments must be flexible enough to enable services to migrate from one node to another in a swift and reliable manner if other nodes are more appropriate, for instance in terms of system resources. Since services often interact with or are dependent on each other, runtime environments have to support an architecture of high cohesion and low coupling, i.e. services should be responsible for only a few particular functionalities and should be independent from other services as far as possible.

Communication with Middleware

Since the site-local service management is responsible for the allocation of resources and the deployment of services to the nodes within the distributed peer-to-peer system, the communication between the node-local and the site-local service management in case of connection has to work quickly and reliably such that all resources are perfectly used and information is continuously exchanged.

Autonomous Service Management

In case of a disconnection form the middleware the services have to continue to run autonomously on other nodes such that no external intervention through the user is required. The shift to a local service management for the unattended nodes has to be made instantly while guaranteeing a steady behavior. Upon reestablishment of the connection, the runtime environment has to care about the right state of the services to provide a seamless switching. Apart from node failures, service failures and SLA violations have also to be addressed and handled by an autonomous service management so that user intervention becomes obsolete.

Continuous Monitoring & Control

The runtime environment has to know at each moment the current system resource usage like CPU, RAM, storage or network bandwidth such that load balancing can be applied if necessary. It is crucial that the current state of the service is always available and can be traced so that in case of a system failure the services start automatically without manual intervention. Continuous monitoring is accompanied by controlling services (e.g. start, stop, pause, migrate) in terms of the control-observe interaction between sensors and actuators.

Availability

The runtime environment has to guarantee a high level of availability of services. Thus, an elaborated failure detection mechanism is required such that services can be rebooted instantly after service failures or the runtime environment is reset after a node failure. In addition, services need also be available in case of low system resources.

Service Discovery / Plug&Play

Runtime environments need to discover services and integrate them such that they are immediately available without reconfiguring the system.
In order to deploy newer versions of services, the runtime environment must support a plug-and-play mechanism where services can be dynamically added to or removed from the system.

Dynamic Update

Updates are inevitable to enhance performance and security or to provide further functionality. Firmware systems shut down all open programs to install new updates. In the context of smart spaces, this practice can be fatal if vital services like reanimation systems in hospitals are interrupted. The same is true for normal service updates. System and service updates have therefore to be applied in the background without interrupting the operability of essential services and devices.

2.1.7 Metrics

This section introduces the metrics to evaluate the quality and effectiveness of an applied runtime environment implementation described in chapters 4 and 5.

The service availability metric is adopted from [40]:

Definition 2.1. *Availability* = $\frac{MTTF}{MTTF+MTTR}$

where *MTTF* is the expected time that the service will run before the first failure occurs indicating the average time which a service can run continuously and *MTTR* is the average time required to repair a service, i.e. the average time a service needs to recover from a failure to rerun it as requested.

The service response time is defined as the elapsed time between a service inquiry and the response to that inquiry.
The network load refers to the traffic carried by the network.
The service discovery is defined in this thesis as the elapsed time between the service download from a repository and its registration being available on node site.
The service recovery is defined as the elapsed time between the service downtime after node or service failure and its rebooting.
The convergence time is the measure of how fast the network converges after a change in its topology, i.e. how quick the network's new topology is adopted on reconnecting. Figure 6.4 depicts a simple convergence characterization.

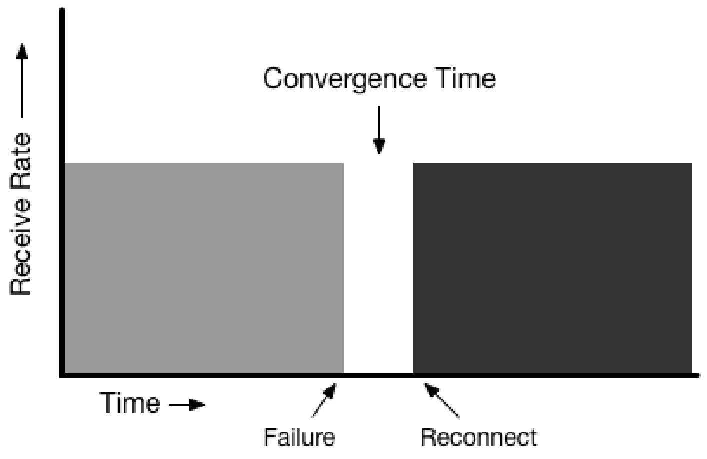


Figure 2.4: Simple Convergence Characterization

2.2 Runtime Environment Approaches

This subsection gives an overview of several runtime environment approaches which seem suitable to fulfill the requirements listed above. The sequence of the approaches corresponds to the degree of their application in related frameworks and their state-of-the-art level. Exemplary projects are presented in the related work where these technologies are used.

2.2.1 OSGi

The Open Services Gateway Initiative (OSGi) framework is the state-of-the-art execution environment for service-oriented architectures in Java being the dynamic module system for Java.
The original mission was to enable Java to be used in networked and embedded devices using core Java constructs like classloaders and manifests.
It provides functionalities like service registration, service discovery, component management or Java class loading [15]. This section is intended to present the basics of OSGi at a high level and its peculiarities in terms of service management because covering OSGi in its entirety would go beyond the scope of this thesis.

2.2.1.1 OSGi Framework

The core elements of OSGi are bundles, lifecycle management and service infrastructure which are supported by the OSGi layered framework as depicted in figure 2.5.

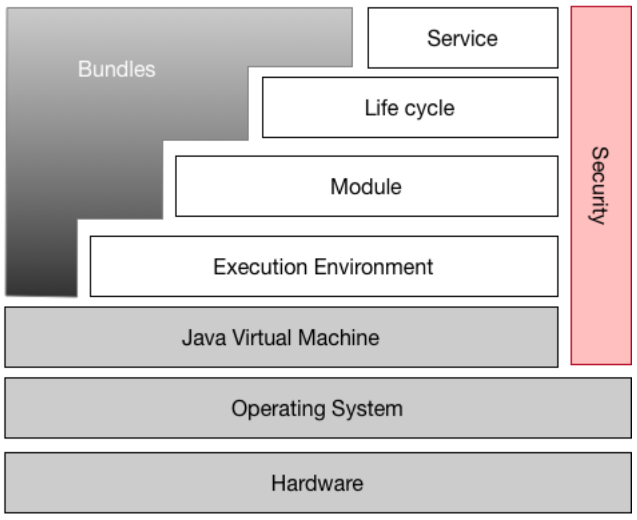


Figure 2.5: OSGi Framework adopted from [3]

The security layer provides extensions to the Java security architecture adding some constraints and filling some of the blanks that Java leaves open. In addition, it defines a secure packaging format and the runtime interaction with the Java security layer.
The module layer defines the modularization model for Java having strict rules for sharing Java packages between bundles or hiding packages from other bundles.
The provision of a life cycle API for bundles by the life cycle layer enables a runtime model for bundles.
A dynamic, concise and consistent programming model for developers of Java bundles is provided by the service layer, thus facilitating the development and deployment of service bundles by decoupling the service's specification (being a Java interface) from its implementations.
The execution environment defines what methods and classes are available in a specific platform.

2.2.1.2 Bundle Life Cycle

The most basic component of OSGi are bundles. These ordinary JAR files with some extra headers contain additional metadata in its manifest to identify them as OSGi bundles. A special dependency management and classloading behavior allowing greater modularity is only one feature making these constructs stand out from normal JARs. Another aspect is that they have their own lifecycle.
Being a dynamic platform by dividing classloading responsibility among several classloaders OSGi bundles are not static entities living on the classpath indefinitely unlike most JAR files on the standard Java classpath [4, p.15]. Thus, bundles may be installed, started, updated, stopped and uninstalled at any time during the running of the framework. In the following the possible states of an OSGi bundle are listed. [4], [41]

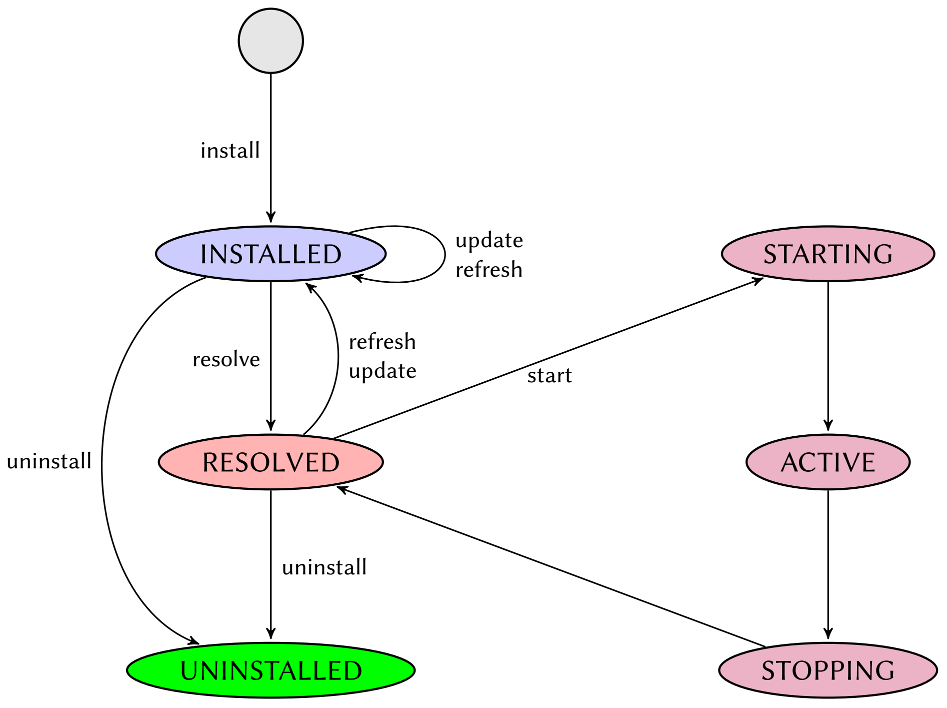
The following lifecycle state diagram shows the OSGi bundle lifecycle as adopted from [41, p. 83].

The installation is the process where new bundles are added to an existing OSGi framework at runtime. To install a bundle the `BundleContext` can be used using a URL pointing to a bundle file.
After the installation into a framework, the OSGi bundle is in the INSTALLED state. It should be mentioned that an INSTALLED bundle does not have a classloader, neither can it provide code or packages.
The real "magic" of OSGi is the resolution process where the framework resolver tries to "resolve" the bundle when all bundle dependencies are available in the OSGi framework. This is where fixed wires between package imports and exports are created obeying the versioning criteria declared in the metadata. If a consistent set of wires can be generated for a bundle, this bundle is called RESOLVED having now a classloader. Thus, dependencies are available at runtime and the risk of a `NoClassDefFoundError` can be

Table 2.1: Bundle States

Status of OSGi Bundle	**Description**
INSTALLED	The bundle has been installed into the OSGi container but some of the bundle's dependencies have not yet been met. The bundle requires packages that have not been exported by any currently installed bundle.
RESOLVED	The bundle is installed and the OSGi system has connected up all the dependencies at a class level and made sure they are all resolved. The bundle is ready to be started. If a bundle is started and all of the bundle's dependencies are met, the bundle skips this state.
STARTING	A temporary state that the bundle goes through while the bundle is starting after all dependencies have been resolved.
ACTIVE	The bundle is running.
STOPPING	A temporary state that the bundle goes through while the bundle is stopping.
UNINSTALLED	The bundle has been removed from the OSGi container.

Figure 2.6: OSGi Bundle Lifecycle



eliminated.
After entering the RESOLVED state, a bundle is suitable for being STARTED. Apart from

the invocation of a bundle's `BundleActivator` (if it has one) and moving the bundle to the ACTIVE state, the crucial change on starting a bundle is the creation of an active `BundleContext` for interaction with the framework, influencing the life cycle of other bundles and accessing the OSGi service registry where the management and discovery of services is handled.
As expected, stopping a bundle is more or less the inverse of starting it. The bundle is moved back into the RESOLVED state by the framework, the stop method is called on the bundle's `BundleActivator` and it is ensured that any services registered by the bundle are unregistered from the service registry.
When a bundle is no longer needed in a runtime it can be uninstalled which does not, however, remove it from the runtime. Rather, uninstalled bundles are marked so that they are no longer suitable for providing packages when the framework resolver tries to resolve new bundles. Nonetheless, they resume providing packages to existing bundles. Uninstalled bundles are only able to be discarded by the framework when no other resolved bundle is wired to them.
After a bundle has been uninstalled, the framework is usually refreshed to resolve any bundles using packages from an uninstalled bundle, thus allowing the framework to tidy up the uninstalled bundles.
The update process is like an atomic uninstall/reinstall operation allowing the bundle content to be changed. The full description would go beyond the scope of this thesis but common side effects resulting from the behavior when uninstalling bundles demonstrate its impact. Given bundle $\mathbf{A_1}$ being updated to $\mathbf{A_2}$ and providing packages to another bundle **B**. Then the *old* version of $\mathbf{A_1}$ cannot be removed without breaking **B**. Thus, $\mathbf{A_1}$ and $\mathbf{A_2}$ are available at the same time while **B** keeps using $\mathbf{A_1}$ until it is refreshed but all future resolutions will wire to $\mathbf{A_2}$. Similarly to uninstalling bundles, refreshing the framework after an update allows to tidy up old bundle versions.

2.2.1.3 Selected OSGi Features

The core of OSGi is its component system. There are a lot of advantages resulting from this architecture.

Modularization

Modularity is the base for service oriented architectures because modules are in control of which classes are completely encapsulated and which are exposed for external use by creating a logical boundary. Thus, each service being handled as module can access other services via provided APIs while encapsulating its classes and explicitly declaring its external dependencies.
Java's object orientation enables partially modularity at the class and package level by declaring methods and classes public or restricting access to the owning class or members of its package. However, if classes are packaged together in a JAR file, encapsulation is not provided because every class inside the JAR is

externally accessible.
OSGi solves this lack of encapsulation through its components called bundles which is the same as modules. Thus, their internal can be hidden from other bundles and communicate via well-defined services.
Apart from the insufficient encapsulation of Java's existing modularity, OSGi also addresses the so-called classpath hell problem. This phenomenon occurs because of Java's dynamic linking mechanism. Most JAR files depend on other libraries or frameworks but due to Java's first encountered policy (i.e. if different versions of the same class appear in several libraries, Java uses the first it encounters) unknown ordering dependencies can lead to problems. OSGi handles this error-prone process by using metadata in the JAR manifest defining the imports and exports as shown in line 6 and 7 of Listing 2.1 so that bundles are wired on a per-package basis.

```
1       Manifest-Version: 1.0
2       Bundle-ManifestVersion: 2
3       Bundle-SymbolicName: serviceHostingEnvironment
4       Bundle-Version: 1.0.0
5       Bundle-Name: Service Hosting Environment Manifest
6       Import-Package: smart.api.pkg;version="[1.0.0,2.0.0)"
7       Export-Package: she.pkg;version="1.0.0"
```

Listing 2.1: Simple Bundle Manifest

Versioning

OSGi recommends a scheme called semantic versioning where each version consists of four parts: major, minor, micro and qualifier. In case of change of the major part (e.g. 2.3.0 to 3.0.0) the code change is not backward compatible as is the case with removing methods or changing argument types. If the minor part changes, backward compatibility is guaranteed for consumers of an API but not for the implementation providers (for instance if a method is added to an interface). Bug fixes and other changes that do not affect the externals are reflected in the micro version. The addition of information like a build date is indicated by the qualifier. The benefits of this semantic versioning are on the one hand the guarantee of compatibility (modules are bytecode compatible with any versions of its dependencies where only the minor or micro versions differ). On the other hand, versioning allows different versions of a module to coexist in the same system [4, p. 341]. Thus, OSGi classloading allows each module to use the version of its dependencies that suits best as depicted in figure 2.7.

All bundles being versioned in the OSGi environment only bundles which are able to collaborate are wired together in the same class space. Hence, the JAR hell problem is solved where libraries can often only work with other libraries of special versions.

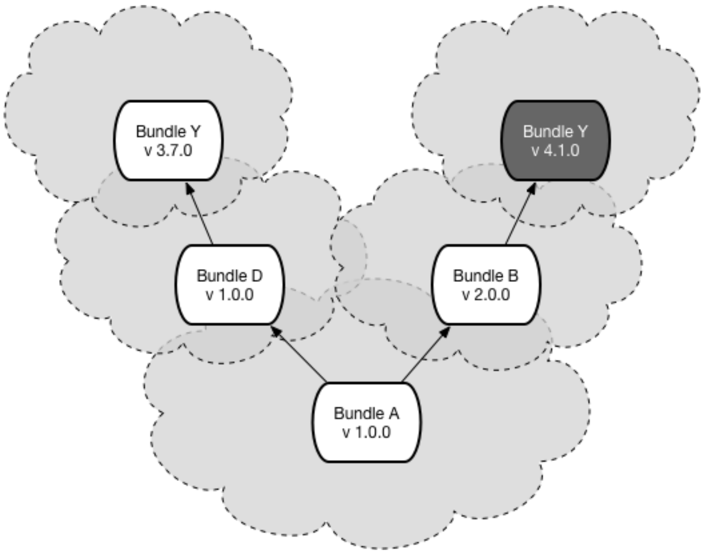


Figure 2.7: Coexistence of Different Version Modules adopted from [4, p. 342]

Dynamic Updates

According to [42], dynamic updates are defined as follows:

Dynamic update is a mechanism that allows software updates and patches to be applied without loss of service or down-time.

This definition implies that updates are made on the fly, i.e. no interruption of the running service occurs. In fact, the OSGi Alliance indicates that the installation/uninstallation, starting, stopping and updating of bundles is possible without requiring a reboot of the whole system and Cummins et al. explicitly mention that bundles can be dynamically updated removing previously exported packages or introducing new package requirements [4, p. 353]. In fact, the reason for this "magic" feature lies in the above mentioned versioning system. Thus, both the old and updated version of the same bundle can reside on the framework which implicitly refreshes the wiring of classes between bundles. The specific manner how this process works would go too much into detail but a simple example is supposed to demonstrate its impact.

Assumed there is an interface definition `Log` in bundle *LogInterface* providing the method `logMessage(String message)`, a `LogProvider` bundle providing an implementation of the interface and a `LogUser` bundle using the interface functionality. All three bundles are installed and started in the OSGi framework as bundle versions 1.0.0. Now suppose that the new method `logMessage(int severity, String message)` is added to the interface definition so that *LogInterface* version 1.1.0 and *LogProvider* version 1.2.0 are installed and started. The *LogUser* would now no longer be able to retrieve the newly provided service from the service registry registered by the *LogProvider* because it is still wired to *LogInterface* version 1.0.0 while the new *LogProvider* is loading its interface definition from *LogInterface* version 1.1.0. In this case a bundle refresh has to be performed to the *LogUser* in order update to the new interface version. This means, however, that the *LogUser* bundle has to be stopped, rewired to the new *LogInterface* version 1.1.0 and restarted.

Size

The OSGi Release 6 Framework can be implemented in about a 300KB JAR file. Therefore, the overhead of the deployed service is very small. In addition, OSGi requires only a minimal Java Virtual Machine (JVM) to run on top of it.

Isolation

Being built on top of the JVM the OSGi framework supports not only code-level isolation, i.e. the internal types of a module cannot be accessed by another one unless it is explicitly allowed by that module to do so, but also true isolation [43, p. 29]. However, there is no recipe against badly behaved modules (for instance services congesting the heap memory). Nonetheless, this security aspect can be handled by integrity checks which are, though, not yet supported by OSGi.

Miscellaneous

OSGi enables a direct connection between services via sockets and provides features related to dynamic life cycle management. Together with its modular architecture, service migration is facilitated because bundles can be easily and quickly deployed and their states can be synchronized. This implies the controlling and monitoring of services (e.g. in which state the bundle currently is). The monitoring quality of OSGi has been extended by a resource monitoring package observing the CPU, memory, disk storage and the network bandwidth.

The table below gives an overview of the provided features of OSGi and compares them in terms of our runtime environment prerequisites.

Table 2.2: Target/Performance Comparison OSGi

Features	**OSGi**
Security	–
Availability	–
Migration	✓
Continuous Monitoring/Controlling	✓
Dynamic Update	✓
Autonomous Service Management	–
Service Discovery / Plug&Play	✓

2.2.2 Jigsaw

After OSGi has been investigated in detail with focus on the most important features in terms of contributing to an adequate remotely manageable runtime environment within a distributed peer-to-peer system, another concept is presented which represents the flagship feature of Java 9.

The most widely used runtime environment is the Java Runtime Environment (JRE). It provides the minimum requirements for executing Java applications and consists of library classes and the Java Virtual Machine (JVM). This abstract machine makes Java platform independent, thus contributing to overcome the heterogeneity of smart devices by providing access transparency [44]. Project Jigsaw was developed under Open JDK and now adds the new scalable module system to Java 9 by applying the module system to the Java Development Kit (JDK) called Java Platform Module System (JPMS) [43, p. 20]. This modularization of the Java platform was overdue because the JDK has grown so much since its first release that installing JDK on small devices can be awkward due to a lack of CPU, memory or disk space (JDK 1.1 was less than 10MB in size, JDK 8u77 for Mac OS X is 227MB). Apart from that, installing the entire JDK can be a waste of memory if requiring only a fraction of it (e.g. when using microservices). The Java Linker (Jlink) coming with Java 9 uses this modularization to dynamically link modules [43, p. 105]. Therefore, targeted JREs can be created containing only modules and their dependencies which are really required, thus conserving system resources.

With OSGi and JPMS two module systems are now available for Java. Thus, the question arises as to whether OSGi has become redundant. The following comparison in some aspects show which approach is preferable when focusing on certain features:

Versioning

In OSGi versioning is completely supported. Bundles and exported packages are versioned. Imports of packages refer to compatibility ranges, usually represented with an inclusive lower bound and an exclusive upper bound (e.g. [1.0.0,2.0.1), i.e. every version from 1.0.0 up to but excluding 2.0.1). OSGi uses semantic versioning. This means that the first segment is the major version indicating pioneering changes in functionality and interfaces while the second segment stands for the minor version denoting less important improvements and the third segment denotes only patches to existing functionalities. In addition, OSGi allows the simultaneous deployment of several versions of a module in a single application so that it is no problem if dependencies have transitive dependencies to different versions of a common library [45]. Services can thus be updated to newer versions. In JPMS, however, the version can only be defined as a meta attribute. JPMS modules can only require other modules but not by version.

Dynamic Behaviour

Due to OSGi's class loader based design for implementing isolation dynamic updating, loading and unloading of bundles at runtime is supported. It provides direct support for on-the-fly updates with no reboot required by only sending the bundles which actually changed. In addition, services in OSGi can come and go and components bound to services are notified in real time because of the dynamic service registry [45]. JPMS, by contrast, does not provide dynamic downloads and loading of modules from a repository while a service and the the JVM are running [43, p. 29].

Isolation

Isolation is the most crucial feature of any module system. Both OSGi and JPMS provide code-level isolation, i.e. the internal types of a module cannot be accessed by another module unless it is explicitly allowed by that module to do so. But OSGi offers true isolation being built on top of the platform while the modules in JPMS are built inside the platform and only provide isolation programmatically according to the way they are designed into the platform. However, neither OSGi nor JPMS can protect the system against badly behaved modules (e.g. blocking the available memory in the JVM, spinning up a myriad of threads or taking up the CPU with a busy loop) [46].

Miscellaneous

Besides these features, OSGi offers an upgraded security model and provides some features related to dynamic life-cycle that are not supplied by JPMS [43, p.29]. However, JPMS offers modularity at compile time and built-in support for native libraries which OSGi does not support. While the security mechanism of OSGi can be bypassed this is not the case for JPMS.

However, Jigsaw is not intended to replace OSGi. While OSGi is a good option for offering application and service modularity, the contribution of JPMS is the modularization of the Java platform itself such that small runtime images can be constructed which contain only necessary pieces of the Java platform for a specific workload. In fact, they can cooperate very well by running OSGi on top of JDK 9. According to Jecan [43, p. 29], OSGi should even be capable of treating Jigsaw modules as OSGi bundles. In a blog post, a proof-of-concept is described how OSGi is running on JPMS [47] where OSGi bundles denote dependencies on particular JPMS modules.

The table below searches for matches between the JPMS characteristics and our runtime environment requirements.

Table 2.3: Target/Performance Comparison Jigsaw

Features	**Jigsaw**
Security	–
Availability	X
Migration	–
Continuous Monitoring/Controlling	X
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	X

2.2.3 Docker

The underlying concept of DS2OS is that the heterogeneity of devices is overcome by enabling service portability. This means that the service executables do not have to consider the system specifications of the devices such that the same result is achieved for different device systems (e.g. the same service controlling the air conditioning leads to the same result for different air conditioners). To achieve this portability, the use of the JVM as intermediate platform independent format for executables is selected [1, p. 248].
So far our focus on finding adequate runtime environments was therefore restricted to Java specific technologies. However, the concept of containers in contrast to virtual machines offers another perspective how to provide autonomous computing.

The following section relies on the research work of [5] and is adopted in most parts. Docker is a virtualization software enabling applications and their dependencies to be packaged into lightweight containers. These can start up quickly and are isolated from each other. Docker containers run directly on the host system. Thus, the operating system (OS) is not required to be virtualized in contrast to other virtualization techniques. The difference between the container and virtual machine architecture is depicted in figure 2.8.

Virtual machines (VM) require a hypervisor managing the system resources and simulating separate hardware for each VM. On top of that, each VM is to run its own operating system. Docker containers, however, are not required to incorporate their own OS running directly on the host OS kernel. However, containers have a limited access although running on the same kernel. Therefore, they are suited for encapsulating microservices. Nonetheless, a machine running the Docker Engine is required to run Docker containers. This is responsible for managing the allocation of the disposable resources using Linux kernel features like namespaces or cgroups. Thus, processes are isolated from each other as well as from the host file system. Unless explicitly configured, Docker containers do not have access to each other. To make them available to the environment port mappings or shared volumes have to be defined.

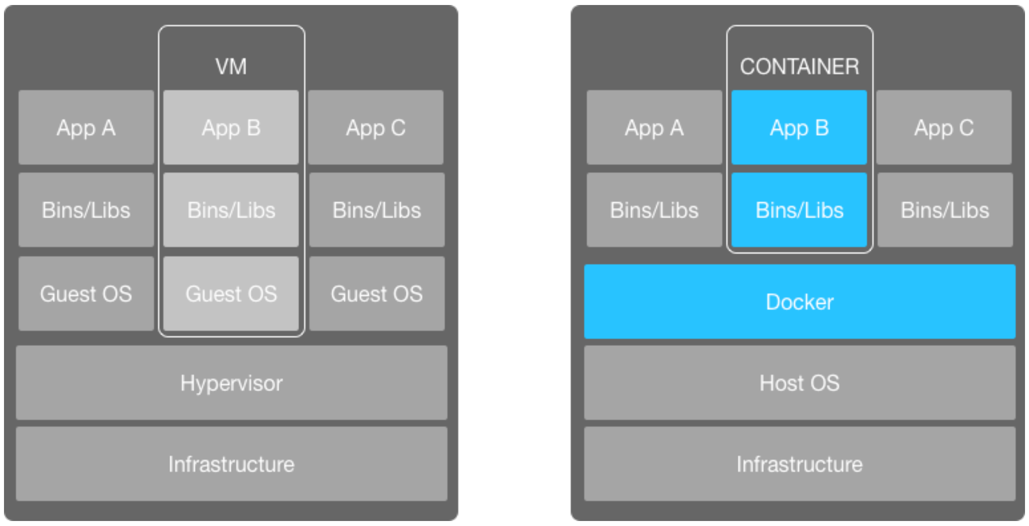


Figure 2.8: Difference between Containers and Virtual Machines adopted from [5]

The lifecycle of a Docker container is presented in figure 2.9.

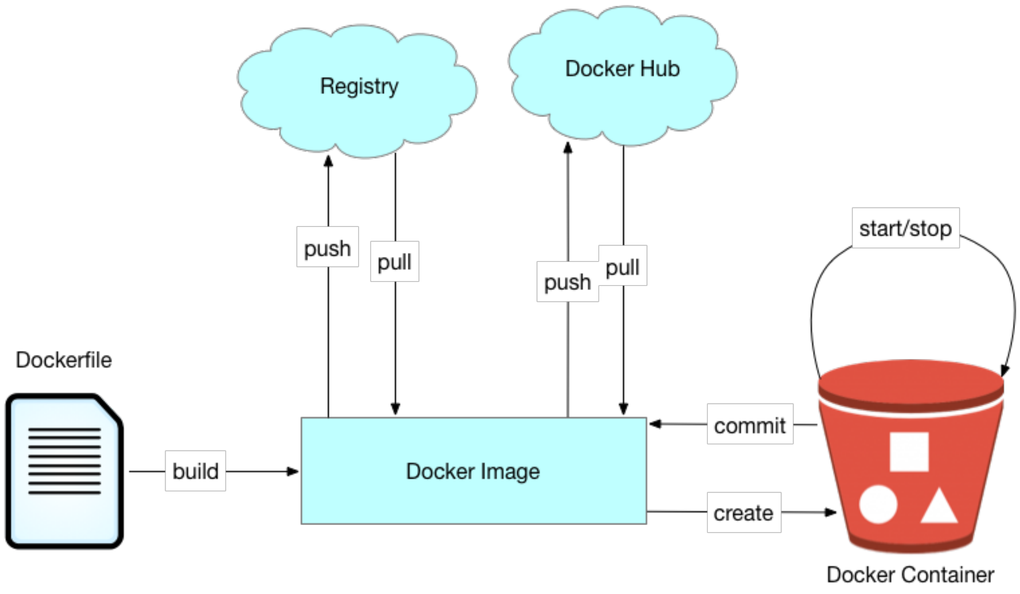


Figure 2.9: The Docker Lifecycle adopted from [5]

It starts with a *Dockerfile* defining the content and configuration of each container. The outcome of its build process is a Docker *image* which can be considered as a blueprint for containers. These can be distributed either through a Docker registry or directly to the several machines.

The registry can be considered as a central image repository. It can be downloaded and applied in local environments. If an image is provided on a client, an arbitrary amount of containers is able to be instantiated from it. Their control is carried out through straightforward commands such as *start* and *stop*. At each moment, the state of the running container can be saved by *committing* the container, thus producing a Docker image. This is ready for distribution and instantiation in copies of the original container.

The following summary is intended to extract the most important features of Docker.

Creation
Docker containers can be built using a simple script. Thus, the container itself can be shared and reproducibly rebuilt with a single text file (Dockerfile).

Versioning
The Docker container itself can be versioned and forked, similar to a Git repository, and shared directly using Docker's hosted repository, the Docker Hub.

Consistency
Unlike virtual machines, the content of the Docker container is restored to its original condition each time it is launched, thus ensuring a consistent computational environment.

Service Discovery
According to [48], a basic service discovery functionality is provided by Docker via so-called service links which are based on Docker Compose links. They create environment variables allowing containers to communicate with each other or other services. The directional links are recorded in the environment variables. After their discovery, the services can automatically be installed and started.

Continuous Monitoring/Controlling
The state of the Docker containers can be monitored using a third party open-source systems monitoring and alerting toolkit called Prometheus [49]. The Docker deamon can be controlled and interacted with through Docker's REST API using scripting or direct command line interface commands.

Miscellaneous
Docker has a high overhead shipping each time the whole JVM and a suboptimal dependency management.
Its security mechanism is based on its encapsulation capability. In addition, Docker does not provide dynamic updates because the container has first to be stopped before being replaced by the updated version which can then be started.
Although Docker provides lifecycle management of containers, it does not include the internal management of the contained services. However, special orchestration tools like Docker Swarm can undertake this job.
Due to the monitoring and controlling capability migration of services is possible. Containers can easily be deployed elsewhere while keeping their states.

The table below compares the Docker qualities with the required runtime environment features.

Table 2.4: Target/Performance Comparison Docker

Features	**Docker**
Security	–
Availability	–
Migration	✓
Continuous Monitoring/Controlling	✓
Dynamic Update	X
Autonomous Service Management	–
Service Discovery / Plug&Play	✓

Before reviewing the results and drawing a conclusion which technology to use, it is worth noting that OSGi and Docker are not mutually exclusive. By contrast, they can complement each other. In the research of [50], Docker is used as Platform as a Service (PaaS) model, thus isolating the application components from each other and the components from the underlying infrastructure. OSGi is adopted as the modularization technology being responsible for the lifecycle management of the services. Their middleware orchestration is OneM2M.

The idea is that the generation of a JAR file containing the OSGi implementation, services, resources and configurations is accompanied by the creation of a Docker image of the application component containing all dependencies which is deployed as container. After the installation, the container registers the offered services. The approach of [50] seems promising, but the concrete way of interaction between OSGi and Docker is not mentioned. In addition, nothing is said about the benefits (if there are any) of using both technologies. Thus, there is no evaluation in terms of response time, network load or discovery time for instance. Finally, the authors do not address possible drawbacks like the overhead produced when shipping a whole JVM each time a component is deployed.

2.2.4 Conclusion

After the collection of preselected potential runtime environment approaches, this section compares them with respect to enable autonomous computing and providing service availability.

The short target/performance comparisons at the end of each technology provides a brief overview of how well the runtime environment requirements are met.
The following table summarizes the insights.

Feature	OSGi	Jigsaw	Docker
Security	–	–	–
Availability	–	X	–
Migration	✓	–	✓
Continuous Monitoring/Controlling	✓	X	✓
Dynamic Update	✓	X	X
Autonomous Service Management	–	X	–
Service Discovery / Plug&Play	✓	X	✓

Table 2.5: Target/Performance Comparison OSGi, Jigsaw and Docker

It is obvious that Jigsaw drops out as runtime environment candidate. Being very novel at the time this thesis is written the benefits of Java 9 are yet to be proven. Thus it is too daring to use it. However, the chance of combination with the other approaches remains.
The choice between OSGi and Docker is more difficult than expected. Both have an architectural based original security mechanism due to their encapsulation capability. Nevertheless, the requirement is not met because the integrity of the binaries cannot be guaranteed. None of the technologies allows to draw conclusions on the degree of service availability. This depends, however, on several factors like service complexity, used infrastructure etc. Due to the variability of this quality no statement can be made as to which approach provides a better availability.
In contrast, the migration capability is provided by both OSGi and Docker because of their lifecycle management so that service states can be restored. The same is true for the continuous monitoring and controlling property.
While OSGi supports dynamic updates at least to a certain degree, Docker does not. This implicitly allows conclusions to the availability property. If each time a service is updated the container is replaced, it can take significant time until a new container is deployed. Finally, OSGi provides via its framework and bundle registry a dynamic service discovery and plug&play mechanism once the services are installed. Docker

also provides service discovery and plug&play and allows to run multiple containers at the same time and to start and stop them as required. However, orchestration software is needed for further service and container management.

To sum up, OSGi supports most of the required features compared to the other approaches. Therefore, it is considered as optimal runtime environment and is applied in the creation of the prototype.
However, it may be interesting to compare the different technologies with respect to the assessment of autonomous computing and service availability.

2.3 Research questions

This part summarizes the elaborated elements which are necessary to realize autonomous service management. After the analysis of runtime environments for Smart Spaces is done, the research questions this thesis aims to address are presented. The main goals can be summarized as follows:

<R 1> How can an autonomous and local service management be provided (→unattended nodes after disconnection from the middleware)?

<R 2> How can migration of services without loss of information quickly be performed?

<R 3> How can the monitoring/controlling quality and response time be optimized (→real-time system)?

<R 4> How can updates after deployment be installed on the fly?

<R 5> How can the overhead be kept to a minimum?

<R 6> How can the recovering of a Service Level Agreement (SLA) violation or service failure be performed?

<R 7> Which information should be exchanged between the node-local and the site-local service management via a well-defined interface?

<R 8> How can the integrity of the binaries of the services be ensured?

These research questions are cited throughout the thesis when topics related to them are addressed.

Chapter 3

Related Work

This section summarizes work with a similar problem domain to this thesis.

There are several partial solutions to subproblems this thesis aims to solve but there is no comparable overall approach.

This chapter is structured as follows:
Section 3.1 gives a short comparison between OSGi and REST based approaches and sensor networks.
In section 3.2 the propagation and handling of Service Level Agreement Violations is addressed.
Section 3.3 has a look at other IoT systems using OSGi.
Section 3.4 investigates other approaches how to enable updates on the fly.
Section 3.5 covers the migration topic.
Section 3.6 is about the way other systems perform integrity checks to prevent unauthorized access.
Section 3.7 has a look on how to enable a working system despite of disconnection from the middleware.

3.1 Comparable Approaches in Sensor Networks

3.1.1 RESTful Smart Space Gateway (RSSG)

As the name indicates, a RESTful Smart Space Gateway (RSSG) represents a gateway for RESTful operations. It is the proposition of an IoT-based user-driven service modeling environment for smart space management systems.

The goal of RSSG is to provide interoperability of devices from different manufacturers in spite of the data heterogeneity generated by different communication protocols and networks. In addition, RSSG provides continuous monitoring.

RSSG provides multiple communication protocols, data aggregation, object management and translation functions. In addition, it uses both Bluetooth and Zigbee as interface drivers to support many types of wireless communication protocols. The devices are discovered by the RSSG and the sensor data and device information are received. The latter is listed in the device Information database. The device information includes the device type and the required data type of the relevant device to manage the connected devices.

RSSG is situated in a space where it aggregates device information which is sent to a service platform where the information is managed and processed for context awareness.

All devices send their sensor data within a certain time interval to the RSSG. RSSG classifies devices into sensors and actuators and proposes an object data format which is depicted below. While sensor data consist of uniform resource identifier (URI), unit, type, value, accuracy, ownership and location, actuator data comprise URI, status, name, location, ownership and a function list.

The object data is addressed by the URI and can be accessed via RESTful operations.

When the RSSG receives the data transmitted by the detected sensor nodes, the data are analyzed by the sensor information contained in the device information database. The data are encoded according to the object data format as shown in table 1 and stored in the measurement information database. In case of a request received through a URI, the RSSG web server processes the request via RESTful operations and the query manager gets the information from the measurement database.

However, RSSG does only cover the acquisition and transmission of sensor data. The observe-control interaction between sensors and actuators is performed via RESTful operations, but service management such as updating or installing a service is not addressed.

3.1.2 REST vs OSGi

A comparison of architectures for service management in IoT and sensor networks by means of OSGi and REST is done by [51]. They state that the OSGi-based approach is better suited for homogeneous sensor networks while REST-based frameworks seem suitable for heterogeneous and widely distributed IoT devices and services. Their outcome is presented in the following table.

Table 3.1: Comparison between OSGi and REST based service management

Requirement	**OSGi based service management**	**REST based service management**
Registering the device	✓	✓
Providing service management (start, stop etc.)	✓	✓
Real-time monitoring of device state	X	✓
Device allocation	✓	✓
Device orchestration	X	✓
Device administration	X	✓
Library sharing	✓	X
Managing distributed access	X	✓

However, the authors' focus is on providing an implementation for an IoT middleware with centralized service management of distributed resources which contradicts our goal of providing manageable runtime environments for the bottom-up service management. Therefore, they come to the conclusion that the OSGi-based approach is better suited for sensor networks and REST-based frameworks are more convenient for dynamic environments because their primary research goal is the provision of distribution of proxies and devices and the device/service orchestration like device registering and allocation which is managed by the Virtual State Layer in the DS2OS. Diving deeper into service management is, however, not covered.

3.2 Service Level Agreement Violations

The ability to recover from and the timely reaction to possible SLA violations was briefly sketched in the analysis part. However, before the recovering from a SLA violation can be executed the system triggering this process needs first of all information about the failure sources responsible for this violation to decide about reactive actions <R 6> .

Brandic et al. try to address in this context the problem of mapping low-level resource metrics to high-level SLA parameters and provide a concept how SLA violations can be identified and propagated [52]. Their layered approach for SLA violation propagation in self-manageable cloud infrastructures (LAYSI) is embedded into the FoSII project (Foundations for Self-governing ICT Infrastructures) for developing self-adaptable Cloud services.

Furthermore, they introduce the concept of threat thresholds being more restrictive than the SLA violation thresholds for detecting future SLA violation threats. Whereas a violation threshold is a value indicating the least acceptable performance level for a service, e.g. a response time $\leq$ 2ms, the threat threshold could be about 1.5ms allowing the system to have 0.5ms of reaction time. For our purposes this can e.g. mean that threat thresholds are determined lying below the actual Service Level Objectives so that migration of services can be triggered before SLA violations arise.

The decision making in terms of executing the appropriate reactive actions by utilizing case based reasoning (i.e. solving problems based on past experience) is applied by retrieving the information from knowledge databases. The current *case* is solved by searching for similar cases from the past and reusing these cases. In the active mode of the knowledge database the SLA parameters are continuously stored. Hence, cases are received based on observed violations and correlated system states. In addition, the quality of the reactive actions can be evaluated based on the utility functions and threat thresholds can be generated. In the passive mode the SLA manager sends notifications and invokes the self-management interface of the upper layer if the announced SLA violation threat cannot be solved by the current layer's SLA manager.

The autonomic manager receives notifications from the lower level and tries to find reactive actions in the database once the SLA violation threat is detected. Its decision components decide whether to defer the SLA violation threat. If it cannot be deferred, they are propagated and all listeners (the components of the upper layer) are notified. If the SLA violation threat cannot be handled at layer n, the SLA manager publishes it to layer $n + 1$ until the meta negotiator is reached informing the user about the issue of a possible renegotiation or stopping the service.

Thus, monitored low-level metrics are periodically mapped to the high-level SLA parameters based on the predefined mappings stored in a database. For instance, if the resource metrics uptime and downtime are continuously monitored, the service availability can be calculated so that the mapping rule is the following:

$$Availability = (1 - \frac{downtime}{uptime}) \times 100$$

The recovery from a SLA violation <R 6> can, however, only be made via service migration which is discussed in detail in chapter 3.5.

However, for our test purposes this goes far beyond because the relevant Service Level Objectives are stated in the meta data files of each service. Thus, in case of SLA violations, the relevant service is migrated to another node containing more resources.

3.3 OSGi based Environments

OSGi has seen a remarkable growth in its adoption as basis to build Smart Space systems upon providing a managed and extensible framework to connect various services and devices around. Thus, it serves as solid grounds for several research projects. An extract of some of these projects is presented in the following giving a glimpse of the different deployment scenarios.

The idea to build Smart Homes based on an OSGi-based architecture has often been realized through a client-server paradigm, thus being a server-centric model with a single point of failure risk in the home gateway.
The shift from the traditional client-server architecture to one of mobile agents in distributed networks using OSGi is investigated by [12].
They propose a service-oriented architecture based on a peer-to-peer interaction model with several OSGi hosts and mobile agent technology so that the resources in the Smart Home environment are efficiently used by distributing the workload among the different platforms. Compared to DS2OS, the knowledge agents are represented by the mobile agents dealing with dynamic situations. The agent hosts are implemented as service bundles so that they can be dynamically downloaded by the devices, thus building the execution environment for the mobile agents.
However, the approach of [12] does not incorporate a strategy for unattended nodes or the recovery from SLA violations. Security aspects are neither considered. Since migration is not provided, service availability cannot be guaranteed. The monitoring and controlling of the states of the devices is supported by the so-called device agents. Nevertheless, they do not monitor or control the states of the running services. The discovery of services is provided by the OSGi framework. A concept for dynamic updates does, however, not exist. The following table compares the specified requirements with the features realized by [12].

Features	**Approach by [12]**
Security	X
Availability	X
Migration	X
Continuous Monitoring/Controlling	–
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	✓

Table 3.2: Reconciliation of Requirements with [12]

Lee et al. recognized very early the benefits of the concept of bundle collaboration through service registry and applied it in a Smart House case study [13].
They adopted OSGi completely as their own execution environment without any exten-

sions. However, their project is not a P2P based architecture but rests on a server-centric model. Thus, autonomous service management is impossible for lack of migration possibilities so that service availability is only limited supported. Using the first release version of OSGi dynamic updates are not supported. Additional security mechanisms are neither implemented. The same applies to the monitoring and controlling of the services. Their discovery, by contrast, is covered from the very beginning. The following table shows how the applied features match our requirements.

Features	**Smart House by [13]**
Security	X
Availability	X
Migration	X
Continuous Monitoring/Controlling	X
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	✓

Table 3.3: Reconciliation of Requirements with Smart House project by [13]

Other pioneers in using OSGi were Gu et al. who have also seen the advantages of this technology like platform independence, various levels of system security, the hosting of multiple services or the support for several home-networking technologies [14].
Their Service-Oriented Context-Aware Middleware (SOCAM) is one of the first approaches to create an OSGi based infrastructure for context-aware services. Thus, a distributed network is generated. Nevertheless, SOCAM does not incorporate autonomous computing. The migration of services is considered just as little as their availability. Although the gateway is monitored, this is not the case for services. The main focus here is on service discovery as in the case of the Smart House project from [13]. The following table illustrates the correspondence between the applied features and our requirements.

Features	**SOCAM by [14]**
Security	X
Availability	X
Migration	X
Continuous Monitoring/Controlling	X
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	✓

Table 3.4: Reconciliation of Requirements with SOCAM by [14]

Another project integrating OSGi in a middleware architecture is the Gator Tech Smart House (GTSH) from [53].

This approach does, however, not differ significantly from SOCAM. Therefore, the upper table also applies to the GTSH.

Ahn et al. focus on the reliability issue OSGi was lacking at that time in terms of network, device and service failure. Furthermore, they propose a proxy-based approach to make the OSGi framework more reliable [15]. The concept is implemented into Oscar, an open source OSGi implementation.
This proxy wrapper enables OSGi to monitor the status of each service instance, isolate failed services and provides an advanced recovery mechanism. Although considering a service-oriented architecture (SOA), Ahn et al. do not apply a distributed environment. Thus, their fault detection extension is only limited to one local platform. Therefore, autonomic computing or migration issues are not covered.

Features	**Oscar by [15]**
Security	X
Availability	X
Migration	X
Continuous Monitoring/Controlling	✓
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	✓

Table 3.5: Reconciliation of Requirements with Oscar by [15]

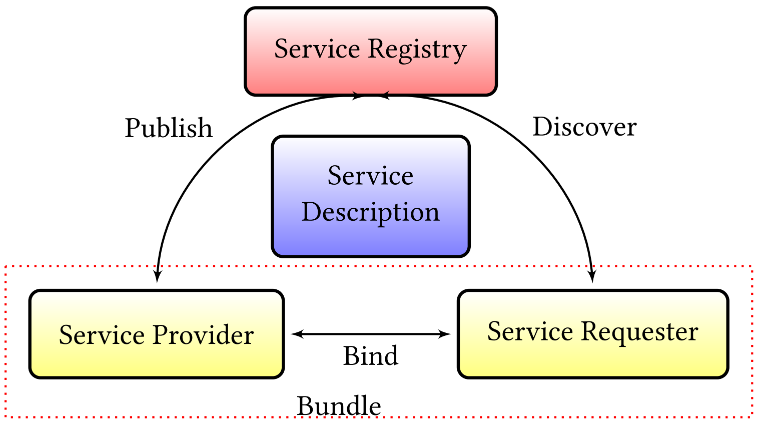
The autonomic aspect of OSGi in terms of its lifecycle management is investigated by [16]. Their research environment was a home area network with a home gateway as central managing device instead of a P2P network. The autonomic element is therefore limited to the home gateway using the MAPE-K loop. Thus, the states of the services are monitored and failures can be handled within the scope of the gateway. However, autonomous computing is not achieved. Therefore, service availability is only partially provided.

Features	**Approach by [16]**
Security	X
Availability	–
Migration	X
Continuous Monitoring/Controlling	✓
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	✓

Table 3.6: Reconciliation of Requirements with [16]

The application of OSGi for heterogeneous networks is addressed by Cheng et al. highlighting the discovery and plug-and-play characteristics. They state that OSGi can be

Figure 3.1: Service Oriented Interactions



divided into two main elements: the services platform and the deployment infrastructure. The former is a software platform supporting the service orientation interaction as depicted in figure 3.1. Thus, service providers publish service descriptions and service requesters discover services based on a service description to bind them to the service providers. The deployment procedure, however, follows the bundle lifecycle as already explained in 2.6 [17].
Nonetheless, their focus on discovering services within a heterogeneous network environment excludes special consideration of security or monitoring issues. In addition, their infrastructure is not distributed so that autonomous computing including migration is becomes irrelevant. The following table gives a quick overview of the issues covered by [17].

Features	**Approach by [17]**
Security	X
Availability	X
Migration	X
Continuous Monitoring/Controlling	X
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	✓

Table 3.7: Reconciliation of Requirements with [17]

Finally, Kim et al. emphasize the device discovery mechanism of OSGi, i.e. the plug and play of heterogeneous devices during runtime which in case of DS2OS is handled by the VSL [18]. Besides, they contribute a semantic model of a Smart Home system to achieve semantic interoperability. However, they do not use a P2P system. Thus, neither availability nor migration are addressed. Therefore, autonomous computing is not an option. The other focus on access control only refers to devices. The table below represents their contribution.

Features	**Approach by [18]**
Security	X
Availability	X
Migration	X
Continuous Monitoring/Controlling	X
Dynamic Update	X
Autonomous Service Management	X
Service Discovery / Plug&Play	✓

Table 3.8: Reconciliation of Requirements with [18]

3.4 Dynamic Updates

Although the OSGi framework offers a policy to update components without bringing down the whole system, it does not mean that updates are made on the fly. Indeed, the updated cooperating bundles have temporarily to be shut down and the state of the services gets lost.

Pohl and Gerlach study the use of the bridge design pattern to decouple service replacement from bundle updates by registering only references to instances of automatically generated bridge classes instead of services. Thus, the problem of dangling references is solved and the stopping and restarting of dependent bundles can be avoided [54]. However, this approach does not provide dynamic updates because it still requires the updated bundle to temporarily shut down.

Chen and Huang study the problem of dynamically updating service bundles in the context of OSGi applications by two techniques <R 4> . The first one is simply replacing the outdated service by its updated version selecting the ideal time to process the update while simultaneously finishing the state transfer.
The new service with the same name as the outdated one is registered in the OSGi framework because the coexistence of services with the same name but different versions is possible as already mentioned in 2.2.1. From that moment, the client resolving the name gets a reference to the new version. The update point has then to be specified for the correct state transfers between the different versions. However, the way how to determine this point is not explicitly mentioned. Assuming that the update point has been identified the state is transferred via a State Transfer Function (STF) which takes the state of the outdated service and writes it into the new one. Finally, the old client bundles are restarted getting references to the new versions of the updated service as depicted in figure 3.2.

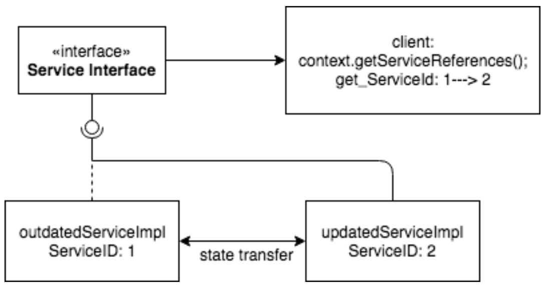
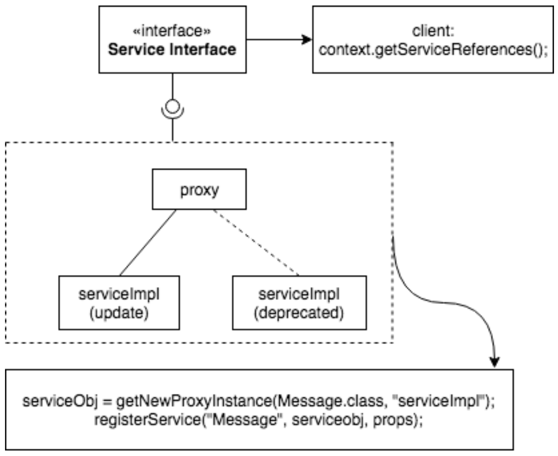


Figure 3.2: Service Replacement adopted from [6]

The other technique is about adding an indirection level through dynamic proxies acting as intermediaries between the clients and service bundles. They wrap the current references to the service bundles. If no updates have to be performed, the client invocations of the service methods are simply forwarded to the service bundles. When a dynamic update has to be applied, these proxies are notified and refresh the references by discarding the old ones and getting new updated ones, thus hiding the update from client code as illustrated in figure 3.3.



```
serviceObj = getNewProxyInstance(Message.class, "serviceImpl");
registerService("Message", serviceobj, props);
```

Figure 3.3: Service Updates with Proxies adopted from [6]

3.5 Availability through Service Migration

The benefit of service migration and its prerequisite on DS2OS site have been discussed in section 2.1.4 <R 2> .
The base for decision whether to migrate and which parts is investigated by [7]. SCANDEX (Service Centric Networking for Challenged Decentralized Networks) is a straw man service centric network architecture for deploying and managing services in decentralized networks. Services are modeled as unikernels which are small virtual machines dedicated to execute special tasks that can be in a stored or instantiated state. The first state is not running whereas the instantiated state is running and able to accept service execution requests. These services are produced by publishers and requested by subscribers.

A SCANDEX network consists of Service Execution Gateways (SEG), Forwarding Nodes (FN), Edge Gateways (GW) and Brokers. The SEGs are the points of attachment for clients and servers hosting and executing services on behalf of its attached client. FNs are responsible for routing requests for services towards available copies caching services locally. GWs are responsible for connecting different domains like two separate networks. They can also serve as publishers and subscribers of services. Brokers perform the service resolution and are responsible for performing intra domain forwarding. SEGs, FNs and GWs have first to register with the Broker.

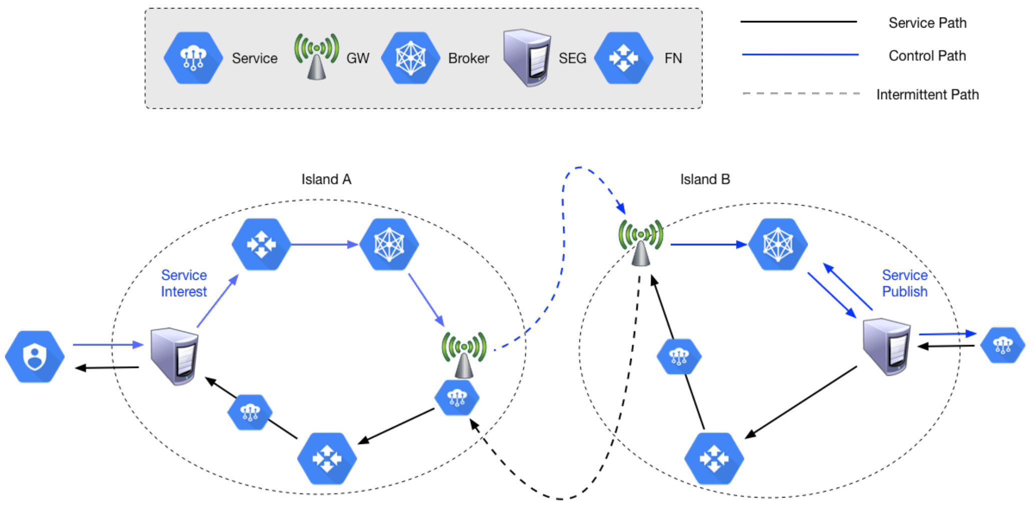


Figure 3.4: SCANDEX network adopted from [7]

The operation of SCANDEX can be best explained by an example as depicted in figure 3.4. The client node in island A wants to access a services located in island B where each client is attached to its local network through a SEG. Thus, service requests are first passed from the client to the SEG.
If the SEG contains locally a copy of the service in its cache, the request is simply passed

to the local service instance and is executed immediately. If, though, the SEG does not have a local copy, the request must be forwarded on. This service interest is then passed through the network via FNs to a broker being in charge of indexing all services that are locally disposable in the network (intradomain). Thus, services hosted by SEGs must be registered with the broker. If the broker knows an intradomain instance of the service, it requests the device with the instance to migrate the service to the SEG. After the migration of the service instance, the SEG instantiates the migrated service and forwards it to its client. If the broker is not aware of any service copy within its own network, the interest is forwarded on an interdomain level to the GW such that the interest is passed into island B. The interest is then passed by the GW of island B to its network broker. If the service is located in island B, the service is discovered by the broker instructing the host to pass it to the GW which has now the the migrated instance locally in its cache. The GW is then instructed by the broker to forward the service instance to the GW of island A caching and publishing it to its own broker. The broker recognizes that a pending interest exists and instructs the GW to pass the service instance to the subscriber.

The interdependency of services and the resulting challenge in terms of migration is also addressed by [7]. Service dependencies can be represented as directed acyclic graphs. Explicitly embedded in the header of a service interest the interaction with only the root of a service directed acyclic graph is required.
Figure 3.5 illustrates an example. Service **A** is the requested root service relying on **B** and **C** which depend on **D**. Each service is located in a separate SEG node represented by dashed squares. The cost of running a node is indicated as its weight and the traffic cost between communicating services is shown as the weight of an arrow. Due to significant

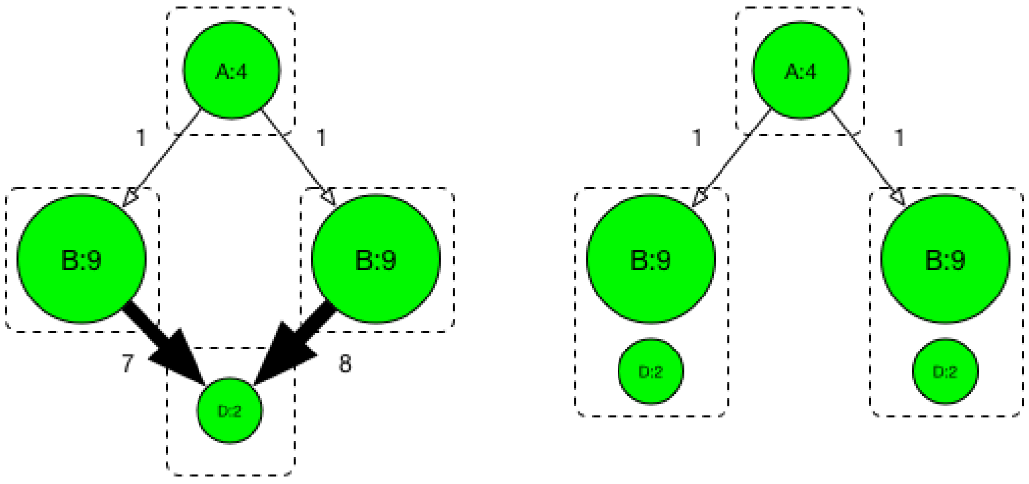


Figure 3.5: Service Dependency and Migration Strategy adopted from [7]

traffic from both **C** and **D** to **D** traffic footprint can be reduced by duplicating **D** at both nodes of **B** and **C** through migration.

The explicitly recorded cost combined with the directed acyclic graph of dependent services is the decision basis for a SEG whether to migrate, and if so, which parts of the service.

The task of the broker is realized by the SLSM in DS2OS which has the knowledge over all available nodes and makes the migration decision based on the available resources. Nonetheless, in a second stage the cost aspect in terms of traffic can be thought of a decision criteria for the migration strategy. However, this requires the SLSM to not only know all service dependencies but also their costs. Even if such information can be obtained, the overhead is inflated and state synchronization becomes even more difficult.

3.6 Integrity checks

The security mechanism of the analyzed runtime environments are mostly based on their encapsulation property resulting in the creation of bundles or containers. However, this does not protect the integrity of the binaries itself as already sketched in section 2.1.5 <R 8> .

Android's security decision made during the app installation process is based on update integrity, i.e. whether to treat the installation as a new app or as an update overwriting previous versions.

App signing is the primary security mechanism of Android that protects the integrity of the app after it is released by the developer ensuring that only he or she can issue an update to an already installed app. One alternative to the entire reputation-based model of Android is to use a full-fledged public key infrastructure where developers prove their identity to a certificate authority (CA) and are issued a certificate.

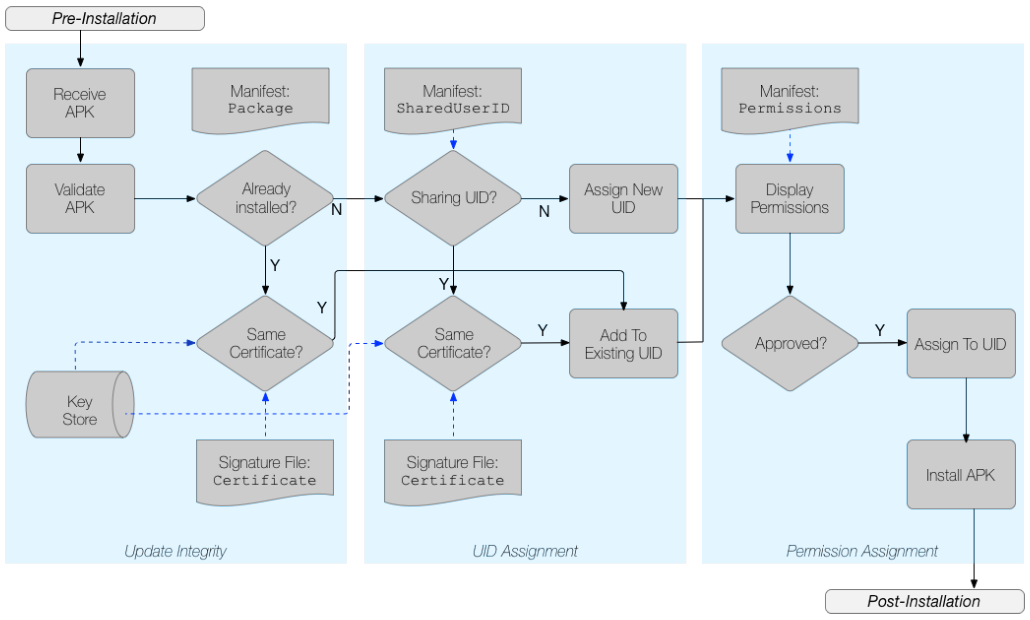


Figure 3.6: Model of the Android Installation Process for an App Package (APK) adopted from [8]

Android's app installation process from the Google Play Store begins with the approval of permissions and the verification of the app package validity where it is assured that it has not been altered or corrupted since its signature and that it comprises a valid certificate for the signing key [8]. Then, it is decided whether the app is a new installation or an update according to the package attribute in the manifest. In case of an update, the certificates are compared and the installed binary is replaced if the signature keys are identical. Otherwise, the app is handled as a new installation.

After that, Android assigns a unique Linux user ID (UID) to the app. In case of an initial installation, it is verified if the manifest comprises the `sharedUserId`. Otherwise, the UID of the overwritten app is used. The permission assignment is the last step where permissions listed in the manifest are assigned to the UID. The whole process is depicted in figure 3.6.

The update integrity concept is roughly sketched in the following. A self-signed certificate is generated by the developer including X.509 attributes like organization, name or validity period usually using RSA as signature algorithm. For every file in the app package (binary, app manifest) a manifest is created by the `jarsigner` in `META-INF/MANIFEST.MF` including an entry with the path and an SHA1 hash for each file. The signature file `META-INF/NAME.SF` is also generated by the `jarsigner` including a hash of `META-INF/MANIFEST.MF` and an individual hash of each entry in `MANIFEST.MF`. So, Android can verify the signature in `NAME.SF` using the public key in `NAME.RSA` and the hashes in `NAME.SF` and `MANIFEST.MF` during installation.

3.7 Autonomous Service Management

The goal of high availability of services at any time and the approach to migrate services if necessary has already been discussed in section 2.1.4. However, in order to talk about autonomous service management this method is only one side of the coin because possible delays during the migration process can lead to longer periods of inactivity. Therefore, different research papers address another aspect in terms of autonomous service management promising to guarantee continuous running of services even in case of node failures <R 1> .

3.7.1 Availability through Service Replication

Thus, Guerrero-Contreras et al. apply autonomic computing techniques to improve service availability in dynamic network topologies. Their self-adaptive context-aware architecture provides a distributed approach to support dynamic service replication and deployment. While following a component-based design, five subsystems are provided: the *Monitor* subsystem, the *Context Manager*, the *Replica Manager*, the *Communications* subsystem and the service itself as depicted in figure 3.7.

The *Monitor* subsystem contains components monitoring device and network capabilities. Whereas device capabilities are easily obtained being local information, collecting network topology information is expensive in terms of bandwidth and energy consumption. Routing protocols such as the *Optimized Link State Routing Protocol* (OLSR) seem to be suitable to estimate the network topology building and providing the routing tables which contain the reachable nodes and for each node the gateway as well as the number of hops.
The *Context Manager* is responsible for storing and processing the information from the monitors being used by the *Replica Manager* to adjust the service deployment according to the changes in the execution context. The exchange between the *Replica Manager* is twofold: on the one hand, the *Replica Manager* is notified about events by the *Context Manager* (publish-subscribe); on the other hand, the *Context Manager* is requested about information to assess the node quality by the *Replica Manager* (request-response). The diagram in figure 3.8 demonstrates the context model managed by the *Context Manager*.

To reduce the bandwidth consumption, only the score of the nodes is shared between each other while sharing the same context model with locally stored information. Taking into account node features such as battery, CPU or memory allows to choose the most suitable nodes as service hosts.
Moreover, the consideration of specific computational service requirements enables the system to find the best matching between services and nodes. Guerrero-Contreras et al. propose an XML structured data model containing the optimal, critical and normal values for each computational service requirement as shown in Listing 3.1.

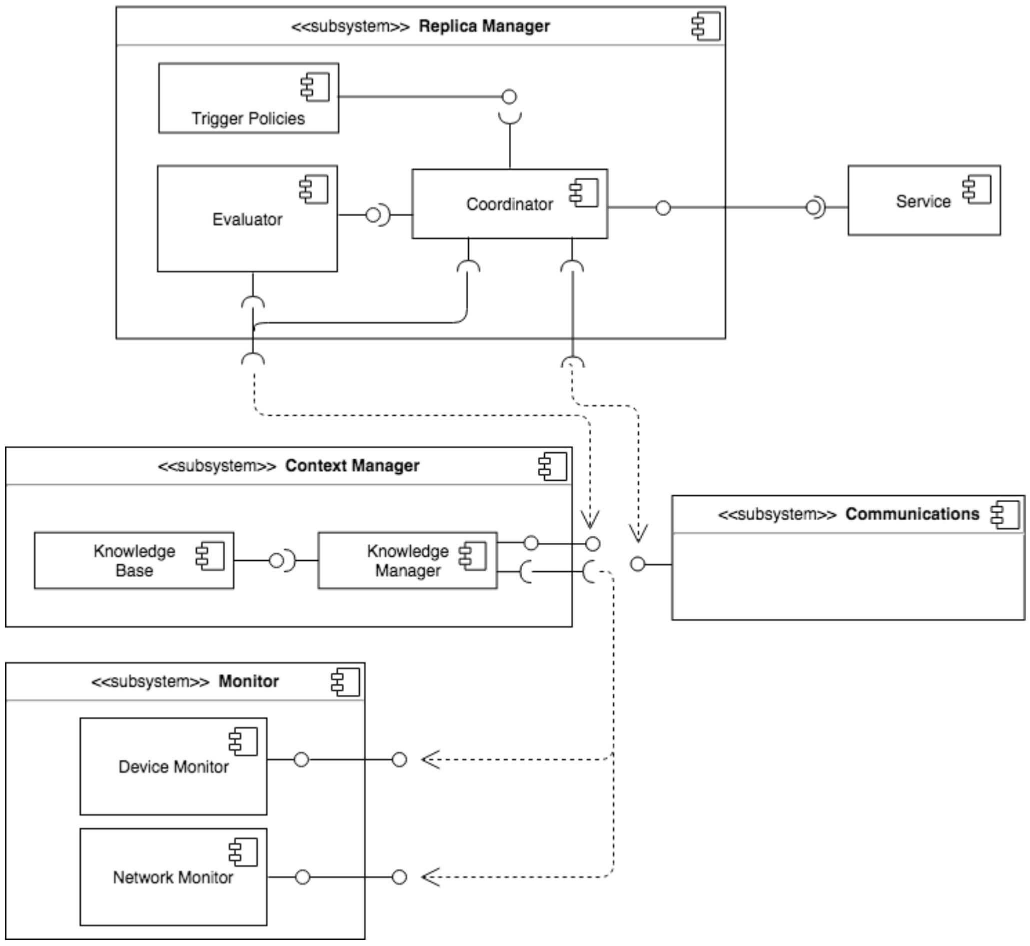


Figure 3.7: Service Replication adopted from [9]

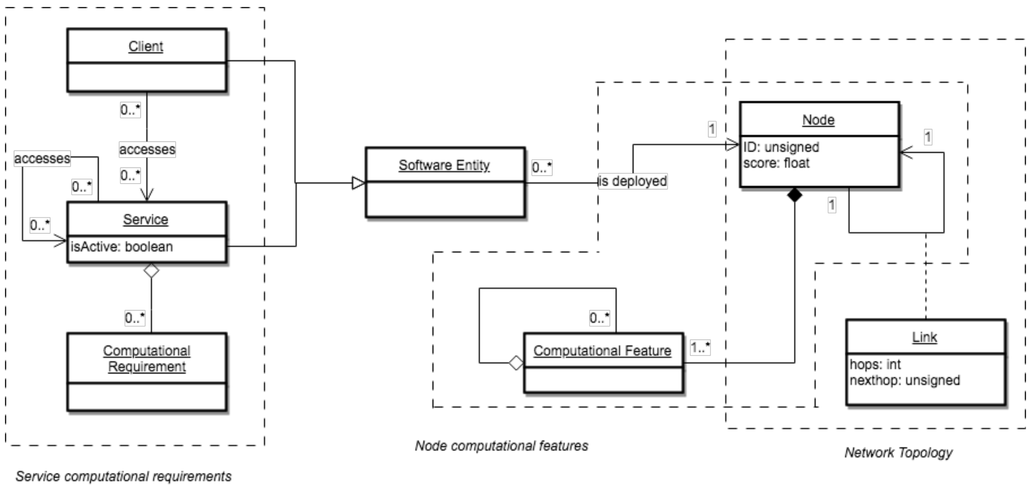


Figure 3.8: Context Model for Self-Adaptive Service Activation adopted from [10]

```
<resourceName="CPU"
 resourceID="TS03"
 optimumValue="2MHz"
 normalValue="1MHz"
 criticalValue="0.5MHz"/>
```

Listing 3.1: Data Model for Computational Service Requirements

Furthermore, the service computational requirements part includes the number of clients (e.g. other services), dependencies between services and the number of service replicas as relevant information to provide an adjusted service deployment solution.
The number of direct links for each node to provide a coordinated activation configuration of the replicas within the network is part of the network topology.
The relevance of the context information is assessed by an election algorithm performed by the different replicas of the *Replica Manager* [10]. This algorithm enables the nodes to exchange their score calculated by the evaluator of the *Replica Manager*, a server request to establish client-server connection between the requesting and receiver node as well as a server rejection and acceptance depending on whether the receiver node of a server request accepts or rejects the request to act as server for the sender node.
The *Communications* subsystem allowing the *Replica Manager* to communicate with other nodes of the framework and the service itself being passive or active replica according to the decision of the *Replica Manager* does not require any further explanation.
The *Replica Manager* embedded in each service replica encapsulates the adaptation logic in terms of the service replication and deployment. The evaluator component of the *Replica Manager* implements the evaluation function used by the coordinator component to assess whether the node is suitable to host a specific service based on the information of the *Context Manager*. The trigger policies indicate through a rule-based system under which circumstances a service should be replicated or migrated (e.g. battery power is below 10%). The coordinator organizes the replicas in the system. If a change context threatens the service availability based on the trigger policies, each coordinator votes for the node evaluated to be most suitable to host the replication service.

Replications are efficient by just restarting a service but also expensive because each node has to carry service implementations [1, p.282]. This contradicts the goal of keeping the overhead to a minimum <R 5>. Therefore, the prototype does not contain any replication mechanism. The implementation of a promising replication strategy taking into account <R 5> would demand a sophisticated probabilistic approach that would go beyond the scope of this thesis.

DS2OS uses periodical alive pings exchanged between knowledge agents via the multicast transport to discover each other and synchronize the context repository node structure [1, p.260]. Thus, the disconnection of a node from the middleware generates an isolated island without pings being exchanged. After the removal of the disconnection reason, this island is merged into DS2OS again <R 1> .

Apart from migrating services, the concept of ad hoc environments can also be seen as a solution for unattended nodes or disconnection from a middleware [55, 56] However, this requires mulit-hop overlays whereas the current VSL architecture is limited to a single-hop architecture.

3.7.2 The MAPE-K Control Loop

The MAPE-K autonomic management control loop as basic technique for autonomous computing has been explained in section 2.1.4.

DS2OS applies the MAPE-K control loop as depicted in figure 3.9. The example shows the observe-control interaction between a sensor (thermometer) and an actuator (air conditioning) until the desired room temperature of 24°C is reached.

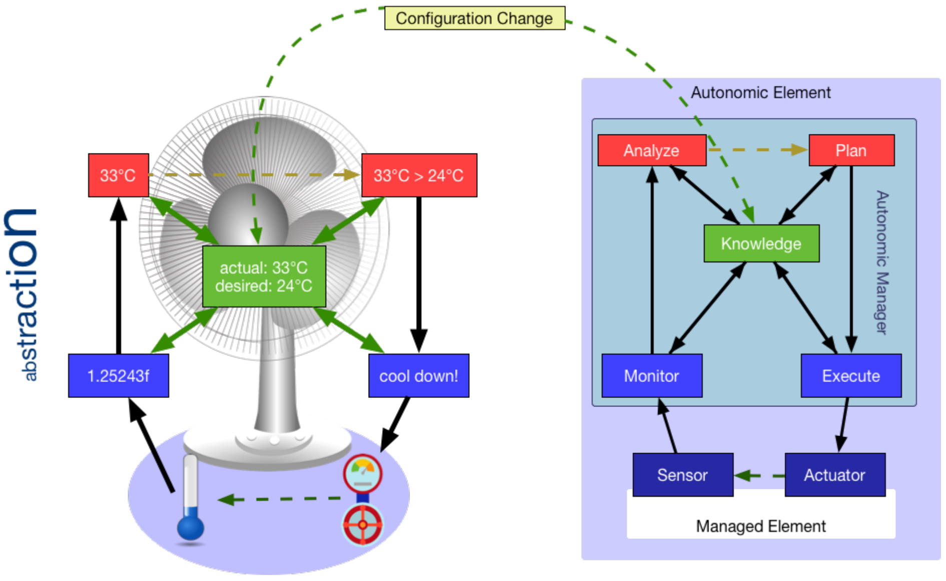


Figure 3.9: MAPE-K cycle in DS2OS adopted from [1, p. 90]

The data represented as floating-point number of the monitoring component (thermometer), i.e. the current room temperature, is retrieved by a sensor service. The value is interpreted by the analysis component (e.g. a converting service) as 33 degree Celsius. This information is squared with the context of the knowledge agent indicating that the desired temperature is 24°C. The deviation of 9°C is recognized by the planning component which instructs the execution component (air conditioning) to further cool down the air.

Chapter 4

Design

In this section the embedding of the OSGi framework as underlying execution environment into the DS2OS context (particularly the SHE) and the connection between the top-down and bottom-up service management are designed. Since the focus of the runtime environment is on the provision of service availability, the architectural elements to achieve this goal are especially stressed <R 3> . The challenge is, on the one hand, to integrate the OSGi framework into the DS2OS architecture and, on the other hand, to synchronize the communication between the NLSM and the SLSM via a well-defined interface <R 7> .

The self-healing capabilities of the generated autonomous service management are made transparent by showing the concrete treatment of SLA violations <R 6> .

Finally, different scenarios for autonomous service management in Smart Spaces are introduced to create test cases for the evaluation part.

This chapter is structured as follows:
Section 4.1 outlines the interaction between SHE, NLSM and SLSM and describes how the resources are monitored.
Section 4.2 describes the publish-subscribe mechanism via callbacks between NLSM and SHE.
Section 4.3 defines the interface between SLSM and NLSM.
The treatment of SLA violations is discussed in section 4.4.
Finally, this chapter ends with the description of three autonomous service management scenarios in Smart Spaces in section 4.5

4.1 OSGi Embedding

The OSGi framework is integrated in the SHE and handles the lifecycle of the services under the hood. The concrete interaction mechanism with the VSL and the communication between SHE, NLSM and SLSM is described in more detail in the next section. Afterwards, interface specific design decisions are discussed.

4.1.1 VSL Connection

The VSL connector is responsible for the communication with the KAs. It is extended with functionality for service management. Virtual Nodes provide the interface to the additional functionality through an additional context model being instantiated if a service is started using the DS2OS connector [1, p. 278].

Since the purpose of the runtime environment is to provide high availability of the services, a quick communication via the knowledge agents contributes to this goal.

The sequence diagram 4.1 shows the desired communication flow between SHE, NLSM and SLSM.

Before users can install and start services, the setup of the environment has first to be prepared. This begins with the initialization of the SHE by invoking its main method. There, the configuration details are parsed from the configuration file including the agent URL where the SHE is running, the service keystore where the certificates are retrieved from and the working directory. After that, the knowledge agent is started and the manifest for the VSL service is generated. Finally, the ServiceHostingRunner is triggered where an instance of the SHE is created and run where the Apache Felix framework as the implementation of the OSGi framework is set up and the NLSM as first service is installed as OSGi bundle and started. Thus, the execution environment is established being ready for further services to run on it.

The real workflow begins with the user installing a service. This triggers the UI Server sending the install request to the SLSM. This entity checks if the service is already available on the SLSM or NLSM site. If it is not available, a GET request to receive the service package for the installation is sent from the SLSM to the S2Store transmitting the package back to the SLSM. This is now in charge of finding the optimal node to run the service on. If the load balancing strategy calculated the optimal node, the service is deployed there and the service is installed and started on the SHE. If everything went well, a callback with the success information is sent to the SLSM, which updates its network model to know each node capacity, and finally via the UI Server to the user.

In case of disconnection from the VSL the SHE can automatically reconnect to the middleware. This is realized through the connector. However, the problem that arises

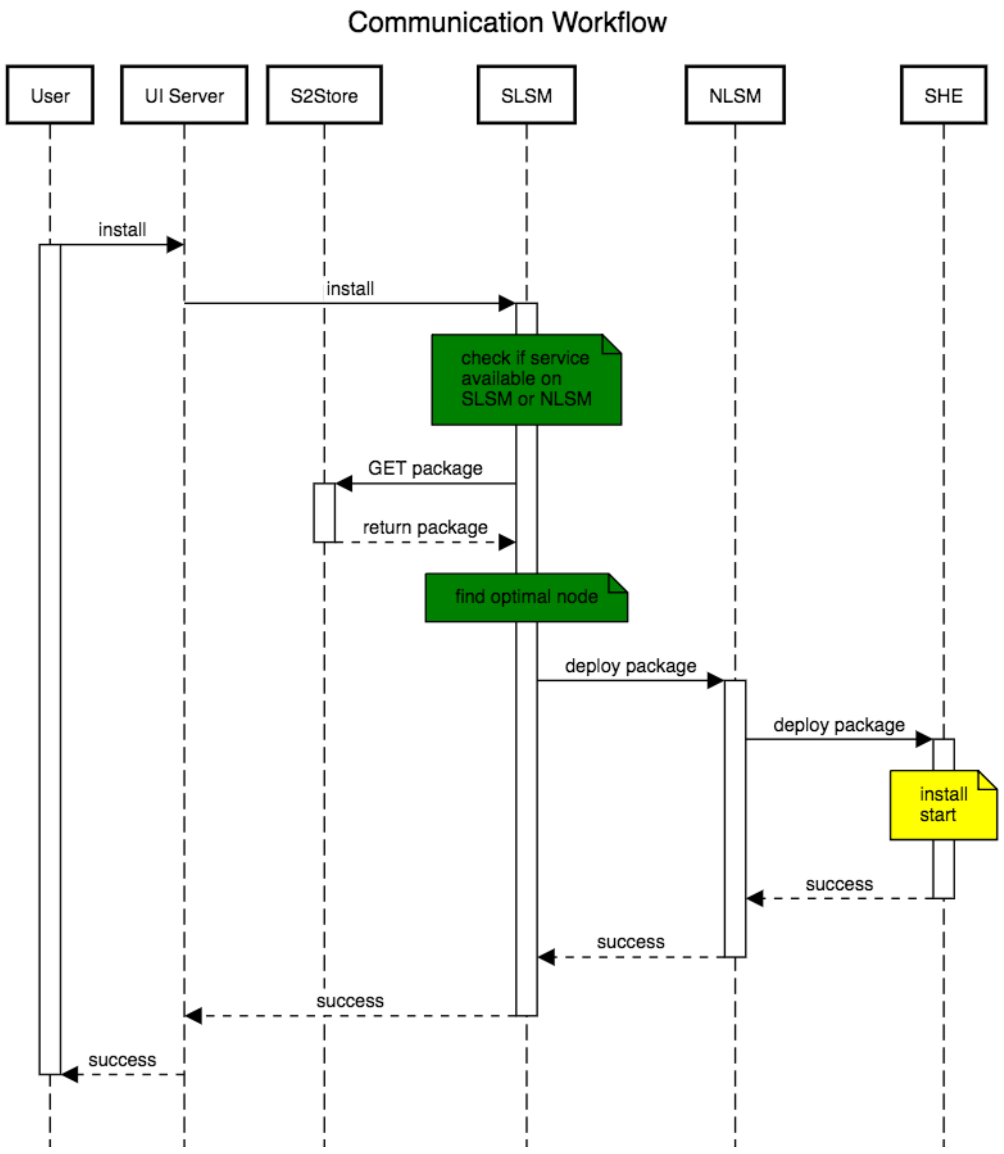


Figure 4.1: Communication Workflow

is how to decide if a disconnection exists.
Therefore, the context model is extended by a connection entry. Immediately after the SHE initialization, a VSL node is created indicating that the connection is given. During the remaining execution time, GET requests are continuously sent to the node to check if the connection still exists. If the request fails due to connection failure, the connector is reactivated and the virtual nodes are registered again. Thus, the SHE is reconnected to the VSL without any user intervention.

4.1.2 Module Invocation Sequence

The structure that the NLSM triggers the SHE may be considered logical from the point of view of the deployment process. However, this contradicts the idea that the NLSM can start the SLSM in case of its failure. In addition, the NLSM is an autonomously running service itself so that it seems reasonable to be started as the first service by the SHE.

If initializing the NLSM before the SHE, the former would need an instance of the latter. This, however, leads to cyclic dependencies between the NLSM and the SHE because both NLSM requires the SHE module and vice versa. This loop is not allowed in Maven.

4.1.3 Resource Monitoring

The frequent and accurate exchange of metrics to monitor the state of the different nodes (CPU load, memory usage, storage allocation and bandwidth usage) <R 3> is essential for knowing when and where to migrate services.

The obvious idea to use OSGi's own interfaces (`CPUMonitor`, `DiskStorageMonitor`, `MemoryMonitor` and `SocketMonitor`) to measure these metrics turned out to be problematic. Although the specification of OSGi describes their features and functions, nothing could be found about their usage. Even the response of one of the specifier of the resource monitoring feature did not lead to the desired perception.
Therefore, the workaround via Java Management Extensions (JMX) is applied. This standard component of the Java platform is a technology supplying tools for managing and monitoring applications. Therefore, it can also measure the CPU and memory consumption of the JVM. To draw conclusions about the impact of running services, the consumption of the resources is measured before and after the starting of services. Thus, the difference represents the current resource consumption. A detailed storage for each service is not necessary for only deciding when to migrate. Apart from that, only the CPU consumption could be measured for each service by summing up the monitored CPU usage of each thread created by the bundle. However, this approach does not apply to the memory consumption. The definition of how much memory a thread and a bundle is using is controversial because Java threading uses shared memory so that memory is not strictly owned by any particular thread.

4.2 Interaction between SHE and NLSM

To enable the communication between NLSM and SLSM the well-defined interface has to be considered. However, the information come originally from the SHE. Therefore, the publish-subscribe mechanism is used on requests of the SLSM. Thus, an intermediary abstraction is created by the NLSM to match the information from the SHE with the interface definition of the SLSM.
This allows, on the one hand, to make counterchecks of the NLSM data against the SHE data. On the other hand, the extension through new requirements or the change of existing data can easily be performed. Instead of matching objects, context models of the knowledge agents have to be matched which involves more abstraction.

Both SHE and NLSM have their own context models. The context model of the NLSM is tailored to support SLSM invocations. Thus, there is little scope to make changes here. Even another SLSM interface definition would lead to major modifications in the NLSM and SHE architecture.
Therefore, the collection of information is outsourced in the SHE context model. This can be extended and changed regardless of the interface requirements between NLSM and SLSM. If, for instance, a new feature is added to the SHE, the code on NLSM side is not affected. Furthermore, new resources can be easily monitored without changing the NLSM logic.

An extract of the SHE and NLSM context model is given by table 4.1. The nodes run on the knowledge agents and can be accessed via GET requests followed by some optional parameters.

Component	**Node Address**	**Method**	**Parameters**
NLSM	/installService	GET	/serviceId=*
NLSM	/uninstallService	GET	/serviceId=*
NLSM	/startService	GET	/serviceName=*
NLSM	/stopService	GET	/serviceName=*
NLSM	/resources	GET	–
NLSM	/stoppedServices	GET	–
NLSM	/runningServices	GET	–
SHE	/isServiceStopped	GET	/serviceName=*
SHE	/isServiceRunning	GET	/serviceName=*
SHE	/stoppedServices	GET	–
SHE	/runningServices	GET	–
SHE	/startService	SET	/serviceName=*
SHE	/stopService	SET	/serviceName=*
SHE	/updateService	SET	/serviceName=*
SHE	/resources	GET	–

Table 4.1: SHE and NLSM Context Model Nodes

The sequence diagram 4.2 depicts the way from receiving the query of the SLSM, redirecting the request to the SHE and calling the result back to the NLSM.

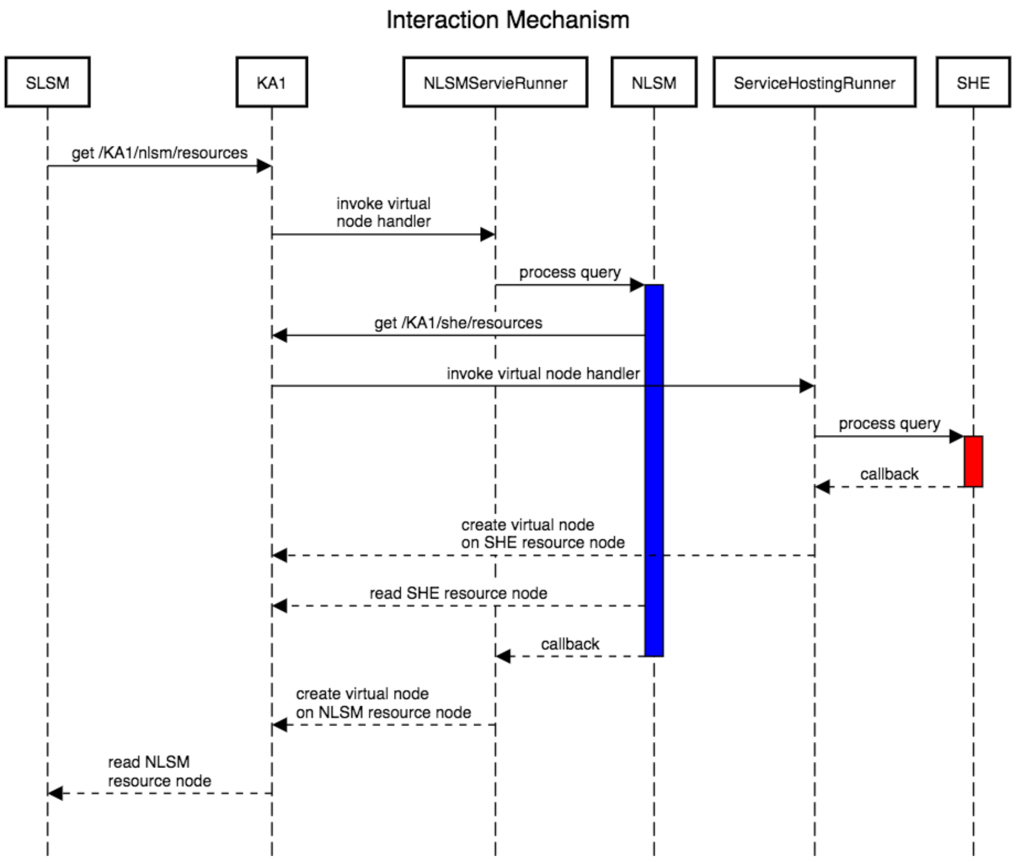


Figure 4.2: Interaction Model between NLSM and SHE

The interaction is triggered by the SLSM GET request of the memory resource consumption on a node. The NLSM context model node *resources* on the running knowledge agent is invoked which is further processed by the NLSMServiceRunner. This makes a callback on the NLSM which is redirected via get request to the SHE context model node *resources* of the agent. The ServiceHostingRunner is responsible for dealing with this request and opens a callback on the SHE where finally the main logic is executed. The resource consumption information are collected and written back to the SHE node. From there, the NLSM can access the data and transfer them to NLSM node. The SLSM can then read the String and parse it into its parts.

4.3 Communication between NLSM and SLSM

The interaction between NLSM and SLSM has already been sketched in figure 4.1. The focus was on the simple workflow of installing a package. Beside the installing process, the communication between NLSM and SLSM goes much further.

One of the most important communication issues is the transmission of monitoring data. These are generated by the SHEs and collected by the SLSM. Containing the resource consumption for each node the SLSM can then perform load balancing and decide to which node services are to be deployed. Furthermore, in case of node overloads or node failures the services involved can be migrated to other nodes with more capacity. In terms of an autonomous service management, this information can help to prevent bottlenecks and thus user intervention.

Another information exchange consists of updating the alive signals of the NLSMs. The knowledge whether a node is still alive and its update frequency is of utmost importance.

One possibility would be to use pings. The SLSM transmits periodically a ping signal to all NLSMs within short intervals. Each NLSM then sends the message back to the SLSM. If the signal arrives at the SLSM within a given period, the NLSM is considered alive and the update procedure starts again.
This mechanism is very resource consuming. The update cycle demands a lot of bandwidth and can lead to network congestions so that other operations are blocked by sending and receiving pings.

Therefore, the concept of heartbeats is applied in the prototype which seems more suitable.
Each NLSM sends a message containing its state (i.e. being alive) to the SLSM. While the SLSM has to send back a signal when using pings, this is not the case for heartbeats. Thus, less bandwidth is required, the overhead can be severely reduced <R 5> and other method calls are not blocked. The heartbeat frequency can be modified so that according to the current circumstances the optimal frequency can be set resulting in reduced network congestion while optimizing the update cycles.
Graphic 4.3 shows the worst and best case scenarios of the occurrence of node failures and their detection.

Two NLSMs are running and connected to one SLSM instance. Each of the NLSMs periodically emits a heartbeat signal which is received by the SLSM to indicate whether they are still alive.
In case of NLSM1 a node failure occurs immediately after sending the heartbeat. Thus, the SLSM receives NLSM1's last sent heartbeat shortly after its failure. The period between the failure and its detection due to the absence of the heartbeat signal is indicated by t_w.
This is the worst case scenario with a maximum downtime until the failure is recognized

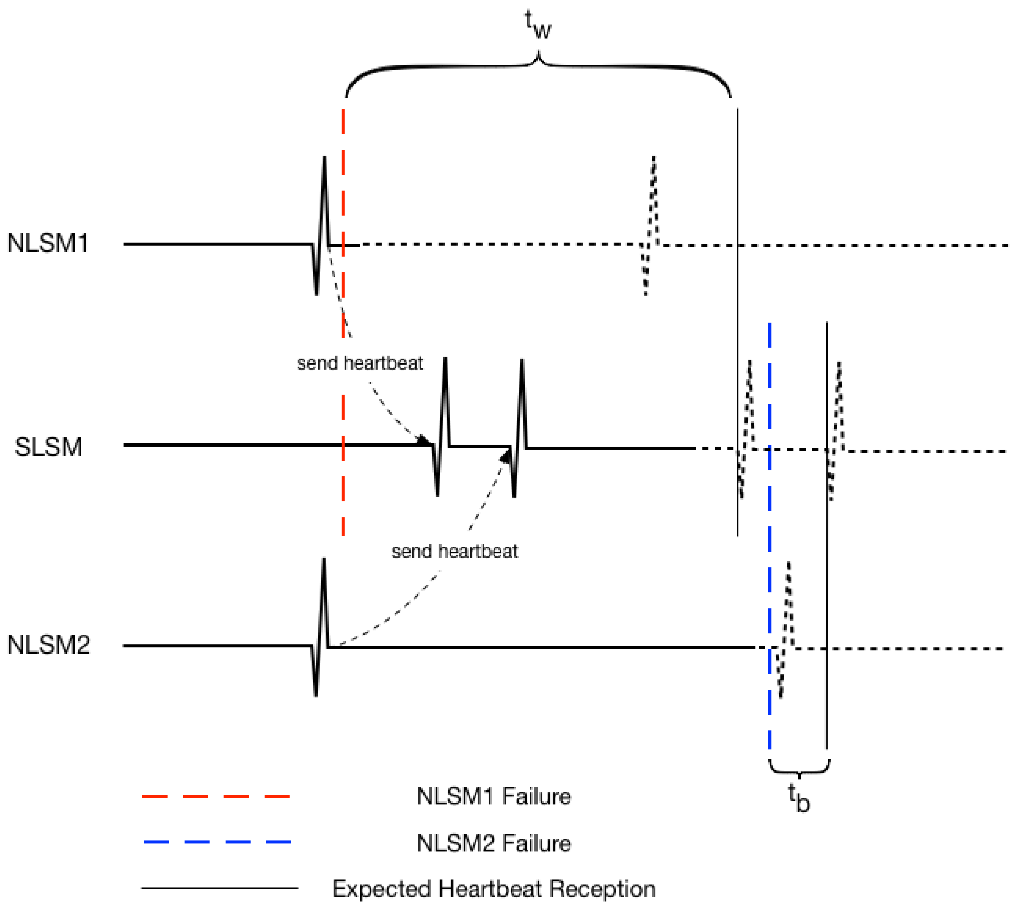


Figure 4.3: Worst and Best Case Scenario of Node Failure Detection

and the node becomes a migration candidate.

The best case scenario, by contrast, is demonstrated by NLSM2. Here again, heartbeat signals are sent to the SLSM indicating the status. The signal is received by the SLSM after NLSM1's. However, this time the node fails at the end of the heartbeat period, i.e. shortly before the emission of the new signal. Thus, the time between the failure and its detection indicated by t_b is minimized.

The outcome of this reflection suggests that the heartbeat frequency should be set very high to undermine this effect. However, this conflicts with preventing network congestion and application responsiveness.
Finally, the increase of the heartbeat frequency can even have the opposite effect. Node failures may be detected where this is not the case because the bandwidth cannot support the frequency. Thus, the reception of the heartbeat signals is expected before they can be transmitted.

4.4 SLA Violations

The provision of an autonomous service management is characterized by little user intervention. Thus, the treatment of SLA violations can be seen as an indicator to achieve this goal.

The SLA parameters are recorded in the OSGi bundle manifest. Their implementation as dictionary allows to retrieve the values for each key. But before reading the information, the entries have first to be accessed. This is done by retrieving the bundle manifest headers calling `bundle.getHeaders()`. The following listing 4.1 depicts an exemplary bundle manifest containing SLA parameters.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-SymbolicName: serviceHostingEnvironment
Bundle-Version: 1.0.0
CPU-Requirement: 3.23%
RAM-Requirement: 5.23E7
Bundle-Name: Service Hosting Environment Manifest
Import-Package: smart.api.pkg;version="[1.0.0,2.0.0)"
Export-Package: she.pkg;version="1.0.0"
```

Listing 4.1: Simple Bundle Manifest

This means that the bundle requires at least 3.23% of the available CPU and at least 52.3 Megabytes of memory.

The examination of whether an SLA violation exists is made by comparing the required resources with the actual resource consumption.JMX allows only to measure the percentage of the CPU so that both the SLA parameters and the resource consumption recording have to be content with this measurement unit. This is a serious disadvantage of JMX because the CPU data should be represented as absolute value in Hertz because each hosting environment has different capacities. However, in order to assess whether SLA violations are properly propagated and recovered the information in percent is sufficient, particularly when testing in a homogeneous environment like the iLab.

Another drawback resulting from using JMX is that the current resource consumption for each bundle cannot be accurately retrieved. Nevertheless, the application of a threshold alleviates the impact of this data.

The selection of which service to migrate is kept simple for test purposes and follows the LIFO (Last In First Out) principle. That means that among the running services those which have only been installed at the end are migrated first.
A more sophisticated approach would be to iterate through each bundle and read their SLA parameters. The bundle with the lowest resource requirements is the first to be migrated. However, this implies that each SLA parameter has the same priority and

that the bundle with the lowest resource requirements is always the best choice for migration. But this implies that the service with the least resource requirements is also the least important one. This is an assumption that is not true because the importance of services is a very subjective matter. While some consider a service crucial, the same service can be insignificant for others.
Therefore, the mandatory indication of the priority expressed as a floating point is suggested as an additional bundle manifest entry.

4.5 Autonomous Service Management Scenarios

This section introduces three different scenarios for autonomous service management in Smart Spaces. They are chosen because they cover the most frequent cases where user intervention is very high. However, they represent scenarios that can occur in every sort of Smart Spaces and that can be made more comfortable by an autonomous service management. The evaluation of the runtime environment designed in the previous sections is based on these scenarios.

4.5.1 Node Failure

The most harmful event is certainly the failure of a complete node. It implies that all services running on the failed node become unavailable. This means that the services need a new host until the node is rebooted.
The autonomous service management is responsible for bringing theses services back to life. The SLSM is informed about the node's failure by not receiving its heartbeat anymore. It then migrates the affected services to other nodes according to their capacities.

The service failure scenario without explicit node failure cannot be realized. The lifecycle management of the runtime environment takes care of the failed services under the hood so that an intervention of the SLSM is not necessary and the process cannot be monitored.

4.5.2 VSL Disconnection

In contrast to node failures, the deficit of a VSL disconnection assumes that the affected nodes keep working.
The disconnection is autonomously repaired by reconnecting to the middleware. The connector is reactivated and callbacks are brought back to life.

4.5.3 SLA Violation Recovery

Services or nodes do not have to fail to make user intervention necessary. It suffices that SLA parameters cannot be guaranteed.
Such SLA violations are propagated to the SLSM which finally decides what to do, i.e. which service is to be migrated and where it is to be migrated.

Chapter 5

Implementation

This section provides a brief overview of interesting implementation details of the integration of the OSGi framework into the SHE and the connection between SHE and NLSM. To enable the communication between NLSM and SLSM the well-defined interface has to be considered. However, the information come originally from the SHE. Therefore, the adapter design pattern is used on requests of the SLSM. Thus, an intermediary abstraction is created by the NLSM to match the information from the SHE with the interface definition of SLSM. This allows, on the one hand, to make counterchecks of the NLSM data against the SHE data and, on the other hand, to easily react to changed or new requirements. Instead of matching objects, context models have to be matched which involves more abstraction.

This chapter is structured as follows:
Section 5.1 presents the setup of the OSGi framework embedded into SHE.
Section 5.2 describes the publish-subscribe interaction between SHE and NLSM.

5.1 OSGi Setup

The OSGi framework as the selected execution environment is embedded into the SHE. Each lifecycle relevant operation is handled by OSGi under the hood. The OSGi framework is implemented with Apache Felix.

The Felix framework is implemented by the `org.apache.felix.framework.Felix` class. Listing 5.1. shows the initialization of the Felix environment in the *run()* method.

```
public void run() throws  Exception {

   this.vslNodeFactory = this.connector.getNodeFactory();
   Map configMap = new HashMap();
   this.activator = new HostActivator();
   List list = new ArrayList();
   list.add(this.activator);
   configMap.put(FelixConstants.SYSTEMBUNDLE_ACTIVATORS_PROP, list);

   try {
      //Initialize Felix
      framework = new Felix(configMap);
      framework.start();
      this.context = framework.getBundleContext();
      //Invocation of NLSM as first bundle
      Bundle nlsm = this.context.installBundle("file:"
               + this.workingDirectory + "/nlsm.jar");
      nlsm.start();
   } catch (Exception e) {
      System.err.println("Could␣not␣␣create␣framework:␣" + e);
      e.printStackTrace();
   }

   ...

}
```

Listing 5.1: OSGi Framework Initialization

The *start()* method is used to start the framework instance which implicitly invokes the *init()* method. While *init()* transfers the framework instance in the Bundle.`STARTING` state, the *start()* method results into the Bundle.`ACTIVE` state. The *init()* method is necessary because the framework does not have a `BundleContext` when being created for the first time, so a transition to the Bundle.`STARTING` state is required to acquire its context for performing various tasks, such as installing bundles.

5.2 SHE-NLSM Interaction

The communication between SHE and NLSM is done using the publish-subscriber design pattern. Thus, both SHE and NLSM have their own context models to exchange service information.

Listing 5.2 shows a characteristic SHE context model node invocation triggered by the NLSM.

```java
public void installService(String serviceName) {
    try {
        this.connector.get(connectedAgent + "/" + SHE_NAME
                + "/installService/serviceName=" + serviceName);
    } catch (VslException e) {
        e.printStackTrace();
    }
}
```

Listing 5.2: SHE Node Invocation by NLSM

The node *installService* is invoked on the SHE context model. The *get()* method triggers a callback function on the SHE side which is depicted in listing 5.3. Finally, the OSGi framework is invoked here by installing and starting the bundle.

```java
@Override
public void installBundle(String serviceName) {
    if (jarLocation != null) {
        try {
            Bundle bundle = this.context.installBundle("file:"
                                + serviceName);
            bundle.start();
        } catch (BundleException e) {
            e.printStackTrace();
        }
    }
}
```

Listing 5.3: Callback on SHE Side

The communication via service connectors does not only allow to use callbacks but also to provide a better transparency for service requests.

Chapter 6

Evaluation

In this chapter the service management with a focus on the runtime environment designed in chapter 4 is evaluated. Simulating different failure scenarios the influence of the runtime environment and the autonomous service management on the service availability is tested <R 1>. The results prove that the runtime environment has a positive influence on the resilience of the service management <R 2> and that service downtime can be reduced.

This chapter is structured as follows:
Section 6.1 introduces the testbed used for the evaluation.
Section 6.2 assess the impact of the runtime environment on the availability of the services, thus on the resilience of DS2OS.
In section 6.3, the convergence time is measured between the dis- and reconnection of a node to the VSL.
The autonomy on a local level is assessed in section 6.4 by examining the triggering of SLA violations.
Finally, section 6.5 analyzes the performance of the SHE and the embedded runtime environment in terms of their resource consumption.

6.1 Testing Environment

This section describes the hard- and software setup used to assess the runtime environment embedding in the following sections.

For the measurement, a computer with the following specification is used: Intel Core i5 processors with 2.5 Ghz, SSD drive and 16 GB of main memory where the macOS operating system is running on. A knowledge agent is running on the computer for each experiment.

All experiments are conducted with different VSL services. Furthermore, OSGi services as variable units with almost the same size are used. They are created via the OSGi framework and contain different characteristics such as special entries in their manifest files. The way in which these services are used for experiments are described in the following sections 6.2-6.4.

6.2 Availability / Resilience

Autonomous service management enables services to remain available even if the nodes on which they are located fail and become disconnected from the middleware <R 1>, thus increasing the resilience of DS2OS. This experiment makes the attempt to assess the advantages of the runtime environment in the failure scenario described in section 4.5.1.

6.2.1 Setup

This experiment utilizes a computer to simulate the interaction between SHE and NLSM running on it. The SHE initializes automatically an NLSM instance. The used computer has a 2.5 GHz Intel Core i5 processor and 16 GB of main memory.
The test is repeated 20 times to reduce random factors. After each run, the bundle cache is deleted to reinstall and restart the services. The metrics are logged and prepared to be visualized as diagrams.

6.2.2 Expected Results

Mission-critical systems often have goals of 4 nines (99.99% availability), which is less than an hour downtime per year, or even 5 nines (99.999% availability), which is 5 minutes of downtime per year. This is, however, not realizable for a constantly changing network topology such as DS2OS.
The service replication mechanism by [11] has already been mentioned in the related work. Their distributed self-adaptive system is broadly similar to DS2OS's. They measure the service availability under both TCP and UDP which is affected by the time to choose a host acting as server. Thus, latency in the communication is crucial for the performance. Both TCP and UDP result in a similar service availability depending on the number of nodes the network consists of. Thus, the metric under UDP ranges between 99.84% for a network of 4 nodes and 97.51% in case of 16 nodes as depicted in figure 6.1.

However, their experiment does not mention any node or service failures. Their definition of service availability is therefore the time in which a node can access a service replica when being connected to other nodes. This corresponds to the case how long the SLSM needs to choose a node and to deploy a service on that node.

But even without such a replication mechanism the service availability for the autonomous service management of DS2OS is expected to be better than 97% for a P2P network of 4 to 16 nodes in case of one node failure due to its migration capability. The performance is supposed to depend in this case on the deployment time, thus also on

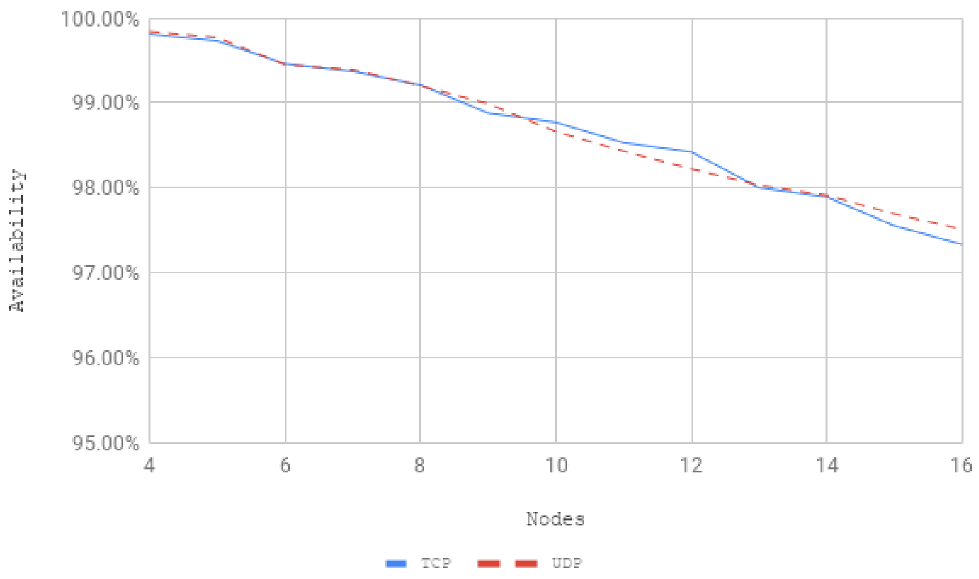


Figure 6.1: Service Availability Experiment from [11]

the latency. Furthermore, it is expected to depend on the heartbeat frequency, i.e. how frequently the nodes send their status messages to the SLSM.

6.2.3 Evaluation

Before assessing the service availability of the autonomous service management system, the influence of the number of services on the downtime is investigated. To exclude the impact of the deployment time, the services have nearly the same size and the SLSM is mocked.

Figure 6.2 illustrates the impact of the number of failed services on the total downtime. The services have nearly the same size so that their deployment time should not differ significantly. The downtime of the services is simulated by stopping them and reinstalling as well as restarting them on another node. The downtime is adjusted by the period of manual interaction.
Migrating and starting only one service produces a downtime between 35 ms and 51 ms. The combination of two services, however, increases the range of the total downtime partially significantly. While the minimum downtime is 54 ms, thus almost the upper bound of the downtime of one service, the maximum is 320 ms. The declaration is that in the latter case two services are installed which interact. The `HelloProvider` bundle in this example provides the logic of a simple *Hello World* output. The receiver of this logic is the `HelloConsumer` service where the string is printed by the `BundelActivator`. The partial unilateral dependence can also be recognized in the case of three failed

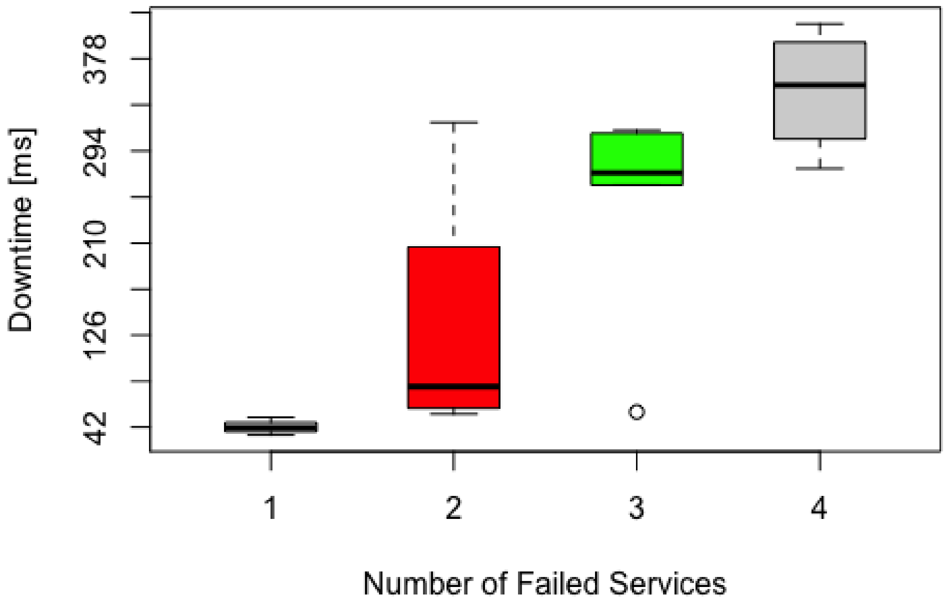


Figure 6.2: Service Downtime relating to Number of Failed Services

services. The minimum of 56 ms is very low because the failed services do not depend on each other while the other values range between 274 ms and 313 ms where at least two services interact. In case of four failed nodes the two dependent services are always involved.
This small example demonstrates the impact of the unilateral dependence of services on their downtime. The number of failed services itself, by contrast, does not have such an effect. Thus, it can be concluded that the deployment time and the dependence of services affect their downtime regardless of the network size or number of node failures.

Keeping this in mind the evaluation of the impact of node failures on the availability has to be done without dependent services. The same amount of non-dependent services has to run on all nodes.
For comparison purposes, the time of the simulated execution is also one hour as in the case of [11]. Furthermore, the number of nodes ranges under these conditions between 4 and 16. The simulation has been performed 20 times to eliminate the influence of random factors. The availability is measured by recording the downtime and uptime. If, for instance, a service has 3 minutes downtime during the measurement of one hour, the availability is calculated as follows:
1 hour corresponds to 60 minutes. The downtime expressed as a percentage is therefore

$(3/60 * 100) = 5\%$. Thus, the availability equals $100\% - 5\% = 95\%$.
If several services are running, their availability is measured by adding them up:

$$\text{Total Availability} = \sum_{i=1}^{n}(1 - \frac{downtime_i}{uptime_i}) \times 100\% \tag{6.1}$$

where n is the number of failed services.

The impact of the SLSM has not been considered yet. The migration was mocked and the timestamps adjusted so that an ideal SLSM is simulated which immediately deploys the failed services on preselected nodes without any computation time. Thus, the variations in the heartbeat frequency and in the choice of the migration strategy are hidden. Furthermore, the variability of latency is bypassed.

Since the network size does barely have an effect on the service load balancer, i.e. the decision to which node a service is to be migrated according to [5], the consideration of a heartbeat frequency of 5 seconds and a deployment time of 10 seconds could lead to the output as depicted in the following graph.

For lack of preparation time for an integration test of this work and that of [5] the interaction between SLSM on the one hand and NLSM as well as SHE on the other hand is not investigated in detail in the iLab.

However, the following graph shows a possible simulation of the whole system in terms of service availability in dependence of the network size considering the partial results of both measurements.

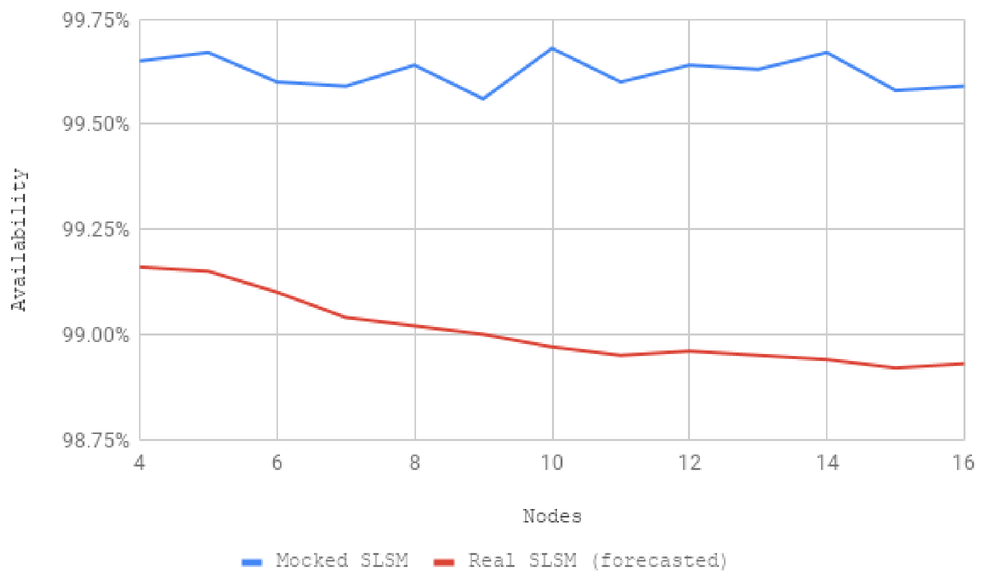


Figure 6.3: Service Availability relating to the Network Size

The measured values with a mocked SLSM are represented as blue line. The variability are due to measurement inaccuracies. However, the ideal mocked SLSM instantly knows after the absence of the heartbeat signal that the node failed and migrates the service to a predefined node. The red line indicates the forecasted service availability in case of interaction with the real SLSM. Due to the deployment time and the heartbeat frequency the graph will be under the mocked SLSM line. The line gradually decreases due to the network size. However, the descent is very small due to the performance of the service load balancer.

As already explained in section 4.5.1 the treatment of failed services is not able to be monitored due to the inner OSGi processes. However, during the experiments where SHE and NLSM instances were running an hour without interruption, the log file did not produce any service failures.

6.3 Self-Healing

The ability of the SHE to reconnect to the middleware in case of connection failure has already been discussed in section 4.1. The emphasis in this case is on its autonomous function.
In contrast to node failures where the instances crashed for some reasons, the reconnection scenario requires a working node with the limitation of temporary disconnection from the middleware.
This section evaluates the convergence time between the disconnection of a node and its reconnection to the VSL.

6.3.1 Setup

This experiment utilizes a computer to simulate the dis- and reconnection of the SHE to the middleware. The SHE initializes automatically a separate NLSM instance with its own process id to eliminate side effects of the SHE-NLSM communication. The used computer has a 2.5 GHz Intel Core i5 processor and 16 GB of main memory.
The disconnection is simulated by shutting down the connector (`connector.shutdown()`). From then on, the system time is measured. Immediately after that, the connection node is tried to be requested. Due to its failure the reconnection mechanism is triggered (`connector.activate()`, `connector.registerService(manifest)`). After successful reconnection, the time is stopped and recorded.
The test is repeated 20 times to reduce random factors. After each run, the bundle cache is deleted. The convergence time is calculated and logged to be visualized as box plot.

6.3.2 Expected Results

The requests answered by a local KA have a delay of some milliseconds [44]. The delay of the reactivation of the connector and the registering of the virtual nodes should also lie within the range of milliseconds.

6.3.3 Evaluation

The outcome of the experiment is depicted in figure 6.4.

The test series confirms that the reconnection mechanism works reliably whenever the SHE becomes disconnected from the VSL.

The box plot shows that the expected results apply. The range of the convergence time lies between 270 ms and 441 ms. Thus, the downtime of the affected services

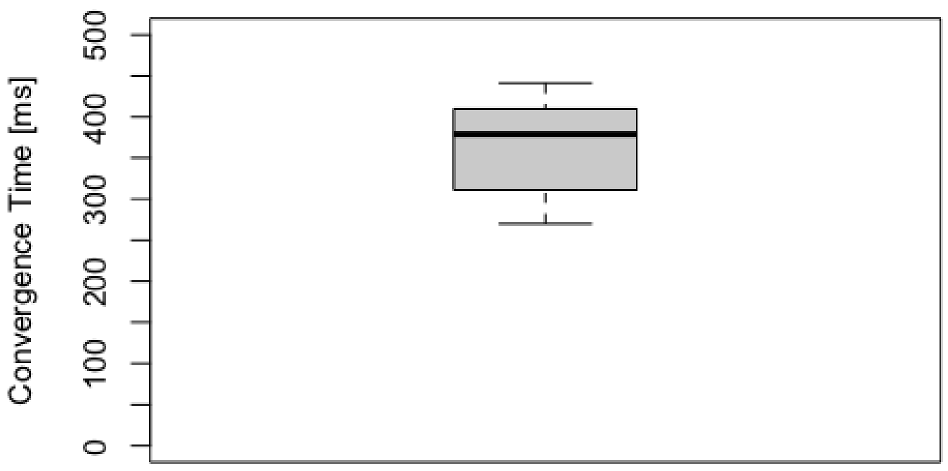


Figure 6.4: Convergence Time between Dis- and Reconnection to the VSL

running on the disconnected node is insignificant because the bundle cache prevents their redeployment.

6.4 Autonomy

The autonomous service management has already been qualitatively evaluated in section 6.2. However, the recovering from SLA violations seems suitable to assess autonomy on a node level. Therefore, the following experiment assesses whether SLA violations are properly treated <R 6> in the failure scenario described in section 4.5.3.

6.4.1 Setup

This experiment utilizes a computer to simulate the triggering of the SLA violation on which a KA, SHE and NLSM are running. The SHE initializes automatically an NLSM instance. The used computer has a 2.5 GHz Intel Core i5 processor and 16 GB of main memory. After each iteration, the bundle cache is deleted to reinstall and restart the service with a memory requirement of at least 3.8 GB (SLA parameter). Thus, the SHE which provides a maximum of 3.8 GB of main memory is expected to detect the SLA violation and trigger the migration of the last installed service.
The test is repeated 20 times to reduce random factors. The time between bundle initialization and migration invocation is logged and prepared to be visualized as box plot.

6.4.2 Expected Results

As qualitative outcome, the continuous detection of the SLA violation is expected as well as the triggering of the migration of the last installed service.
The quantitative measurement is dependent on the runtime of the SHE and therefore on the embedded runtime environment. The delay between installation and migration as inner processes should therefore be low.

6.4.3 Evaluation

The qualitative result confirms the expectation that SLA violations are reliably detected and the reactive actions in terms of migration triggering are properly propagated.

The quantitative result of the experiment is illustrated in figure 6.5. It shows that the delay between the installation and the triggering of the migration of the service whose SLA is violated ranges between 200 ms and 1430 ms.

Due to the simple and fast access of the SLA parameters by just reading the manifest headers, the variability can be explained by the size of the tested services resulting in different installation times.

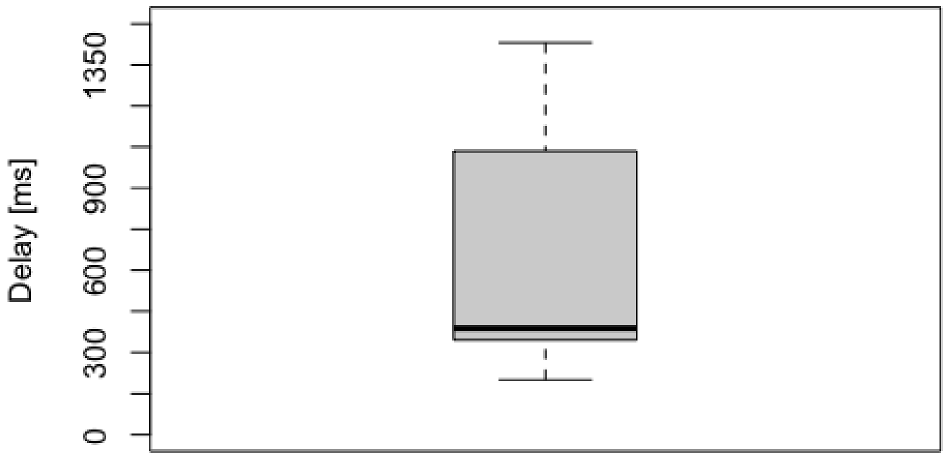


Figure 6.5: SLA Violation Triggering Time

This experiment demonstrates that SLA violations are not only qualitatively treated (i.e. they are reliably detected and propagated), but also that the period between bundle installation and the triggering of the service migration is quickly processed. Thus, service availability in a broader sense can be guaranteed by the local autonomous service management <R 1> .

6.5 Resource Usage

The decision when and which services to migrate depends not only on node failures but also on their capacity. Thus, nodes operating to full capacity should not receive new services. Instead, their load is to be reduced. Therefore, the monitoring of specific metrics like memory or CPU is crucial to assess the functionality of the nodes <R 3>. This experiment tries to evaluate the resource consumption of the runtime environment embedding SHE.

6.5.1 Setup

This experiment utilizes a computer to simulate resource consumption of the SHE running on it. The SHE initializes automatically a separate NLSM instance with its own process id to eliminate the influence of NLSM specific resource consumption. The used computer has a 2.5 GHz Intel Core i5 processor and 16 GB of main memory.
The test is repeated 20 times to reduce random factors. After each run, the bundle cache is deleted to reinstall and restart the services. The metrics are logged and prepared to be visualized as diagrams.

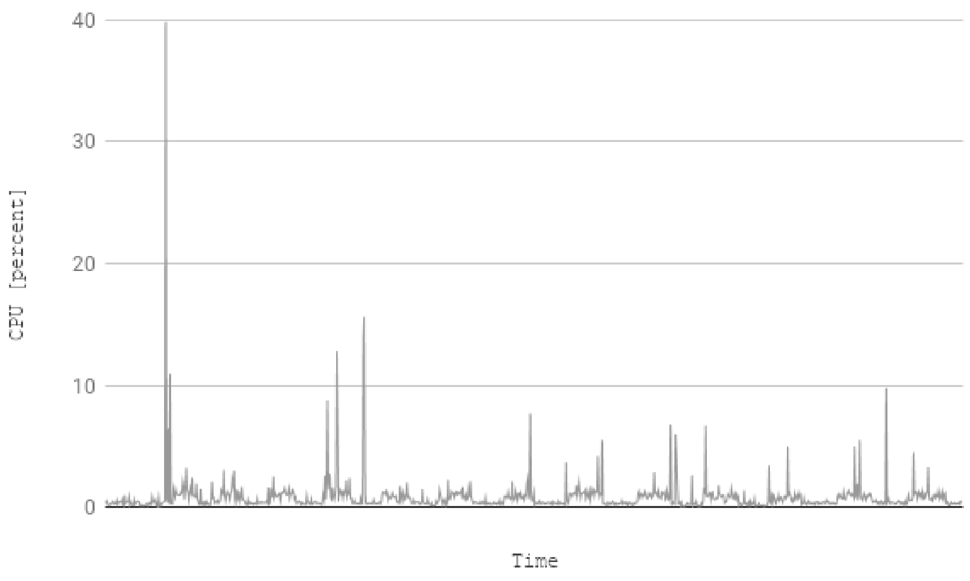
6.5.2 Expected Results

The use of JMX instead of OSGi's proper resource monitoring allows only to measure the JVM as a whole. The starting process of SHE is supposed to consume most resources, especially in terms of CPU.

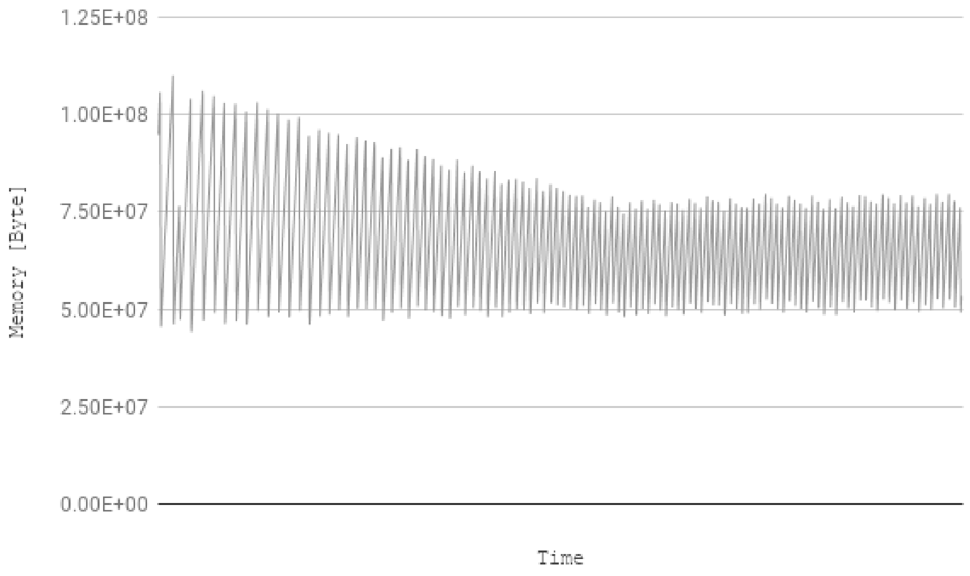
6.5.3 Evaluation

The results are qualitatively checked by comparing them to the output of Java's embedded jconsole. The metrics were recorded during a period of 20 minutes.
Figure 6.6a depicts the CPU consumption during this time. The big amplitude at the beginning up to 40% characterizes the instantiation of the Apache Felix framework. Other significant fluctuations thereafter ranging between 9% and 15% are due to the installation and starting of services.

The memory counterpart is depicted in figure 6.6b. Both heap and non-heap memory of the JVM are recorded. It can be seen that the measurement values range between 50 MB and 100 MB. The high variability can only be explained by the internal processes of the execution environment. In the beginning, the Apache Felix framework creates the bundle cache. This is a separate cache only for installed OSGi bundles which is internally managed by the framework. The OSGi documentation [3] gives a hint that the creation of a class loader for each bundle starting can significantly increase the memory footprint. At the same time, the JVM garbage collector may try to shrink the heap size or not for very sound CPU optimization reasons. The comparison of the results with jconsole show the same picture.



(a) CPU Consumption of SHE



(b) Memory Consumption of SHE

Figure 6.6: Resource Consumption of SHE

Chapter 7

Conclusion

The main goal of this thesis was to assess whether the embedding of an execution environment in the context of an autonomous hierarchical service management can improve service availability. For this purpose, different runtime environments have been investigated and compared in chapter 2 in terms of the collected requirements for an autonomous service management.

Chapter 3 has analyzed different projects which already used OSGi as runtime environment. These demonstrated that the utilization of a runtime environment can provide more or less an autonomous system depending on its architecture and the network environment. Furthermore, approaches of other research areas have been investigated separately for each elaborated requirement. Thus, concepts could be adopted or compared to other solutions to find the best suitable results.

Keeping in mind the research questions formulated in chapter 2 and considering the research results from the related work in chapter 3 a runtime environment to be embedded into an autonomous service management was designed in chapter 4.
The implementation of OSGi was done via the Apache Felix framework to provide the lifecycle management of the services. Thus, the system is always aware of the states of the services and can react appropriately to changes <R 3>.
The monitoring of the resources of the underlying hosting environment enables to trigger migration requests so that node congestions are prevented <R 2>. This capability provides autonomous service management without user intervention so that services continue to run on other nodes increasing their availability <R 1>.
Furthermore, the recovering of SLA violations is triggered in time resulting in an autonomous local service management <R 6>. The interface between SLSM and NLSM is provided <R 7>. Using OSGi allows services to be updated on the fly, thus improving their availability <R 4>.
Finally, the overhead is kept to a minimum <R 5> by using heartbeats instead of pings and the integrity of the service binaries is guaranteed by a separate certificate

management system developed by [25] <R 8> .

Chapter 6 evaluated a prototype implementation of the bottom-up service management including the runtime environment. The results proved that the runtime environment ameliorates the availability of services within a distributed S2O system. The comparison of the results for this metric with those of a related work with a similar architecture like DS2OS shows that the created bottom-up autonomous service management with an embedded runtime environment performs better in terms of service availability than the comparative project resulting in values above 97%. Thus, the lack of a replication mechanism due to the overhead does not reduce the service disposability. Thus, the resilience of DS2OS can be increased while reducing user intervention.
In this context, the effect of dependent services emerges as a significant factor compared to the amount of services to be migrated apart from their deployment time.
Finally, the triggering of SLA violations and the resource consumption of the SHE with the embedded runtime environment is measured.

To sum up the major results of this thesis, the embedding of a runtime environment into a hierarchical service management can increase service disposability and reduce user intervention. Therefore, further investigation is promising as the development of S2O systems proceed.

Chapter 8

Future Work

The implemented prototype contains the core features for a resilient and remotely manageable runtime environment with a focus on providing availability. However, the spectrum of additional and advanced features around service management in general an runtime environments in particular which occurred while working on this thesis is very broad.
Therefore, the very last part of this thesis is supposed to recommend some selected suggestions for future work in this area.

The selection of OSGi as runtime environment was discussed in 2.2.4. However, there were sometimes only slight differences between the other suitable approaches. Therefore, comparing the other technologies with regard to the prerequisites should be an interesting aspect.
The interaction of Docker and OSGi was already mentioned in the analysis and related work. Nonetheless, this approach is very new and the research has only just begun. There are already some open source projects like Amdatu-Kubernetes making the Kubernetes[1] REST API available to Java and which is tailored to work with OSGi [57]. But it covers only a small part of the possible applications of Docker.
To take yet another step forward, the combination of OSGi, Docker and Java 9 could bring to light surprising results.

Another research area which is promising for service management in terms of guaranteeing higher availability of services is to provide a probabilistic node failure detection to implement a replication mechanism such that services are deployed or migrated to new nodes immediately prior to the failure of the current nodes. This would reduce the downtime of services significantly. However, this requires to collect empirical data before a forecast of node failures can be made. For this reason and because the de-

[1]Kubernetes is an open source container orchestration platform allowing a large number of Docker containers to work together seamlessly by interacting with the Docker engine to coordinate the scheduling and execution of the containers

velopment would go beyond the scope of this thesis a replication mechanism is not implemented.

Heartbeats as indication if a node is still alive are already realized in the prototype. A more sophisticated approach would, however, be to generate dynamic heartbeats. That means that the nodes define their own heartbeat intervals according to their capacities in terms of bandwidth, CPU or memory for instance. Thus, the frequency of heartbeat signals can be optimized such that node failures can be detected earlier.

Finally, future work could also include machine learning approaches identifying SLA violations before they happen, e.g. by collecting data about node capacity utilization to predict future bottlenecks.

Bibliography

[1] M.-O. Pahl, "Distributed Smart Space Orchestration," Ph.D. dissertation, Technische Universität München, 2014.

[2] P. H. Su, C. S. Shih, J. Y. J. Hsu, K. J. Lin, and Y. C. Wang, "Decentralized fault tolerance mechanism for intelligent IoT/M2M middleware," in *2014 IEEE World Forum on Internet of Things (WF-IoT)*, March 2014, pp. 45–50.

[3] OSGi Alliance, "OSGi Core Release 7 Specification," https://osgi.org/specification/osgi.core/7.0.0/ch01.html, 2018, accessed: 2018-05-24.

[4] H. Cummins and T. Ward, *Enterprise OSGi in Action: With Examples Using Apache Aries*. Greenwich, CT, USA: Manning Publications Co., 2013.

[5] D. Celik, "Semi-Autonomous IoT Service Management on Unattended Nodes," Master's thesis, Technical University of Munich, August 2018.

[6] J. Chen and L. Huang, "Dynamic Service Update Based on OSGi," in *2009 WRI World Congress on Software Engineering*, vol. 3, May 2009, pp. 493–497.

[7] A. Sathiaseelan, L. Wang, A. Aucinas, G. Tyson, and J. Crowcroft, "SCANDEX: Service Centric Networking for Challenged Decentralised Networks," in *Proceedings of the 2015 Workshop on Do-it-yourself Networking: An Interdisciplinary Approach*, ser. DIYNetworking '15. New York, NY, USA: ACM, 2015, pp. 15–20.

[8] D. Barrera, J. Clark, D. McCarney, and P. C. van Oorschot, "Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android," in *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ser. SPSM '12. New York, NY, USA: ACM, 2012, pp. 81–92.

[9] G. Guerrero-Contreras, S. Balderas-Díaz, C. Rodríguez-Domínguez, A. Valenzuela, and J. L. Garrido, "An Approach Addressing Service Availability in Mobile Environments," in *Intelligent Environments (Workshops)*, ser. Ambient Intelligence and Smart Environments, D. Preuveneers, Ed., vol. 19. IOS Press, 2015, pp. 46–57.

[10] G. Guerrero-Contreras, J. L. Garrido, S. Balderas-Díaz, and C. Rodríguez-Domínguez, "A Context-Aware Architecture Supporting Service Availability in Mobile Cloud Computing," *IEEE Transactions on Services Computing*, vol. 10, no. 6, pp. 956–968, Nov 2017.

[11] G. Guerrero-Contreras, J. L. Garrido, M. J. R. Fórtiz, G. M. P. O'Hare, and S. Balderas-Díaz, "Impact of Transmission Communication Protocol on a Self-adaptive Architecture for Dynamic Network Environments," in *Recent Advances in Information Systems and Technologies*, Á. Rocha, A. M. Correia, H. Adeli, L. P. Reis, and S. Costanzo, Eds. Springer International Publishing, 2017, pp. 115–124.

[12] C. L. Wu, C. F. Liao, and L. C. Fu, "Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 2, pp. 193–205, March 2007.

[13] C. Lee, D. Nordstedt, and S. Helal, "Enabling Smart Spaces with OSGi," *IEEE Pervasive Computing*, vol. 2, no. 3, pp. 89–94, July 2003.

[14] T. Gu, H. K. Pung, and D. Q. Zhang, "Toward an OSGi-Based Infrastructure for Context-Aware Applications," *IEEE Pervasive Computing*, vol. 3, no. 4, pp. 66–74, Oct 2004.

[15] H. Ahn, H. Oh, and C. O. Sung, "Towards Reliable OSGi Framework and Applications," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, 2006, pp. 1456–1461.

[16] J. E. López de Vergara, V. A. Villagrá, C. Fadón, J. M. González, J. A. Lozano, and M. Álvarez Campana, "An Autonomic Approach to Offer Services in OSGi-based Home Gateways," *Comput. Commun.*, vol. 31, no. 13, pp. 3049–3058, Aug. 2008.

[17] S.-T. Cheng, C.-H. Wang, and G.-J. Horng, "OSGi-based smart home architecture for heterogeneous network," *Expert Systems with Applications*, vol. 39, no. 16, pp. 12 418 – 12 429, 2012.

[18] J. E. Kim, G. Boulos, J. Yackovich, T. Barth, C. Beckel, and D. Mosse, "Seamless Integration of Heterogeneous Devices and Access Control in Smart Homes," in *2012 Eighth International Conference on Intelligent Environments*, June 2012, pp. 206–213.

[19] ZigBee Alliance, http://www.zigbee.org/, 2018, accessed: 2018-01-02.

[20] Bluetooth SIG, https://www.bluetooth.com/, 2018, accessed: 2018-01-01.

[21] H.-S. Choi and W.-S. Rhee, "IoT-Based User-Driven Service Modeling Environment for a Smart Space Management System," *Sensors*, vol. 4, no. 11, pp. 22 039–22 064, 2014.

[22] T. Kawashima, J. Ma, R. Huang, and B. O. Apduhan, “GUPSS: A Gateway-Based Ubiquitous Platform for Smart Space,” in *2009 International Conference on Computational Science and Engineering*, vol. 2, Aug 2009, pp. 213–220.

[23] M.-O. Pahl, “Data-Centric Service-Oriented Management of Things,” in *2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)*, May 2015, pp. 484–490.

[24] Apple Inc., “HomeKit,” https://developer.apple.com/homekit, 2017, accessed: 2017-10-27.

[25] L. Donini, “Autonomous Certificate Management for Microservices in Smart Spaces,” Master’s thesis, Technical University of Munich, August 2018.

[26] M.-O. Pahl and G. Carle, “The Missing Layer - Virtualizing Smart Spaces,” in *2013 IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, March 2013, pp. 139–144.

[27] S. Blom, M. Book, V. Gruhn, R. Hrushchak, and A. Köhler, “Write Once, Run Anywhere - A Survey of Mobile Runtime Environments,” in *2008 The 3rd International Conference on Grid and Pervasive Computing - Workshops*, May 2008, pp. 132–137.

[28] K. Keahey, M. Ripeanu, and K. Doering, “Dynamic Creation and Management of Runtime Environments in the Grid,” in *In Workshop on Designing and Building Grid Services*, 2003.

[29] Techopedia. Runtime Environmnet (RTE). Accessed 2017-12-27. [Online]. Available: https://www.techopedia.com/definition/5466/runtime-environment-rte

[30] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 4th ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2014.

[31] S. Liebald, “Caching content in smart spaces,” Master’s Thesis, Technische Universität München, 2016.

[32] MuleSoft, “Services in SOA,” https://www.mulesoft.com/de/resources/esb/services-in-soa, 2018, accessed: 2018-03-26.

[33] R. Carbou, M. Diaz, E. Exposito, and R. Romain, “Service Management,” in *Digital Home Networking*. Chichester, UK: John Wiley & Sons, Inc, February 2013, pp. 259–307.

[34] A. F. Antonescu, A. M. Oprescu, Y. Demchenko, C. d. Laat, and T. Braun, “Dynamic Optimization of SLA-Based Services Scaling Rules,” in *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*, vol. 1, Dec 2013, pp. 282–289.

[35] J. Sen, "Secure and Privacy-Aware Searching in Peer-to-Peer Networks," in *Data Privacy Management and Autonomous Spontaneus Security*, J. Garcia-Alfaro, G. Navarro-Arribas, N. Cuppens-Boulahia, and S. de Capitani di Vimercati, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 72–89.

[36] J. Ferreira, J. Leitão, and L. Rodrigues, "A-OSGi: A Framework to Support the Construction of Autonomic OSGi-Based Applications," in *Autonomic Computing and Communications Systems*, A. V. Vasilakos, R. Beraldi, R. Friedman, and M. Mamei, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 1–16.

[37] V. Stankovski, S. Taherizadeh, I. Taylor, A. Jones, C. Mastroianni, B. Becker, and H. Suhartanto, "Towards an Environment Supporting Resilience, High-Availability, Reproducibility and Reliability for Cloud Applications," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, Dec 2015, pp. 383–386.

[38] N. Suri, "Dynamic Service-oriented Architectures for Tactical Edge Networks," in *Proceedings of the 4th Workshop on Emerging Web Services Technology*, ser. WEWST '09. New York, NY, USA: ACM, 2009, pp. 3–10.

[39] M.-O. Pahl and G. Carle, "Taking Smart Space Users into the Development Loop: An Architecture for Community Based Software Development for Smart Spaces," in *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, 2013, pp. 793–800.

[40] L. Chen, D. Xiong, H. Wang, and P. Zou, "Web Service Run-Time Monitoring and Visualization Analysis Based on Probe," in *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*, June 2016, pp. 446–451.

[41] R. Hall, K. Pauls, S. McCulloch, and D. Savage, *OSGi in Action: Creating Modular Applications in Java*, 1st ed. Greenwich, CT, USA: Manning Publications Co., 2011.

[42] A. Baumann, J. Appavoo, D. D. Silva, O. Krieger, and R. W. Wisniewski, "Improving Operating System Availability With Dynamic Update," in *Proceedings of the Workshop on Operating System and Architectural Support for the On-Demand IT Infrastructure*, October 2004.

[43] A. Jecan, *Java 9 Modularity Revealed*, 1st ed. Apress, 2017.

[44] M.-O. Pahl, G. Carle, and G. Klinker, "Distributed Smart Space Orchestration," in *NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium*, April 2016, pp. 979–984.

[45] K. H. Neil Bartlett, "Java 9, OSGi and the Future of Modularity (Part 2)," https://www.infoq.com/articles/java9-osgi-future-modularity-part-2, 2018, accessed: 2018-01-29.

[46] K. H. Neil Bartlett, “Java 9, OSGi and the Future of Modularity (Part 1),” https://www.infoq.com/articles/java9-osgi-future-modularity, 2018, accessed: 2018-01-29.

[47] N. Bartlett, “OSGi and Java 9 Modules Working Together,” https://blogs.paremus.com/2015/11/osgi-and-java-9-modules-working-together/, 2018, accessed: 2018-01-29.

[48] Docker, https://docs.docker.com/docker-cloud/apps/service-links/, 2018, accessed: 2018-01-02.

[49] Docker, https://docs.docker.com/config/thirdparty/prometheus/, 2018, accessed: 2018-01-02.

[50] P. Lillo, L. Mainetti, and L. Patrono, “A Public-Private Partnerships Model Based on OneM2M and OSGi Enabling Smart City Solutions and Innovative Ageing Services,” in *Cloud Infrastructures, Services, and IoT Systems for Smart Cities.* Springer International Publishing, 2018, pp. 49–57.

[51] J. Rykowski and D. Wilusz, “Comparison of architectures for service management in IoT and sensor networks by means of OSGi and REST services,” in *2014 Federated Conference on Computer Science and Information Systems*, Sept 2014, pp. 1207–1214.

[52] I. Brandic, V. C. Emeakaroha, M. Maurer, S. Dustdar, S. Acs, A. Kertesz, and G. Kecskemeti, “LAYSI: A Layered Approach for SLA-Violation Propagation in Self-Manageable Cloud Infrastructures,” in *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, July 2010, pp. 365–370.

[53] S. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen, “The Gator Tech Smart House: a programmable pervasive space,” *Computer*, vol. 38, no. 3, pp. 50–60, March 2005.

[54] H. Werner Pohl and J. Gerlach, “Using the Bridge Design Pattern for OSGi Service Update,” pp. 717–726, 01 2003.

[55] T. A. B. Nguyen, C. Meurisch, S. Niemczyk, D. Böhnstedt, K. Geihs, M. Mühlhäuser, and R. Steinmetz, “Adaptive Task-Oriented Message Template for In-Network Processing,” in *2017 International Conference on Networked Systems (NetSys)*, March 2017, pp. 1–8.

[56] C. Groba and S. Clarke, “Opportunistic Service Composition in Dynamic Ad Hoc Environments,” *IEEE Transactions on Services Computing*, vol. 7, no. 4, pp. 642–653, Oct 2014.

[57] Amdatu-Kubernetes, https://bitbucket.org/amdatulabs/amdatu-kubernetes, 2018, accessed: 2018-02-01.