A Scalable Architecture for Distributed OSGi in the Cloud

Hendrik Kuijs[1], Christoph Reich[1], Martin Knahl[1] and Nathan Clarke[2]
[1]*Institute for Cloud Computing and IT Security, Furtwangen University, Furtwangen, Germany*
[2]*Centre for Security, Communications and Network Research, Plymouth University, Plymouth, U.K.*

Keywords: OSGi Service Architecture, Load Balancing, Distributed OSGi, PaaS Management, SaaS.

Abstract: Elasticity is one of the essential characteristics for cloud computing. The presented use case is a Software as a Service for Ambient Assisted Living that is configurable and extensible by the user. By adding or deleting functionality to the application, the environment has to support the increase or decrease of computational demand by scaling. This is achieved by customizing the auto scaling components of a PaaS management platform and introducing new components to scale a distributed OSGi environment across virtual machines. We present different scaling and load balancing scenarios to show the mechanics of the involved components.

1 INTRODUCTION

The average life expectancy in developed countries world-wide is increasing continuously while the birth-rate at the same time is declining (United Nations, 2001). This leads to a decreasing proportion of the young working population: Families are getting smaller in general and are no longer able to care for their elderly relatives, like extended families once used to do. On the other hand, the traditional concept of extended families gets replaced by welfare models and pension systems. Politics try to support this trend by introducing programs for new nursing or day-care facilities, but there is a still a lack of trained personnel and the progress of expansion is still slow.

With the goal to reduce the workload upon professional care personnel and facilities, Ambient Assisted Living (AAL) is seen as a possible solution. AAL is defined as transformation of the known living environment of the elderly people and people in need of help by new technological concepts to assist both the caring person and the person needing care. This means that people are enabled to live at home longer, which is an often desired effect (Grauel and Spellerberg, 2007).

Existing AAL platforms extend the concepts of smart-home environments combining input-devices like sensors, user-interfaces or cameras and output-devices like alarms, emergency call functionality, databases or graphical user interfaces. All devices are connected by a configurable middleware and all components including the compute-power have to be installed in the user's living environment. This existing computing equipment has to be renewed or extended, if new demanding services are introduced into the AAL environment.

Therefore we are focusing on delivering cloud based services for AAL. Service providers (e.g., care givers or day care facilities) should be able to deliver services without the need for investing in expensive technical equipment in advance. With cloud computing, the high start-up costs can be reduced significantly for service providers and it will be feasible for users to try out new or innovative services without the need of a high investment. Giving AAL service providers a dynamic load balanced, managed, easy provisioned and easy to use AAL service platform hosted in the cloud to be deployed on demand is a highly desirable goal.

This flexibility can be provided by a customizable Platform as a Service (PaaS) that is run by the AAL platform provider for the AAL service provider and used as Software as a Service (SaaS) by the end user.

The security and privacy enhanced cloud infrastructure for AAL (speciAAL) focuses on delivering personalised and adapted services for information, communication and learning. It is based on the project Person Centered Environment for Information Communication and Learning (PCEICL) (Kuijs et al., 2015) which is developed in the Collaborative Centre for Applied Research on Ambient Assisted Living (ZAFH-AAL, 2014). The PaaS is considered to run in a private cloud, as adaptation of the system to the user's need is heavily based on personal user data.

Kuijs, H., Reich, C., Knahl, M. and Clarke, N.
A Scalable Architecture for Distributed OSGi in the Cloud.
In *Proceedings of the 6th International Conference on Cloud Computing and Services Science (CLOSER 2016) - Volume 2*, pages 109-117
ISBN: 978-989-758-182-3

SpeciAAL is based on OSGi (OSGi Alliance, b): OSGi supports installing, starting and stopping software bundles during runtime. By introducing tools for managing and resolving of dependencies between bundles, the application can be extended or updated during runtime without the need to restart the whole environment. OSGi has also been chosen, since it is a seen as a standard technology for Smart Home environments and AAL platforms (OSGi Alliance, a).

With this paper we are focusing on the scaling and load balancing process in the PaaS layer for speciAAL. As speciAAL is a customizable and extensible application, we present a way of horizontal scaling of an OSGi environment based on distributed OSGi (DOSGi). New mechanics and modules in the PaaS layer are introduced to support the scaling process for the SaaS layer.

In Section 2 we describe different concepts of scaling in cloud environments and the basis of distributed OSGi. Related Work in the field of AAL and scaling OSGi applications in cloud environments is presented in Section 3. Section 4 gives an overview of the architecture for speciAAL and its main concepts. Components that have to be adopted to support the load balancing approach for speciAAL are described in Section 5. To show the main functionality of load balancing in speciAAL we present four common scenarios in Section 6, followed by the conclusion in Section 7.

2 SCALING AT IaaS, PaaS AND SaaS

The main idea of the presented architectural approach is to provide a platform for OSGi applications in the cloud. When leveraging services to the cloud it is often required to have the ability to scale resources according to the computational demand.

At an Infrastructure as a Service (IaaS) provider scaling can be achieved by providing bigger or smaller virtual machines (VMs) in terms of computational power or memory resources, by providing bigger or smaller storage nodes or by load balancing network traffic among different VMs or with different priority.

At the SaaS level scaling is often achieved by load balancing requests between more or fewer nodes of the same application. For this the application has to be either stateless, has to provide synchronized states among all instances of the application or has to store its current state on client side (which is often done in web-application sessions).

PaaS management organizes scaling on the IaaS (vertically) and SaaS level (horizontally) and is often used to automate the scaling process. The AAL service developer wants to use the scaling service from the PaaS to develop and provide an AAL service to the customer.

Strictly speaking PaaS just provides platforms for development or deployment (Mell and Grance, 2011). One scaling scenario would be to request a bigger platform to test or run an application. This would translate to a bigger VM on the IaaS level. To address user load, a scaling scenario would be to start new nodes of the same application behind a load balancing infrastructure on high demand or to stop the nodes when there is a decline of demand for the application. This would trigger the SaaS scaling like explained before and also be coordinated by the PaaS management environment.

The focus of this approach lies on the demand of a growing modular application on the computational resources itself. By adding more functionality to the modular application, the provided environment on one platform node can get too small with respect to memory or CPU power. The presented research supports this scenario by scaling out horizontally to being able to put the new module on a new node in a distributed OSGi platform (see Fig. 1).

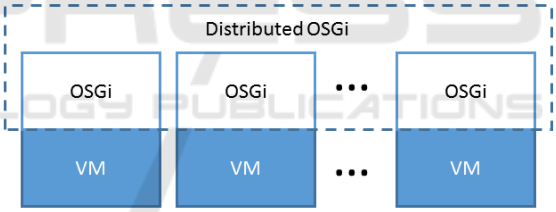

Figure 1: Horizontal Scaling with Distributed OSGi.

In a distributed OSGi environment, different OSGi VMs and their running bundles are connected to each other to compose one big OSGi environment by imports and exports of endpoints. Communication between the different nodes is usually done by calling HTTP interfaces. The OSGi Core Specification 4.3 (OSGI Alliance, 2011) introduces the main concept of distributed OSGi but does not recommend a specific way of implementation for the required components and functionality.

As shown in figure 2, a *Provider Bundle* in *Node 1* exports an OSGi service as an interface by using additional service properties (e.g., which interface is accessible remotely). This interface is registered at the *Distribution Provider* within the node and an *Endpoint* for a remote call of the service is created. Through a *Discovery Service* this existing *Endpoint* is announced at the *Distribution Provider* of the remote node, where the *Distribution Provider* creates a

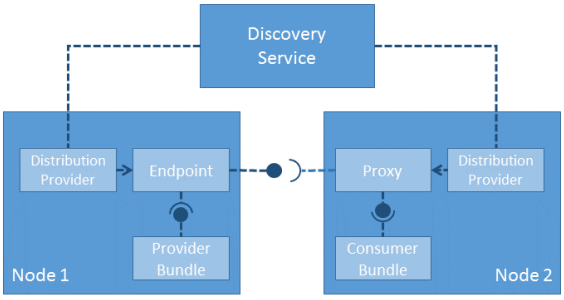

Figure 2: Basic Functionality of Distributed OSGi (based on (Apache Software Foundation, 2015b)).

Proxy for the remote service. This *Proxy* can be imported by a *Consumer Bundle* like a locally deployed bundle. The *Proxy* and *Endpoint* provide the implementation for remote message exchange.

The OSGi Compendium Specification 4.3 (OSGi Alliance, 2012) splits the *Distribution Provider* into several modules, to make it possible to enhance or exchange parts of the *Distribution Provider*. The *Discovery* module notifies *Endpoint Listeners* upon detection of available *Remote Endpoints*. The *Topology Manager* uses also an *Endpoint Listener* to monitor remote OSGi services and is able to monitor locally available OSGi services. The creation and destruction of *Endpoints* and *Proxies* is delegated to the *Remote Service Admin* module.

The project Apache CXF (Apache Software Foundation, 2015b) has implemented these components in a framework to support distributed OSGi in several specification compliant OSGi platforms (e.g., Equinox, FELIX or Knopflerfish). The *Discovery* module is implemented with an Apache Zookeeper server (Apache Software Foundation, 2015), to discover and announce endpoints in a highly dynamic environment.

3 RELATED WORK

Existing platforms in the field of AAL are mainly focused on delivering customizable middleware for smart home environments and the whole computing power has to be installed in the living environment of the end-user himself. Example projects are SOPRANO (Balfanz et al., 2008), AMIGO (Janse, 2008), ProSyst (Petzold et al., 2013) and universAAL (Sadat et al., 2013) that are providing middleware based on OSGi. They introduce techniques (e.g., software agent platforms) to gather user data, do reasoning based on ontologies and recognize events or incidents that lead to corresponding system reactions.

The projects that introduce cloud functionality are few and use the cloud-services for specific problems: A platform approach to share health data in the cloud in a secure way is presented by Kim et al. (Kim et al., 2012). The patient-centric solution is entirely governed by the patient and provides strong security and privacy characteristics. Health data can be shared between hospitals, trained care-personnel or relatives to indicate changes in the health conditions amongst the different support groups of the user.

The extensible OSGi-based architecture of Ekonomou et al. (Ekonomou et al., 2011) is focused on the integration of new devices by using a cloud-based service for discovering drivers for highly heterogeneous smart home systems in a manual, semi-automatic and automatic way.

The project Cloud-oriented Context-aware Middleware in Ambient Assisted Living (CoCaMAAL) (Forkana et al., 2014) uses cloud services for the context generation and classification of incidents. Data of installed sensors and devices in the smart living environment is gathered by a *Data Collector* on-site and transferred to a context aggregator in the cloud that sends back appropriate actions which are performed inside the users environment.

AIOLOS (Verbelen et al., 2012), a framework for scalable component-based cloud applications, focuses on offloading demanding tasks from mobile devices to remote VMs. The middleware based on OSGi uses *Remote Endpoints* to achieve this functionality. However, automatic scaling of the provided environment is so far not discussed in this approach.

Paul Bakker and Bert Ertman describe a way how to build a modular cloud app with OSGi (Bakker and Ertman, 2013). They use Amdatu (Amdatu, 2015) as a tool to provide the needed functionality for enabling OSGi applications to work on the SaaS layer (e.g., RESTful webservices, support for NoSQL-DBs) by being able to scale horizontally across nodes. To manage this scalability they use Apache ACE (Apache Software Foundation, 2015a) as a deployment-tool in conjunction with the auto scaling (Amazon Web Services, 2015a) and elastic load balancing (Amazon Web Services, 2015c) provided by Amazon AWS in the IaaS layer. This setup is used by PulseOn (Luminis, 2015), an e-learning application that is deployed with different compositions of functionality to different schools in the Netherlands and worldwide. The scaling is done by adding more nodes of the same application during periods where there is higher load (e.g., during school hours).

Although many concepts for OSGi applications in the cloud and programming guidelines of Bakker and Ertman can be applied to the SaaS layer of speciAAL, their concept differs from the presented approach of

scaling the environment by adding nodes to a single combined distributed environment.

4 ARCHITECTURE OF speciAAL

Figure 3 shows an overview of the speciAAL architecture. The architecture is divided into three layers: The IaaS layer, the PaaS layer including the Paas Manager and the SaaS layer. To explain the details of each layer, the different components are described in the following subsections.

4.1 IaaS Layer

The IaaS layer is considered to be an Private Cloud infrastructure provider (e.g., an OpenStack installation hosted at the institution) or a Public Cloud infrastructure provider (e.g., Amazon AWS, Rackspace). IaaS providers support starting and stopping virtual machines based on specified VM templates (compute power and storage services), network routing services, database and persistent datastore services, accounting services and monitoring.

The IaaS layer is controllable through command line tools (Amazon Web Services, 2015b) or API calls (The OpenStack project, 2015) to manage nodes based on decisions made in the PaaS layer. To be able to control different APIs of different IaaS providers there are tools, that wrap the system specific or vendor specific API for being able to exchange the IaaS infrastructure based on different service requirements. jClouds (Apache Software Foundation, 2015d) or BOSH (Cloud Foundry, 2015) try to standardize the usage of Cloud infrastructure APIs.

4.2 PaaS Layer

Open PaaS management systems simplify the provisioning of developed applications (Apache Software Foundation, 2015c; Linux Foundation Collaborative Projects, 2015). It reduces deployment time of a platform or a framework needed to run an application. For this the PaaS management systems can be configured to certain environments (e.g., PHP, JVM, Ruby, OSGi) that are ready to be started and used by application developers. This can reduce deployment time for the developer or the system operator and therefore also cost for the company they work in. Open PaaS management should be IaaS provider independent to enable provisioning of platforms on different IaaS environments like OpenStack, Open Nebula or Amazon AWS. The presented PaaS layer architecture is based on Apache Stratos (Apache Software Foundation, 2015c) and customized with components needed to provide horizontal scaling of distributed OSGi.

The *Manager* (see Fig. 3) is the central component for interacting with the PaaS environment. The *Manager* provides a UI or console line interface (CLI) for registered PaaS users. It holds information for available platform environments, scaling and deployment policies and is the central component to deploy environments or load balancers. The *Manager* provides tools for multi-tenancy: virtual computing resources and platform environments can be administered and shared among PaaS users, for deploying the applications, but also divided to provide specialized platforms or set different resource limits to various developer groups.

The *Cloud Controller* component communicates with the IaaS infrastructure and holds all the topology information of the PaaS system. The communication with the IaaS layer is done through an implementation layer and can be provided by the aforementioned jClouds. This makes the API of the IaaS layer accessible to the PaaS layer and allows management of different VMs or other service functionality (e.g., network routing) of the IaaS provider. All information retrieved from the IaaS is stored in the topology configuration at the *Cloud Controller* and can be published to other components that need to be aware of changes in the topology of the PaaS system. Examples for components where topology data is crucial information to work properly are the *Load Balancer* or the *Distribution Coordinator* component.

The *Distribution Coordinator* holds a set of deployable artifacts to update the deployed platforms. This component can also be used to automate the deployment process for the complete application on all subscribed running nodes. The artifacts are usually received from a connected repository service and deployed based on deployment policies, by events triggering the deployment or by manual commands to the *Manager* component. The *Distribution Coordinator* is triggered on start-up of a new node and is considered to hold all nodes of the same type at the same deployment state by the concept of deployment synchronization.

To take advantage of the possibility to add and remove computational resources to or from an application or distributed environment, the *Auto Scaler* component evaluates load and health information of the *Complex Event Processor* based upon policies (e.g., deployment policies or scaling policies). This evaluation is executed by a rule engine and applied according to the current topology retrieved from the *Cloud Controller* component. The information ex-

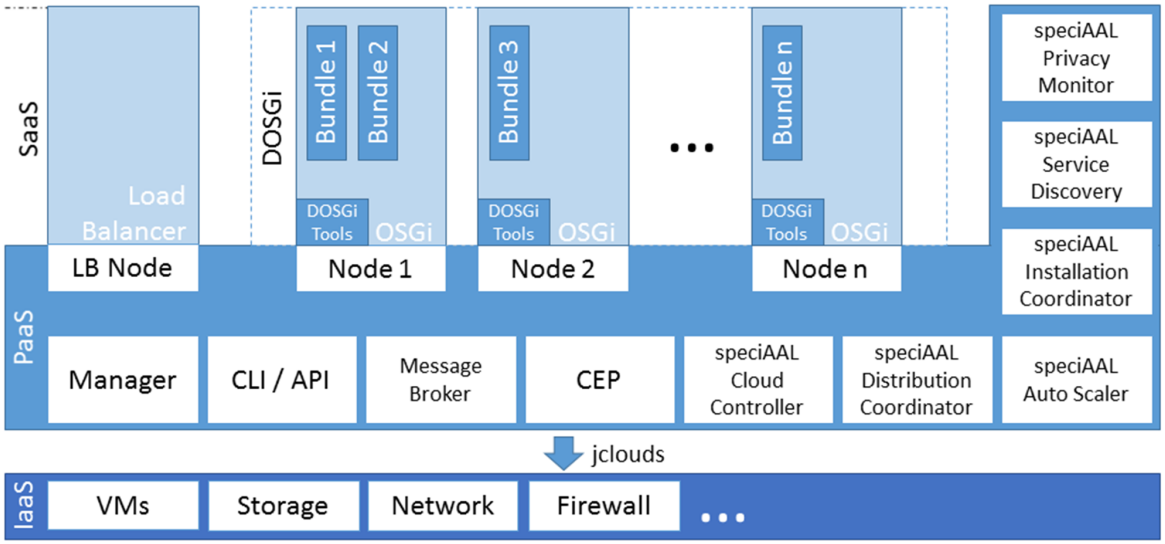

Figure 3: Distributed OSGi PaaS for speciAAL.

change between components is realized by a *Message Broker* component and a message bus based on a publish/subscribe pattern. This pattern enables the PaaS system to add or remove components (e.g., load balancers) dynamically by preserving the information exchange between components.

The *Nodes* (or *Cartridges*) are the actual environment in which the application is running. They are started, stopped and registered by the *Cloud Controller* module. In modern PaaS Management systems the focus lies on fast deployment of a completed application to a cloud node. For this purpose the *Distribution Coordinator* is able to provision the whole application out of its repository into an new node and to keep existing nodes up to date. In the presented approach this task is reduced to deploying just the environment and the minimum required components for setting up the initial bundles of the application in a distributed OSGi environment. The deployment of new functionalities is later triggered by functionality in the SaaS layer. This leads to a flexible and extendible application that is able to interact with the PaaS layer.

The *Service Discovery* is integrated in the PaaS layer but directly connected to the SaaS layer to translate the currently active application topology to coordinated actions in the PaaS layer. It is further described in section 5.

The *Installation Coordinator* is a new concept for the speciAAL architecture. It is able to receive installation requests out of the SaaS layer and evaluate the best place for installing a new functionality in the distributed environment of the application. Its main characteristics are detailed in section 5 and the process of installation is explained in section 6.

The *Privacy Monitoring Module* is able to collect log-data of actions within the SaaS layer and generating reports according to privacy policies and data access policies. The detailed architecture of this component is still work in progress and at this point it is shown for completeness.

4.3 SaaS Layer

In the presented approach the SaaS layer provides services for the platform speciAAL in a distributed OSGi environment. All running bundles are added up to one adapted application for the user with the functionalities configured to his needs. It consists of system bundles, core bundles and configurable application bundles.

The system bundles are part of the environment and provide general services for distribution (e.g., DOSGi bundles for Apache CXF), discovery (e.g., bundles for communication with Apache Zookeeper) and monitoring.

The core bundles are part of the application and provide core functionalities, like an Web-Interface, an Address Book, Communication bundles for Smart-Home Control and Sensors or a Customizing bundle.

The configurable application bundles are individually chosen by the user and installed via a Bundle Store. These bundles can be installed and configured during runtime and also have the ability to adapt to user behavior (Fredrich et al., 2014). They can be compared to apps on a smart phone, that can be installed, tested and also deleted if they do not provide a desired functionality.

5 LOAD BALANCING IN speciAAL

In figure 3 some components are marked as "speciAAL" components: The *Cloud Controller*, *Distribution Coordinator*, *Auto Scaler*, *Installation Coordinator* and *Service Discovery*. These components have to be adopted to provide the required functionality in order to provide the distributed load balancing approach for speciAAL as described in section 4.

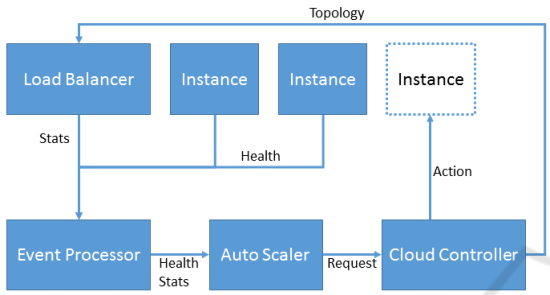

Figure 4: Auto Scaling in Apache Stratos.

In Apache Stratos scaling is achieved by creating or destroying instances (so called cartridge instances) of an application (Fig. 4): A real time event processor receives statistical information (e.g., requests or failed requests) of the *Load Balancer* component and the health status (e.g., load average, memory consumption or request count) of the running cartridge instances. This data is evaluated, summarized and published to a *Health Stats* topic in the *Message Broker*. The *Auto Scaler* receives this information and decides, backed by a *Rule Engine*, whether new nodes are needed or existing nodes are expendable. The *Auto Scaler* then sends a request to the *Cloud Controller* to create or destroy instances. After the performed action the altered topology information is published to the *Message Broker* and the *Load Balancer* is updated with the new topology.

One simple solution to provide horizontal scaling for a distributed OSGi environment would be, to work with small nodes and put each new part of the application on a single node. On installation of a new functionality the environment would be extended by a node and the corresponding bundles would be deployed on the started node. It would then be easy to delete the functionality by simply destroying the node. However, this would lead to a fragmented environment and unbalanced resource use across the application. The presented approach is focusing on making use of the available computational resources of one node before extending the environment by adding another node.

For speciAAL the *Distribution Coordinator*, *Service Discovery* and *Installation Coordinator* have to be added to the work flow of auto scaling (see Fig. 5):

Analogous to the *Cloud Controller* holding the topology of VMs and network interfaces on the PaaS and IaaS layer, the *Service Discovery* holds the topology for all of the distributed services in the SaaS layer. On creation of a new node for the distributed OSGi environment, it is getting contacted by a previously configured startup script and the new node and further installed services will be updated in the topology and service registry.

Furthermore, the *Service Discovery* is also monitoring the availability of the registered remote services on the node: If the node is deleted it will also de-register the node and all formerly running remote services on this missing node.

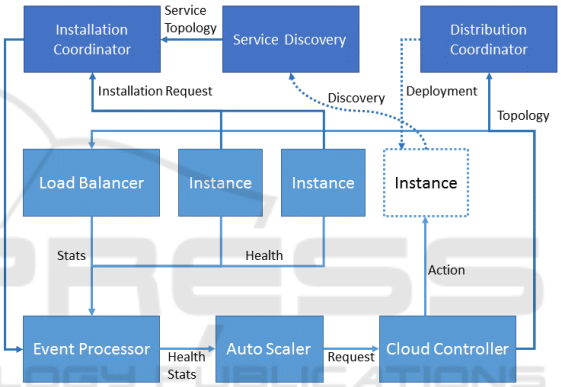

Figure 5: Auto Scaling in speciAAL.

The *Distribution Coordinator* has to configure and setup a new node after starting to provide the required services to integrate the node in the distributed environment. As described before it has the capability to keep the nodes on the same patch-level (distribution synchronization) and is able to push configuration updates as well (e.g., on what address or port a node is able to contact the *Service Discovery*). For speciAAL the *Distribution Coordinator* is used to set up the initial environment with all services that are needed for the application for configuration by the end user (e.g. authentication bundle, database access agent bundle, web interface bundle and a installation wizard bundle). Later the *Distribution Coordinator* provides additional nodes that are stripped down to only provide the minimal configuration for extending the platform as a distributed OSGi environment.

The *Installation Coordinator* plays a central role in this setup as it enables for detailed decisions where to install a new service. Before the bundle installation is performed, load information on the nodes is evaluated and based on this different actions are ex-

ecuted: If the nodes are able to run another service, it receives the information where (on which node) to install the new bundle in the distributed environment. If the nodes are on a load limit, it triggers the extension of the environment by a new node and receives the information to install the bundle on the newly provisioned node.

To further broaden the basis for these decisions, the *Complex Event Processor* has to be extended to collect information from inside the distributed OSGi environment (e.g., request/response time between servers and services) as well as from the *Service Discovery* component (e.g., registration or de-registration of services and nodes).

6 SCENARIO-BASED EVALUATION

The auto scaling during operation is influenced by two main aspects: User triggered events or monitoring events. If a user adds a new functionality to the platform or deletes an existing functionality from the platform, the *Auto Scaler* component has to decide whether a rebalancing of nodes has to be performed. Besides this, monitoring events, like high load, low load or health issues, are gathered at the *Complex Event Processor* and can lead to a reevaluation at the *Auto Scaler* component. In addition, there is load balancing during deployment time as well.

To show the systems flexibility, we describe the starting situation, occurring events and the involved components of the system.

6.1 Setting up the Application

If a new end-user wants to use speciAAL, a new PaaS environment is created. This means, that the basis template is deployed by the *Deployment Coordinator* to a newly created VM by the *Cloud Controller*. This action is triggered by the *Manager* and the node will be already enabled for the distributed environment. Although there is just one node in the beginning, the *Service Discovery* has to be active and the node has to be registered for being able to access remote services later on. The new environment is created with the goal to give each user (e.g., an elderly person) an isolated, customizable and adoptable environment.

6.2 Installing a New Service

When the user wants to add a new service to the application, the installation request is sent to the *Installation Coordinator*. The *Installation Coordinator* informs the *Complex Events Processor* which combines the request with the current health and load information of the nodes and triggers the evaluation process of the *Auto Scaler*. Based on the rules engine, the decision-making can lead to one of the two results:

- There are still resources available on a specific node. This information is sent back to the *Installation Coordinator* and the bundle installation is performed on the specified node. On start of the new bundle, exported services are registered by the *Service Discovery* and can be used by other services or the end-user.
- All nodes are exceeding a defined threshold and the new functionality is likely to exceed the computational resources of a node. The *Auto Scaler* requests an additional node (VM) at the *Cloud Controller*. After the node is started, the *Deployment Coordinator* gets an updated topology information and deploys the extension template to the new node. After the new node is registered at the *Service Discovery*, the *Installation Coordinator* is updated to install the requested bundle on the new node. After deploying and starting the bundle, the exported services are registered by the *Service Discovery* and can be used by other services or the end-user.

6.3 Removing an Existing Service

If an end-user decides to remove a certain functionality out of his application, the associated bundles that are no longer needed by other services are stopped and removed. This triggers the de-registration in the *Service Discovery* and the exported remote services are no longer present in the system.

One special case to this scenario is when the last exported remote service on a node is deleted. This may indicate an orphaned node that might as well be deleted. To verify this sufficient condition, the current set of bundles on the node has to be compared to the set of bundles of a node extension template with only the vital services for distribution.

- If the set is the same, the node is removed from the distributed environment by simply deleting the VM on the IaaS level by the *Cloud Controller*.
- Otherwise, no further actions are required at this phase and the node will be re-evaluated in the scenario “Low Load” (see Section 6.5).

6.4 High Load

At first this scenario does not differ from the auto scaling mechanism of Apache Stratos (see section *speciaal*). One node is monitored as having high cpu load,

memory swapping or long running requests. The *Auto Scaler* triggers the creation of a new node at the *Cloud Controller* and it is deployed as extension node by the *Deployment Coordinator*. As Apache CXF has the ability to monitor request/response times on a service level inside the OSGi environment, this is used in a second step to get an estimation, what bundles are causing the overload this time.

As the new node is started up and registered bundles are "moved" to the new node. This is realized by duplicating bundles at first on the new node and deleting the bundles on the busy node afterwards to make sure, that a remote service to handle the requests is always available in the environment.

There are two conceivable strategies to move services from a node with high load:

- Moving the service that is detected as causing the high load. This can mean that the long running requests are prohibiting the service from stopping and the moving scenario will take longer than expected.
- Freeing up resources on the node by moving other services that are not causing the high load. This will leave the problematic service untouched and running but is not helpful if the service is crashed and producing the high load (e.g., by continuously looping).

6.5 Low Load

The opposite situation is too much idle time for one node in the environment. The environment can then be consolidated to fewer nodes. But before a node is ready to be deleted we have to provide a way for reassuring that all needed services are migrated: As the *Service Discovery* knows what remote services are running on a specific node, the according OSGi bundles have to be started on a different node prior to stopping the services on the node that is to be deleted.

The other information that is crucial is, from which node to which other node the migration of services is applied. There are different requirements that have to be considered for the migration of services:

- For the migration scenario there have to be at least two nodes with low load. If it is just one node and the services are consolidated on another node with medium load, this can lead to a higher than expected load on the target. In this case, the "High Load" scenario would be triggered, and this would lead to another migration in the opposite direction.
- The definition of threshold for low load should be set low enough, that it is viable to consolidate the two nodes into one.

If the *Complex Event Processor* gets notice of two nodes with low load, it triggers the consolidation process on the *Installation Coordinator*. This is applied the same way like in the High Load scenario: The corresponding bundles have to be started on the target node, before the bundles can be stopped and deinstalled on the source node. After the last remote service is de-registered from the *Service Discovery* the node is ready for deletion and can be shut down and removed by the *Cloud Coordinator*.

7 CONCLUSIONS

We presented an architecture of a PaaS Management platform for speciAAL, a SaaS scenario based on distributed OSGi. The main focus of the architecture lies on simplified deployment of the application and on elasticity of the distributed OSGi environment by utilizing the IaaS layer functionalities and monitoring the SaaS layer events.

These events and collected statistics can be used to do load-balancing during deployment and operation of the application. This paper presented the main functionality and the components that are involved in the load-balancing and auto scaling process and defined the differences that have to be considered when scaling an application in a distributed environment.

In the scaling process, there are still particularities that have to be further examined: Should there be a migration of services based on a mobility strategy (e.g., are there sets of services that have to remain on the same node?) or policy (e.g., some services are not allowed to leave a node, because it is the primary configuration node or the first/last of the application).

Another challenge will be if it is viable to combine *load balancing during deployment* (applied to the IaaS layer) with *load balancing of requests* by multiplying the involved bundles on the platform and redirecting requests between exported services. A mechanism for failover-handling backed by the *Service Discovery* and common OSGi mechanics is another interesting topic for future research.

REFERENCES

Apache Software Foundation (2015). ZooKeeper: Because Coordinating Distributed Systems is a Zoo. https://zookeeper.apache.org/doc/r3.5.1-alpha/.

Amazon Web Services (2015a). Auto Scaling. https://aws.amazon.com/de/autoscaling/.

Amazon Web Services (2015b). AWS Command Line Tools. http://docs.aws.amazon.com/general/latest/gr/GetTheTools.html.

Amazon Web Services (2015c). Elastic Load Balancing - Cloud-Load Balancer. https://aws.amazon.com/de/elasticloadbalancing/.

Amdatu (2015). Amdatu – Philosophy. http://www.amdatu.org/philosophy.html.

Apache Software Foundation (2015a). Apache ACE. https://ace.apache.org/.

Apache Software Foundation (2015b). Apache CXF – Distributed OSGi. https://cxf.apache.org/distributed-osgi.html.

Apache Software Foundation (2015c). Apache Stratos - Open Enterprise PaaS. http://stratos.apache.org/.

Apache Software Foundation (2015d). jclouds - The Java Multi-Cloud Toolkit. https://jclouds.apache.org/.

Bakker, P. and Ertman, B. (2013). *Building Modular Cloud Apps with OSGi*. O'Reilly Media, 1 edition.

Balfanz, D., Klein, M., Schmidt, A., and Santi, M. (2008). Partizipative Entwicklung einer Middleware für AAL-Lösungen: Anforderungen und Konzept am Beispiel SOPRANO. In *GMS Medizinische Informatik, Biometrie und Epidemiologie*, volume 4(3), http://www.egms.de/static/de/journals/mibe/2008-4/mibe000078.shtml.

Cloud Foundry (2015). bosh. https://bosh.cloudfoundry.org/.

Ekonomou, E., Fan, L., Buchanan, W., and Thüemmler, C. (2011). An Integrated Cloud-based Healthcare Infrastructure. In *Third IEEE International Conference on Cloud Computing Technology and Science*, pages 532–536. IEEE Computer Society.

Forkana, A., Khalil, I., and Tari, Z. (2014). CoCaMAAL: A cloud-oriented context-aware middleware in ambient assisted living. In Fortino, G. and Pathan, M., editors, *Future Generation Computer Systems*, volume 35, pages 114–127.

Fredrich, C., Kuijs, H., and Reich, C. (2014). An ontology for user profile modeling in the field of ambient assisted living. In Koschel, A. and Zimmermann, A., editors, *SERVICE COMPUTATION 2014, The Sixth International Conferences on Advanced Service Computing*, volume 5, pages 24–31. IARIA.

Grauel, J. and Spellerberg, A. (2007). Akzeptanz neuer Wohntechniken für ein selbstständiges Leben im Alter. In *Zeitschrift für Sozialreform*, volume Heft 2 Jg. 53, pages 191–215.

Janse, M. D. (2008). AMIGO - Ambient Intelligence for the networked home environment. Final activity report.

Kim, J. E., Boulos, G., Yackovich, J., Barth, T., Beckel, C., and Mosse, D. (2012). Seamless Integration of Heterogeneous Devices and Access Control in Smart Homes. In *Eighth International Conference on Intelligent Environments*.

Kuijs, H., Rosencrantz, C., and Reich, C. (2015). A Context-aware, Intelligent and Flexible Ambient Assisted Living Platform Architecture. In *Cloud Computing 2015: The Sixth International Conference on Cloud Computing, GRIDs and Virtualization*. IARIA.

Linux Foundation Collaborative Projects (2015). Cloud Foundry — The Industry Standard For Cloud Applications. https://www.cloudfoundry.org/.

Luminis (2015). PulseOn - Personalized Learning — Maximizing human potential through personalized learning. http://www.pulseon.nl/en/.

Mell, P. and Grance, T. (2011). The NIST Definition of Cloud Computing. Special Publication 800-145, National Institute of Standards and Technology.

OSGi Alliance. Smart home market. http://www.osgi.org/Markets/SmartHome.

OSGi Alliance. The OSGi Architecture. http://www.osgi.org/Technology/WhatIsOSGi.

OSGI Alliance (2011). OSGi Service Platform Core Specification. Technical Report Release 4, Version 4.3, The OSGi Alliance.

OSGi Alliance (2012). OSGi Service Platform Service Compendium. Technical Report Release 4, Version 4.3, The OSGi Alliance.

Petzold, M., Kersten, K., and Arnaudov, V. (2013). OSGi-based E-Health / Assisted Living. Whitepaper, ProSyst, http://www.prosyst.com/fileadmin/ProSyst_Uploads/pdf_dateien/ProSyst_M2M_Healthcare_Whitepaper.pdf.

Sadat, R., Koster, P., Mosmondor, M., Salvi, D., Girolami, M., Arnaudov, V., and Sala, P. (2013). Part III: The universAAL Reference Architecture for AAL. In Sadat, R., editor, *Universal Open Architecture and Platform for Ambient Assisted Living*. SINTEF.

The OpenStack project (2015). Application Programming Interfaces. http://developer.openstack.org/.

United Nations (2001). World Population Ageing: 1950-2050. Report, UN: Department of Economics and Social Affairs - Population Division, http://www.un.org/esa/population/publications/wpolda geing19502050/.

Verbelen, T., Simoens, P., Turck, F. D., and Dhoedt, B. (2012). Aiolos: Middleware for improving mobile application performance through cyber foraging. *Journal of Systems and Software*, 85(11):2629 – 2639.

ZAFH-AAL (2014). ZAFH-AAL - Zentrum für angewandte Forschung an Hochschulen für Ambient Assisted Living. http://www.zafh-aal.de.