Technische Universität München

Department of Informatics

Master's Thesis in Information Systems

Semi-Autonomous IoT Service Management on Unattended Nodes

Deniz Celik

Technische Universität München

Department of Informatics

Master's Thesis in Information Systems

Semi-Autonomous IoT Service Management on Unattended Nodes

Semiautonomes Management von IoT Services auf unüberwachten Knoten

Author	Deniz Celik
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Dr. Marc-Oliver Pahl, Stefan Liebald, M. Sc.
Date	August 13, 2018

Informatik VIII

Chair for Network Architectures and Services

I confirm that this thesis is my own work and I have documented all sources and material used.

Garching b. München, August 13, 2018

Signature

Abstract

Within the past years computing paradigms like the Internet of Things and Pervasive Computing led to a broad spectrum of application scenarios. One example are smart spaces: indoor environments containing smart devices which provide and consume microservices and which are able to work together in a useful way to create value for users. In the future, non-experts should be able to install such devices easily in their homes and to deploy new services to their smart spaces in one click, similar to the way apps are installed on smartphones. This entails multiple challenges since smart spaces are distributed systems instead of single devices, and users are going to expect the same level of usability and dependability as for their smartphones. Also, the software must have a low footprint since it will be deployed on constrained and energy-efficient devices like the Raspberry Pi. This thesis focuses on centralized and autonomous management of distributed services using the existing Virtual State Layer (VSL) middleware. An extensive literature review on different research fields which deal with autonomous service management is provided in order to extract concepts that can be applied to the problem domain. Based on the findings, a service management prototype is designed. In the end, the design is implemented and evaluated regarding performance and applicability in the target domain. Open challenges are discussed to motivate further research in the area.

Zusammenfassung

In den letzten Jahren sind durch Paradigmen wie das Internet of Things und Pervasive Computing viele Anwendungsszenarien entstanden. Ein Beispiel hierfür sind Smart Spaces, also Räume die mit Smart Devices ausgestattet sind, welche Microservices anbieten und konsumieren und dadurch in der Lage sind, zusammenzuarbeiten und dadurch Nutzen für Anwender zu schaffen. In Zukunft sollten auch Laien in der Lage sein, solche Geräte einfach in ihre Räume zu integrieren und deren Funktionalität durch das Installieren von Services zu erweitern - ähnlich dem Installieren von Apps auf Smartphones. Diese Vision bringt allerdings einige Herausforderungen mit sich, da es sich nicht um einzelne Geräte sondern um komplexe verteilte Systeme handelt, während Nutzer eine vergleichbare Zuverlässigkeit und Nutzbarkeit wie auf ihren Smartphones erwarten werden. Auch sollte die Software performant arbeiten, da diese typischerweise auf Geräte mit eingeschränkten Ressourcen verteilt wird, zum Beispiel den Raspberry Pi. Diese Forschungsarbeit beschäftigt sich mit zentralisiertem und autonomem Management von verteilten Services am Beispiel der Virtual State Layer (VSL) Middleware. Verschiedene Forschungsgebiete, welche sich mit autonomem Service Management befassen, werden in einer umfangreichen Literaturrecherche vorgestellt. Ziel ist hierbei die Identifikation von Konzepten welche auf die Problemdomäne dieser Arbeit anwendbar sind. Basierend auf den Ergebnissen wird ein Design für einen Service Management Prototypen erarbeitet. Zum Schluss wird der Prototyp implementiert und hinsichtlich seiner Performanz und Eignung für den Einsatz in der Problemdomäne evaluiert. Offene Herausforderungen werden diskutiert um für weitere Forschungsarbeit in dem Gebiet zu motivieren.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the past years, Internet of Things and Pervasive Computing have become very popular computing paradigms with the potential of changing the interaction between users and computers fundamentally. The idea is to connect small constrained devices and to let them work together in a way that the devices themselves are not directly recognized anymore. Emerging applications target companies as well as private end users. A good example are smart homes or smart spaces where rooms are equipped with numerous distributed devices providing services like closing the shutters or turning on the lighting. Although many of those applications are easily viable today, many open issues exist in research. On the one hand, controlling smart spaces should be as easy as possible for end users. In the best case, services can be installed to the environment with one click, similar to smartphone apps. The problem is that we are facing complex distributed systems composed of heterogeneous hardware and running a variety of services which must work together somehow without requiring users to deal with complex configurations or error handling. Promising research can be found in the field of autonomous computing, for example.

1.1 Thesis Goals

This thesis focuses service management in smart spaces consisting of distributed, heterogeneous nodes which run services that control the environment via sensors and actuators. From a user perspective, managing services in such a scenario is currently a complex task because of hardware requirements, dependencies between services and error handling. The ideal future smart space management system should manage itself completely autonomous, without requiring users to intervene. Deploying and starting new services should become an easy task one day - similar to installing and running apps on smartphones. With this in mind, the following thesis goals can be formulated:

1. Providing an extensive literature review on autonomous service management in different application areas.
2. Identifying concepts and mechanisms suitable for the presented problem domain and deriving a list of requirements for a service management solution.
3. Developing a comprehensive system design based on the requirements.
4. Developing a prototypical solution and evaluating it regarding applicability to the problem domain.

1.2 Methodology and Outline

The topic of this thesis cuts across several research areas, since mechanisms for the management of services and distributed systems are widely used and applicable. Because of this, a clear methodology for the literature review is even more necessary in order to stay on top of things. Figure 1.1 shows the process which has been conducted in order to find relevant literature. Most of the papers cited in this thesis were found in the IEEE Xplore Digital Library [7].

The fist step involved an unstructured keyword search which brought up papers in a variety of research areas. From this collection, relevant areas could be extracted.

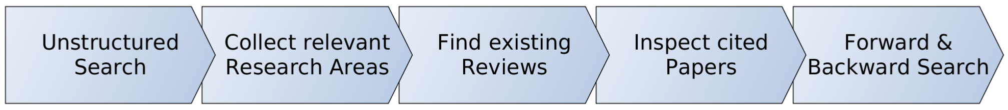


Figure 1.1: Literature Search Process

For each area, the goal was to find existing literature reviews, since they often provide good overview over a topic and help in finding relevant primary work. The identified primary papers were then examined and the relevant ones were used as the basic input for the analysis and related work of this thesis. If more specific information on a topic was sought, similar papers could be easily derived via forward and backward search.

The thesis is structured as follows. In chapter 2 important aspects used in the remainder of this work are analyzed and relevant research is presented. Based on this research, requirements for the solution design are collected. In chapter 3 a selection of existing approaches and solutions is presented. These fulfill multiple of the identified requirements and have similar goals to this thesis. Therefore differences to this work and eventually our research contribution will be worked out. A concrete solution design based on previous theoretical research findings is constructed in chapter 4. A feasible prototype fulfilling a subset of the requirements is defined and interesting implementation details are documented. In chapter 5 the prototype is evaluated and setup and results are presented. In the end a conclusion and outlook to future work is provided.

Chapter 2

Analysis

This chapter analyzes important building blocks of the topic and derives important aspects and requirements from existing research. In chapter 2.1 the problem domain and related research fields are introduced. Chapter 2.2 analyzes the DS2OS system which will be designed and implemented in the course of this thesis as well as the existing underlying *Virtual State Layer* (VSL) middleware. In chapter 2.3 basics around service quality and dependability are explained. Chapter 2.4 presents two common technologies for service lifecycle management. Afterwards, self-management or autonomous computing are introduced and a classification of different approaches is provided in chapter 2.5. As monitoring and the management of the produced data are important features of the planned solution, chapter 2.6 presents interesting research around these topics. All derived requirements are listed and condensed into higher-level research questions in chapter 2.7. The chapter also defines which components and aspects of the DS2OS system are focused in this thesis, and which not.

2.1 Problem Domain

This section presents the problem domain and an introduction to existing issues. Also, related research areas are shortly introduced.

2.1.1 Smart Space Service Management

This thesis addresses smart spaces and smart devices as well as related open challenges. A typical challenge with smart devices is their heterogeneity, as there are different manufacturers and platforms, as well as different data formats, facing a still existing lack of standardization [8, p. 732]. Certainly connecting the devices and benefiting from their data and functionality implies the existence of a standard or abstraction, in order to create a common language and understanding between them. There are several service-oriented approaches which create an abstraction over the specific functionality from the device level. Two examples which provide a comprehensive Service Management in the form of a middleware are Gaia OS [9] and the Distributed smart space Orchestration System (DS2OS) [1]. The latter will serve as the system for the design and implementation part of this thesis, and will be analyzed in chapter 2.2.

One set of issues with existing systems concerns the management of microservices in IoT scenarios. For example, the installation or deployment of services still requires system experts in most cases, since there is no easy way to install distributed services without complex configuration. In the future, users will want to change the functionality of their smart spaces easily by installing, removing or combining services. This is already possible on our smartphones, where services are bundled as apps and installed through a central software store. Transferring the concept of apps to smart spaces entails some challenges, as we have to deal with a distributed system instead of a single device. Instead of keeping one app on a single device running, services must be executed on different distributed nodes and could even be composed in order to model complex use cases. End users in that case should not be aware of the system architecture behind it and experience a high dependability, similar to software running on single devices. Autonomous computing research shows that there is promising work regarding complex systems that are able to manage themselves to a high degree. These self-management capabilities are a main concept which will be analyzed in chapter 2.5.

2.1.2 Related Research Areas

The unstructured literature search for this thesis brought up multiple research areas which are relevant to the topic. Although use cases sometimes differ significantly from our topic and specific algorithms are not directly applicable, suitable approaches and mechanisms can be found in any of these research areas. Figure 2.1 visualizes the identified research areas which will be introduced in the following. It will be pointed out how each area contributes to this thesis by presenting similarities.

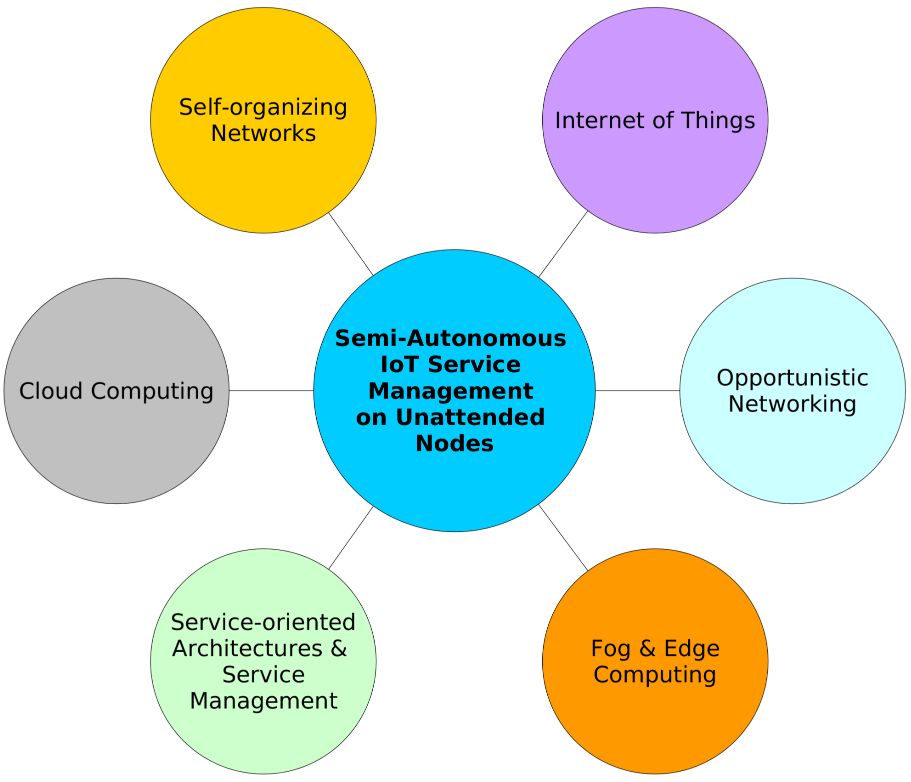

Self-organizing Networks
Internet of Things
Cloud Computing
Semi-Autonomous IoT Service Management on Unattended Nodes
Opportunistic Networking
Service-oriented Architectures & Service Management
Fog & Edge Computing

Figure 2.1: Related Research Areas

The Internet of Things (IoT) is a wide technological field covering many applications. Basically it means that smart objects are connected to the internet [10, p. 11]. This thesis targets smart spaces, where smart objects could be light controllers, regulate the heating or be fridges equipped with sensors for controlling their contents. Therefore IoT as a research area is obviously relevant to this thesis. Services will not be running on powerful servers, but on a multitude of constrained devices which are connected to a loosely coupled distributed system. These restrictions have to be considered throughout this thesis.

As already mentioned, this research focuses on managing microservices in distributed IoT systems. In contrast to monolithic applications, service-oriented computing (SOC) or service-oriented architectures (SOA) propose providing functionality via loosely coupled, small services which can be composed to more complex applications. We assume knowledge of basic SOA concepts which will be important throughout the thesis. We refer to [11] which provides a good basis on the topic.

Self-organizing Networks (SON) research is mostly concerned with cellular networks like LTE, though there are applications in other areas like sensor networks, for example.

A typical issue in SON research are cell outages in mobile networks and how to deal with them. In such a case, the system should be able to heal itself in order to maintain the signal quality and therefore the service quality for the end users. [12] provides an extensive survey over the topic and defines three key attributes of self-organization. These include adaptive behavior, distributed control and emergent behavior [12, p. 338]. The first attribute means that SON need to adapt their state or behavior when the system changes. Second, control must be distributed, there is no central administrative component. Emergent behavior includes that *"patterns [...] can be observed in a system without being explicitly programmed to exhibit that behaviour"* [12, p. 338]. It is obvious that there are several intersections and similarities between SON research and the topic of this thesis:

1. Both research focuses on distributed systems.
2. Autonomy respectively self-management are central issues in each case.
3. There exist similar typical tasks like load balancing, error detection and system configuration.

These similarities are not meant to be exhaustive but prove the relevance of SON research for this thesis.

Opportunistic Networking (ON) focuses on distributed networks where nodes are highly mobile. An example could be networks formed by private persons' smartphones. When those people are walking, driving car or taking the bus, their phones are moving with them all the time. Communication channels between devices are often only available in the order of seconds, resulting in a highly dynamic network structure. According to the survey in [13, p. 1102], it would require a significant amount of energy to keep the devices scanning for surrounding nodes all the time in order to create a complete network topology. Therefore ON relies on short-term network paths, enabling communication between nodes even if there is no end-to-end path available [13, p. 1101]. The survey focuses on neighbor discovery, which is a basic feature in ON. It provides three basic challenges which have to be tackled. First, nodes need to recognize the presence of other nodes in their neighborhood. Second, mobility models must be developed in order to understand when and where nodes are present. Third, the system must be able to learn occurring patterns and store this knowledge which can be reused in future similar situations. Thus, the better a node is able to anticipate where and when a connection becomes available, the less resources are wasted. Another interesting survey in the area of ON is presented in [14]. The paper targets opportunistic offloading which describes ONs where nodes are able to offload network traffic or computational tasks. Traffic offloading helps reducing the load on cellular networks: some nodes transfer data via an ON to several subscriber nodes, so that there is no need for each one to download the data via the internet [14, p. 1]. Computational offloading can be used to transfer tasks from constrained devices to nodes with free capacities, which send the result back as soon as computation is finished [14, p. 2]. These mechanisms are interesting for smart space research since devices are constrained and load can be distributed unevenly. In the case of multi-hop smart space networks, also routing and neighbor discovery are interesting concepts which could be valuable to our research.

Cloud Computing is a paradigm where infrastructure (IaaS), platforms (PaaS) or software (SaaS) are outsourced into global data centers. Computing tasks are executed by a resource pool consisting of multiple machines utilizing virtualization techniques and can be provided on a pay-per use basis [15, p. 347]. Customers have the great advantage that configuration and maintenance tasks are outsourced. Also the usage is dynamically scalable based on the customer's demands and agreed Service Level Agreements (SLAs) must be satisfied by the cloud provider. To make this possible, a lot of resource management on the provider's side is necessary. The survey in [15, p. 352] specifies three tasks related to resource management:

1. *Resource Provisioning* assigns resources to customers.
2. *Resource Allocation* distributes resources between multiple users or programs.
3. *Resource Scheduling* takes care of the temporal assignment of resources.

Other research focuses on energy-efficient resource management in clouds. A comprehensive survey can be found in [16]. The paper states that the goal is to minimize the number of active machines in order to reduce energy consumption. It differentiates two major categories of cloud resource management systems: *reactive* approaches which act after a specific symptom has been monitored, and *proactive* or *predictive* ones which try to anticipate future system states like the workload of a server.

A related paradigm to cloud computing is *fog computing*. It can be seen as an extension to the cloud, where processing is partly moved to the edge of the network, resulting in a lower latency between applications and the devices at the network edge [17, p. 416]. Although some work differentiates fog and edge computing, we will use these terms interchangeably. There exists a lot of research regarding the management of services and devices in this area. [17] provides a comprehensive overview and a classification of existing research. Many papers are concerned with resource management in this area. Similar to opportunistic networking research, fog nodes are often considered to be highly mobile, resulting in a continuously changing network topology. Devices could stop service execution suddenly due to errors. Therefore service allocation and migration are important management functions. Interesting research handles deployment and requirements matching [18], migrations and related algorithms [5] and resource provisioning using virtualization [19].

2.2 The Distributed Smart Space Orchestration System (DS2OS)

In the following section, the Distributed smart space Orchestration System will be analyzed. Its functionality and design is based on the Virtual State Layer (VSL), a middleware which enables the implementation of smart space services. The main source for this chapter is the dissertation in [1].

2.2.1 The Virtual State Layer

The VSL is a µ-Middleware. This means it provides only basic, non domain-specific functionality and extensibility at runtime through services [1, p. 246]. One major problem with existing middleware is that it often implements domain-specific functionality, which leads to *middleware silos* - this prevents that services running on different middleware can interact with each other [20, p. 3]. However the design of the VSL enables dynamic extensibility at runtime through domain-specific services [1, p. 246], only providing the basis for running these services. The system's architecture will be analyzed in the following.

2.2.1.1 The VSL Architecture

The VSL is a self-organizing unstructured peer-to-peer middleware; each peer is called Knowledge Agent (KA) in this context [20, p. 3]. Figure 2.2 visualizes the VSL architecture. Each KA is running on one network node, all nodes are connected over a Local Area Network (LAN). On top of each KA, different services can be started, which implement domain-specific functionality. A typical case would be a service reading sensor data from peripheral devices connected to the machine it is running on. Depending on measured environmental parameters, the service can execute actions in order to adjust the smart space by sending commands to connected actuators.

The VSL is written in Java, what makes it highly portable since it can be run on different machines. Services are accessed via an API using *Remote Procedure Calls* (RPCs) which provides 13 different operations [1, p. 251]. Thus, language-specific *Service Connectors* can be implemented. The advantage is that only the connectors have to be adapted to the specific language, and developers can freely choose their favorite language [1, p. 251]. A Java connector has already been implemented and will be used for implementation later. The interface is described in subchapter 2.2.1.3.

The VSL separates data from the service logic [20, p. 4]. Data is stored outside the services in the *Context Model Repository* (CMR). This simplifies coupling of services as well as sharing data [1, p. 253]. The CMR contains the service's context models and is designed as part of the global S2Store, which provides service packages. For

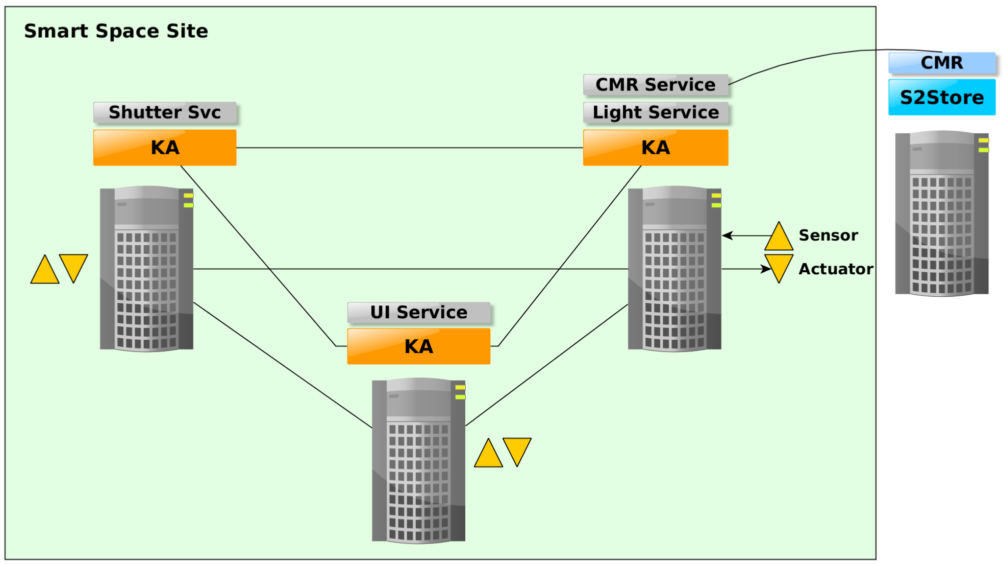


Figure 2.2: The VSL Architecture following [1, p. 249]

each service, exactly one context model is created [20, p. 4]. These context models are based on the *Extensible Markup Language* (XML) and serve as an interface between the services [20, p. 4].

2.2.1.2 Self-Organization Features

The VSL takes care of several tasks autonomously, which facilitates the design of the service management solution. Some useful autonomous features are taken from [1, ch. 6.4] and will be explained in the following.

Service Registration When a service is started, it is registered at one KA by the VSL. This includes that the service's certificate is checked and a service identifier is generated. Also the VSL checks if a context model already exists for that service, and instantiate it if necessary.

Context Handling Each knowledge agent persists context data in a local database. Context is only stored locally at the KA where the service is connected to and is not distributed. KAs are directly contacted when context is requested. The node structures and related access rights are synchronized regularly between all agents.

Virtual Node Handling Virtual context nodes can be registered easily via the API described in the next chapter. When a virtual node is requested, a callback is triggered at the service defining the node. In the background the VSL handles all required context operations transparently.

Agent Discovery Agents send periodic alive pings to each other for discovery. When a new agent is started, an authentication routine is triggered. This includes that the agent's certificate is checked autonomously. With each incoming ping the *time to live* of the agent is updated. If this time is up, the agent is removed from the VSL overlay network.

Self-Management The VSL implements all four self-management properties. These are explained in detail in chapter 2.5. New KAs are automatically added to the VSL overlay, and new services are automatically registered. Therefore the VSL is *self-configuring*. *Self-healing* applies since agents keep the network topology up-to-date and are able to recognize node failures, a network partitioning or a merger of two separate networks. The VSL is *self-optimizing* when removing unreachable context and recording unreachable KAs. *Self-protection* The VSL implements features like access control for context models and encrypts communication between agents.

2.2.1.3 Service Connectors and API

Each service connects to exactly one KA when started. The respective KA instantiates a context model for each service, which act as abstract interfaces for the services [20, p. 4]. This concept separates the service or application logic from the data each service produces. The XML-based context models provide an extensible data typing system. Each service is able to read and write another service's context nodes, unless it doesn't have the permissions needed to perform the operation.

The VSL provides a simple API for context access, and managing access of different services to the system ([1, ch. 5.3]). This Service Interface provides simple operations like *get* and *set* for reading and manipulating context nodes. There are 13 functions for context access, access control and virtual node handling [1, ch. 5.3.1]:

Context Access This includes simple operations like get and set for reading and manipulating data stored in context nodes. Also, subtrees can be locked (and unlocked) for temporary exclusive access. Another useful function are subscriptions, where the subscribing service is notified via callback as soon as the respective node has changed.

Access Control A service can be registered at a specific KA. Other functions allow to add and remove certificates for a service.

Virtual Node Handling Virtual nodes can be registered via the API. In case the node changes, the service is notified via callback.

2.2.2 Site-Local Service Management

One goal of this thesis is to create a smart space service management solution which runs on the site level. The solution design will be based on the DS2OS system which is built upon the VSL middleware. DS2OS is part of the dissertation in [1, ch. 7.2]. First, we will discuss the architectural concept of the system (see fig. 2.3). The DS2OS system comprises three important components: the nodes which are each managed by the *Node-local Service Manager* (NLSM), the *Site-local Service Manager* (SLSM), which manages all nodes in the smart space, and the S2Store which acts as a central service repository - similar to an app store for smartphones. This thesis is concerned mainly with the design of the SLSM. Other theses focus on the NLSM [21] and the distribution of certificates via the S2Store [22]. The specific requirements for the solution based on the SLSM concept are collected in the course of the analysis chapter.

The Run Time Environment (RTE) for executing services on each node is based on Java. On top of the RTE, one can find the *Service Hosting Environment* (SHE), which is responsible for starting and stopping the services. The prototype from the PhD thesis uses an OSGi implementation in order to perform these service lifecycle operations. This choice is analyzed in chapter 2.4 and compared to widely used containerization approaches. The Node-Local Service Manager (NLSM) represents the next level on each node. It is responsible for monitoring services, hardware metrics and errors on its local machine, and reporting the data to the SLSM. The SLSM in turn regularly passes directives to each NLSM, which executes them.

Therefore the interface between the NLSM and the SLSM will be used constantly for managing the smart space. The SLSM is mainly concerned with collecting information about the network and service states and making decisions in order to maximize the dependability of the whole system. The second interface lies between the SLSM and the S2Store. The SLSM downloads service packages from the global store and manages them inside its smart space site. According to the official concept in [1, ch. 7.2.4] it is responsible for a number of tasks, which will be explained hereafter. Several basic requirements can be adopted unchanged from the DS2OS concept.

Service Repository The SLSM must store all service packages which are downloaded from the S2Store **[R3.2]**. This makes sense because a package needs to be downloaded only once and can be deployed to any number of nodes. Therefore the SLSM "acts as site-local repository for DS2OS services" [1, p. 280]. For a simplified handling of services, it makes sense to pack services into one single file, because they must be transferred between different components **[R3.1]**. These packages include all application code and dependencies as well as potential meta data about the service.

Service Deployment One of the most common tasks of the SLSM is installing services in the smart space. This comprises multiple requirements which have to be

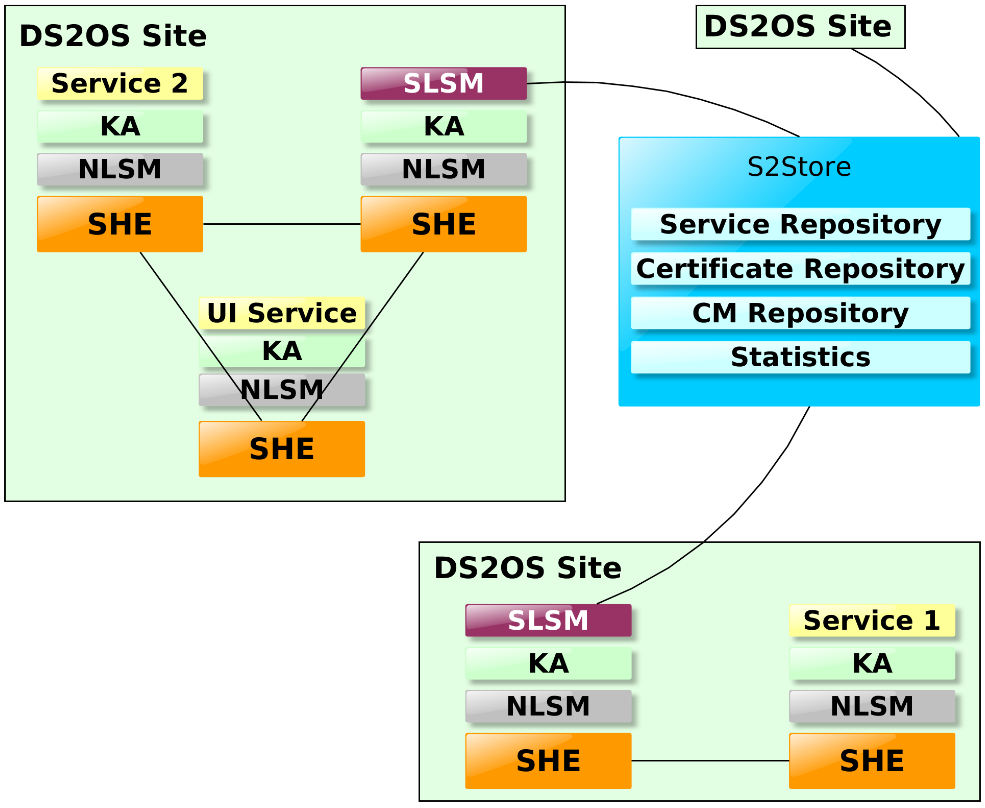


Figure 2.3: The DS2OS Architecture following [1, p. 277]

fulfilled. First, the SLSM must be able to communicate with the S2Store in order to request a list of available services and related meta data as well as to download and store the services inside the smart space when requested by the user **[R4.1]**. Since it is unclear in which form services will be handled, it should be possible to transfer any type of binary data between the two components. If the solution will be based on OSGi, the services could be in the form of jar packages, for example.

The next step for installing a service requires deploying it to an available node in the VSL network **[R3.3]**. This comprises sending the service binary to the designated node, as well as directives which tell the NLSM what to do with it. For example, whether to run the service package instantly or just to store it for later use.

Service Migration and Replication The VSL separates the service logic from its state, which simplifies the migration of services - this enables a strong mobility of services [1, p. 281]. That means that services can "dynamically be moved to other computing hosts" [1, p. 281]. Obviously several situations could require such a migration routine, e.g. if a node running a service fails due to some reason.

Therefore the SLSM must be able to migrate services from one node to another **[R3.4]**. The decision on which service to move to which node is part of the SLSM's self-management capabilities discussed in chapter 2.5.

A similar functionality is service replication, where a service's context is synchronized regularly between two service copies. In case the original service fails, the second one can be started by the NLSM containing the surrogate service [1, p. 282]. The SLSM should feature a replication mechanism because it helps increasing the dependability of services [1, p. 282] **[R3.5]**.

Service and DS2OS Updates As effort for smart space owners must be minimized, service updates should be checked and installed automatically when available **[R3.6]**. The service package is downloaded from the S2Store and sent to the node where the service is currently installed or running. The NLSM tries to start the new service; in case of an error the SLSM is informed [1, p. 283].

An automated update mechanism is important not only for the services, but also for all DS2OS components. This contributes to a good usability, and improves the security of the system [1, p. 283]. Since the SLSM and NLSM will be implemented as services themselves, they can be updated like every other service. In any case, service interruption should be avoided or at least minimized as far as possible.

Statistics Collection smart space optimization can only be performed if the SLSM has certain information about the system. This includes knowledge of available nodes and services with their lifecycle state as well as performance data, for example. Besides being input for optimization algorithms, data is prepared and sent to the S2Store, in order to provide feedback for developers [1, p. 283]. Therefore SLSM and S2Store must be able to exchange data bidirectionally - for sending feedback data to the store and for sending service meta data to the service manager **[R4.2]**. Research around monitoring in distributed systems and derived requirements are analyzed in chapter 2.6.

Optimization Strategies If multiple nodes are available for deployment, the SLSM needs to make a decision regarding where to deploy the service. This node selection requires a specific strategy; according to [1, p. 284], the SLSM collects performance data from the nodes, and selects the node with the most free resources. This represents a classic load balancing algorithm. For experienced users, it could be beneficial to have control over the node selection. However an unexperienced, non-expert user is considered as the typical smart space user in this thesis. Services can have complex requirements regarding available sensors or dependencies to other services, for example. Checking manually if requirements are met can be a complex task and should be executed autonomously. This can be achieved through autonomic computing or self-management approaches, which will be analyzed in chapter 2.5. Other tasks like migration and replication could also be executed autonomously in order to improve system dependability and user experience.

Security The system design includes securing the microservices with X.509 certificates. The idea is to extend the SLSM by autonomous certificate management functionality. This topic is covered in Donini et al. [22] and will not be discussed further in this work.

2.3 Service Level Agreements and Dependability

Research in Service-oriented Computing shows that service quality is a crucial issue. Especially when services are managed autonomously, there is a need to measure the execution quality of the services. Adellina et al. [23] conducted an interesting review around the term of dependability. The result involves a list of criteria which are "expected from a dependable system" [23, p. 3]:

- Availability: The system must be available on request.
- Reliability: The system must complete running tasks without failures.
- Safety: The system execution must not lead to danger.
- Integrity: The system state cannot be altered unauthorized.
- Maintainability: The system can be maintained with reasonable effort.

Burkert et al. [6] refer to the norm EN 50126 defining the abbreviation RAMS as reliability, availability, maintainability and safety. It defines availability as time intervals where no failures occur. Availability is defined as the ratio of time where the system is available and total time. The paper defines the time needed to repair a system as a measure for maintainability. Examples where safety is crucial are fire alarms and evacuation systems [6]. These criteria are high-level definitions of the goals which should be achieved by a dependable system. In order to define dependability on a more specific and technical level, especially between customers and service providers, service level agreements (SLAs) have established [24].

In the Smart Space scenario the system should be able to manage services autonomously. Therefore it depends on input regarding which dependability targets to achieve by executing optimization algorithms. For example, service developers could be constrained to categorize each service regarding the expected availability. A service controlling an alarm system should have a higher availability than services regulating the color of the lighting. It is obvious that a service management system needs to be able to understand such requirements and to align its strategy to meeting those:

> **[R5.5]** A service management solution should be able to interpret dependability requirements and to align its optimization strategies to meeting them.

2.4 Service Lifecycle Management

2.4.1 Overview

One important requirement to service management is that the system must be able to manage the lifecycle of services. This lifecycle includes installing services, starting and stopping as well as uninstalling them. Research shows there are two prevalent approaches for managing the lifecycle of microservices, which differ substantially. The first approach uses an implementation of the OSGi specification - formerly known as *Open Services Gateway initiative*. Applications of OSGi in the field of service management can be found in [25] and [26]. The latter makes use of an OSGi-based service registry, where service bundles are registered and can be discovered.

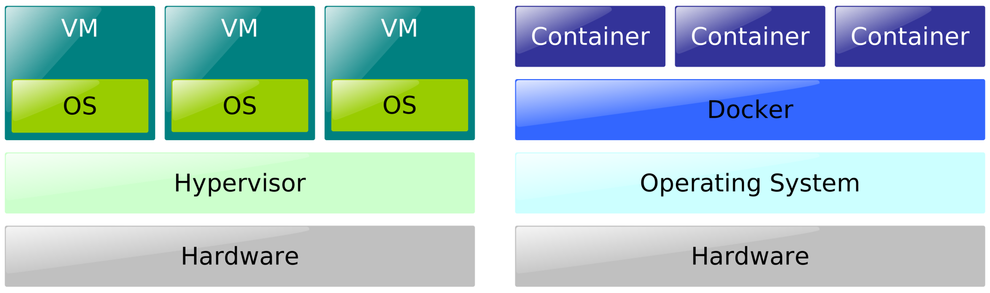
The deployment of software often includes packaging the application code and the dependencies into a single file which can be distributed to different computers easily. This is considered as a requirement for this research to enable an easy handling **[R3.1]**. An example are Android apps, which are bundled into one .apk file, which contains everything needed to run the app. OSGi instead uses jar files which contain a manifest file for storing service meta data. Storing service related data inside the package makes sense since data cannot get lost and is always available in place when it is needed. So next to storing all dependencies in one single file, storing meta data inside it is another important requirement for handling services **[R2.1]**.

The second approach to enable service management are Docker containers. It is a more lightweight alternative to established virtual machines and enables application in microservice environments and on constrained devices. Instead of having jar files which are managed by a language-specific framework, Docker allows free choice of programming language for each service, since the execution environment can be shipped inside each container. For example, one could deliver a service implemented in Java on top of a Java Runtime Environment in a specific version - all in one container.

As defined in the DS2OS concept in [1], the services are bundled using an OSGi implementation. Hereafter both approaches will be analyzed, presenting some relevant work. In the end a conclusion will be provided as a basis for the concrete solution design in chapter 4.

2.4.2 Docker

Docker is a virtualization software which allows to package applications or services into containers. Compared to existing virtualization techniques, there is no need to virtualize a complete operating system, because Docker containers are running on the host system directly. Figure 2.4 illustrates the difference between the two. VMs depend on a hypervisor, which manages the physical computing resources and simulates a separate hardware for each Virtual Machine. Each VM then has to run its own operating system on top of it. In contrast, Docker containers do not need to include an own operating system, since they run directly on the host OS kernel. This makes it a lightweight solution and therefore suitable for use cases where we want to encapsulate microservices. Even though containers run on the same kernel, the access of each container is limited [27, p. 27]. Smart spaces usually contain constrained devices with low computing power. There are several papers which suggest using Docker virtualization for service management, e.g. [28], [29] and [30].



(a) Virtual Machine Architecture (b) Container Architecture

Figure 2.4: Comparison of Virtual Machines and Containers following [2]

Prerequisite for running Docker containers is a machine running the Docker Engine, which manages the allocation of the available resources to the containers, using Linux kernel features like cgroups and namespaces. These enable isolation of processes and host filesystem [27]. Nevertheless, Docker Containers cannot access each other unless they are configured that way. One has to explicitly define port mappings or shared volumes to open containers to their environment.

Figure 2.5 shows the lifecycle of a Docker container. All information can be taken from the official documentation in [31]. The lifecycle starts with a *Dockerfile*, which defines the content and configuration of each container. This file can then be built, resulting in a Docker *image*. Images can be seen as a blueprint for Containers, similar to a Java Object which is generated from a Java Class. They can be distributed to different machines directly or via a Docker *registry*.

This registry acts as a central image repository where images can be *pulled* from and *pushed* to. An official public registry can be found under [32], containing many predefined images with open source software like Apache Tomcat. The registry can otherwise be downloaded and deployed in a local environment. Once an image is available on a client, any number of containers can be instantiated from it. These containers can then be controlled via simple commands, for example *start* and *stop*. Although containers can be compared to Java objects, they differ in keeping their state when being stopped.

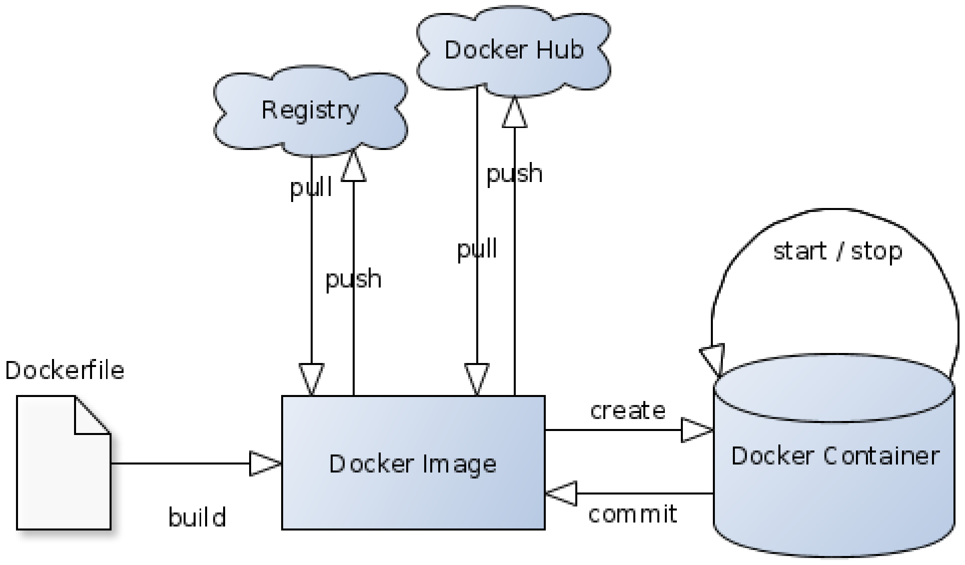


Figure 2.5: Docker Container Lifecycle

When a container changes over the time being executed, for example because it produced some data it could be interesting to have a snapshot of that exact state. This is possible by *committing* the container, which produces an image from it. This image can then again be distibuted and instantiated into any number of exact copies of the original container.

If we think about introducing Docker into a smart space environment, where we consider running lots of microservices on a number of constrained devices, there is a need to examine the performance of this approach. As our literature search shows, there are several papers concerned with evaluating the performance of Docker containers. [33] compared the performance of Linux Containers (LXC) to the virtualization software Xen in the field of High-performance Computing. As Docker is based on LXC, this paper can be seen relevant for this survey. As their results show, Linux Containers are faster than Xen for CPU, memory and disk performance, and even similar to native execution without virtualization.

In [34], the performance of Docker executed on a Raspberry Pi 2 using Hypriot [35] - a Docker distribution running on ARM processors - is analyzed. The paper provides a comparison of executing benchmark tests once native and once using Docker virtualization. The results show that the differences in performance are minor. For example, the CPU performance differs by only 2.67 percent [34, p. 1]. This shows the relevance of Docker to the Internet of Things, because it is possible to run containers on constrained hardware like the Raspberry Pi.

The paper [36] presents an extension to the Hypriot project, the *Hypriot Cluster Lab*. It offers cloud functionality similar to platforms like Kubernetes, while running on constrained devices [36, p. 1]. The paper demonstrates two interesting use cases. First, an overlay network is created, in order to access a web server running on one node from another node. The second use case shows that a load balancer can be instantiated, which distributes requests to other nodes in the cluster [36, p. 2].

Besides all performance-related work, [28] analyzes some interesting functional advantages of using containerization in microservice architectures:

1. The software and all its dependencies are packed into a single image, which is highly portable.
2. Docker simplifies service deployment. In case of errors, rollbacks can be easily performed.
3. Containers are more efficient compared to virtual machines and enable good scalability of cloud infrastructure services.

The use of containerization for smart space service management could definitely make sense under certain circumstances. The advantage of having images which contain all meta data, application code, dependencies and even the runtime environment clearly has some advantages like freedom regarding programming languages. Also, the risk that required software is not installed (e.g. minimum Java version) one the target machine is minimized since the approach only depends on a working Docker engine. In addition Docker provides basic lifecycle management functions and an open source registry implementation. However, depending on the runtime environment needed to execute the service each images become bulky and performance suffers, especially if multiple Java Runtime Environments are executed simultaneously on a node. In smart space scenarios this can become a real problem since everything is running on constrained devices.

2.4.3 OSGi

The Open Service Gateway initiative (OSGi) specifies a platform to support modularization of Java applications. It provides a service registry for managing the lifecycle of services. These services are called bundles and are simply jar files with a special manifest file [21, chp. 2.2.1]. Bundles and their lifecycle can be managed by the framework during runtime. They can be installed, started and stopped for example. According to [21, chp. 2.2.1], OSGi provides various advantages which are summarized in the following:

Modularization OSGi enables better encapsulation of functionality than standard Java possibility of restricting access through private and protected modifiers.

Versioning OSGi features semantic versioning which enables co-existence of different versions of a bundle.

Dynamic Updates Services can be installed and updated during runtime without the need to restart the whole system.

Size OSGi has a small overhead.

More information about the suitability of OSGi for node-local service management can be found in [21]. The decision is not relevant to the SLSM - the choice on how to manage the service lifecycle mainly affects the SHE and NLSM.

2.5 Self-Management

This section presents a summary of self-management or autonomic computing basics. In this thesis, these two terms will be used interchangeably. Concrete related work will be presented in detail in chapter 3.

2.5.1 Introduction

The foundation for autonomic computing has been created in 2001, when IBM proposed a reference architecture for autonomic computing [37, p. 25]. IBM has anticipated a tremendous increase in the complexity of software systems, especially in distributed systems [38, p. 41]. One can easily comprehend this complexity from today's perspective, where many tasks are automated because the manual effort would be far too high. Considering the Internet of Things, where vast amounts of small computing devices are connected and are expected to work together in order to ensure a high quality of service, the IBM reference model has become a widely used standard for autonomic computing.

According to [38], IBM stated four aspects of self-management, which are part of the autonomic computing vision. First, *Self-configuration* aims at systems that can configure themselves. The only input they need are high-level business objectives, that "specify what is desired, not how it is to be accomplished" [38, p. 43]. *Self-optimization* implies that the system regularly checks if there is optimization potential. For example, if a load balancing algorithm identifies a high load on one machine, it could migrate one running application to another machine. *Self-healing* refers to a system's ability to detect failures and handle them autonomously. Considering the smart space scenario, one would not expect a non-expert user to browse log files in order to find what caused the error and to fix it by changing complex configuration or even the application's source code. *Self-protection* implies that the system anticipates possible problems and tries to avoid them [38, p. 43].

The concept of autonomous computing is highly relevant for this thesis, since smart spaces can be large and complex systems, running a variety of services which require continuous configuration and optimization. Since the users are considered non-experts, these tasks must be automated wherever possible - although it will not be possible to handle every single situation autonomously.

2.5.2 The IBM Reference Model

The term autonomic comes from the autonomic nervous system, which monitors and controls important body parameters without human effort. This reactive behavior was transferred to computer science as the *MAPE-K* control loop, which stands for monitor, analyze, plan, execute and knowledge. It is executed by *autonomic managers* in order to control any type of resource in the system.

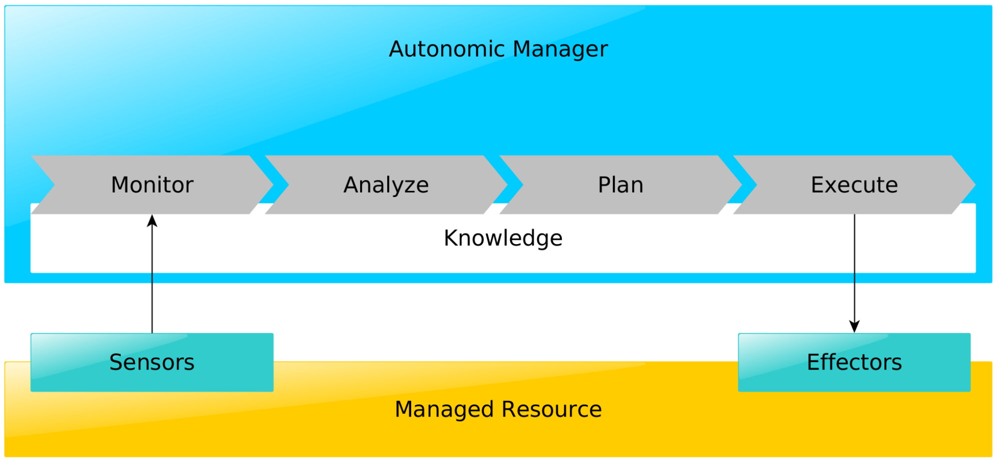


Figure 2.6: The MAPE-K loop, based on [3, p. 7:6]

The loop starts with the manager *monitoring* a specific resource, for example a database, a web service or a hardware device. Typical parameters could be the CPU load or the reaction time of a software component. Next the collected data is *analyzed* in order to learn and derive decisions on what to do. The concrete actions which will be executed, are selected in the *planning* step. In the end, the planned actions are *executed*. The term *knowledge* stands for a knowledge source, which is used by the managers. It can be a registry, database or repository storing information like policies, which will be applied by the managers [39, p. 12]. According to [3, p. 7:6], autonomous systems can ideally be configured by entering high-level objectives - in most cases, these are event-condition-action (ECA) or utility function policies. The paper states that conflicts can occur between two or more ECA rules. In such a case, the system will not know how behave unless there is another rule handling the specific conflict. According to the paper, utility functions can be a solution, since they define a "quantitative level of desirability".

2.5.3 Self-Management in Practice

Self-management features are applied in many different research areas. This chapter will present categorizations of approaches and algorithms in these areas. Table 2.1 summarizes the findings. Aliu et al. [12] provide a classification of algorithms utilized in SON research. Among learning algorithms, Bayesian networks (BN) represent a popular approach. In cloud computing, [40] uses BN in order to manage resources. They propose a decision making module which is able to predict future system states. Jules et al. [41] present a framework for choosing a reliable cloud provider and meeting SLA requirements. Their model is based on a Bayesian inference engine which computes the probabilities of violating SLAs for each provider. An example for mathematical optimization can be found in [42]. The paper proposes an *Integer Linear Programming* (ILP) approach for cloud application placement consisting of multiple optimization objectives like minimizing number of migrations and number of computation nodes. Kientopf et al. [43] propose using the Dijkstra algorithm combined with some performance metrics like hop count for making service migration decisions in fog networks. Arabnejad et al. [44] differentiates four categories for auto-scaling in clouds. In the threshold-based approach, rules and conditions are defined by cloud providers or users over different load metrics. Control theory is concerned with aligning the output of the system with defined goals, e.g. SLAs. Time series analysis uses prediction models which operate on historical data. The paper mainly deals with reinforcement learning (RL), where an agent tries to maximize its reward by executing specific actions. RL does not need prior knowledge and is able to learn from observing the environment [44, p. 66]. Khazaei et al. [45] propose using a function model which incorporates CPU, memory and network usage in order to scale services based on the load. The function incorporates adjustable parameters which assign a weight to each load variable.

Area	Paper	Classification
Self-organizing Networks	Aliu et al. [12]	Supervised and unsupervised learning, mathematical optimization, stochastic optimization
Cloud Computing	Arabnejad et al. [44]	Threshold-based rules, control theory, time series analysis, reinforcement learning
	Lu et al. [16]	Predictive vs. reactive approaches
Fog Computing	Mouradian et al. [17]	Exact algorithms, graph-based algorithms, heuristics, policies, schemes

Table 2.1: Algorithm Classification According to Different Surveys

Lu et al. [16] analyze approaches to cloud resource management and divide them into reactive and proactive ones. They state that most predictive algorithms use time series data in order to make predictions. Anticipating the future state of a system can be quite unreliable in some cases, because not all events are predictable. Because of that, reactive approaches are also proposed in research [16, p. 34]. For interested readers, Mouradian et al. [17] provide a very comprehensive survey of fog computing algorithms. The paper classifies algorithm types over different applications like resource sharing and offloading.

One can see that there is a variety of specific algorithms for different domains and applications. An idea to implement a variety of algorithms is proposed by Velasquez et al. [5]. The paper presents a service migration solution for fog computing. Their service orchestrator module is able to apply different strategies depending on the system's needs which are defined by the administrator [5, p. 108]. We will adopt this idea of exchangeable algorithms as the next solution requirement **[R5.1]** and refer to this concept as *Optimization as a Service* (OaaS). In DS2OS, the SLSM as the central managing component should have control over all available algorithms and activates or disables them. They could be handled like any other service package and provided via the S2Store. This clearly brings some advantages. The developer community can contribute not only by providing smart space services, but also by uploading and enhancing management and optimization algorithms. Also, algorithms perform differently under varying service compositions, node conditions etc. - so switching between them can be beneficial under certain circumstances. This process could even be automated, for example by defining rules. In this case the SLSM could be enabled to react to environmental changes by switching its strategy.

As the SLSM is a standard VSL service, it can also fail due to some reason. If this happens, there is no optimization and service management. Because of that, it makes sense to have a number of self-repair features for the SLSM implemented. The VSL autonomously recognizes when peers enter and leave the network. Since the SLSM regularly updates its network representation, the same applies if the leaving node is not the one running the SLSM service. If the SLSM node is leaving, this can have two reasons. Either there is a network split since one or more communication channels are unavailable, or the single node failed due to some internal error. In the first case, the network transformed into two P2P networks where the node running the SLSM will remain in one of the two networks depending on the location of the split. In both cases, a new SLSM instance must be started immediately to ensure that service management interruption time is minimized **[R5.2]**. If the network split is temporary, one must expect that the two subnetworks merge back after a time. As soon as this happens, there would be two SLSM instances running in one network. This must be handled since two leading components could impede each other without recognizing. The solution would be to kill one running SLSM in that case to ensure there is only one instance running **[R5.3]**. These actions could be executed by one of the NLSMs, for example.

An interesting self-managing solution from the fog computing domain is *Fogernetes* [18]. Since fog computing devices are considered heterogeneous (like in smart spaces), services must be deployed to nodes which meet their hardware requirements. The paper proposes a mapping between node capabilities and service requirements by using labels. These are controlled during service deployment in order to meet the service's requirements. In DS2OS, we propose to enrich service packages with meta data **[R2.1]**.

This data should contain hardware requirements, which are compared to each node's hardware capabilities **[R5.4]**. This could be implemented as a pre-filtering mechanism for the exchangeable OaaS which ensures that only nodes that meet the minimum requirements can be chosen by the algorithm. Other nodes are eliminated, in the worst case leading to services not being deployed at all. Labeling the packages could also used for matching service dependencies. This is the case if stand-alone services are composed in order to build a more complex application. For example, there could be numerous services which just need to turn on a green LED in the smart space as a signal that an operation was successful. Each of these would have a dependency on the LED service in that case - if the latter is not running at the same time, the functionality of the composed application is not complete. There are numerous other options for integrating self-managing behavior into the SLSM, but due to limited time we picked the most interesting features from our view.

2.6 Monitoring and Data Infrastructure

This thesis focuses on creating a basic service management infrastructure which enables running services as well as exchangeable management strategies on top of it. Running microservices in general results in a continuously changing architecture; therefore information must be created and kept up to date automatically [46, p. 66]. Comprehensive monitoring must be enabled, which regularly collects data from the distributed nodes. Since there is a variety of parameters which could be measured, the goal of this chapter is to identify which data must be collected. Basically, in a DS2OS smart space two components must be monitored. First, each node must be monitored since the SLSM needs to know if they are still alive **[R1.3]**. Also hardware parameters like CPU load are interesting for load balancing. Second, the services running on each node must be controlled regularly regarding their lifecycle status **[R1.2]**.

Interesting research on monitoring can be found in the area of cloud computing. A main goal there is to manage resources efficiently in order to decrease operational costs and to meet service level objectives or to decrease operational costs or improve energy efficiency. One interesting paper about monitoring cloud environments is [47]. Their approach is based on adaptive monitoring, which means that monitoring can be dynamically adjusted to current requirements while at the same time, certain levels of QoS must be guaranteed. The paper shows that monitoring can have significant impact on the whole system performance, making it necessary to enable configuration at runtime. In smart spaces, mainly constrained devices will be installed due to their low energy consumption. This requires that service management does not introduce a high overhead since this could completely compensate the performance improvements achieved by service management itself. The suggested solution from [47] is based on a publish-subscribe mechanism, where consumers can subscribe to different channels. The monitoring component pushes monitored data into each channel. Using higher-level policies, different monitoring parameters can be adjusted. This comprises *what* should be monitored *in which detail*, *how to deliver and process* measured data, and *how to control* monitoring systems [47, p. 127]. This adaptive monitoring will be incorporated as next requirement for the service management solution:

[R1.1] Adaptive monitoring must be implemented to minimize overhead.

Therefore it is not required to define a set of fixed monitoring parameters in the beginning. Instead, optimization services inform the SLSM about required monitoring data. The SLSM then asks each NLSM to monitor exactly these required parameters, collects the data and passes it to the currently running algorithm.

As already mentioned, it is beneficial that feedback data is collected and sent to the S2Store. This enables developers to find bugs and improve the stability of their services.

[R4.2] The SLSM must be able to generate statistics from monitored data and send it to the S2Store on a regular basis.

This data should be collected over a period of time and regularly transmitted to the store. The SLSM is therefore required to store the collected data in a structured and efficient way. We propose that all collected data should be stored inside a site-local database **[R2.2]**. This brings several advantages. Optimization services which apply time series analysis could request historical data as input for algorithms. As monitoring will produce a lot of data over time, a database represents an efficient way for data storage and access. Also, statistical feedback information for the S2Store can be better extracted, filtered and manipulated from a database than from log files, for example. There are a lot of possibilities for the specific design of the monitoring solution. A database type must be selected and the communication mechanisms for collecting the data must be discussed. See chapter 4 for design details.

2.7 Boundaries and Research Questions

In the following section the research questions are derived from the collected requirements. Before that, the focus of the thesis and its boundaries are defined.

2.7.1 Focus and Boundaries

This thesis focuses on service management on the smart space site level. The corresponding functionality shall be realized by the Site-Local Service Management component (SLSM) according to the official concept in [1]. There are interfaces to other components in the DS2OS system architecture, that need to be defined. This is important because other components rely on the SLSM's functionality, and vice versa.

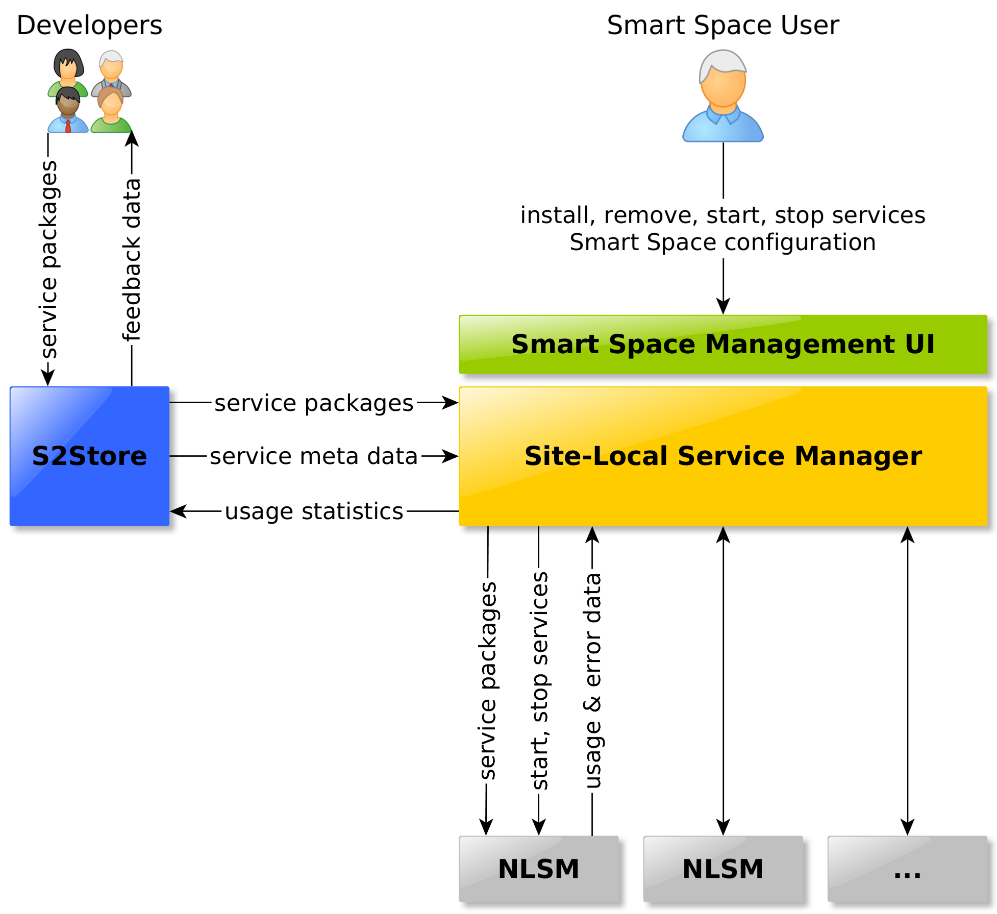


Figure 2.7: The SLSM in the DS2OS Context

Figure 2.7 visualizes the interfaces on a conceptual level. The first interface lies between the SLSM and the S2Store, which is like an App Store for service packages. In one direction, the SLSM depends on the S2Store in terms of providing the service packages in any form. Also, it will need to access meta data about the packages - for example to provide a list of available services to the user. In the opposite direction, the S2Store needs to receive usage data from the SLSM regularly. This data will be forwarded to the developers as a feedback ([1, p. 287]).

Another interface is located between the SLSM and the NLSM. Service packages must be sent to the NLSM, which has to be able to execute them. In return, usage data is transmitted on a regular basis back to the SLSM. This data is necessary for the automated management capabilities of the SLSM and as feedback for the developers. The functionality to start, stop and pause packages will not be part of this thesis. We assume it is possible to trigger these actions by sending simple commands to the respective NLSM.

In this thesis, a solution for the SLSM will be designed and implemented as a prototype. The other components and their inner functionality will not be part of the designed solution. The inner design of the NLSM is targeted in another current thesis [21]. For evaluation purposes, a simple prototype of the S2Store will be implemented. Further features regarding app store functionality will be future work. Security mechanisms concerning the whole lifecycle of services from the S2Store to deployment on nodes is targeted in [22].

2.7.2 Research Questions and Requirements

This chapter will provide a classification of the identified requirements. These will be mapped to higher level research questions in order to gain a comprehensive overview over the whole topic. Figure 2.8 summarizes five central aspects derived from the requirements. The first class of requirements concerns the data infrastructure of the solution. We identified the need to label services with meta data, for example to provide static information like the service version. This kind of meta data must be bundled inside each service package, in order to provide an easy handling and to prevent that data gets lost somehow. All data that is monitored by the SLSM should be stored inside a database. On top of that, a simple API must be provided so that strategies can easily access monitored information. This is a prerequisite for autonomous service management, since all algorithms require data; some even need historical data for learning.

The second class involves monitoring the distributed system. Since DS2OS is based on a P2P network, nodes must be monitored continuously in order to know if they are still alive or overloaded, for example. Also it is important to monitor the lifecycle state of services. Without these information, the optimization strategies running on top of the SLSM cannot work properly. The execution of management decisions requires that basic deployment routines are available. This forms the third class of requirements. These

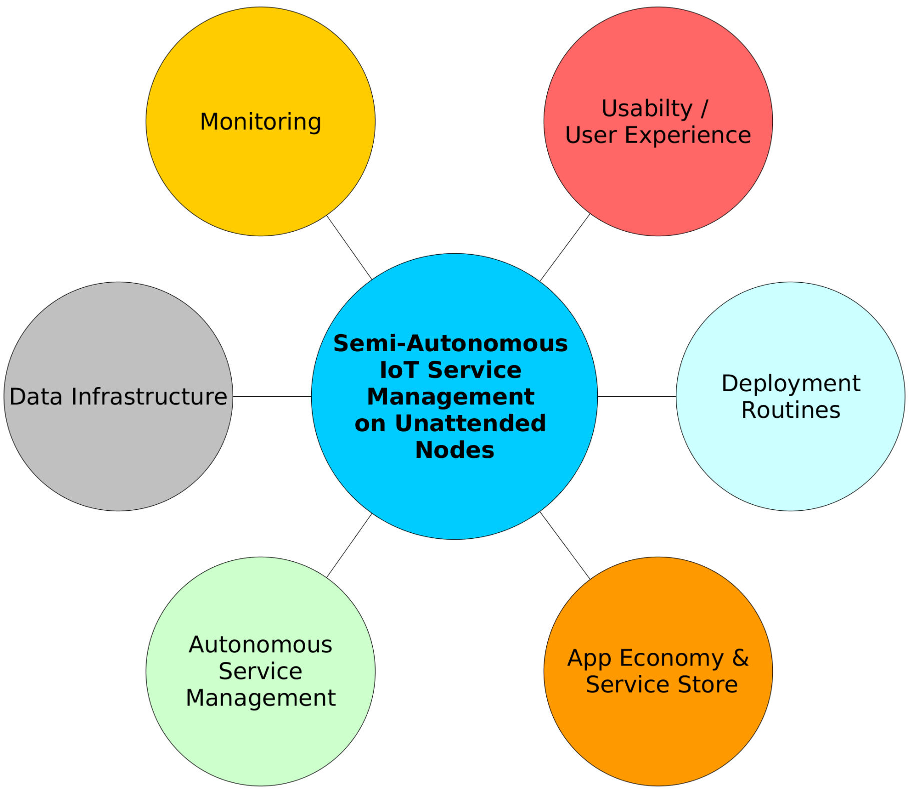

Monitoring
Usabilty / User Experience
Data Infrastructure
Semi-Autonomous IoT Service Management on Unattended Nodes
Deployment Routines
Autonomous Service Management
App Economy & Service Store

Figure 2.8: Requirements classification

comprise deployment, migration, replication and service updates. The intended solution must be able to communicate with the S2Store. This store can be compared to an app store on smartphones since it is a global, stand-alone component providing packaged services in an easy way. The deployment of services requires bidirectional data and binary exchange, since the SLSM needs to download service packages and meta data (e.g. a list of services available). In the other direction, usage statistics should be reported back to the S2Store as a feedback for service developers. Autonomous service management forms the fifth class of requirements. This comprises self-repair features of the SLSM itself, as well as extensibility regarding optimization strategies. Also, the matching of service dependencies and requirements is classified here. A superior non-functional requirement is usability of the solution. We assume smart space users are non-experts which expect high service dependability, quick and easy service installations and no or minimal configuration effort. Based on the classification, the following five research questions can be formulated. In table 2.2, each requirement is directly mapped to one of the research questions in order to provide a clear structure.

[Q1] Which entities, parameters and metrics in a smart space should be monitored to which extent by a service management solution?	**[R1.1]** Adaptive monitoring must be implemented to minimize overhead. **[R1.2]** Services and their lifecycle state must be monitored. **[R1.3]** Nodes must be monitored regarding their alive status, available hardware and performance metrics.
[Q2] How and where should meta data and monitoring data be stored and used by a service management solution?	**[R2.1]** Service packages must be shipped with service meta data. **[R2.2]** Collected monitoring data should be stored in a site-local database for efficient and secure access.
[Q3] Which functional features regarding deployment are must-have functionality for a service management solution?	**[R3.1]** Services must be packed into one single file for easy handling. **[R3.2]** The solution must hold a service repository. **[R3.3]** The solution must be able to deploy a service to a specific node. **[R3.4]** The solution must be able to migrate a service from one node to another. **[R3.5]** The solution must be able to replicate services. **[R3.6]** The solution must be able to update services and system components.
[Q4] How can services be easily deployed from a global service store to a smart space and which data should be provided as feedback for service developers?	**[R4.1]** The solution must be able to download service packages from the S2Store to the smart space. **[R4.2]** The solution must be able to exchange service meta data with the S2Store bidirectionally.
[Q5] Which features regarding self-management are important for the solution design?	**[R5.1]** The solution must be extensible regarding optimization strategies. **[R5.2]** A new SLSM instance must be spawned in case of the node running it leaves the P2P network. **[R5.3]** If multiple P2P networks merge via the VSL, only one SLSM instance may remain running. **[R5.4]** The solution must be able to fulfill service dependencies and match hardware requirements with node capabilities in order to meet each service's conditions. **[R5.5]** The solution must be able to interpret SLAs stated by developers and smart space users. **[R5.6]** The solution must offer a good usability and dependability, while minimizing effort for smart space users.

Table 2.2: Summary of all Requirements

Chapter 3

Related Work

This chapter will give an overview about relevant related work. The papers presented in this section address related problems and propose approaches that are considered relevant to this thesis.

3.1 Related Research

Cao et al. [4] propose a design for an operating system for the Internet of Everything called *EdgeOS*. One focus of the paper is managing data produced by heterogeneous devices. The design consists of different layers. The communication layer is responsible for forwarding commands to devices and to collect data from them. Data is processed on the data management and self-management layer by an event hub and stored in a database. The data is used by a self-learning engine to create a user model based on usage behavior in order to provide better user experience [4, p. 1759]. Also, a comprehensive API for service development and registration is proposed. Context data storage, service registration and a service API are already made available by the VSL. Interesting insight for SLSM design can be taken from self-management features of EdgeOS. The paper differentiates two phases of device monitoring. The first one is concerned with checking if a device is alive at all. The second phase checks the status of alive devices to see if there are other execution problems. When a device fails the proposed system suspends all services executed by the device [4, p. 1760]. EdgeOS has the capability to restore configuration of the failed device to a replacement device. Occuring conflicts between services are solved by assigning priorities to each service. This mechanism could be useful in DS2OS, if a service A wants to turn on the lights and a second service B wants to do the opposite, for example. Via meta data, priorities could be added to services and utilized for decision making by exchangeable strategies. Autonomy in EdgeOS is based on self-learning from monitored data about user behavior. Exchangeability of algorithms enables applying such machine learning algorithms in DS2OS. This requires efficient storage and access of time series data in a database. EdgeOS focuses on smart home applications with non-expert users. However, the paper states that user experience is still an open issue regarding simple installation, configuration and good user interfaces [4, p. 1762]. The solution manages devices and services autonomously through machine learning algorithm.

(+)	(-)
Centralized solution	Open usability issues
Device and service monitoring	No app store / app concept
Self-learning	
Efficient data storage	

Table 3.1: EdgeOS [4]

Velasquez et al. [5] propose a service placement solution designed for fog computing. The goal is to reduce service latency by migrating services to optimal locations. The proposed architecture consists of three main modules. The first module is a service repository. It stores all services as well as related meta data and is located centralized in the cloud. The second module is concerned with monitoring the fog network. The solution is based on the *Application-Layer Traffic Optimization* (ALTO) protocol which enables creation of an overlay network. It provides a *network map* and a *cost map* describing the topology of the network as well as different networking metrics. The third module is the *service orchestrator* which implements different strategies that use monitoring data and service meta data in order to make decisions regarding service placement [5, p. 108]. The proposed architecture and the SLSM have similar features. Both are centralized solutions which target management of services on distributed and unattended nodes. Exchangeable algorithms for optimizing service quality are executed during system runtime. These make decisions based on monitored data reflecting the network state as well as restrictions defined in service meta data, for example resource requirements or dependencies to other services.

(+)	(-)
Centralized solution	Only network distance metrics covered
Network monitoring	Administration tasks done by system experts
Processing of service requirements	No app store / app concept
Exchangeable strategies	

Table 3.2: Velasquez et al. [5]

However, the paper targets multi-hop networks where mainly topological metrics are interesting for optimization. The paper states that algorithms are installed through administrators [5, p. 108]. In DS2OS, there are no system experts to perform such adjustments to the system. Therefore it is crucial that strategies are easily exchangeable as part of the app concept.

Several papers suggest a rule-based approach to realize a MAPE control loop [48], [6]. The latter targets autonomic self-management in IoT scenarios as well as meeting dependability requirements for building automation systems including heating, ventilation and alarm systems among others. Like in the smart space problem domain, one target is to enable heterogeneous devices to work together through services. The main focus of the paper lies on self-management capabilities of the management system with the main goal to "get human out of the management loop" ([6, p. 2]). It provides several interesting concepts which could be applied to the smart space scenario as well:

Dynamic service associations These allow to change associations between services at runtime. A service association in DS2OS could be a relation of type *depends on*, for example. If a service dependency fails, a service copy could be started on another node, providing the same functionality to the consuming service. This exchange needs to be done at runtime, requiring the management system to dynamically change the association between two services. In DS2OS, service requirements and dependencies could be defined in the service meta data inside packages, which can be interpreted by the SLSM **[R5.4]**. The SLSM itself or the active optimization strategy can then use this information in order to satisfy these conditions. If those requirements cannot be fulfilled, appropriate actions could be initiated like starting a service dependency autonomously.

Data model The paper proposes an "*implementation and protocol-independent definition of management data*" and that data must be accessible in a "*homogeneous and well-defined manner*" [6, p. 5]. There is a need to handle services equally in order to enable an efficient and partly autonomous management. Homogeneous service meta data is a basic requirement for this. Exchangeability of different strategies requires that they can access monitoring data homogeneously via a well-defined interface. In our design a database will be used for efficient data access **[R2.2]**. The specific data model will be discussed in chapter 4.

Management tree Data is exchanged between different management agents via a hierarchical tree structure. This provides an overview over the whole system's state. It contains configuration variables, which can be changed by the management system to change a component's behavior. In DS2OS, we can use VSL context nodes that represent a similar concept. The VSL provides the possibility to construct hierarchical structures via XML, as well as CRUD (create, read, update, delete) operations. VSL agents can subscribe to specific context nodes in order to receive notifications over changes and adapt their behavior.

Rule-based runtime management The paper proposes a rule engine, which is able to execute rules based on changes in the management tree. On the one hand, rules could be a good solution as long as there is a comprehensive set of rules for most situations. There could be problems if the set of rules is insufficient to handle common error situations. A second issue is that such a solution requires input - either the rules are directly entered into the system or the specific rules are automatically derived from higher-level specifications. This is the case in [6], which depends on the input of requirements entered by system experts. In the case of smart spaces, one cannot expect end users to create sophisticated input in the form of requirements or specific rules. However a rule engine could be added to the SLSM as a service in order to optimize the smart space via rules that are created by the user **[R5.1]**.

(+)	(-)
Tree-based data structure Service monitoring Processing of service dependencies Extensibility through rules OSGi-based service management	Management rules created by system experts No app store / app concept

Table 3.3: Burkert et al. [6]

Dai et al. [48] provide another interesting approach for autonomous service management in industrial control systems. The central component is the *Autonomic Service Manager* (ASM) which communicates with a number of external services responsible for tasks like monitoring, discovery and authorization. The proposed system is based on a MAPE-K loop, where the knowledge base is realized as a *Web Ontology Language* (OWL) ontology. Domain-specific entities are modeled in this ontology which form the basic process steps in the control loop. These include symptoms which are created from monitored system data. For example, if a resource is not responding in a specific time window, a symptom of type *alarm* would be created. For lower priority symptoms like the discovery of a new resource, the symptom type *info* would be used. Based on these symptoms, administrators need to specify rules which tell the system what to do in case a symptom occurs. These rules are written in the *Semantic Query-Enhanced Web Rule Language* (SWQRL) which is both machine and human-readable and easily extensible [48, p. 729]. The paper states that ontologies can become quite large which is problematic because devices have low memory capacity. The rule-based knowledge ontology is highly extensible, but there could be significant effort to apply it to other domains. One can see that although semantic web rules are human-readable, it definitely requires experts to create or modify them. The paper focuses on monitoring **[R1.2], [R1.3]** and implements a management knowledge base **[R2.2]**. The rule-based approach offers good extensibility **[R.5.1]**. However, the paper does not target basic service management functionality like deployment and migration. Also, self-repair features of the ASM itself are not clearly stated and adaptive monitoring is not mentioned although these are crucial features of the SLSM design.

(+)	(-)
System monitoring	Complex semantic rule engine
Extensibility	Domain-specific ontology

Table 3.4: Dai et al. [6]

3.2 Conclusions

The presented papers have been selected out of a multitude of candidates. Since autonomous service management is relevant to various research fields like cloud and edge computing, the amount of related work is not surprising. Many solutions are based on MAPE or MAPE-K loops which can be seen as de facto standard in the area of autonomous computing. Interesting approaches regarding monitoring, data infrastructure, service lifecycle management and self-management can be found. However some research gaps around the topic of this thesis could be identified:

- Many solutions focus on node-local management only, e.g. in opportunistic networks.
- Many papers focus on approaches or algorithms for specific application domains and use cases, e.g. improving network latency in multi-hop networks.
- Most papers do not target smart space scenarios, where end users have no or little knowledge about technical details of the system.
- A comparable design of a service store, from which services can be downloaded and deployed to a smart space easily, has not been found in the literature review.
- Some related work does not target constrained devices.

This thesis focuses on developing a design for a centralized service management solution for IoT smart spaces. Specific algorithms are not directly interesting - instead the basic infrastructure should be analyzed and designed here. Key features include monitoring, data infrastructure, communication mechanisms, extensibility and self-management properties. Another important point is that non-expert end users are targeted by transferring the concept of an app store to a distributed smart space system. The system is designed towards running on constrained hardware. Therefore approaches from areas like cloud computing might not be suitable since some algorithms could require too much computing power.

Chapter 4

Design

The following chapter describes the design of the smart space management solution. In chapter 4.1 a subset of the identified requirements is chosen for design and evaluation. Chapter 4.3 deals with the overall system architecture and communication infrastructure. In chapter 4.2, service lifecycle management and service package structure are explained. The inner architecture of the SLSM is developed in chapter 4.4. In the end, chapter 4.5 covers interesting details on implementation.

4.1 Solution Requirements

Implementing all identified requirements would go far beyond the scope of this work. Therefore a subset will be selected and implemented as a prototype which can be evaluated in view of the goals of this thesis. The requirements were selected in a way that enables to test the complete process beginning with the S2Store to deploying services to a specific node. Also, the optimization through exchangeable strategies is a core feature which has to be implemented. Every decision made by the SLSM is based on accurate data about the network - therefore monitoring capabilities are an essential requirement. Table 4.1 summarizes the selected requirements. The intended service management solution must be able to monitor nodes regarding performance metrics and services and their lifecycle state. This monitoring is a crucial basis since no self-managing algorithm works without up-to-date system data. Monitoring must be adaptive, since continuous measurement of many metrics on each node could cause significant performance issues. Monitoring requires that data is stored so that it can be accessed efficiently. Some machine learning algorithms utilize time series data in order to make predictions on the system's state, for example. Storing monitored data and accessing historical data must therefore be possible in an efficient way. Services and their meta data should be stored inside one single file to simplify their transfer between the distributed components. This requires that the solution is able to unpack and parse

Req. #	**Requirement**	**Description**
[R1.1]	Adaptive Monitoring	Monitoring overhead should be kept as low as possible.
[R1.2]	Service Monitoring	The SLSM must keep an up-to-date view over all services.
[R1.3]	Node Monitoring	Offer monitoring as many different metrics as possible.
[R2.1]	Service Meta Data	Service packages must be enriched with meta data.
[R2.2]	Monitoring Database	Monitoring data should be stored in a DB for efficient access.
[R3.1]	Service Package Format	Services and meta data must be stored in one single file.
[R3.2]	Service Repository	Services are stored at the node running the SLSM.
[R3.3]	Service Deployment	The SLSM should offer functionality to deploy services.
[R3.4]	Service Migration	The SLSM should offer functionality to migrate services.
[R4.1, R4.2]	Communication with S2Store	The SLSM must be able to communicate withs the S2Store.
[R5.1]	Optimization as a Service	The SLSM can be extended with management algorithms.
[R5.4]	Check Service Constraints	Service requirements and dependencies must be checked.

Table 4.1: Solution Requirements

the meta data. All service packages that are downloaded to a computing node must be stored and registered in a service repository.

Node managers and SLSM should be able to easily check if a package is already available locally in order to prevent unnecessary file transfers. The prototype should support service deployment and migrations for a start. Other mechanisms like service replications and updates are not considered. Communication with the S2Store is necessary since the full deployment process should be evaluated - from the user triggering the installation until the deployment and start of the service on a specific node. During system runtime, the monitored network status must be regularly checked by one optimization strategy. In case optimization can be achieved by migrating services, the necessary actions are triggered by the currently active strategy. These strategies must be easily exchangeable. Service dependencies and hardware requirements of each service must be fulfilled any time, regardless of the active strategy. Therefore the system needs to check these constraints before calling the strategy to ensure minimum service requirements.

4.2 Service Lifecycle Management

In chapter 2.4 two alternatives for service packaging and lifecycle management have been presented: Docker and OSGi. To compare suitability one has to examine the target environment where technology should be applied to. Although the VSL is language-independent [1, p. 251], the DS2OS prototype will be completely written in Java since VSL service connectors are available in that language. These connectors provide an easy-to-use API for interacting with the VSL overlay. Methods for registering services and virtual node handlers are available, for example. This facilitates deployment in the beginning. Another reason is that VSL agents are implemented in Java, too - so each computing node requires an installed Java Runtime Environment anyway. Docker has the great advantage that services can be shipped inside images which also contain all dependencies and the execution environment. This would enable freedom of choice regarding implementation language of services. However using Docker for deploying a Java-based DS2OS system would imply a significant overhead, since each container has to run an own Java Runtime Environment inside.

In contrast to Docker, OSGi does not imply virtualization and runs directly on the host computer. It provides functionality for controlling the lifecycle of service bundles and a service registry for management. Details on the choice of OSGi for the Service Hosting Environment implementation can be found in [21].

```
service.zip
├── service.jar ............................ OSGi Service Package
│   ├── META-INF
│   │   └── MANIFEST.MF .................... OSGi Service Manifest
│   └── service ............................ Class File Dir
├── manifest.json .......................... DS2OS Service Manifest
└── service.xml ............................ VSL Context Model
```

Figure 4.1: Service Archive Structure

OSGi bundles are simple jar files containing a manifest file with dependencies and meta data. These files can be easily transferred between the DS2OS components via TCP, for example. However, additional meta data needs to be stored to add domain-specific information like the service identifier, hardware requirements and the required VSL context model. When developers upload their OSGi-conform jar bundles to the S2Store, they are prompted to enter all meta data required. This process avoids that mandatory data could be accidentally omitted. The store puts all information together into a DS2OS-specific manifest file and creates an XML-based VSL context model. Figure 4.1 shows the structure of such a zip file. The OSGi bundle is not modified and only copied into the zip archive. For the DS2OS manifest, JSON is chosen since it is human-readable and can be easily processed by JavaScript for displaying meta data in user interfaces. JSON can be converted into Java objects and back easily with the Google Gson library [49]. This enables DS2OS components to easily extract meta data from the zip archives and create service representations as Java objects. The service context model is needed for registration in the VSL and is the basis for creation of context nodes for interacting with other services.

Once a node receives a package, the service executable and manifest are extracted to the NLSM's working directory. The context model must be extracted to a location matching the path from the VSL agent's config file. The bundle is started using the OSGi framework implementation and the service registers to the agent running on the node. Design of the NLSM and runtime environment on the nodes is not part of this thesis but targeted in [21].

4.3 DS2OS Architecture

The proposed design is based on the VSL middleware and uses available functionality as far as possible. All components running inside the smart space are implemented as VSL services which communicate over VSL context nodes, which results in loosely coupled actors with few interfaces. This decreases complexity by *separation of concerns* and improves maintainability and testability. Any text-based information is exchanged via context nodes. However the transfer of service packages is done via a side channel, since the VSL implementation currently does not support transferring binaries. We consider it best practice to handle binary exchange via the VSL as soon as this functionality is available. The S2Store is designed as global stand-alone web server which runs outside any VSL network. Therefore communication between smart space services and the S2Store does not run over the VSL but via *Representational State Transfer* (REST) services.

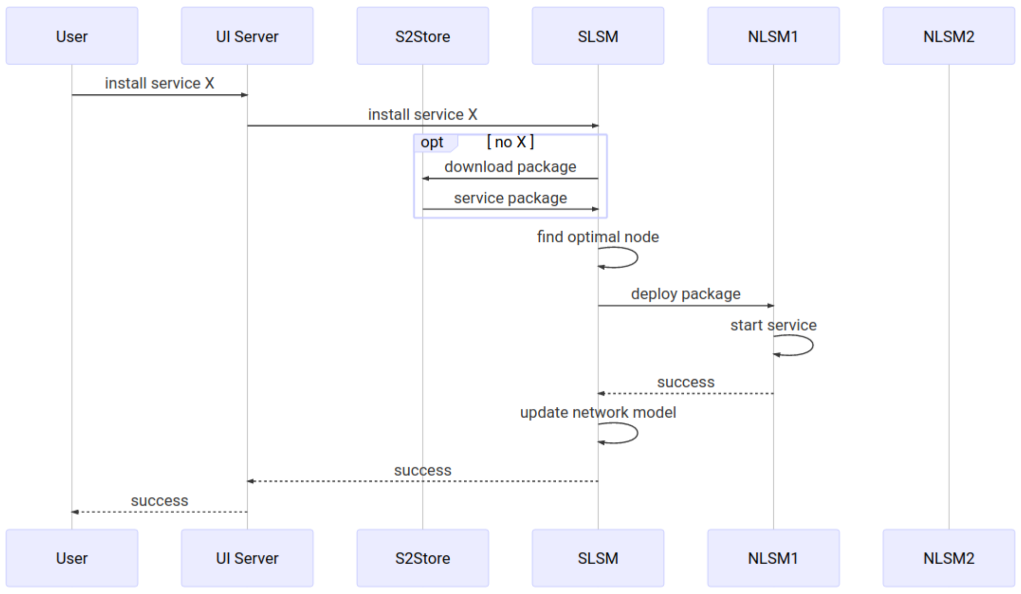


Figure 4.2: DS2OS Deployment Process

Figure 4.2 visualizes the deployment process in DS2OS. The site-local UI server provides a user interface for managing the smart space. It communicates with the S2Store via HTTP requests in order to obtain meta data of all available services. The user can trigger installation with one simple click in the web-based user interface. Since both UI server and SLSM are implemented as VSL services, the installation is triggered by calling the SLSM's virtual context node */installService* containing the service identifier. The SLSM then checks its local service repository for a service package with the same identifier. If it does not exist, it is directly downloaded from the S2Store via HTTP GET. As soon as the package is available, the currently active management strategy chooses a node for deployment based on the service's hardware requirements and dependencies to other services. The package is deployed to the selected node through a TCP socket side channel if it is not already available locally. The NLSM starts the service and returns a success or error message which is forwarded through SLSM to the UI server and presented to the user. The SLSM updates its network model accordingly. The whole deployment process requires only one click from the user since all communication and decisions are executed autonomously.

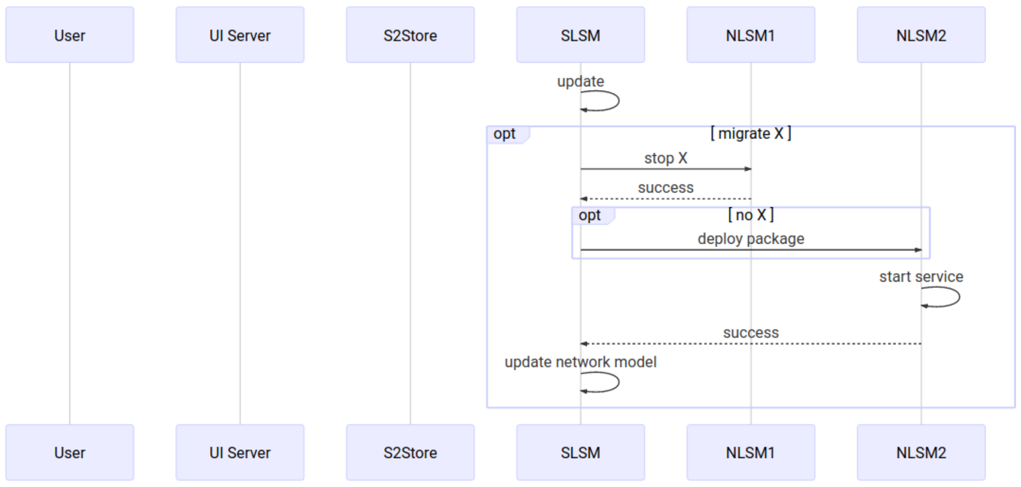


Figure 4.3: DS2OS Migration Mechanism

The migration or optimization process is visualized in figure 4.3. During runtime, the SLSM regularly updates its network state by requesting information from the nodes. Based on this latest network information, the currently active strategy determines if optimization can be achieved through migrating one or more services. The migration itself involves stopping the service on the current node, transferring the service package if necessary and starting the service on the new node. Afterwards, the SLSM updates its internal network representation accordingly. The migration process is completely autonomous. No interactions with the UI, S2Store and users are required.

Figure 4.4 visualizes the heartbeat mechanism of DS2OS. The SLSM must always stay updated about the state of each node. If one node fails, actions ensuring service quality must be triggered in a minimum of time. The reaction time mainly depends on the heartbeat rate and update cycle length. The NLSMs use UDP sockets for sending heartbeat messages to the SLSM, each containing the node identifier. Each heartbeat received by the SLSM is directly stored in the node model. It checks if a node with the identifier already exists in the model, creates a new node object if necessary and updates its latest heartbeat timestamp. In each update cycle, this timestamp is checked for each node. If it is older than a predefined threshold, the node's state is set to *FAILED*. As a consequence, services that were running on the failed node must be migrated to another node immediately. Heartbeat rate, update frequency and the node failure threshold are interesting parameters for testing different configurations regarding service downtime

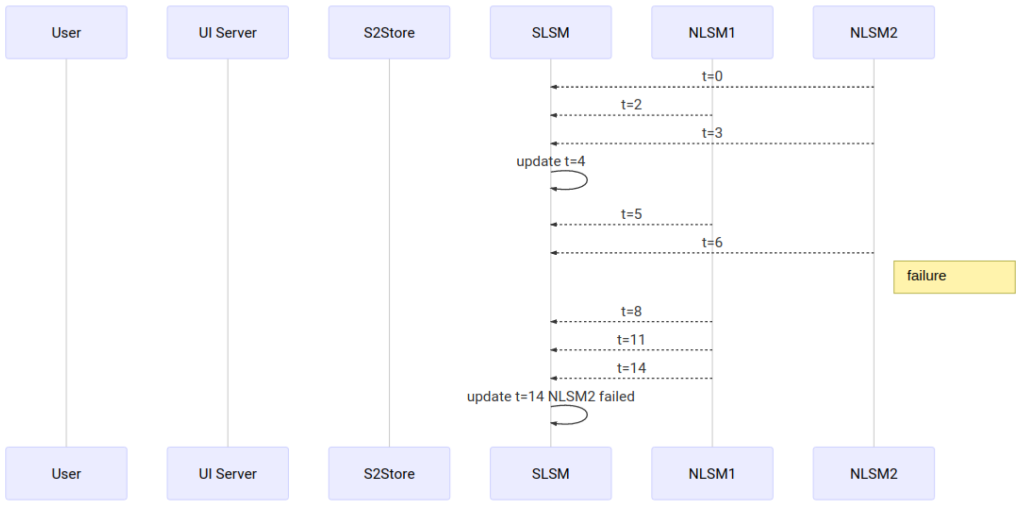


Figure 4.4: DS2OS Heartbeat Mechanism

and monitoring overhead that should be evaluated in future work. In the sequence diagram, the NLSMs send a heartbeat every three seconds, and the SLSM updates every ten seconds. If only one heartbeat gets lost on the way, the sending NLSM is not considered as failed immediately. Instead, the heartbeat must fail multiple times. Here, it takes two failed heartbeats and about eight seconds for the SLSM to recognize node NLSM2 has failed. It is beneficial to have a high heartbeat and update frequency to improve reaction times. However, each update cycle triggers optimization strategy algorithms which could produce certain overhead if executed within short spaces of time.

Table 4.2 summarizes all communication channels to be implemented. All VSL nodes are implemented as virtual nodes, which provide callback functions that are executed when a get or set is performed on them. Most management activities are executed behind the scenes, requiring no human input. Only four actions are interesting for users: *install/uninstall* and *start/stop* services. The simplicity of this interface is intended as users are considered non-experts and the goal of this thesis is to enable self-management to a high degree. The actions are controlled via the user interface provided by the UI service and are communicating solely with the SLSM. This is intended as complexities of the smart space being a distributed system should be hidden from the user. All interaction concerning the NLSMs must be performed by the SLSM autonomously. It determines the affected nodes and triggers the respective actions on them. Two virtual nodes are utilized for hardware monitoring. As adaptive monitoring is required, the SLSM is able to configure which parameters should be monitored by setting */config/metrics* node with a string containing all resource types. It makes sense for future development to include other configuration options via this node, for example the monitoring frequency. In turn, the NLSM regularly updates the */metrics* node with new measurements, for example the current CPU usage. These measurements are requested by the SLSM in each update cycle and forwarded to the strategy to make decisions on service migrations.

Component	**Node Address**	**Method**	**Parameters**
SLSM	/installService	GET	/serviceId=*
SLSM	/uninstallService	GET	/serviceId=*
SLSM	/startService	GET	/serviceId=*
SLSM	/stopService	GET	/serviceId=*
NLSM	/installService	GET	/serviceId=*
NLSM	/uninstallService	GET	/serviceId=*
NLSM	/startService	GET	/serviceId=*
NLSM	/stopService	GET	/serviceId=*
NLSM	/metrics	GET	none
NLSM	/config/metrics	SET	CPU&RAM&...

Components	**Purpose**	**Side Channel Type**
SLSM - NLSM	Heartbeat Mechanism	User Datagram Protocol (UDP)
SLSM - NLSM	Package Transfer	Transmission Control Protocol (TCP)
S2Store - SLSM	Package Transfer	Hypertext Transfer Protocol (HTTP)

Table 4.2: VSL and Side Channels Communication

4.4 Site-Local Service Manager Architecture

This chapter will describe the most important classes of the SLSM and the interfaces to the NLSM and S2Store. Management functionality is distributed over three classes. The *NetworkDataCollector* is responsible for monitoring the nodes in the network. It provides methods for querying data from the VSL context, for example a list of connected agents and running NLSMs. The *ServiceLifecycleManager* contains methods which execute service deployments and migrations. The *StrategyManager* acts as repository for optimization strategies and offers funtionality for switching between them. The *MongoDBConnector* is used for connecting to the MongoDB and storing and querying monitoring data. Since service packages are not stored in the database, these files are deposited on the filesystem under the SLSM's working directory. Here, the *SiteServiceRepository* class deals with extracting the packages, reading the contained service meta data and creating representations as Java objects.

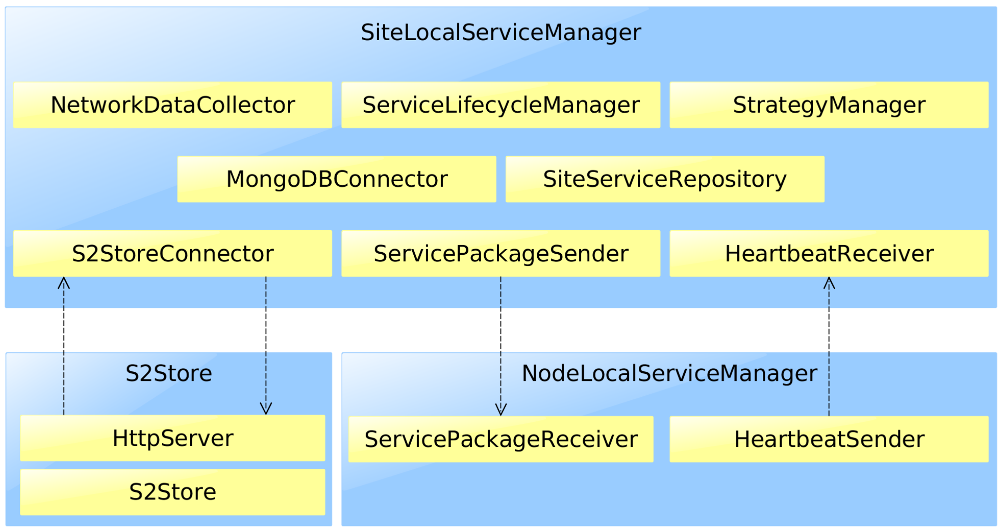


Figure 4.5: Class Structure of the SLSM and Interfaces to NLSM and S2Store

The *S2StoreConnector* class contains methods for calling the REST resources of the S2Store HTTP server - for example for downloading a service package. The S2Store logic provides a similar service repository implementation like the SLSM in order to read meta data. For deploying packages to the NLSMs, sender and receiver classes exist. The binaries are transferred via a TCP socket. Another class pair for communication between NLSM and SLSM implement UDP sockets for sending heartbeats on a regular basis. Smaller helper classes and logic for managing certificates are omitted here. The class structure has been created under the idea of *separation of concerns*. Future changes and extensions should be easily possible. For example, the temporary side channels for heartbeats and package transfer can be easily exchanged without changing core code on both SLSM and NLSM side. The *StrategyManager* implementation follows a strategy pattern - therefore new strategy implementations can be added and activated without great effort.

4.5 Implementation

The DS2OS prototype implementation is the result of a collaboration with two related research projects. Donini et al. [22] targets autonomous management of certificates and access rights. Ohlenforst et al. [21] focuses on node-local service management and developed a design for the runtime environment (SHE) and the NLSM. In total, seven modules have been developed:

```
ds2os
 ├── *common ...................... Common classes used in multiple modules
 ├── she ........................................ Service Hosting Environment
 ├── *nlsm ........................................ Node-local Service Manager
 ├── *slsm ......................................... Site-local Service Manager
 ├── slca ...................................... Site-local Certificate Authority
 ├── *s2store .................................................... Service store
 └── *uiserver .......................................... Smart space UI server
```

Figure 4.6: DS2OS Project Structure

Relevant modules regarding the topic of this thesis are marked with an asterisk.

4.5.1 Optimization Strategies

The current implementation features a strategy manager class which holds a map of all available strategies and enables the SLSM to switch between them by calling the *setActiveStrategy(int strategyId)* method. The strategies themselves have to extend the abstract class *OptimizationStrategy* in order to guarantee that the methods *optimize(Map<String, SiteLocalNode> nodes)* and *deployServicePackage(Map<String, SiteLocalNode> nodes, String serviceId)* are implemented. The first method takes all node representations as parameter and has to implement a logic that checks if the current network status could be optimized. As an example, a load balancer has been implemented that adjusts the number of services running on each node. The second method is called by the SLSM as soon as a service should be installed. The strategy is accountable for finding the optimal node for deployment. Strategies can trigger both deployment and migration after internal calculations via an interface callback that is implemented by the SLSM.

4.5.2 Network and Service Representation

Most management functions rely on a Java object representation of the current network status. For this, classes and interfaces exist for representing nodes and services. The

SLSM holds a map of node objects. Each node object holds a map of service objects that are available locally and implements the *INode* interface which contains common methods, for example for starting and stopping services. Each SLSM and NLSM hold own classes implementing the node interface, since different logic is required. From the perspective of the SLSM, a service is started by requesting the corresponding VSL context node at the NLSM. From the node manager perspective, this is done by communicating the action to the SHE. Another example concerns the timestamp of the last heartbeat sent by a specific node - this information is only relevant to the SLSM and therefore only implemented in the *SiteLocalNode* class.

4.5.3 Database and Data Structure

In the current VSL implementation, nodes persist their context in a local HSQLDB [50]. However we decided to use a MongoDB for storing the SLSM's data because of several reasons:

Popularity MongoDB is widely used today, especially in web applications.

Web interfaces Since we implement two web-based user interfaces for the S2Store and the DS2OS UI service, JSON-based data structures are a good choice for being processed with JavaScript in the frontend.

Performance comparison We are interested in a performance comparison between the VSL context which is stored in a relational HSQLDB database, and a document-oriented database like MongoDB.

Data model Using relational data structures requires a strict data model design in advance. Instead, data model in document-oriented databases can be changed freely over time. This will become handy in the future when new metrics can be monitored and need to be stored in addition to existing ones, for example.

As a wide range of optimization algorithms should be supported by the SLSM, it must be able to provide detailed data about the network. Also, the SLSM could restore configurations from the database after it failed. We plan to store representations of the current network state each time the SLSM updates its network information. Therefore each data set is stored with a current timestamp as its identifier. The data itself contains all relevant data that can be collected in the current implementation. If additional monitoring metrics are added in the future, no redesign of the database is necessary since documents stored in the same collection may differ in their structure. The structure of the prototype's data model can be abstracted from the following document:

```
1533717095 : {
  dn89r3438h3ir329339rur9u393rjir338: {
    agentRoot: "/agent1",
    nodeManagerRoot: "/nlsm1",
    runningServices: ["ledservice", "uiserver"],
    stoppedServices: [],
    hardware: {
      CPU: 1200,
      RAM: 2000
    }
  }, rrh4hr498393jfuih39rhfrh48trh4tk3n: {
    agentRoot: "/agent2",
    nodeManagerRoot: "/nlsm2",
    runningServices: [],
    stoppedServices: ["helloworld"],
    hardware: {
      CPU: 700,
      RAM: 512
    }
  },
  ...
}
```

The network state is stored under the current timestamp. Each node is identified by its certificate's public key. The data stored about each node comprises its VSL agent and service address, running and stopped services as well as node hardware capabilities. Service meta data is not persisted in the database, since it can be obtained from the service packages on the file system. Therefore it is sufficient to store only service identifiers in the database.

Chapter 5

Evaluation

This chapter describes evaluation of the SLSM implementation. Criteria are specified in chapter 5.1. An isolated evaluation of write and read latencies is provided in chapter 5.2. Performance evaluation is described in 5.3.

5.1 Evaluation Criteria

Evaluation of the presented design includes quantitative criteria which are concerned with performance of the prototype as well as qualitative criteria targeting quality of the design. In particular, the following points are evaluated:

1. `[Quantitative]` Measurement and comparison of read and write latencies MongoDB versus VSL context nodes.
2. `[Quantitative]` Measurement of performance metrics on the node running the SLSM in a realistic scenario.
3. `[Qualitative]` Quantitative evaluation will show how the system behaves in realistic situations and if the proposed design is suitable.

The database is applied to store relevant data about the network state as described in chapter 4.5. The data size increases with number of nodes, services and measured metrics in the network. If machine learning algorithms are applied, historical data could be valuable which again increases size of the data to be read. Therefore it makes sense to evaluate how fast data can be stored and read using the database. The same tests are applied to a VSL node in order to have a relation on how performant the MongoDB works. The main evaluation includes testing common system functionality under realistic conditions. The results will reveal the applicability of the system in real world scenarios, both quantitatively regarding performance as well as qualitatively.

5.2 Data Storage and Access Latencies

For data storage latency evaluation, information about the current network state is used. In case of the MongoDB, the data must be transformed into a specific Document object before it can be stored via the Java connector. For VSL, exactly the same data is converted into a comparable JSON string before being stored in the context node. Data is stored on the agent running on the same node to achieve comparable results. Both data preparation steps are excluded from measuring write latencies. Also, no deserialization is included in the measurements of read latencies. For testing a higher number of nodes, the measured node data is multiplied by the number of nodes needed before storage. In any case, all current nodes are serialized and written to the MongoDB or VSL node under the current timestamp. Reading data always includes the node set stored under the latest timestamp.

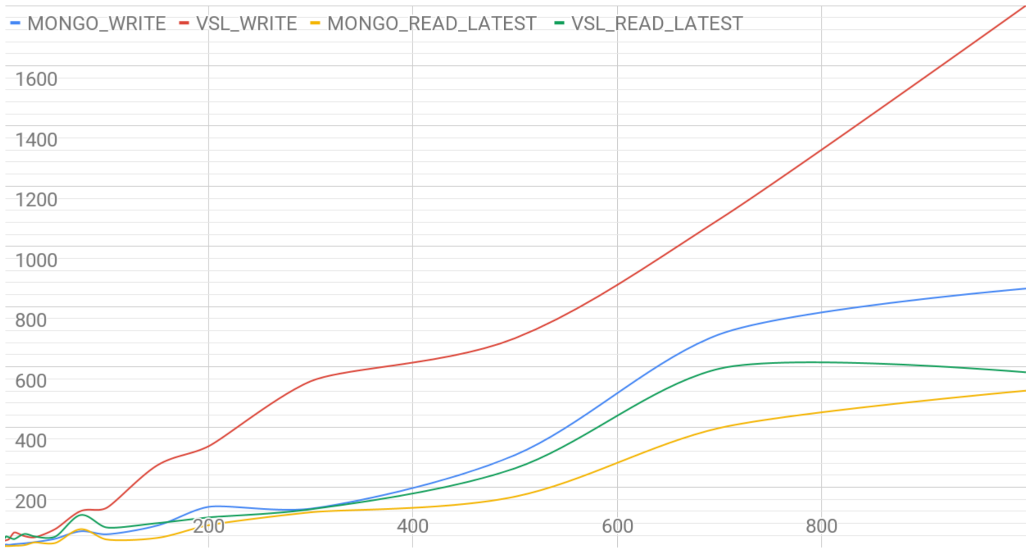


Figure 5.1: Read and Write Latencies of VSL Context and MongoDB

Figure 5.1 shows the measured latencies in milliseconds for different dataset sizes. One can see that read and write latencies are comparable for up to 50 nodes. However, the test data currently contains only few services and monitoring values so in a typical smart space the values would start differing significantly already under a lower number of nodes. Read latencies stay comparable over large data sizes. Significant difference makes writing larger data to the VSL. Latencies increase up to the order of two seconds, which is extremely inefficient considering realistic update cycles of the SLSM in the order of ten seconds. In contrast, MongoDB write latency remained clearly under one second and is therefore much more efficient. Also we observed the problem that setting VSL nodes with large strings generates errors in some cases. The results show clearly that using a separate database for storing large amounts of data improves system performance and reliability. Since data is only stored and used by the SLSM, there is also no need to store it in the VSL overlay because no distributed access is required during normal system execution. However, replicating the data in case the SLSM fails and needs to be migrated to another node is considered important future work. There may be more efficient database systems compared to MongoDB, but document-orientation is adequate for storing data about a continuously changing system that can be extended at runtime through additional services. For example, if nodes are extended by additional monitoring capabilities, these values can be added to the data model without any problems.

5.3 Performance Measurements

5.3.1 Experimental Set-Up

For quantitative measurements we created the testbed displayed in figure 5.2 which features typical IoT hardware. The main device running all services relevant for site-local management is a Raspberry Pi 3 Model B. It features a 1.2GHz 64-bit quad-core ARMv8 Cortex-A53 and 1024 MiB of RAM. With its low power consumption it is considered suitable for IoT and smart space scenarios. It is used as the smart space management node running an agent, the SLSM, the MongoDB, an NLSM instance and the UI server service. The second node is a relatively old Raspberry Pi 1 Model B with 700MHz ARM CPU and 512 MB of main memory. Such hardware could lead to interesting results regarding minimum hardware requirements. It will only run an agent and an NLSM service instance. The third network node is a standard laptop, which is used for running another NLSM, but mainly for monitoring the DS2OS via the console service and for executing the S2Store. Also, a browser is running on the laptop for accessing the S2Store and UI server web interfaces and triggering service installation. Running a browser on one of the Raspberry Pis would use significant hardware capacities and distort the measurements. All nodes are connected to the same router so the agents are able to discover each other for building the VSL overlay network.

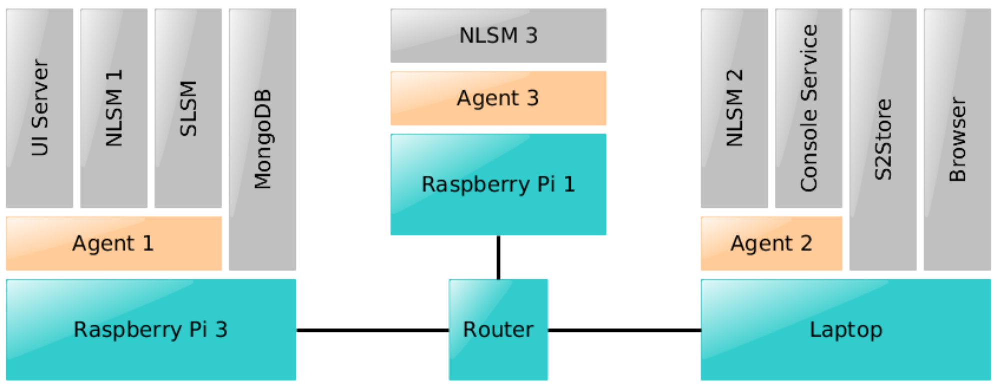


Figure 5.2: Evaluation Testbed

5.3.2 Execution and Results

A typical application scenario of the DS2OS involves a number of constrained devices, each running VSL agents and DS2OS components. On top of this configuration, several service instances must be executed. Therefore the resource usage of the DS2OS system is an interesting aspect which is evaluated in the following. Services are not executed but only deployed to the respective nodes, since the Service Hosting Environment implementation is not finished currently. DS2OS components are exported as runnable jar files and executed manually via terminal, instead of being started over OSGi by the SHE. Overhead of the SHE must therefore be evaluated separately in the future. As typical performance metrics, CPU load and RAM usage are measured to show which minimum hardware is required to run the basic DS2OS system and how much resources are left to run services in the end. Performance metrics are measured on both Raspberry Pis; additionally power consumption is monitored on the RPi3. Energy consumption is mainly interesting because in a real life scenario multiple devices are running day and night and should not cause severe energy costs.

The following charts display the results for CPU and RAM usage on both Raspberry Pis as well as energy consumption on the newer device. The test process begins with starting the MongoDB service on the RPi3 (event **M**). Next, all three agents are started successively on each node (**A1, A2, A3**). SLSM and UI server service are started on the RPi3 after the agents have formed the VSL overlay network. Afterwards the three NLSMs are started subsequently on each node (**N1, N2, N3**). As soon as all node managers are started and the SLSM discovered them via heartbeat mechanism, a test service package is installed (**SRV**). The node manager of the chosen target node is then terminated to see if the service package is deployed to another node and how fast this migration process is executed.

Figures 5.3 and 5.4 show CPU and RAM usage on the Raspberry Pi 3 (node number 1). Starting the VSL agent **A1** temporarily increases CPU load up to almost 70 percent and uses around 90 megabytes of main memory. The following discovery process between the agents does not produce significant CPU load and uses no further RAM. Starting the SLSM and NLSM 1 also has a mainly temporal effect on the processor. In contrast the UI server creates continuous stress. All three events increase main memory usage significantly. The SLSM uses roughly 160, the UI server around 110, and the NLSM around 150 megabytes. The strategy selected NLSM 1 for service deployment, so this node is manually terminated (**X1**). Since services are not started currently, the decrease in RAM is completely caused by the terminated NLSM service. The migration process is triggered shortly afterwards. It took 37 seconds until the package was received by NLSM 2.

Performance on the Raspberry Pi 1 brought very clear results. After starting the agent **A3**, CPU load already increased to nearly 100 percent, leaving no resources left for node manager or additional services. This became apparent when the NLSM was terminated because of a timeout during connecting to the VSL agent (**T3**). RAM usage had a maximum of 400 megabytes leaving only few buffer for running more services.

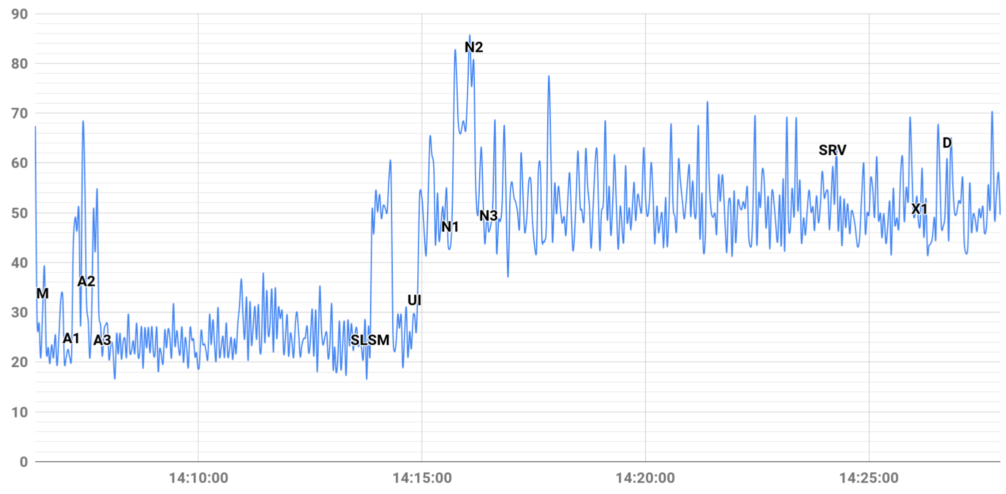


Figure 5.3: CPU Usage on the RPi3

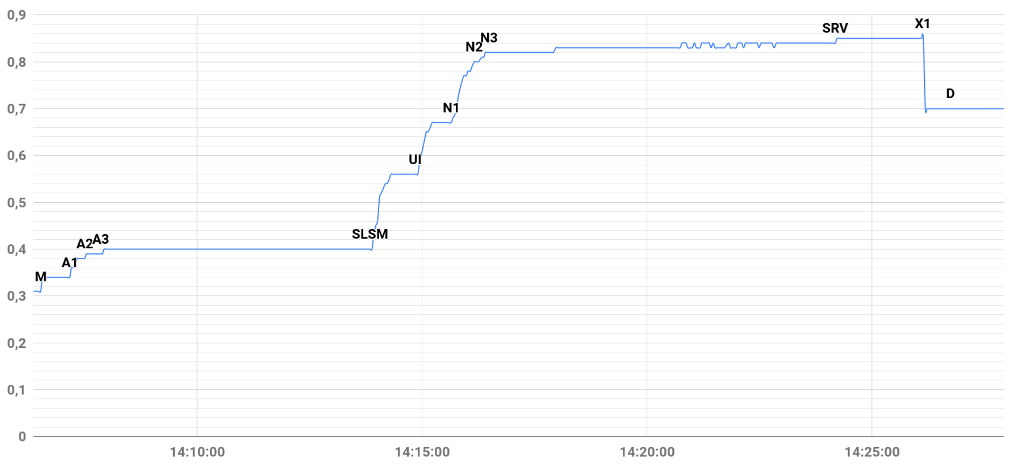


Figure 5.4: RAM Usage on the RPi3

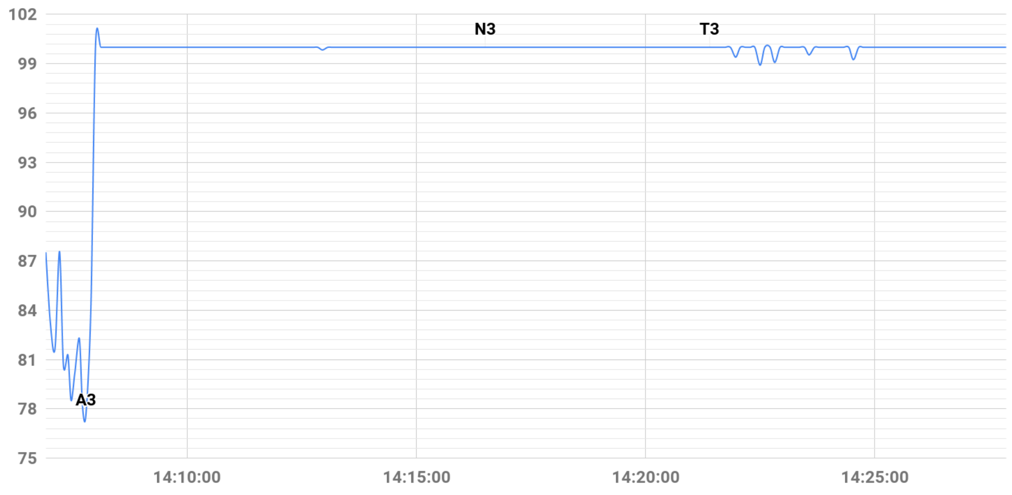


Figure 5.5: CPU Usage on the RPi1

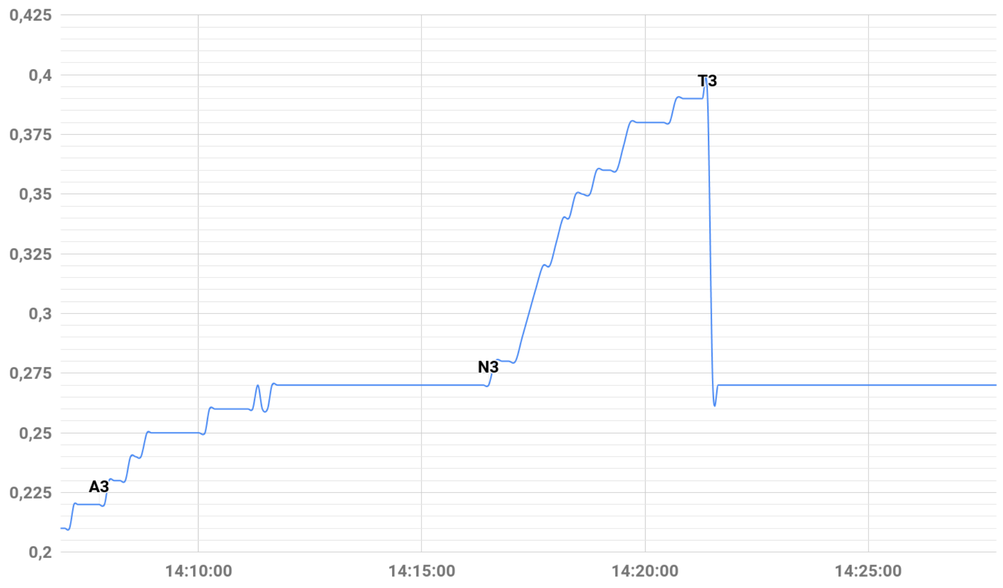


Figure 5.6: RAM Usage on the RPi1

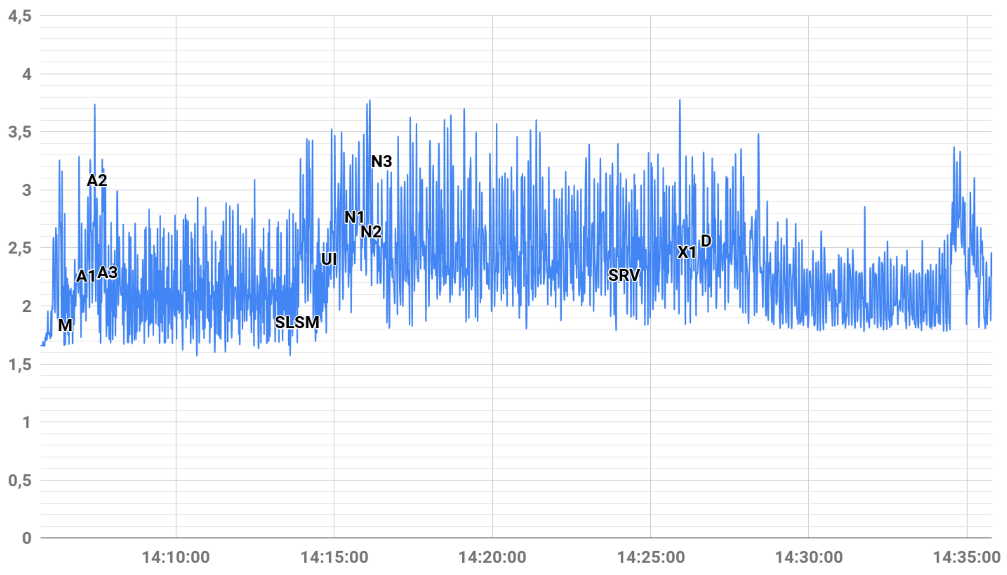


Figure 5.7: Power Consumption on the RPi3

Power consumption measurements are displayed in figure 5.7. Slight increases can be observed after starting of agent **A1**, NLSM **N1** and when NLSM 1 is terminated (**X1**). Consumption reaches its peak at around 3.738 watts. The Raspberry Pi 3 Model B officially requires up to 2.5 ampere current. Assumed that voltage is constant at five volts the device has a maximum power consumption around 12.5 watts. Therefore measured power consumption is considered moderate.

5.3.3 Conclusion

Evaluation showed that hardware performance of typical IoT devices can be sufficient for executing the Java-based DS2OS system on top of the VSL. However, modern hardware like the Raspberry Pi 3 is required, especially if a number of services should be executed on top of the evaluated configuration. Also additional overhead must be considered for the currently lacking Service Hosting Environment. Power consumption of the device turned out adequate for continuous operation.

Future work should consider another programming language for DS2OS implementation in order to improve system performance. The JVM certainly adds some significant overhead compared to native languages like C.

Chapter 6

Conclusion and Future Work

The goal of this thesis was to analyze service management approaches from different research areas and to apply the results to IoT smart spaces using the Virtual State Layer middleware as basis. A comprehensive literature review revealing different research areas with similar problem statements has been presented. Various requirements and solution approaches have been identified and presented in detail. A solution design has been developed based on the requirements. In the end, a prototype was implemented and evaluated regarding applicability in the presented problem domain.

The presented solution design provides a foundation for centralized service management in distributed smart space environments. The basic data infrastructure, deployment mechanisms and communication channels between the different component as well as important management features have been developed. However, the design does not provide a completely operational solution - plenty of future work is required to create such a dependable system:

1. Provide further monitoring metrics to support many different optimization strategies.
2. Improve user interfaces of SLSM and S2Store, e.g. by implementing a smartphone app.
3. Provide further management functionality like service replication and updates.
4. Analyze and evaluate specific optimization strategies and algorithms regarding applicability and performance in the problem domain. Especially machine learning algorithms are interesting in our context in order to enable autonomy.
5. Send error reports to the S2Store as feedback for service developers.
6. Implement optimization strategies as VSL services and enable users to easily switch between them.

7. Improve performance of the whole system to support older hardware.

The possibility of exchanging management strategies has been implemented, but further research is required to examine different specific algorithm types and evaluate applicability and performance in the problem domain. The current implementation only features CPU and RAM usage data which limits the range of algorithms that could be applied. Other interesting metrics could be network bandwidth, free storage capacity or connected sensors and actuators. The strategy implementation currently features node selection for service deployment and migration. However there are other important functional features which should be considered, too. These include replication of services to reduce downtime in case a node fails and services have to be restarted. Also automated updates for services and the DS2OS system should be implemented. Especially distributed systems should be up-to-date to maximize dependability and security. Further research has to be conducted to improve usability and user interfaces of the smart space. Also, user experience, power consumption and support of older hardware can be improved regarding performance of the components.

One can see there are many open challenges which have to be met in the future to realize the idea of a fully autonomous, reliable and easy to use smart space. If the concept of having a service store becomes prevalent for such distributed systems, its success and user experience will depend in a large part on developers which contribute by providing innovative services and help improving service management continuously by creating efficient and reliable optimization strategies. However applications of the Internet of Things in homes or offices definitely have the potential of permanently changing the way we perceive technology and interact with it and to improve quality of living by automating and simplifying our daily tasks.

Bibliography

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

[2] *What is a Container, available at https://www.docker.com/what-container*, Jan. 2017. [Online]. Available: https://www.docker.com/what-container

[3] M. C. Huebscher and J. A. McCann, "A Survey of Autonomic Computing—Degrees, Models, and Applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 7:1–7:28, Aug. 2008. [Online]. Available: http://doi.acm.org/10.1145/1380584.1380585

[4] J. Cao, L. Xu, R. Abdallah, and W. Shi, "EdgeOS_h: A Home Operating System for Internet of Everything," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2017, pp. 1756–1764.

[5] K. Velasquez, D. P. Abreu, M. Curado, and E. Monteiro, "Service placement for latency reduction in the internet of things," *Annals of Telecommunications*, vol. 72, no. 1-2, pp. 105–115, Feb. 2017. [Online]. Available: http://link.springer.com/article/10.1007/s12243-016-0524-9

[6] M. Burkert, H. Krumm, and C. Fiehe, "Technical management system for dependable Building Automation Systems," in *2015 IEEE 20th Conference on Emerging Technologies Factory Automation (ETFA)*, Sep. 2015, pp. 1–8.

[7] "IEEE Xplore Digital Library." [Online]. Available: https://ieeexplore-ieee-org.eaccess.ub.tum.de/Xplore/home.jsp

[8] S. A. Al-Qaseemi, H. A. Almulhim, M. F. Almulhim, and S. R. Chaudhry, "IoT architecture challenges and issues: Lack of standardization," in *2016 Future Technologies Conference (FTC)*, Dec. 2016, pp. 731–738.

[9] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "Gaia: A Middleware Platform for Active Spaces," *SIGMOBILE Mob. Comput. Commun. Rev.*, vol. 6, no. 4, pp. 65–67, Oct. 2002. [Online]. Available: http://doi.acm.org/10.1145/643550.643558

[10] E. Ahmed, I. Yaqoob, A. Gani, M. Imran, and M. Guizani, "Internet-of-things-based smart environments: state of the art, taxonomy, and open research challenges," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 10–16, Oct. 2016.

[11] M. N. Huhns and M. P. Singh, "Service-Oriented Computing: Key Concepts and Principles," *IEEE Internet Computing*, vol. 9, no. 1, pp. 75–81, Jan. 2005. [Online]. Available: https://doi.org/10.1109/MIC.2005.21

[12] O. G. Aliu, A. Imran, M. A. Imran, and B. Evans, "A Survey of Self Organisation in Future Cellular Networks," *IEEE Communications Surveys Tutorials*, vol. 15, no. 1, pp. 336–361, 2013.

[13] R. Pozza, M. Nati, S. Georgoulas, K. Moessner, and A. Gluhak, "Neighbor Discovery for Opportunistic Networking in Internet of Things Scenarios: A Survey," *IEEE Access*, vol. 3, pp. 1101–1131, 2015.

[14] D. Xu, Y. Li, X. Chen, J. Li, P. Hui, S. Chen, and J. Crowcroft, "A Survey of Opportunistic Offloading," *IEEE Communications Surveys Tutorials*, pp. 1–1, 2018.

[15] A. Yousafzai, A. Gani, R. M. Noor, M. Sookhak, H. Talebian, M. Shiraz, and M. K. Khan, "Cloud resource allocation schemes: review, taxonomy, and opportunities," *Knowledge and Information Systems*, vol. 50, no. 2, pp. 347–381, Feb. 2017. [Online]. Available: https://link.springer.com/article/10.1007/s10115-016-0951-y

[16] Z. Lu, S. Takashige, Y. Sugita, T. Morimura, and Y. Kudo, "AN ANALYSIS AND COMPARISON OF CLOUD DATA CENTER ENERGY-EFFICIENT RESOURCE MANAGEMENT TECHNOLOGY," *Services Transactions on Services Computing*, vol. 2, pp. 32–51, Oct. 2014.

[17] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos, "A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges," *IEEE Communications Surveys Tutorials*, vol. 20, no. 1, pp. 416–464, 2018.

[18] C. Woebker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and Management of Fog Computing Applications," 2018.

[19] S. Agarwal, S. Yadav, and A. Yadav, "An Efficient Architecture and Algorithm for Resource Provisioning in Fog Computing," *International Journal of Information Engineering and Electronic Business*, vol. 8, pp. 48–61, 2016.

[20] M.-O. Pahl, G. Carle, and G. Klinker, "Distributed Smart Space Orchestration," in *Network Operations and Management Symposium 2016 (NOMS 2016) - Dissertation Digest*, May 2016.

[21] F. Ohlenforst, "Providing a Remotely Manageable Runtime Environment for IoT Services," Master's thesis, Technical University of Munich, Aug. 2018.

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

[23] A. R. Adellina, Suhardi, N. B. Kurniawan, and J. Sembiring, “Dependability model of services computing systems,” in *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)*, Nov. 2017, pp. 1–6.

[24] R. Garg, H. Saran, R. S. Randhawa, and M. Singh, “A SLA framework for QoS provisioning and dynamic capacity allocation,” in *IEEE 2002 Tenth IEEE International Workshop on Quality of Service (Cat. No.02EX564)*, 2002, pp. 129–137.

[25] M. Burkert, J. Volmer, H. Krumm, and C. Fiehe, “Rule-based technical management for the dependable operation of networked building automation systems,” in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, May 2017, pp. 612–615.

[26] H. Zhang, F.-Y. Wang, and Y. Ai, “An OSGi and agent based control system architecture for smart home,” in *Proceedings. 2005 IEEE Networking, Sensing and Control, 2005.*, Mar. 2005, pp. 13–18.

[27] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, “Performance Evaluation of Microservices Architectures Using Containers,” in *2015 IEEE 14th International Symposium on Network Computing and Applications*, Sep. 2015, pp. 27–34.

[28] H. Kang, M. Le, and S. Tao, “Container and Microservice Driven Design for Cloud Infrastructure DevOps,” in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, Apr. 2016, pp. 202–211.

[29] H. Zhang, H. Ma, G. Fu, X. Yang, Z. Jiang, and Y. Gao, “Container Based Video Surveillance Cloud Service with Fine-Grained Resource Provisioning,” in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, Jun. 2016, pp. 758–765.

[30] B. I. Ismail, E. M. Goortani, M. B. A. Karim, W. M. Tat, S. Setapa, J. Y. Luke, and O. H. Hoe, “Evaluation of Docker as Edge computing platform,” in *2015 IEEE Conference on Open Systems (ICOS)*, Aug. 2015, pp. 130–135.

[31] “Docker Documentation,” May 2018. [Online]. Available: https://docs.docker.com/

[32] “Docker Hub.” [Online]. Available: https://hub.docker.com/

[33] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, “Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments,” in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb. 2013, pp. 233–240.

[34] R. Morabito, “A performance evaluation of container technologies on Internet of Things devices,” in *2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Apr. 2016, pp. 999–1000.

[35] “Docker Pirates ARMed with explosive stuff · Docker Pirates ARMed with explosive stuff.” [Online]. Available: https://blog.hypriot.com/

[36] M. Großmann, A. Eiermann, and M. Renner, “Hypriot Cluster Lab: An ARM-Powered Cloud Solution Utilizing Docker,” May 2016.

[37] M. B. Alaya, S. Matoussi, T. Monteil, and K. Drira, “Autonomic Computing System for Self-management of Machine-to-machine Networks,” in *Proceedings of the 2012 International Workshop on Self-aware Internet of Things*, ser. Self-IoT ’12. New York, NY, USA: ACM, 2012, pp. 25–30. [Online]. Available: http://doi.acm.org/10.1145/2378023.2378029

[38] J. O. Kephart and D. M. Chess, “The vision of autonomic computing,” *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[39] IBM, “An architectural blueprint for autonomic computing,” *IBM White Paper*, vol. 31, pp. 1–6, 2006.

[40] F. Ramezani, M. Naderpour, and J. Lu, “Handling uncertainty in cloud resource management using fuzzy Bayesian networks,” in *2015 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*, Aug. 2015, pp. 1–8.

[41] O. Jules, A. Hafid, and M. A. Serhani, “Bayesian network, and probabilistic ontology driven trust model for SLA management of Cloud services,” in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, Oct. 2014, pp. 77–83.

[42] H. Moens, B. Hanssens, B. Dhoedt, and F. D. Turck, “Hierarchical network-aware placement of service oriented applications in Clouds,” in *2014 IEEE Network Operations and Management Symposium (NOMS)*, May 2014, pp. 1–8.

[43] K. Kientopf, S. Raza, S. Lansing, and M. Güneş, “Service management platform to support service migrations for IoT smart city applications,” in *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, Oct. 2017, pp. 1–5.

[44] H. Arabnejad, C. Pahl, P. Jamshidi, and G. Estrada, “A Comparison of Reinforcement Learning Techniques for Fuzzy Cloud Auto-Scaling,” in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 64–73.

[45] H. Khazaei, H. Bannazadeh, and A. Leon-Garcia, “SAVI-IoT: A Self-Managing Containerized IoT Platform,” in *2017 IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, Aug. 2017, pp. 227–234.

[46] B. Mayer and R. Weinreich, "A Dashboard for Microservice Monitoring and Management," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, Apr. 2017, pp. 66–69.

[47] S. Agarwala, Y. Chen, D. Milojicic, and K. Schwan, "QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems," in *2006 IEEE International Conference on Autonomic Computing*, Jun. 2006, pp. 124–133.

[48] W. Dai, V. N. Dubinin, J. H. Christensen, V. Vyatkin, and X. Guan, "Toward Self-Manageable and Adaptive Industrial Cyber-Physical Systems With Knowledge-Driven Autonomic Service Management," *IEEE Transactions on Industrial Informatics*, vol. 13, no. 2, pp. 725–736, Apr. 2017.

[49] "gson: A Java serialization/deserialization library to convert Java Objects into JSON and back," Jun. 2018, original-date: 2015-03-19T18:21:20Z. [Online]. Available: https://github.com/google/gson

[50] "HSQLDB." [Online]. Available: http://hsqldb.org/