Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Tobias Eichler

Agentenbasierte Middleware zur Entwicklerunterstützung in einem Smart-Home-Labor

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Tobias Eichler

Agentenbasierte Middleware zur Entwicklerunterstützung in einem Smart-Home-Labor

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Kai von Luck
Zweitgutachter: Prof. Dr. Gunter Klemke

Eingereicht am: 2. Oktober 2014

Tobias Eichler

Thema der Arbeit

Agentenbasierte Middleware zur Entwicklerunterstützung in einem Smart-Home-Labor

Stichworte

agentenbasiert, Middleware, Smart-Environments, Entwicklerunterstützung

Kurzzusammenfassung

Software für Smart-Environments besteht oft aus vielen Komponenten, die kontextabhängige Entscheidungen treffen und zusammen ein komplexes System bilden. Diese Arbeit befasst sich mit dem Entwurf einer agentenbasierten und verteilten Middleware für Smart-Home-Labore, mit dem Ziel, die Entwicklung neuer Komponenten zu erleichtern und die Komplexität des Systems zu reduzieren. Die Middleware wurde innerhalb des Living Place Hamburg evaluiert. Latenzmessungen zeigen, dass Nachrichten im System auch noch bei sehr hohen Zahlen von Agenten und Kommunikationsgruppen Latenzen aufweisen, die sich für Benutzerinteraktion eignen. Durch die Auswertung mehrerer mit Hilfe des neuen Systems durchgeführter Projekte wurde gezeigt, dass die Entwicklung neuer Projekte beschleunigt werden kann.

Tobias Eichler

Title of the paper

Agent-based Middleware for Supporting Developers in a Smart Home Laboratory

Keywords

agent-based, middleware, smart environments, support for developers

Abstract

Software for smart environments often consists of many context aware components that together form a complex system. This thesis deals with the design of an agent-based and distributed middleware for smart home laboratories, with the aim to speed up the development of new projects and to reduce the complexity of the system. The middleware was evaluated in the Living Place Hamburg. Latency measurements show that even with high numbers of agents and communication groups the message latency is suitable for user interaction. By evaluating several completed projects using the new system it was shown that the development of new projects can be accelerated.

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Zielsetzung . . . 2

1.2 Gliederung . . . 3

2 Analyse **5**

2.1 Smart-Environments und Context-Aware-Systems . . . 5

2.1.1 Smart-Homes . . . 6

2.1.2 Kontext . . . 6

2.1.3 Sensoren, Aktoren und Kontextinterpreter . . . 7

2.1.4 Kontextverarbeitung und Interpretation . . . 7

2.1.5 Allgemeiner Aufbau eines verteilten Context-Aware-Systems . . . 8

2.2 Agentenbasierte Systeme . . . 10

2.2.1 FIPA - Agent Communication Language . . . 10

2.2.2 JADE . . . 11

2.2.3 Bewertung . . . 12

2.3 Verwandte Arbeiten . . . 13

2.3.1 CHIL . . . 13

2.3.2 Agentenbasierte Middleware für zusammenarbeitende Smart-Objects . 15

2.3.3 AMUSE . . . 15

2.3.4 Vergleich . . . 17

2.4 Living Place Hamburg . . . 17

2.4.1 Javascript Object Notation - JSON . . . 18

2.4.2 Aktuelle Architektur . . . 20

2.5 Anforderungen in einer Smart-Home-Laborumgebung . . . 22

2.6 Analyse der Entwicklungsprozesse . . . 23

2.6.1 Szenario: Eine neue Projektarbeit im Living Place . . . 23

2.6.2 Einteilung in Phasen . . . 26

2.7 Ergebnis der Anforderungsanalyse und Abgrenzung . . . 29

2.7.1 Anforderungen . . . 29

2.7.2 Abgrenzung . . . 31

2.7.3 Zusammenfassung . . . 32

3 Design **33**

3.1 Designentscheidungen . . . 33

3.1.1 Auswahl des Nachrichtenformates . . . 33

3.1.2 Kontextverarbeitung durch Complex-Event-Processing 34
3.1.3 Laufzeitumgebungen 36
3.2 Middleware 39
3.2.1 Schnittstellen von Agenten 41
3.2.2 Kommunikation 43
3.2.3 Kontextverarbeitung 44
3.2.4 Laufzeitumgebungen 46
3.2.5 Speicherung von Zuständen durch Event-Sourcing 48
3.2.6 Monitoring von Agenten 49
3.2.7 Verteilte Ausführung 50
3.3 Framework zur Entwicklerunterstützung 52
3.3.1 Vom Aktor zum Agenten 52
3.3.2 Type-Classes 54
3.3.3 Funktionen 55
3.3.4 Komponenten- und Integrationstests 57
3.3.5 Konfigurationsmanagement 58
3.4 Tools zur Entwicklerunterstützung 60
3.4.1 Versionsverwaltung 61
3.4.2 Build-Tool 62
3.4.3 Build-Server 63
3.4.4 Artefakt-Repositories 64
3.4.5 Webinterface der Middleware 65
3.5 Integration in vorhandene Systeme 65
3.6 Mögliche Weiterentwicklungen 67

4 Evaluation **68**
4.1 Fallstudien 68
4.1.1 Fernbedienung 68
4.1.2 Aktoren 70
4.1.3 Sensoren 71
4.2 Latenzzeiten und Skalierbarkeit des Systems 72
4.2.1 Mögliche Optimierungen 77
4.3 Erfahrungen im Team 78
4.4 Fazit 80

5 Schluss **82**
5.1 Zusammenfassung 82
5.2 Ausblick 83

Abbildungsverzeichnis

2.1 Allgemeiner Aufbau eines Context-Aware-Systems. 9
2.2 Beispiel für eine FIPA-ACL-Nachricht. 11
2.3 Die FIPA-Agent-Plattform. 12
2.4 Der Agenten-Lebenszyklus in CHIL. 14
2.5 Das AMUSE-Framework. 16
2.6 Das Living Place Hamburg. 18
2.7 Innenansichten des Living Place Hamburg. 19
2.8 Beispiel für eine JSON-Nachricht. 19
2.9 Alte Architektur des Living Places. 20

3.1 OSGi-Integration im Living Place. 38
3.2 Blackbox-Ansicht der Middleware. 40
3.3 Ein Teil der Schnittstellendefinition zur Lichtsteuerung im Living Place. 42
3.4 Protokollstack mit Serialisierung von JSON-Nachrichten. 43
3.5 Integration von Complex-Event-Processing in das System. 45
3.6 Fehlertoleranz durch Event-Sourcing. 49
3.7 Aufbau eines Middleware-Clusters mit zwei Knoten und einer Laufzeitumgebung. 51
3.8 Formatierung als Beispiel für die Verwendung von Type-Classes in Scala. . . . 54
3.9 Einsatz eines Agenten zur Nachrichtenfilterung. 59

4.1 Benutzeroberfläche der Fernbedienung des Living Place. 69
4.2 Test mit acht Middleware-Knoten und sieben Laufzeitumgebungen mit unterschiedlicher Anzahl von Agenten. 74
4.3 Test mit 5.000 Agenten und unterschiedlicher Anzahl von Middleware-Knoten mit Laufzeitumgebung. 76
4.4 Test mit 10.000 Agenten und unterschiedlicher Anzahl von Middleware-Knoten mit Laufzeitumgebung. 76

1 Einleitung

Mit der fortschreitenden technologischen Entwicklung können immer kleinere und leistungsfähigere Computer gebaut werden. Es entstehen kleine Computer, die den Menschen in seinem Leben unterstützen, ohne aufzufallen oder ihn abzulenken. Dieser Vorgang wurde bereits in den neunziger Jahren von Weiser (1999) als „Ubiquitous Computing“ beschrieben.

Persönliche Computer werden immer weiter durch Smart-Objects ersetzt. Smart-Objects sind Gegenstände, die um eingebettete Computer erweitert werden, und damit die Fähigkeiten erhalten, Informationen zu ermitteln, diese zu kommunizieren und Entscheidungen zu treffen (vgl. López u. a., 2011). Beispiele hierfür sind intelligente Implantate, kleine Sensoren, die in Kleidung eingenäht werden, oder auch miteinander vernetzte Stromzähler. Das Verhalten von Smart-Objects wird meistens durch einen sich wiederholenden Sense-Reason-Act-Zyklus definiert. Dabei wird versucht, durch die Auswertung von Sensorinformationen für den aktuellen Kontext adäquate Aktionen auszuführen.

In der aktuellen Entwicklung werden immer mehr Gegenstände als „smart“ bezeichnet. Ein Beispiel hierfür ist die Initiative „Smarter Planet“[1], die von IBM gegründet wurde und sich mit den verschiedensten intelligenten Objekten wie dem Stromnetz, der Wasserversorgung, der Verkehrssteuerung und ökologischen Gebäuden auseinandersetzt. Ergänzend dazu finden immer wieder Konferenzen und Workshops zum Thema „Smart-Objects“ statt, wie zum Beispiel die „International Conference on Intelligent User Interfaces“ und der „Workshop on Interacting with Smart Objects“[2].

Ein weiterer Begriff, der in diesem Zusammenhang oft auftaucht, ist „Internet der Dinge“, womit die Vernetzung von Smart-Objects über das Internet gemeint ist (vgl. Uckelmann u. a., 2011). Damit werden Gegenstände der realen Welt über Computernetzwerke zugänglich. Die Verwendung des IPv6-Protokolls ermöglicht es aufgrund des extrem großen Adressraumes,

[1] IBM Smarter Planet Initiative - http://www.ibm.com/smarterplanet/us/en/overview/ideas; letzter Zugriff: 26.09.2014

[2] IUI 2014 Workshop on Interacting with Smart Objects - http://www.smart-objects.org; letzter Zugriff: 26.09.2014

auch kleinsten Geräten global eindeutige IP-Adressen zuzuweisen. Das Ziel bei der Vernetzung ist es, intelligentes Handeln durch Zusammenarbeit zu erreichen.

Durch die Vernetzung von Smart-Objects zu komplexen verteilten Systemen entstehen neue Herausforderungen. In diesen Umgebungen kann eine Analyse des Systemverhaltens oder die Identifikation von Fehlern eine schwierige Aufgabe darstellen. Damit hierfür die vorhandene Theorie zu Multiagentensystemen verwendet werden kann, können Smart-Objects auch als Agenten betrachtet werden, die durch den Austausch von Nachrichten zusammenarbeiten. Zur Reduzierung der Komplexität und Erleichterung der Entwicklung wird hier oft eine Middleware eingesetzt. Eine Middleware ist eine Sammlung von Diensten, die unter anderem dazu eingesetzt werden können, zwischen Anwendungen oder Komponenten zu vermitteln.

Aus der Art der intelligenten Umgebung können sich einige spezielle Anforderungen an das System ergeben. Zum Beispiel handelt es sich bei Smart-Homes um privat genutzte Lebensräume, in denen im Gegensatz zu Arbeitsbereichen kaum vorgeschriebene Abläufe oder Regeln existieren, was die Entscheidungsfindung aufgrund von Kontextinformationen erschwert. Bei der Erforschung dieser Umgebung innerhalb von Laboren kommt hinzu, dass das System kontinuierlich weiterentwickelt, verändert und für Experimente verwendet wird. Smart-Homes bestehen meist aus vielen miteinander kommunizierenden Komponenten. Es ist die Aufgabe einer Middleware, eine Umgebung für die Zusammenarbeit zu schaffen und intelligentes Handeln des Systems zu fördern. Diese Problemstellungen finden sich auch im Living Place Hamburg, einem Smart-Home-Labor an der HAW Hamburg[3], wieder.

1.1 Zielsetzung

In dieser Arbeit soll eine agentenbasierte Middleware für Smart-Home-Labore konzipiert und entwickelt werden. Da es sich bei einem Smart-Home-Labor um eine Forschungseinrichtung handelt, ist dabei die Unterstützung von Entwicklern ein wichtiger Aspekt. Durch die Entwicklung des Systems für ein Smart-Environment ergeben sich weitere Anforderungen, die in dieser Arbeit analysiert und umgesetzt werden sollen. Ein Ziel ist es, dass die Middleware in der Lage ist, das System dynamisch an aktuelle Anforderungen verschiedener Stakeholder anzupassen. Hierfür soll eine Laufzeitumgebung in die Middleware integriert werden, die es ihr ermöglicht, die ausgeführten Komponenten zu überwachen und zu kontrollieren.

[3] Hochschule für Angewandte Wissenschaften Hamburg - http://www.haw-hamburg.de; letzter Zugriff: 26.09.2014

Die Entwicklerunterstützung soll durch die Bereitstellung eines Frameworks erreicht werden. Dieses Framework soll die Entwicklung weiterer Projekte beschleunigen und die Anbindung neuer Agenten an die Middleware erleichtern. Außerdem soll der Entwicklungsprozess mit einer Reihe von Tools unterstützt werden. Durch eine Anbindung der Tools an die Middleware und das Framework sollen Teile des Prozesses, wie zum Beispiel das Deployment, automatisiert werden.

Es ist geplant, die Evaluation der Arbeit in einem Smart-Home Labor, dem Living Place Hamburg, durchzuführen. Dafür sollen einige wichtige Komponenten des vorhandenen Systems auf die neue Middleware umgestellt werden. Das Framework soll innerhalb des Projektteams möglichst schon während der Entwicklung eingesetzt werden, um direkt auf Rückmeldungen eingehen zu können. Die im Rahmen dieser Arbeit entwickelte Software soll im Living Place installiert werden und für die Entwicklung zukünftiger Projekte bereitstehen.

1.2 Gliederung

Diese Arbeit besteht zusammen mit der Einleitung aus insgesamt fünf Kapiteln. Das erste Kapitel, die Einleitung, endet mit dieser Gliederung.

Das Kapitel 2 befasst sich zuerst mit der Analyse von Smart-Environments und agentenbasierten Systemen. Anschließend werden mehrere verwandte Arbeiten vorgestellt und es wird insbesondere auf das Living Place Hamburg eingegangen. Ausgehend von der Analyse werden Anforderungen an das System und die Middleware erarbeitet. Dabei werden auch die speziellen Anforderungen an eine Laborumgebung und der Ablauf der Entwicklungsprozesse mit einbezogen. Das Kapitel wird mit einer Zusammenfassung der Ergebnisse der Anforderungsanalyse abgeschlossen.

In Kapitel 3 wird das Design des Systems beschrieben. Dabei werden zuerst einige wichtige grundlegende Designentscheidungen getroffen und anschließend werden die Designs der Middleware und eines Frameworks für die Entwicklerunterstützung erläutert. Außerdem wird auf verschiedene Tools eingegangen, die zur Unterstützung des Entwicklungsprozesses bereitgestellt werden. Das Kapitel endet mit einer Beschreibung, wie eine Integration der Middleware in bereits vorhandene Systeme vorgenommen werden kann.

Die Evaluation des Designs erfolgt in Kapitel 4. Zuerst werden einige Agenten vorgestellt, die

als Fallstudien mit dem Framework entwickelt und im Labor installiert wurden. Das Verhalten der Middleware wird in einem Testszenario unter Last analysiert. Dabei werden die Latenzen der Nachrichten gemessen und die Skalierbarkeit des Systems wird untersucht. Anschließend geht es um die Erfahrungen anderer Projektteilnehmer mit dem entwickelten System und um ein bereits abgeschlossenes Projekt, welches mit dem Framework entwickelt wurde.

Das Kapitel 5 fasst alle Kapitel und die Ergebnisse der Arbeit zusammen. Abschließend erfolgt ein Ausblick auf weitere interessante Fragestellungen.

2 Analyse

In diesem Kapitel sollen die Anforderungen an ein System und insbesondere die Middleware in einem Smart-Environment analysiert werden. Zuerst wird dafür kurz der typische Aufbau eines Systems im Bereich von Smart-Environments vorgestellt. Darauf aufbauend werden agentenbasierte Systeme und Standardtechnologien, die in diesem Umfeld eingesetzt werden, untersucht. Anschließend werden verwandte Arbeiten und das Living Place Hamburg zusammen mit der aktuell dort vorhandenen Systemarchitektur vorgestellt und die daraus hervorgehenden Anforderungen analysiert. Weitere Anforderungen ergeben sich aus dem Entwicklungsprozess eines neuen Projektes im Living Place. Am Ende des Kapitels werden die Anforderungen zusammengefasst und das Thema der Arbeit wird weiter abgegrenzt.

2.1 Smart-Environments und Context-Aware-Systems

Smart-Environments sind Umgebungen, in denen Menschen durch sich intelligent verhaltende Soft- und Hardwarekomponenten unterstützt werden. Solche Umgebungen können zum Beispiel Büros, Konferenzräume oder Privatwohnungen sein. Mit der fortschreitenden technologischen Entwicklung werden die eingesetzten Geräte immer kleiner und verschwinden immer mehr aus der Wahrnehmung der Benutzer. Dies wird auch als „Pervasive“ oder „Ubiquitous Computing“ bezeichnet (vgl. Weiser, 1999).

Die Anzahl an Komponenten in einem Smart-Environment macht es dem Benutzer unmöglich, diese auf herkömmliche Arten, wie zum Beispiel mit Maus und Tastatur, zu bedienen. Der Benutzer sollte also ohne sein aktives Eingreifen und ohne die Notwendigkeit seiner Aufmerksamkeit unterstützt werden. Eine Möglichkeit dies zu erreichen, ist die Systeme kontextabhängig handeln und eigene Entscheidungen treffen zu lassen. Diese Anforderungen führen zur Entwicklung von Context-Aware-Systems (vgl. Schilit u. a., 1994).

2.1.1 Smart-Homes

Ein Smart-Home ist ein spezielles Smart-Environment, bei dem es sich um einen privat genutzten Lebensraum handelt. Zur Unterstützung der Bewohner werden in einem Smart-Home nicht nur Geräte der Hausautomatisierung oder Haushaltstechnik, wie zum Beispiel die Steuerung der Heizung, Beleuchtung oder Waschmaschine, sondern auch Anwendungssoftware, die ihr Verhalten an den aktuellen Kontext der Wohnung und des Bewohners anpasst, eingesetzt (vgl. Strese u. a., 2010; Robles und Kim, 2010). Neben der Unterstützung der Bewohner können durch die Technologie in Smart-Homes aber auch andere Vorteile, wie zum Beispiel das Einsparen von Energie, umgesetzt werden.

Ein Beispiel für ein Smart-Home in Form einer Laborumgebung ist das Living Place Hamburg (s. Kapitel 2.4).

2.1.2 Kontext

Eine häufig verwendete Definition von Kontext stammt von Abowd u. a. (1999).

> „Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.“ Abowd u. a. (1999)

Kontextinformationen können zur Beschreibung der Situation verwendet werden, in der sich eine Entität gerade befindet. Als ein Beispiel für Kontextinformationen wir oft die Position des Benutzers verwendet. Zur Beschreibung des Kontextes gehört aber deutlich mehr. Zu den primären Kontexttypen gehören neben der Position auch die Identität, Zeit und Aktivität (vgl. Abowd u. a., 1999). Dabei kann sich die Identität einer Person zum Beispiel aus Namen, Adressen und zusätzlichen personenbezogenen Informationen zusammensetzen.

Der Sonderforschungsbereich Transregio 62 (SFB 62) beschäftigt sich mit der Erforschung von Companion-Technologie (vgl. SFB 62). Die Forschungsgruppe besteht aus Wissenschaftlern verschiedenster Fachgebiete, die interdisziplinär zusammenarbeiten.

> „Das Forschungsvorhaben folgt der Vision, dass technische Systeme der Zukunft Companion-Systeme sind – kognitive technische Systeme, die ihre Funktionalität vollkommen individuell auf den jeweiligen Nutzer abstimmen: Sie orientieren sich

> an seinen Fähigkeiten, Vorlieben, Anforderungen und aktuellen Bedürfnissen und stellen sich auf seine Situation und emotionale Befindlichkeit ein.“ SFB 62

Der Begriff „Kontext“ kann also noch deutlich weiter gefasst werden. Die Definition ist abhängig von der konkreten Anwendung und dem Anwendungsgebiet. Eine Herausforderung für Context-Aware-Systems ist es, mit diesen Mengen an verschiedenen Informationen umgehen zu können und Entscheidungen auf deren Basis zu treffen.

2.1.3 Sensoren, Aktoren und Kontextinterpreter

Sensoren, Aktoren und Kontextinterpreter sind drei wichtige Bestandteile von Systemen im Bereich von Smart-Environments und bilden die Grundlage für Context-Aware-Systems (vgl. Meyer und Rakotonirainy, 2003). Sensoren liefern Messwerte über ihre Umgebung und machen sie für den Rest des Systems zugänglich. Aktoren werden zur Steuerung von Gegenständen eingesetzt. Sie können ihre Umgebung auf Befehl von anderen Komponenten verändern. Aktoren haben meistens einen Zustand und führen ihre Aktionen abhängig davon aus. Kontextinterpreter sind Komponenten, die Sensordaten weiterverarbeiten. Diese Weiterverarbeitung kann eine einfache Aggregation von verschiedenen Sensordaten sein. Es sind aber auch komplexe Interpretationen auf Basis der Sensordaten möglich. So können aus den Sensordaten Kontextinformationen erstellt werden, die zum Beispiel die Gesamtsituation der Umgebung beschreiben. Nicht jede Komponente kann klar einer Kategorie zugeordnet werden. Zum Beispiel kann eine Zustandsänderung eines Aktors als Sensorevent interpretiert werden.

2.1.4 Kontextverarbeitung und Interpretation

Die einfachste Art von Kontextinformationen sind Sensordaten. Dies sind unterschiedliche Informationen aus der Umgebung der Anwendung, die direkt von Hardware-Sensoren gemessen werden können. Dazu zählen zum Beispiel Positions- und Temperaturinformationen. Diese Daten können weiterverarbeitet, interpretiert und aggregiert werden, um Informationen der nächsten Ebene zu generieren.

Sensordaten weisen meist Fehler auf, die in einem weiteren Vorverarbeitungsschritt reduziert werden können. Bei Menschen in einer Wohnung ist anzunehmen, dass diese sich nicht schneller als mit einer bestimmten Geschwindigkeit fortbewegen können. Diese Information kann dazu benutzt werden, große Sprünge der Positionsdaten als Fehlmessungen zu identifizieren und herauszufiltern. Eine weitere Möglichkeit zur Verbesserung von Sensordaten ist es, einen Durchschnitt der letzten Messungen zu bilden, um die Schwankungen der Messwerte

zu reduzieren und Fehler auszugleichen. Dieses Verfahren wird auch als Sensor-Smoothing bezeichnet.

Aus einfachen Sensordaten können durch Interpretation Informationen einer höheren Ebene gewonnen werden. Mit einfachen mathematischen Verfahren lassen sich zum Beispiel aus den Temperaturdaten, die Temperaturveränderung berechnen. Positionsdaten ergeben über die Zeit betrachtet Bewegungspfade, aus denen durch weitere Bearbeitung Bewegungsmuster erstellt werden können (vgl. Karstaedt, 2012).

Mit Hilfe der Aggregation von verschiedenen Sensordaten können weitere Informationen erschlossen werden. So wird es möglich, aus kombinierten Messwerten Situationen und Handlungen zu erkennen. Hierfür werden neben festen Regeln oft maschinelle Lernverfahren eingesetzt. Zum Beispiel können einfache Regeln eingesetzt werden, um zu ermitteln, in welchem Raum sich ein Bewohner befindet oder ob er sich im Interaktionsradius von Geräten aufhält. Maschinelle Lernverfahren können gut zur Situations- oder Handlungserkennung verwendet werden, nachdem sie dafür trainiert wurden.

Die Zustandsveränderungen von Aktoren können ebenfalls als Sensordaten betrachtet werden. Dies kann zum Beispiel das Öffnen eines Fensters oder eine Regulierung der Heizungssteuerung sein. Aus diesen Zustandsänderungen lässt sich der Zustand der Umgebung ableiten.

2.1.5 Allgemeiner Aufbau eines verteilten Context-Aware-Systems

Verteilte Context-Aware-Systems bestehen aus vielen verschiedenen Komponenten. Während früher in dem Bereich die verteilten Komponenten direkt mit Sensoren und Aktoren kommunizierten, werden heute immer öfter Infrastruktur-Komponenten eingesetzt. Diese dienen zur Reduzierung der Komplexität und Erhöhung der Wartbarkeit in solchen Systemen (vgl. Augusto u. a., 2010). Henricksen u. a. (2005) beschreiben eine allgemeine Architektur aktueller Context-Aware-Systems und identifizieren häufig eingesetzte Komponenten. Dabei wird das System in fünf verschiedenen aufeinander aufbauenden Schichten beschrieben (s. Abbildung 2.1).

In der untersten Schicht (Layer 0) befinden sich die Sensoren und Aktoren, welche die Sensordaten messen und Aktionen ausführen können. Die Komponenten, die die gemessenen Sensordaten zu Kontextinformationen verarbeiten und die Steuerungsbefehle an die Aktoren verwalten und weiterleiten, befinden sich in der darüberliegenden Schicht (Layer 1).

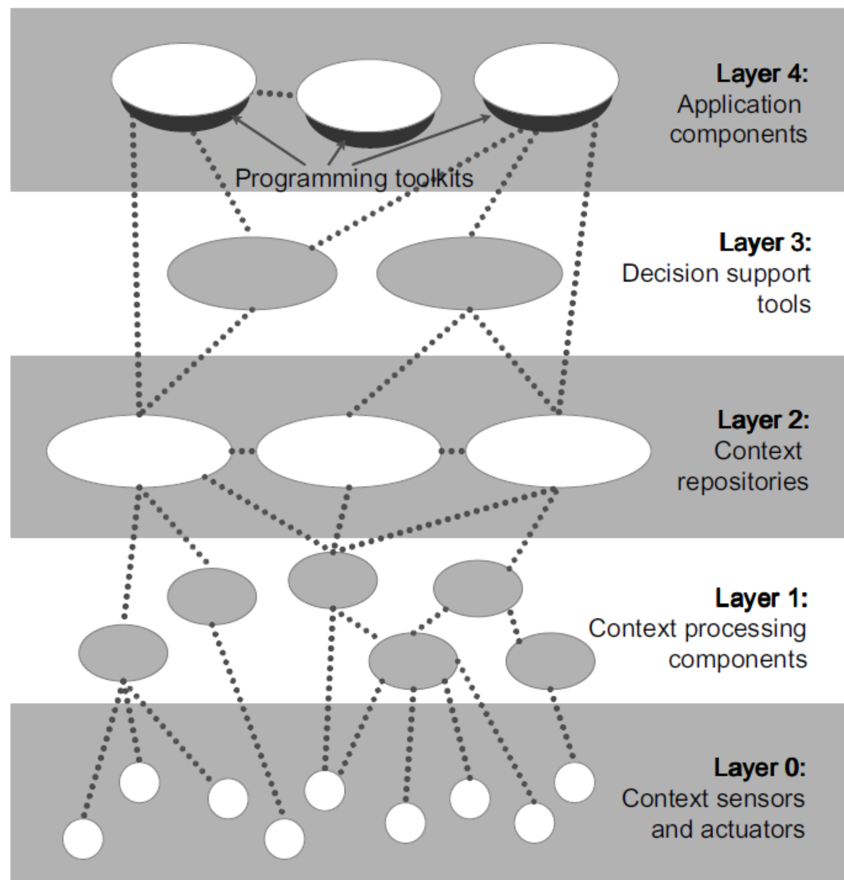


Abbildung 2.1: Allgemeiner Aufbau eines Context-Aware-Systems. Henricksen u. a. (2005)

In Layer 2 befinden sich Kontext-Repositories, die die gesammelten Kontextinformationen strukturiert speichern. Die Informationen können dann von anderen Komponenten über Queries abgerufen werden. Decision-Support-Tools werden eingesetzt, um die Entscheidungsfindung auf Basis von Kontextinformationen zu vereinfachen. Diese Tools befinden sich auf Layer 3 und können außerdem zwischen Anwendungsprogrammen und den Aktoren vermitteln, um für jede gewünschte Aktion die richtigen Aktoren zu finden. In der obersten Schicht (Layer 4) sind die Anwendungsprogramme angesiedelt, welche die Funktionen des Context-Aware-Systems steuern und die Schnittstelle zum Benutzer bilden. Sie können auf die beiden darunterliegenden Schichten zugreifen, um Entscheidungen auf Basis der gesammelten Kontextinformationen zu treffen und Aktionen von Aktoren ausführen zu lassen. Zur Vereinfachung der Entwicklung von Anwendungskomponenten können Toolkits eingesetzt werden, die ebenfalls zur

obersten Schicht gehören. Das Toolkit kann zum Beispiel die Kommunikation mit den anderen Komponenten bereitstellen.

2.2 Agentenbasierte Systeme

Ein Multiagentensystem ist ein System, das aus mehreren Agenten besteht und in dem Problemstellungen durch Zusammenarbeit zwischen Agenten gelöst werden. Diese Zusammenarbeit wird durch den Austausch von Nachrichten erreicht. Multiagentensysteme werden in vielen unterschiedlichen Bereichen eingesetzt. Dazu zählen hauptsächlich verteilte Systeme und verteilte künstliche Intelligenz. Ein großes Anwendungsgebiet stellen Context-Aware-Systems dar. Im Folgenden werden einige wichtige Standards und Frameworks für agentenbasierte Systeme vorgestellt, die auch in vielen verwandten Arbeiten verwendet werden.

Es existiert keine allgemein akzeptierte Definition von Agenten (vgl. Wooldridge, 2002). Je nach Anwendungsfall werden Agenten unterschiedliche essentielle Eigenschaften zugesprochen. Eine häufig geforderte Eigenschaft für Agenten ist Selbstständigkeit, die Fähigkeit, nicht nur reaktiv zu handeln. Hierfür sollte ein Agent Zugriff auf seine Umgebung haben, um von sich aus Aktionen auszuführen. In einigen Anwendungsgebieten wird der Lernfähigkeit ein hoher Stellenwert zugesprochen. In diesem Zusammenhang wird oft von intelligenten Agenten gesprochen, die ihr Verhalten basierend auf ihren Erfahrungen anpassen können.

Jeder Agent hat eine vom Rest des Systems abgegrenzte Aufgabe. Komplexere Aufgaben lassen sich durch die Organisation von Agenten in Gruppen realisieren. Da dies ausschließlich über den Austausch von Nachrichten erfolgt, herrscht eine lose Kopplung zwischen den Systemkomponenten vor. Deswegen eignen sich agentenbasierte Ansätze gut für die Realisierung von komplexen verteilten Systemen (vgl. Jennings, 2000).

2.2.1 FIPA - Agent Communication Language

Die Foundation for Intelligent Physical Agents (FIPA)[1] ist ein internationales Standardisierungsgremium, das 1996 gegründet wurde. Es beschäftigt sich mit Standards im Bereich von Agentensystemen. Das Ziel ist es, dass sich die entwickelten Technologien unabhängig vom konkreten Anwendungsfeld für komplexe, dynamische Systeme mit hohem Kommunikationsaufwand einsetzen lassen (vgl. Poslad, 2007). Ein wichtiger Standard der FIPA ist die Agent Communication Language (ACL).

[1] Foundation for Intelligent Physical Agents - http://fipa.org; letzter Zugriff: 26.09.2014

Die Agent Communication Language ist ein Nachrichtenformat für Agentenkommunikation. Das Encoding, die Semantik und der Aufbau der Nachrichten sind im FIPA-ACL-Standard definiert (vgl. Foundation for Intelligent Physical Agents, 2002b). Dabei schreibt der Standard keine Implementierungsdetails vor, um die Kommunikation zwischen verschiedenen FIPA-konformen Implementierungen zu erlauben. ACL ist auf Agentenkommunikation zugeschnitten und bietet über Interaction-Protokolle Definitionen für einen üblichen Nachrichtenaustausch zwischen Agenten. Ein Beispiel hierfür ist das Query-Request-Protokoll, welches ein einfaches Frage-Antwort-Protokoll darstellt. Das Contract-Net-Negotiation-Protokoll ist ein Beispiel für ein komplexeres Protokoll. Es kann zur Konfliktlösung zwischen Agenten eingesetzt werden.

Abbildung 2.2 enthält eine ACL-Beispielnachricht. Jede Nachricht hat einen Sender und einen oder mehrere Empfänger, die über URLs eindeutig definiert werden. Der Inhalt der Nachricht kann in verschiedenen Formaten angegeben werden, welche im Feld „language“ definiert werden. Diese Nachricht ist ein Beispiel für eine Anfrage im FIPA-Request-Protokoll und kann als Aufforderung an einen Agenten, das linke Fenster zu öffnen, interpretiert werden.

```
0  (request
1    :sender (:name A@server.de:8080)
2    :receiver (:name B@tcp://server2.de:6600)
3    :ontology smart_home
4    :language FIPASL
5    :protocol fipa-request
6    :content
7      (action B@tcp://server2.de:6600
8        (open_window (:position left) ...
9    ))
10 )
```

Abbildung 2.2: Beispiel für eine FIPA-ACL-Nachricht.

2.2.2 JADE

Das Java Agent Development Framework (JADE) ist ein Java-Framework für Multiagentensysteme, welches nach dem FIPA-Standard entwickelt wurde (vgl. Bellifemine u. a., 2001). Dabei handelt es sich vordergründig um die standardisierte FIPA-Agent-Plattform (vgl. Foundation for Intelligent Physical Agents, 2002a), deren Architektur in Abbildung 2.3 abgebildet ist. Eine Agent-Plattform besteht hiernach aus mehreren Komponenten, die über einen internen

Nachrichtentransportservice verbunden sind. Das Agent-Management-System verwaltet alle angemeldeten Agenten und hat die Kontrolle über die Plattform. Der Directory-Facilitator stellt einen Auskunftsdienst für die Plattform dar. Über diesen Dienst können Agenten nach bestimmten Diensten suchen und selber welche registrieren. Ein verlässlicher und geordneter Nachrichtenaustausch nach außen läuft über den Agent-Communication-Channel (ACC).

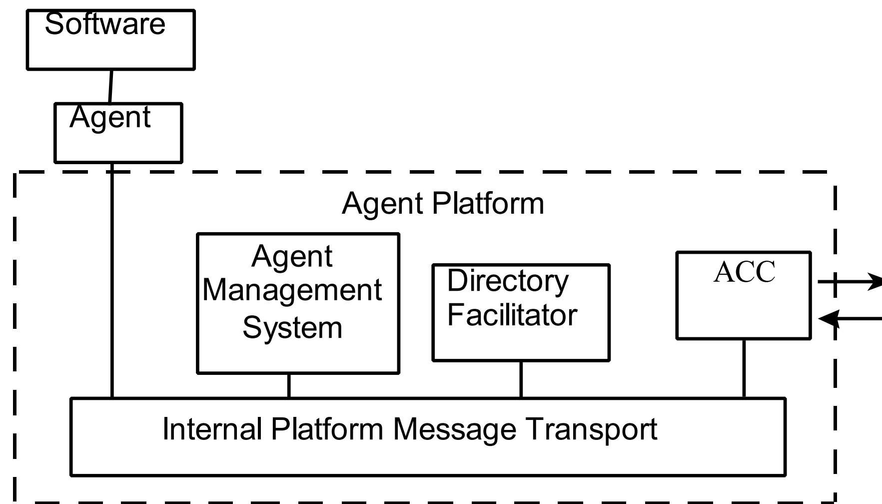


Abbildung 2.3: Die FIPA-Agent-Plattform. Bellifemine u. a. (1999)

Das JADE-Framework ist verhaltensorientiert. Um einen Agenten zu erstellen, muss der Entwickler sein Verhalten definieren. Dies kann zum Beispiel das Senden oder Empfangen einer Nachricht sein. Diese Verhaltensdefinitionen können einmalig, mehrmals oder zyklisch ausgeführt werden. Über diese Kombinationen lassen sich komplexe Agenten definieren. Zusätzlich zur Plattform bietet JADE Tools zum Systemmonitoring.

2.2.3 Bewertung

Die formale Definition der Protokolle durch den FIPA-ACL-Standard hat den Vorteil, dass die Kommunikation fest definiert ist und auch von anderen FIPA-konformen Komponenten verstanden werden kann. Damit ist es möglich, Komponenten zwischen verschiedenen Systemen wiederzuverwenden oder diese Systeme zu verbinden. Allerdings müssen die Nachrichtenformate und Protokolle oder zumindest das Erstellen von Nachrichten in einem Framework von den Entwicklern erlernt werden. Dies stellt einen zusätzlichen Implementierungsaufwand dar. Aufgrund der Komplexität der Nachrichten können Fehler bei der Entwicklung von neuen Komponenten entstehen. Außerdem hat ACL aufgrund der vielen vorgeschriebenen Attribute, wie dem Protokoll-Attribut oder dem allgemeinen Nachrichtenaufbau, einen großen Overhead.

Dies kann besonders für Systeme mit bestimmten Latenzanforderungen ein Problem werden.

Mit JADE existiert bereits ein umfangreiches Framework für Multiagentensysteme auf Basis einer standardisierten Architektur. JADE-Agenten sind jedoch sehr schwergewichtig, weil sie auf Threads basieren, die in großen Mengen sehr viel Arbeitsspeicher benötigen. Damit sind Agenten ebenfalls eine begrenzte Ressource, die vom Programmierer sparsam verwendet werden sollte. Die standardisierte Architektur erhöht zwar die Wiederverwendbarkeit der Systemkomponenten, verhindert aber dabei eine Anpassung an die Anforderungen bestimmter Umgebungen.

2.3 Verwandte Arbeiten

Im Folgenden werden einige Arbeiten vorgestellt, die sich mit einer ähnlichen Fragestellung beschäftigen. Die Auswahl beschränkt sich dabei auf drei agentenbasierte Ansätze mit unterschiedlichen Schwerpunkten.

2.3.1 CHIL

Die CHIL-Middleware wurde im Rahmen des Projektes „Computers in the Human Interaction Loop“ (CHIL) entwickelt. Es ist eine vollständig agentenbasierte Middleware für Smart-Places auf Basis von JADE und ACL (vgl. Soldatos u. a., 2007).

CHIL ist eine Middleware, die für heterogene Systeme ausgelegt ist. Die Verwendung von FIPA-Standards und des JADE-Frameworks erlauben die Implementierung von Komponenten mit verschiedenen Technologien. Das Nachrichtenformat ist mit dem ACL-Standard ebenfalls technologieunabhängig, was den Austausch aller Middleware-Komponenten möglich macht. Dieser Vorteil wird durch den agentenbasierten Ansatz verstärkt. Die Aufgliederung von Diensten und Zuständigkeiten in Agenten führt zu einer stärkeren Kapselung, was die Austauschbarkeit und Wiederverwendbarkeit der Komponenten erhöht.

Jeder Agent in CHIL hat die Fähigkeit, andere Agenten zu überwachen. Dies wird über ein Ping-Reply-Protokoll realisiert, welches einfache Ping-Nachrichten verschickt. Hängt der bereitgestellte Dienst eines Agenten von der Funktion eines anderen ab, schickt er regelmäßige Ping-Nachrichten an ihn. Sollte der überwachte Agent nicht mehr erreichbar sein, können so alle davon abhängigen Agenten darauf reagieren. Die einfachste Reaktion ist, den auf dem ausgefallenen Dienst aufbauenden Dienst ebenfalls zu beenden. In einzelnen Fällen kann aber

auf Ersatzdienste zugegriffen oder der Funktionsumfang des bereitgestellten Dienstes eingeschränkt werden. Ausgefallene Agenten können, nachdem festgestellt wurde, dass sie nicht mehr erreichbar sind, neu gestartet werden. Mit diesen Maßnahmen lässt sich die Verfügbarkeit von Diensten erhöhen.

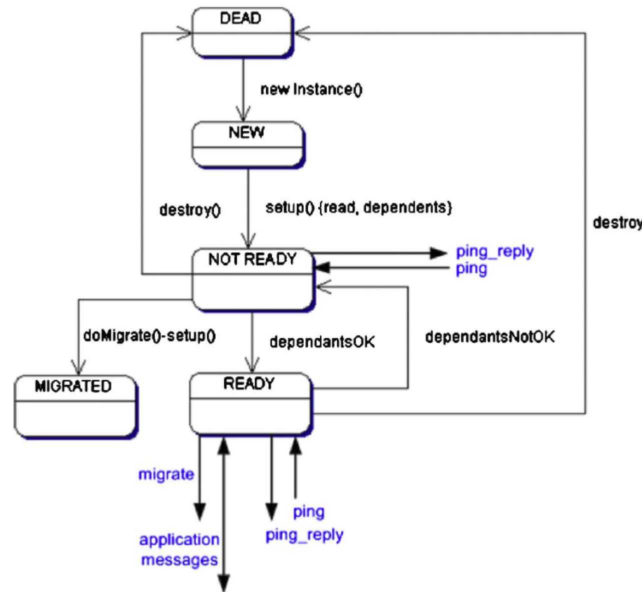


Abbildung 2.4: Der Agenten-Lebenszyklus in CHIL. Soldatos u. a. (2007)

Abbildung 2.4 zeigt den Lebenszyklus eines Agenten in CHIL. Am Anfang befindet sich jeder Agent im Zustand „NEW“. Er initialisiert sich selbst mit der Konfiguration und lädt eine Liste seiner Abhängigkeiten. Danach wechselt er in den Zustand „NOT READY“. Er überprüft mit Ping-Nachrichten die Verfügbarkeit seiner Abhängigkeiten. Erhält er von allen eine Antwortnachricht, sind alle Abhängigkeiten verfügbar und er wechselt in den Zustand „READY“. Ab diesem Zeitpunkt nimmt er seine Arbeit auf und seine Dienste sind für das System verfügbar. Dabei werden die Abhängigkeiten weiterhin aktiv mit Ping-Nachrichten überprüft. Sollte eine Abhängigkeit nicht erreichbar sein, wechselt er wieder zurück in den Zustand „NOT READY“. Der Zustand „MIGRATED“ zeigt an, dass ein Agent für eine Migration auf eine andere Plattform, eingefroren wurde. Dies ist notwendig, um dabei einen konsistenten Zustand übertragen zu können.

2.3.2 Agentenbasierte Middleware für zusammenarbeitende Smart-Objects

Eine weitere agentenbasierte Middleware wurde von Fortino u. a. (2013) vorgestellt. Die Middleware ist eventbasiert und verwendet mit JADE und ACL ebenfalls mehrere Standards für Multiagentensysteme.

Smart-Objects sind physikalische Objekte, die mit Hilfe von Hard- und Softwarekomponenten, mit Sensoren, Aktoren und der Fähigkeit zur Kommunikation mit andern Objekten ausgestattet sind. Die Kommunikation bildet die Grundlage für die Kooperation zwischen den Objekten. Der Begriff Smart-Object ist sehr allgemein gehalten und kann sowohl für kleinere Objekte wie Stifte, Kaffeemaschinen oder Türen als auch für mittlere bis große Objekte wie ganze Gebäude, Busstationen oder Tunnel verwendet werden.

Alle Smart-Objects werden als JADE-Agenten implementiert und bestehen aus mehreren Komponenten. Das Verhalten jedes Smart-Objects wird mit Tasks definiert und direkt im JADE-Framework implementiert. Ein Event-Dispatcher und ein Communication-Management-Subsystem verwalten die Kommunikation innerhalb des Objektes und mit anderen. Zwei weitere Systeme kontrollieren die Sensoren und Aktoren sowie die gesammelten Kontextinformationen. Die Kommunikation zwischen den Objekten basiert auf ACL und einem Publish-Subscribe-Dienst.

Evaluiert wurde die Middleware in einem Smart-Environment, welches sich aus zwei Gruppen von kooperierenden Smart-Objects zusammensetzt. Die zwei Gruppen sind zum einen ein Smart-Office und zum anderen Smart-Body-Sensoren zur Aktivitätserkennung des Benutzers. Als einfaches Testszenario wurde ein normaler Arbeitstag im Büro verwendet. Die beiden Agenten-Gruppen tauschen Sensorinformationen über das Abonnieren von Gruppen und mit ACL-Abfragen aus und arbeiten so zusammen, um den Benutzer in seiner Arbeitsumgebung zu unterstützen. Eine Anwendung der Kontextinformation ist die kontextabhängige Anzeige von Inhalten auf Bildschirmen und die Warnung der Benutzer bei speziellen Situationen, wie zum Beispiel nicht ausgeschaltete Lampen, beim Verlassen des Raumes.

2.3.3 AMUSE

AMUSE - Agent-based Middleware for Context-aware Ubiquitous Services - ist eine agentenbasierte Middleware, die von Takahashi u. a. (2005) entwickelt wurde. Alle Entitäten im System der AMUSE-Middleware werden durch Agenten gekapselt. Entitäten können sowohl Software-

und Hardwarekomponenten des Systems als auch Benutzer und Netzwerkverbindungen sein. Die Kapselung beinhaltet die Ergänzung von Fähigkeiten zur Kontextverwaltung und zur Zusammenarbeit mit anderen Agenten. Dafür werden eine Kommunikationsschnittstelle und die Möglichkeit, Kontextinformationen zu speichern, hinzugefügt. Damit wird es zum Beispiel möglich, Ressourcen wie eine Netzwerkverbindung zu überwachen. Ein Agent kann so auf die Überlastung einer Verbindung reagieren und die Kommunikation umleiten. Dieser Vorgang ist ein zentrales Konzept in AMUSE und wird als „Agentification“ bezeichnet.

In dem AMUSE-System befinden sich auf der untersten Ebene die Entitäten. Durch Agentification werden aus diesen Entitäten Agenten erstellt, die die Ebene darüber bilden. Diese werden auch als primitive Agenten bezeichnet. Gruppen von Agenten können durch Kommunikation im Agent-Relationship-Layer gebildet werden. Diese Gruppen befinden sich im Agent-Organization-Layer. Auf der obersten Ebene sind die Dienste, die von den Agentengruppen bereitgestellt werden, zu finden. Sie sind in der Abbildung 2.5 als Symbole dargestellt und können als ubiquäre Dienste bezeichnet werden.

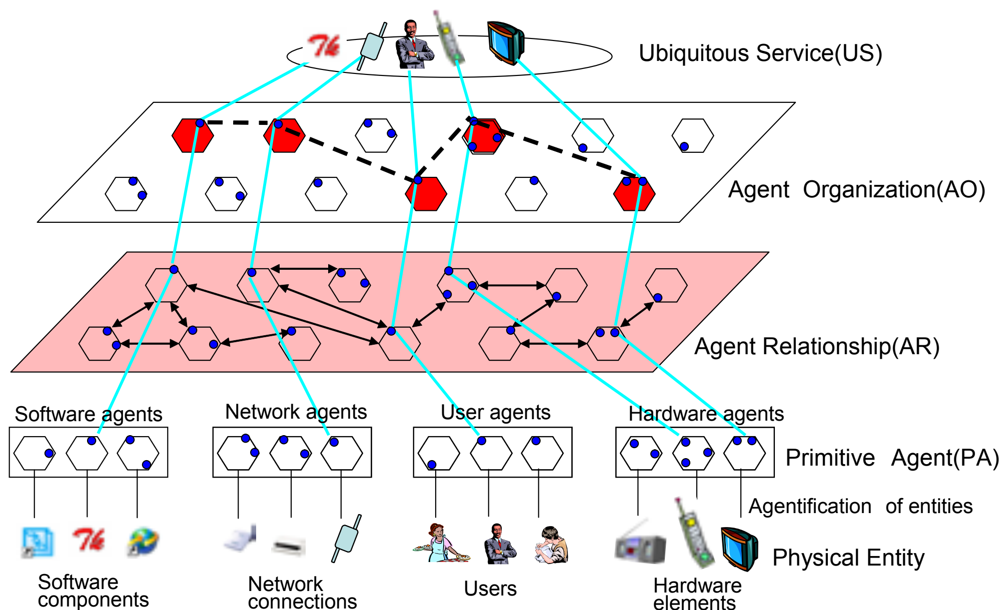


Abbildung 2.5: Das AMUSE-Framework. Takahashi u. a. (2005)

Durch die Kapselung der Entitäten mit Hilfe von Agenten und die Funktionserweiterungen wird es möglich, die Servicequalität der bereitgestellten Dienste zu überwachen. Das System kann selbstständig auf eine Veränderung der Servicequalität reagieren und bei Bedarf zum Beispiel die Gruppenorganisation der Agenten verändern. Die Daten über die Servicequalität

können also zusammen mit anderen Kontextinformationen von Agenten dazu benutzt werden, selbstständig Entscheidungen zu treffen, die dabei helfen können, die Verfügbarkeit der Dienste zu verbessern.

2.3.4 Vergleich

Eine Einordnung der verschiedenen Middleware-Entwürfe ist über einen Vergleich der Fähigkeiten der Agenten in den jeweiligen Systemen möglich (vgl. Sahli, 2008). Bei der CHIL-Middleware wird der agentenbasierte Ansatz hauptsächlich zur Modularisierung der Komponenten eingesetzt. Agenten können nicht wirklich autonome Entscheidungen treffen. Im Gegensatz dazu ist die Selbstständigkeit von Agenten einer der wichtigsten Punkte in AMUSE. Dies äußert sich vor allem in der Selbstorganisation von Agenten in Gruppen, um Services bereitzustellen, und der Überwachung der Servicequalität von Agenten.

Eine weitere Klasse von Middleware konzentriert sich auf die Mobilität von Agenten. Ein Wechsel der Plattform durch eine Migration ist in CHIL zwar vorgesehen, das System ist aber nicht hauptsächlich für mobile Agenten entworfen worden. Ein Beispiel für eine Middleware mit mobilen Agenten ist das System von Harroud und Karmouch (2005), welches für mobile Geräte wie Smartphones entwickelt wurde. Diese Middleware basiert auf MicroFIPA-OS, einer leichtgewichtige FIPA-konformen Agenten-Plattform für mobile Geräte. Das Verhalten von Agenten wird mit Hilfe von Policies definiert. Diese enthalten kontextabhängige Anweisungen für Umgebungen mit unterschiedlichen Eigenschaften. Ein Anwendungsbeispiel hierfür wäre eine angepasste Auflösung von Videos auf Geräten mit eingeschränkten Darstellungsmöglichkeiten oder benutzerabhängigen Einstellungen und Vorlieben.

Die Unterstützung des Entwicklungsprozesses neuer Komponenten hat bei den vorgestellten Ansätzen keine hohe Priorität. Toolkits und das Framework stellen häufig benötigte Implementierungen wie Anbindung an die Kommunikationsinfrastruktur oder die Kontextverarbeitung bereit. Außerdem bietet das JADE-Framework allgemeine Tools mit grafischer Benutzeroberfläche an, die anzeigen können, welche Agenten auf einer Plattform ausgeführt werden.

2.4 Living Place Hamburg

Das Living Place Hamburg ist ein Labor an der HAW Hamburg in Form einer 140 m^2 großen Loft-Wohnung. Das Labor beinhaltet zusätzlich zur Wohnfläche einen Kontrollraum und mehrere Büro- und Arbeitsräume. Der Kontrollraum kann zur Durchführung von Usability-

Tests in der Wohnung benutzt werden. In der Laborumgebung werden neben Fragestellungen zu den Themen Smart-Homes und Urban-Living auch Themen wie Human-Computer-Interaction und New Storytelling bearbeitet (vgl. von Luck u. a., 2010).

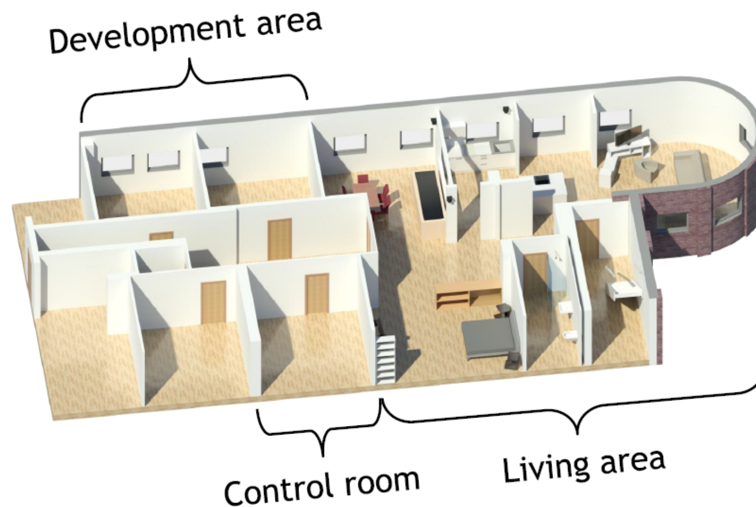


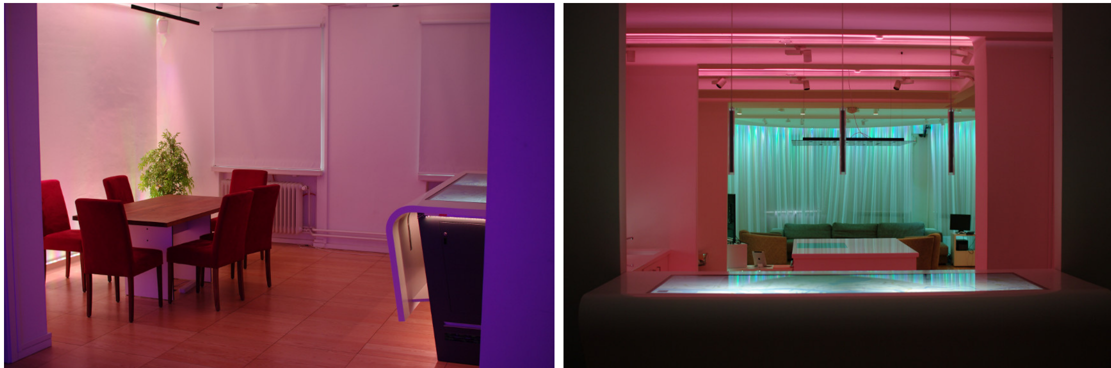
Abbildung 2.6: Das Living Place Hamburg. Karstaedt (2012)

2.4.1 Javascript Object Notation - JSON

Im Living Place wird die Javascript-Object-Notation als Nachrichtenformat für die Kommunikation zwischen allen Komponenten verwendet.

Die Javascript-Object-Notation ist ein Webstandard für ein Datenformat (vgl. Ecma International, 2013). Es wird wie XML häufig zur Beschreibung von Konfigurationen und zur Datenübertragung im Zusammenhang mit Web-Technologien wie AJAX oder Websockets verwendet. Das JSON-Format eignet sich aber auch zur Verwendung für die technologieunabhängige Kommunikation zwischen Komponenten oder Agenten.

Eine JSON-Nachricht ist ein Objekt, welches beliebig viele Schlüssel-Wert-Paare enthalten kann. Im Gegensatz zu den Schlüsseln, die Zeichenketten sein müssen, können verschiedene Datentypen als Werte verwendet werden. Dazu zählen unter anderem Zahlen, boolesche Werte und Arrays. Außerdem lassen sich durch die Verwendung von Objekten als Werte verschachtelte Nachrichten erstellen. Durch diesen einfachen Aufbau ist JSON sowohl einfach für Computer zu verarbeiten als auch für Menschen zu lesen.



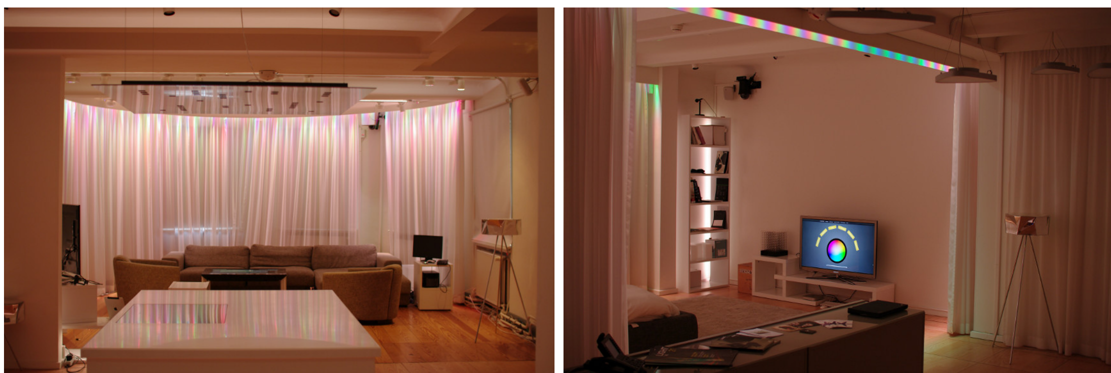


Abbildung 2.7: Innenansichten des Living Place Hamburg.

```
0  {
1    "type": "ControlLightRGB",
2    "id": 177,
3    "color": {
4        "r": 100,
5        "g": 0,
6        "b": 0
7    },
8    "identifier": "sender",
9    "options": ["debug", "timestamp"]
10 }
```

Abbildung 2.8: Beispiel für eine JSON-Nachricht.

Abbildung 2.8 zeigt ein Beispiel für eine JSON-Nachricht. Diese Nachricht könnte verwendet werden, um in einer Wohnung eine farbige Lampe auf einen bestimmten Farbwert zu setzen. Da

der Aufbau der Nachricht und die verwendeten Schlüssel frei wählbar sind, müssen diese per Konventionen oder Dokumentation festgehalten werden, um eine Kommunikation zwischen Komponenten zu ermöglichen.

2.4.2 Aktuelle Architektur

Im Living Place wird eine Blackboard-Architektur verwendet. Das Blackboard ist der zentrale Kommunikationsdienst für alle Komponenten des Systems (vgl. Ellenberg u. a., 2011). Eine Komponente kann Nachrichten auf Gruppen veröffentlichen. Diese Nachrichten werden dann an alle Komponenten verteilt, die diese Gruppe zuvor abonniert haben. Zur Realisierung des Blackboards wird ein Message-Broker verwendet. Alle Nachrichten werden ohne weitere strukturelle Anforderungen im JSON-Format verschickt.

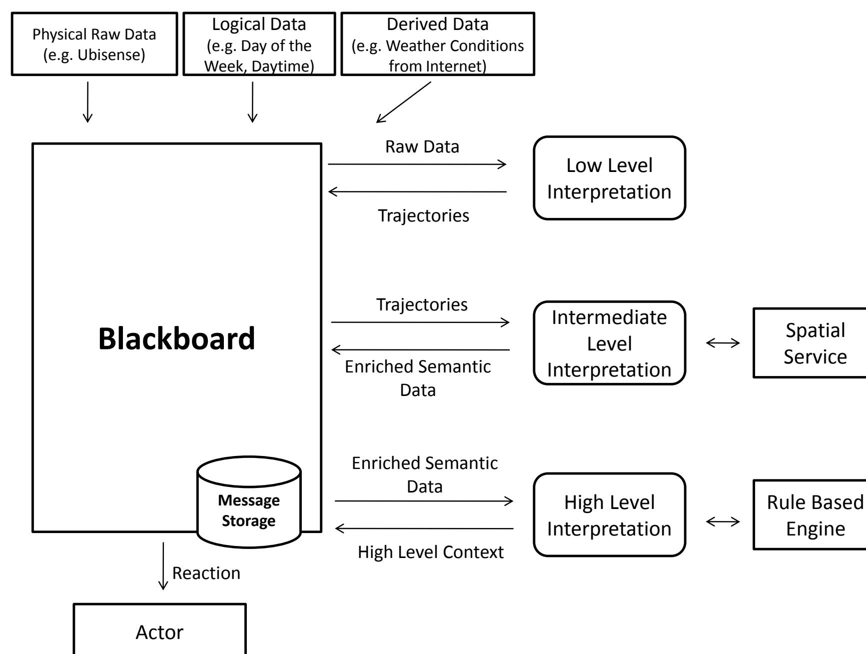


Abbildung 2.9: Alte Architektur des Living Places. Ellenberg u. a. (2011)

Abbildung 2.9 zeigt die Softwarearchitektur im Living Place. Als zentrale Instanz, über die alle Komponenten kommunizieren, dient das Blackboard. Daten wie Positions-, Zeit- oder Wetterinformationen werden von Sensoren auf dem Blackboard veröffentlicht. Weitere Komponenten können diese Informationen abonnieren und weiterverarbeiten. Die Sensordaten durchlaufen verschiedene Interpretationsebenen. Zum Beispiel können aus den Positionsdaten Bewegungspfade (Trajektorien) oder -muster erstellt werden (vgl. Voskuhl, 2011). Mit Hilfe

von zusätzlichen Informationen, wie zum Beispiel die Position von Räumen oder Gegenständen in der Wohnung, werden diese weiter angereichert. Auf der höchsten Ebene können aus den Sensordaten durch vordefinierte Regeln Situationen erkannt werden. Alle abgeleiteten Informationen werden wieder über das Blackboard veröffentlicht.

Die Architektur wurde entwickelt, um möglichst schnell und einfach neue Projekte im Living Place realisieren zu können. Das System ist sehr offen, weil alle Nachrichten von beliebigen Komponenten abonniert werden können. Da alle Nachrichten in Textform übertragen werden und sie überall abgegriffen werden können, kann ein Entwickler ohne großen Aufwand einen guten Einblick in die Kommunikation bekommen.

Durch die Freiheiten bei der Nachrichtenformatierung und bei der Anbindung an den Message-Broker sind über die Zeit viele verschiedene Implementierungen entstanden. Mit einer wachsenden Anzahl von Komponenten durch weitere Studentenprojekte entstand so ein komplexes verteiltes System. Dadurch wird es schwierig, Fehler im System zu finden oder einen Überblick über das System zu bekommen. Beim Start neuer Projekte beginnen viele Entwickler mit der Implementierung der Anbindung an den Message-Broker und der Nachrichtenserialisierung. Dies erfordert viel Zeit und ist fehleranfällig. Außerdem ist aufgrund der Komplexität des Systems der Aufwand für neue Projektteilnehmer, sich in die vorhandenen Systeme einzuarbeiten, sehr groß.

Bei der Einführung einer Middleware in das Living Place müssen die vorhandenen Komponenten ungestört weiterarbeiten können. Neue Komponenten müssen zudem auf die vorhandenen Dienste zurückgreifen können. Sonst wäre es zum Beispiel nicht möglich, die bisher realisierten Aktoren anzusprechen. Bei dem Design der neuen Middleware muss besonders auf die Unterstützung der Entwickler geachtet werden. Die Vorteile der alten Architektur sollen mit dem neuen System weiterbestehen. Die Kommunikation soll offen und für Entwickler einsehbar bleiben. Es muss möglich sein, unterschiedliche Technologien in das System zu integrieren und die Entwickler sollen weiterhin die Freiheiten haben, ihre Projekte nach eigenen Vorstellungen zu realisieren. Zur Aufwandsreduzierung neuer Projekte soll es einen empfohlenen und gut dokumentierten Weg zur Entwicklung neuer Komponenten geben, der durch ein Framework unterstützt wird.

2.5 Anforderungen in einer Smart-Home-Laborumgebung

In einer Smart-Home-Laborumgebung ergeben sich unterschiedliche Anforderungen, die im Folgenden am Beispiel des Living Place Hamburgs analysiert werden.

Das Living Place hat als Laborumgebung an einer Hochschule eine Reihe von Stakeholdern mit unterschiedlichen Interessen. Zu den Hauptbenutzern der Laborumgebung zählen die Informatik-Studenten, die ihre Projekt- und Abschlussarbeiten dort entwickeln und testen wollen. Dies sind meist zeitlich begrenzte Arbeiten, die mit einer Projektausarbeitung oder einer Abschlussarbeit beendet werden. Diese zeitliche Begrenzung führt zu einem sich ständig verändernden Entwicklerteam, in das stetig neue Studenten eingearbeitet werden müssen. Die Tools und die Software in der Laborumgebung sollten diesen Einstieg unterstützen.

Zusätzlich führt der Zeitdruck, unter dem die Studenten stehen, oft dazu, dass der praktische Teil, die Software, die später in der Laborumgebung laufen soll, nicht hundertprozentig ausgearbeitet oder fertiggestellt wird. Wird die Dokumentation bis zuletzt aufgeschoben, fällt sie oft ganz oder teilweise weg. Wenn die Studenten nach ihrem Abschluss die Hochschule verlassen, kann die Software in einigen Fällen nicht mehr benutzt werden oder der Wartungsaufwand ist extrem hoch. Der Entwicklungsprozess sollte also die Dokumentation von neuen Komponenten fördern. Dabei sollte auf eine hohe Wiederverwendbarkeit aller Softwarekomponenten und Tools geachtet werden, damit jede Aufgabe nur genau einmal erledigt werden muss und Zeit für die wichtigen Aufgaben zur Verfügung steht.

Das Labor wird ebenfalls für Untersuchungen von Studierenden aus anderen Fachbereichen und für interdisziplinäre Arbeiten benutzt. Ein Beispiel hierfür sind Medien- und Interaction-Designer. Diese wollen die Laborumgebung für Experimente nutzen, die eine möglichst reale Umgebung erfordern. Dabei kann es sich zum Beispiel um Untersuchungen der Auswirkungen von Lichtfarben und Musik auf die Bewohner handeln. Die Anforderungen für solche Untersuchungen stehen mit den Anforderungen der Informatik-Studenten im Konflikt. Diese wollen neue Software testen, während eine stabile Laborumgebung für die Experimente benötigt wird.

Eine weitere besondere Situation, die in Laborumgebungen auftritt, sind Präsentationen. Hier sollen für Besucher einerseits neue Entwicklungen in der Wohnung vorgestellt werden, andererseits ist eine möglichst fehlerfreie Laborumgebung für eine Präsentation notwendig. Die einzelnen Szenarien müssen für die Präsentationen abrufbar und steuerbar sein. Bei Fehlern

muss eingegriffen oder auf das nächste Szenario übergegangen werden können. Diese Art der Benutzung erfordert also eine Art Fernbedienung für die Funktionen der Laborumgebung.

Für die unterschiedlichen und sich teilweise widersprechenden Anforderungen an die Laborumgebung müssen das System und die Middleware sehr dynamisch sein. Es müssen verschiedene Konfigurationen und Systemzustände hergestellt werden können. Dabei muss verhindert werden, dass die Systeme zu komplex werden, um den Wartungsaufwand einzuschränken und Fehler zu vermeiden. Jeder Benutzer der Wohnung sollte zu jeder Zeit einen Überblick über den aktuellen Systemzustand bekommen können.

2.6 Analyse der Entwicklungsprozesse

Durch die Einführung der neuen Middleware sollen die Entwickler besser bei der Erstellung neuer Projekte unterstützt werden. Um die daraus resultierenden Anforderungen zu ermitteln, wird zuerst der Entwicklungsprozess anhand eines Beispielszenarios analysiert. Anschließend wird der Prozess in verschiedene Phasen eingeteilt und die jeweiligen Anforderungen werden ermittelt.

2.6.1 Szenario: Eine neue Projektarbeit im Living Place

Im Folgenden wird anhand einer fiktiven Studentin der Ablauf eines neuen Projektes im Living Place beschrieben.

Die Studentin Alice ist im zweiten Mastersemester Informatik und möchte im Rahmen der Veranstaltung Projekt 1 eine Projektarbeit im Living Place erstellen. Da sie schon bei der Einführungsveranstaltung zum Master die verschiedenen Projekte an der Hochschule kennengelernt hat und sich am Anfang des ersten Semesters für einen Betreuer entscheiden musste, ist sie schon länger Mitglied des Living-Place-Teams. Durch die im Team abgehaltenen regelmäßigen Treffen kennt sie bereits die anderen Studenten und die Mitarbeiter. In den ersten Wochen hat sie eine Einweisung für die Tools zur Entwicklerunterstützung, wie zum Beispiel den Blog, der die Beschreibungen anderer Projekte und Dokumentationen enthält, und die Versionsverwaltung erhalten.

Innerhalb des Living-Place-Teams werden regelmäßige Coding-Sessions veranstaltet. Dort haben die Studenten die Möglichkeit, zusammen anhand kleinerer Projekte die Entwicklung neuer Komponenten und den Umgang mit den Tools zu erlernen. Außerdem werden hier öfter

Vorträge zu aktuellen Themen der Softwareentwicklung im Labor und den neuen Projektarbeiten gehalten. Im Rahmen dieser Veranstaltungen hat Alice erste Erfahrungen im Umgang mit dem Framework und den theoretischen Konzepten der Middleware gesammelt.

Während der bisherigen Arbeit im Living Place ist ihr die Idee gekommen, eine Komponente zur Entkopplung des dargestellten Inhalts und der Anzeigegeräte zu entwickeln. Im Living Place existieren eine Reihe von Anzeigegeräten, wie zum Beispiel mehrere Fernseher, Tablets, Smart-Phones, Beamer und mit einem großen Tresen und dem Couchtisch zwei Multitouchflächen. Die Komponente soll Inhalte abhängig vom Kontext an die richtigen Anzeigegeräte vermitteln und dabei verschiedenste Eigenschaften der Geräte und Inhalte beachten. Beispiel für solche Eigenschaften sind die Bildschirmgrößen, die Unterstützung für Bild und Ton, die Orientierung des Gerätes und die unterstützten Wiedergabeformate.

Als einfaches Test-Szenario für die Projektarbeit wählt Alice in Rücksprache mit dem betreuenden Professor die Darstellung von verschiedenen Inhalten, wie zum Beispiel Video und Informationen im HTML-Format, auf dem Bildschirm, der dem Benutzer am nächsten ist. Die Distanz wird durch mehrere bereits vorhandene Agenten im Living Place ermittelt. Es existiert eine vollständige 3D-Modellierung der Wohnung, in der die genauen Positionen der einzelnen Geräte eingetragen sind. Zusammen mit den aktuellen Positionsdaten des Benutzers, die von den Sensoren im Labor ermittelt werden, lässt sich daraus die Distanz zum Gerät berechnen.

Im ersten Schritt lädt sich Alice das Beispielprojekt für die Verwendung des Frameworks herunter. Damit hat sie bereits einen vollständig lauffähigen Agenten inklusive Build-Konfiguration. Nachdem sich die Softwarearchitektur ihrer neuen Komponente und die Aufteilung in Agenten definiert hat, startet sie mit der Implementierung. Dabei steht ihr offen, zuerst Funktionen zu realisieren oder mit Unit-Tests zu beginnen. Die Schnittstelle zu ihren neuen Agenten wird in einer Schnittstellenbibliothek definiert. Damit wird festgelegt, welche Nachrichten verschickt werden können und welche Parameter diese haben. Alice importiert alle externen Schnittstellen, die sie benötigt, wie zum Beispiel die Nachrichten für die Positionsdaten, in ihr Projekt. Dafür muss sie nur die Importe des Beispielprojektes für ihre Anforderungen anpassen.

Schon während der Entwicklung kann das Projekt im Build-Server konfiguriert werden. Dieser führt bei jeder Änderung am Quelltext in der Versionsverwaltung alle Unit-Tests aus und erstellt die Artefakte mit Hilfe der Build-Konfiguration. Alice kann so sofort sehen, wenn nach einer Änderung die Tests fehlschlagen oder der Build nicht ausgeführt werden kann.

Außerdem kann sie so zu jeder Zeit Anderen Zugriff auf ihren Quelltext und die aktuellen Artefakte geben, was besonders bei der Klärung von Fragen oder einem Review hilfreich ist.

Nach der Implementierung der Grundfunktionen möchte Alice gerne die Interaktionen mit den anderen Komponenten testen. Hierfür kann sie über ein Plugin im Build-Tool die Middleware lokal auf ihrem Rechner starten. Dabei kann sie andere Agenten oder Dummies für deren Funktionen nachladen. Auf dieser Basis kann sie bei Bedarf auch automatisierte Integrationstests schreiben, die in mehreren Prozessen ausgeführt werden können. Möchte sie ihre Arbeit in der realen Laborumgebung testen, kann sie diese für Tests reservieren. Dabei hat sie vollständige Kontrolle darüber, welche Agenten im Live-System während des Tests ausgeführt werden. Um Zeit zu sparen, kann sie ihren Laptop direkt mit dem System verbinden und so Agenten bei ihr lokal ausführen, die mit der Middleware des Living Places interagieren.

Während dieser Tests fällt Alice ein Fehler bei der Auswahl der Anzeigegeräte auf. Über das Webinterface der Middleware schaut sie nach, welche Agenten gerade im System ausgeführt werden. Sie öffnet die für ihre Tests relevanten Gruppen im Webinterface und wiederholt ihn. Dabei kann sie den Nachrichtenaustausch der einzelnen Agenten über die Gruppen verfolgen. Nach einigen Tests bemerkt sie, dass der Agent, der die Positionsdaten generiert, fehlerhafte Messwerte liefert. Sie schreibt ein Ticket im Issue-Tracker, in dem sie den Fehler beschreibt, damit dieser behoben werden kann. Durch ihre bisherige Mitarbeit im Projekt hat sie aber schon eine Vermutung, woran der Fehler liegen könnte und lädt sich das Projekt aus der Versionsverwaltung herunter. Nach kurzer Zeit hat sie den Fehler gefunden und verknüpft ihr Ticket mit einem Merge-Request, der den Fehler behebt. Dieser Request kann später von einem anderen Studenten oder Mitarbeiter in die Software eingepflegt werden. Da Alice aber nicht solange warten und mit ihren Tests direkt fortfahren möchte, konfiguriert sie die Middleware so, dass der von ihr angepasste Agent mit Fehlerbehebung in das System geladen wird. Das dafür benötigte Artefakt wurde bereits automatisch vom Build-Server aus dem Merge-Request erstellt.

Nachdem Alice mit der Implementierung weiter vorangekommen ist und alle Funktionen für sich getestet hat, sollte die Konfiguration für die Laufzeitumgebungen angepasst werden. Diese Anpassung ist für die meisten Agenten sehr einfach, da sie keine weiteren Abhängigkeiten als das Beispielprojekt haben. Die finale Konfiguration kann bei Bedarf ein erfahrener Student oder Mitarbeiter übernehmen. Mit diesem Schritt ist das Deployment der neuen Komponente automatisiert und kann über das Webinterface der Middleware oder vom System selbst

ausgelöst werden. Genauso wie im Verlauf der Entwicklung kann die Komponente weiterhin außerhalb der Laufzeitumgebungen ausgeführt werden.

Vor der Abgabe ihres Projektes stellt Alice die Dokumentation ihrer Komponente fertig. Diese enthält eine Anleitung zur Benutzung der Anwendungsschnittstelle für die anderen Studenten und spätere Projekte. Der Quelltext und die Artefakte liegen zu diesem Zeitpunkt bereits vor. Um die Fertigstellung der Projektartefakte zu verdeutlichen, ändert Alice deren Versionsnummer in der Projektdatei und lässt den Build-Server die neuen Versionen bauen.

Ab diesen Zeitpunkt steht die neue Komponente offiziell für die Entwicklung anderer Projekte zur Verfügung und kann während einer Präsentation der Laborumgebung benutzt werden.

2.6.2 Einteilung in Phasen

Das beschriebene Szenario lässt sich nach Art der einzelnen Tätigkeiten in folgende sechs Phasen untergliedern. Die Einteilung beschreibt dabei nur grob den zeitlichen Ablauf, da sich alle Phasen überlappen können.

1. **Vorbereitung** - Kennenlernen des Teams, Einarbeitung in Technologien, Workshops, Themenfindung
2. **Design** - Entwurf der Softwarearchitektur, Definition der Schnittstellen
3. **Entwicklung** - Implementierung der Agenten
4. **Tests und Fehlerbehebung** - Implementierung der Testfälle, Auswahl von Testszenarien, Fehlersuche und Fehlerbehebung
5. **Deployment** - Installation der Software im Living Place, Konfiguration der Komponente
6. **Dokumentation und Präsentation** - Beschreibung der Schnittstellen und der Aufbau der Komponente, Vorbereitung für Präsentation im Labor

Vorbereitung

In der ersten Phase, der Vorbereitung, geht es vor allem um das Kennenlernen des Projektteams und das Einarbeiten in die vorhandenen Architekturen und Technologien. Dieser Vorgang wird durch regelmäßige Treffen des Projektteams vereinfacht. Regelmäßige Workshops sollen alle im Team auf den neusten Stand bringen und den technischen Einstieg erleichtern. Die

Workshops bestehen aus Vorträgen von Studenten zu wichtigen Technologien und aktuellen Entwicklungen sowie der gemeinsamen Implementierung von kleineren Projekten. Die Erleichterung des Einstieg in das Projekt ist besonders relevant in Teams mit häufig wechselnden Entwicklern, wie es zum Beispiel bei Studentenprojekten der Fall ist.

Für die Unterstützung dieser Phase existiert bereits eine Plattform im Living Place. Es gibt einen Blog, der Einträge über alle Teammitglieder und deren Projekte, sowie die passende Dokumentation dazu enthält. Außerdem werden hierüber die Organisation der Treffen und Workshops durchgeführt und Themen angekündigt. Da diese Plattform bereits existiert, gehen aus dieser Phase keine Anforderungen hervor.

Design

In der Design-Phase wird ein Entwurf der Softwarearchitektur angefertigt und die Schnittstellen der neuen Komponente werden definiert. Hierfür wird vor allem der Zugriff auf die Dokumentation und den Quelltext vorhandener Projekte benötigt, die als Beispielimplementierungen dienen können. Außerdem sollten die bereitgestellte Software und die Standard-Architektur für Komponenten gut dokumentiert und einfach zu verstehen sein, damit ein schneller Einstieg möglich ist. Hierfür müssen Plattformen geschaffen werden, die den einfachen Zugriff ermöglichen. Die Definition der Schnittstellen sollte unabhängig von der Implementierung möglich sein und ebenfalls durch Tools unterstützt werden.

Entwicklung

Die Entwicklung umfasst die Implementierung der Komponente. Es kann es eine Herausforderung sein, eine neue Komponente für ein komplexes verteiltes System zu schreiben. Deswegen sollte ein Framework bereitgestellt werden, das Implementierungen häufig benötigter Aufgaben bereitstellt und die Komplexität verwendeter Technologien kapselt, um die Entwicklungsphase zu beschleunigen. Zu den häufig benötigten Aufgaben gehören zum Beispiel der Zugriff auf die Middleware, andere Agenten oder Kontextinformationen, aber auch die Serialisierung von Nachrichten.

Für die Entwicklung im Team ist eine Versionsverwaltung unverzichtbar. Es muss eine Plattform geschaffen werden, über die der Quelltext einsehbar ist und kommentiert werden kann. Außerdem sollten Änderungsvorschläge gemacht und anstehende Aufgaben angelegt werden können, die vom Besitzer des Projektes bearbeitet werden und von berechtigten Projektteilneh-

mern diskutiert werden können. Hierfür ist zusätzlich zur eigentlichen Versionsverwaltung eine weitere Plattform notwendig.

Tests und Fehlerbehebung

Ein wichtiger Teil der Entwicklung neuer Softwarekomponenten sind automatisierte Tests. Das Framework und die Middleware sollten sowohl automatisierte Komponenten- als auch Integrationstests unterstützen. Die Ausführung der Tests sollte dabei nicht abhängig von der Laborumgebung sein, damit jeder Entwickler diese ebenfalls in seiner Entwicklungsumgebung ausführen kann. Das Framework sollte die Implementierung der automatisierten Tests unterstützen. Hierfür sollten vor allem Tools zum Testen von agentenbasierten Systemen und Komponenten von Smart-Environments verfügbar sein. Außerdem sollte die Ausführung der Tests und die Ausgabe des Fehlerreports automatisiert ablaufen können.

Die Middleware und die Komponenten sollten so dynamisch sein, dass sich Konfigurationen für Testszenarien in einer Laborumgebung ohne großen Aufwand herstellen lassen. Testszenarien verlangen oft den Austausch von Sensoren und Aktoren durch Dummies um spezielle Situationen herstellen und Ereignisse überprüfen zu können. Um den Entwicklern zu jeder Zeit einen möglichst guten Systemüberblick zu ermöglichen, sollte die Middleware Informationen über das System aufbereiten und darstellen können. Teil davon ist die Darstellung der laufenden Agenten und Gruppen, sowie die Anzeige, von welchen Agenten auf eine Gruppe zugegriffen wird. Es sollte jederzeit auf Logging-Informationen der Komponenten zugegriffen werden können, um Fehler im System analysieren zu können.

Deployment

Das Deployment der Komponenten sollte nach vorheriger Konfiguration vollständig automatisiert ablaufen können. Der Vorgang sollte unabhängig von der Plattform ausgeführt werden können und somit nicht nur in der Laborumgebung, sondern auch auf den Entwicklersystemen benutzbar sein.

Durch das automatische Deployment von Agenten lassen sich für bestimmte Situationen passende Konfigurationen herstellen. Um einen Überblick über das System zu haben, sollte die aktuelle Systemkonfiguration, bestehend aus den aktuell ausgeführten Agenten, angezeigt werden können. Da im Live-Betrieb kein Ausschalten des Systems möglich ist, müssen Änderungen an der Konfiguration zur Laufzeit vorgenommen werden können.

Dokumentation und Präsentation

Vor dem Abschluss der Projekte müssen diese ausführlich für nachfolgende Generationen dokumentiert werden. Hierfür ist eine Plattform erforderlich, die die Dokumentationen verwaltet und strukturiert.

Für Software- und Laborpräsentationen müssen ebenfalls Konfigurationen hergestellt werden können. Diese müssen möglichst stabil und zuverlässig laufen, um eine Präsentation zu ermöglichen.

2.7 Ergebnis der Anforderungsanalyse und Abgrenzung

2.7.1 Anforderungen

Im Folgenden werden die Anforderungen an das System, welche sich aus der bisherigen Analyse ergeben, zusammengetragen.

Allgemeine Anforderungen

Eine Middleware für Smart-Environments muss mit komplexen verteilten Systemen zusammenarbeiten. Hieraus ergeben sich zusammen mit dem Anwendungsfeld Smart-Environment spezifische Anforderungen (vgl. Henricksen u. a., 2005).

1. **Unterstützung für Heterogenität** - Ein Smart-Environment besteht aus vielen verschiedenen Komponenten, die mit unterschiedlichen Technologien realisiert wurden. Die Middleware muss Kommunikationsmöglichkeiten für alle Komponenten bereitstellen und zwischen verschiedenen Komponenten vermitteln können.

2. **Fehlertoleranz** - Das System darf nicht ausfallen, wenn eine der Komponenten oder mehrere fehlerhaft sind. Besonders Sensoren können ausfallen oder falsche Ergebnisse liefern, die herausgefiltert werden müssen, um das Gesamtsystem nicht zu beeinträchtigen. Da die Kommunikation oft über ein Netzwerk realisiert wird, ist zu beachten, dass hier Nachrichten verloren gehen oder doppelt ankommen können.

3. **Reaktionsfreudigkeit** - Viele Komponenten in einem Smart-Environment müssen reaktionsfreudig sein und in für die Situation angemessener Zeit antworten. Hierzu zählen insbesondere vom Benutzer wahrgenommene oder direkt ausgelöste Aktionen. Diese Anforderung steht in engem Zusammenhang mit der Skalierbarkeit und Fehlertoleranz.

4. **Skalierbarkeit** - In einem Context-Aware-System kann es eine große Anzahl von Sensoren und Aktoren geben, deshalb müssen die Middleware und das ganze System mit einer großen Anzahl von Komponenten zurechtkommen. Hierfür muss das System mit einer steigenden Anzahl von Komponenten und dem damit verbundenen Kommunikationsaufwand skalieren. Das System sollte außerdem durch das nachträgliche Hinzufügen von Ressourcen, wie zum Beispiel zusätzliche Prozessorleistung, skalierbar sein.

5. **Unterstützung für Mobilität** - Alle Komponenten des Systems können mobil sein. Sämtliche Kommunikation sollte lokationsunabhängig sein. Deshalb muss die Middleware einen flexiblen Routing-Mechanismus beinhalten. Dies ermöglicht es Systemteilen, bei Kontextveränderungen ihren Ausführungsort zu ändern. Außerdem kann die Mobilität der Agenten zur Umsetzung von Fehlertoleranz und Skalierbarkeit genutzt werden, indem Agenten von fehlerhaften oder überlasteten Systemen auf andere migriert werden.

6. **Installation und Konfiguration** - Das System und die Komponenten müssen einfach und zuverlässig installiert und konfiguriert werden können. Um auf die Bedürfnisse verschiedener Stakeholder innerhalb der Laborumgebung eingehen zu können, muss die Middleware das Herstellen von verschiedenen Konfigurationen erlauben. Die Anpassungen der Konfiguration soll dabei sowohl manuell als auch automatisch nach festgelegten Regeln oder abhängig von Kontextänderungen erfolgen können. Änderungen an der Systemkonfiguration können zur Laufzeit vorgenommen werden, ohne dass ein Neustart erforderlich ist.

7. **Kontextverarbeitung** - Die Verarbeitung von Kontextinformationen ist eine häufige Aufgabe für Agenten in einem Smart-Environment. Die Middleware und das Framework sollten diesen Vorgang vereinfachen und Tools zur kontextabhängigen Entscheidungsfindung bereitstellen.

Anforderungen für die Entwicklerunterstützung

Weitere Anforderungen ergeben sich für das Framework und die Tools aus der notwendigen Unterstützung für neue Projekte.

1. **Zugriff auf Entwickler-Ressourcen** - Alle Projektmitglieder müssen Zugriff auf Quelltexte, Artefakte und Dokumentation inklusive Versionsverwaltung für alle Projekte haben. Außerdem soll die Middleware über Schnittstellen zu den Systemen Zugriff auf die Ressourcen erhalten können.

2. **Framework** - Die Entwicklung neuer Komponenten muss auch für neue Entwickler mit kurzer Einarbeitungszeit möglich sein und das Framework sollte die Entwicklung schneller Prototypen unterstützen. Dazu muss das Framework häufig benötigter Implementierungen bereitstellen, um die Entwicklungszeit zu verkürzen.

3. **Dokumentation** - Eine Dokumentation für das Framework mit Beispielprojekten für einen schnellen Einstieg in die Entwicklung muss für alle Entwickler zugreifbar sein.

4. **Automatisierte Tests** - Das Framework muss automatische Tests ermöglichen. Hierzu zählen Komponenten- und Integrationstests in Form von Unit-Tests. Das Herstellen von Konfigurationen soll mit Systemtests verbunden werden können, um im Zusammenspiel mit generierten Sensordaten automatisiert in kontrollierten Umgebungen testen zu können. Alle Komponenten und die Tests sollen dabei auch außerhalb der Laborumgebung ohne großen Konfigurationsaufwand lauffähig sein.

5. **Übersicht und Kontrolle** - In komplexen Systemen sollte es Dienste geben, die Entwicklern und Benutzern einen Systemüberblick geben können. Im Fehlerfall können diese Dienste eingesetzt werden, um die Suche nach dem Fehler zu vereinfachen. Um Fehler nachzustellen, sollten Systemzustände einfach herstellbar sein und von Entwicklern manipuliert werden können. Dies ist besonders für automatisierte Tests notwendig. Alle Komponenten können Protokollausgaben produzieren, die gesammelt und bereitgestellt werden. Die Informationen müssen sowohl für die Entwickler als auch für andere Agenten bereitstehen.

2.7.2 Abgrenzung

Dieser Abschnitt enthält eine Liste mit Problemstellungen, die sich aus der Analyse ergeben, aber in dieser Arbeit nicht behandelt werden.

1. **Sicherheit und Schutz der Privatsphäre** - Die Themen Sicherheit oder Schutz der Privatsphäre des Benutzers sind nicht Gegenstand dieser Arbeit. Die Middleware und das Framework sollen in einer nach außen hin abgesicherten Laborumgebung ausgeführt werden.

2. **Implementierungen außerhalb der JVM** - Diese Arbeit und die entwickelte Software konzentrieren sich auf die JVM. Die Unterstützung von heterogenen Systemen ist Teil der Anforderungen, aber die Entwicklerunterstützung durch das Framework beschränkt sich auf die JVM.

3. **Lernende Agenten** - Wie in den vergleichbaren Arbeiten werden die potenzielle Lernfähigkeit von Agenten und die damit verbundenen Auswirkungen nicht betrachtet. Die Integration von maschinellem Lernen in die Agenten ist grundsätzlich möglich, wird aber vom Framework nicht abgedeckt.

2.7.3 Zusammenfassung

Ausgehend von dem allgemeinen Aufbau eines Smart-Environments und von Context-Aware-Systems im Zusammenhang mit verwandten Arbeiten und dem Living Place Hamburg wurde eine Liste von Anforderungen für die Middleware und das Framework erarbeitet. Weitere Anforderungen ergaben sich aus der Analyse des Entwicklungsprozesses durch ein Szenario der Erstellung einer neuen Projektarbeit.

Die Analyse ergab vor allem Anforderungen für ein dynamisches verteiltes System wie Mobilität, Fehlertoleranz und Skalierbarkeit, welche durch einen agentenbasierten Ansatz erreicht werden können. Die Entwicklerunterstützung erfordert einerseits eine hohe Testbarkeit, aber auch eine gute System-Übersicht und -Kontrolle für die Entwickler.

Im weiteren Verlauf der Arbeit werden die zusammengetragenen Anforderungen durch das Design einer Middleware und eines Frameworks umgesetzt und anschließend evaluiert.

3 Design

In diesem Kapitel werden Konzepte zur Umsetzung der Anforderungen, die sich aus der Analyse ergeben, entwickelt. Zuerst wird dafür die wichtigste Komponente der neuen Architektur, die Middleware, zusammen mit den verwendeten Konzepten vorgestellt. Dazu gehören die verschiedenen Middleware-Dienste, wie zum Beispiel die Kommunikation, die vom restlichen System verwendet werden sollen. Anschließend wird zur Umsetzung der Anforderungen nach einer verbesserten Entwicklerunterstützung ein Framework entworfen. Zusätzlich zum Framework werden eine Reihe von Tools ausgewählt, die den Entwicklungsprozess vereinfachen sollen. Abschließend wird darauf eingegangen, wie das entwickelte System in eine vorhandene Architektur eingefügt werden kann.

3.1 Designentscheidungen

Im ersten Schritt sollen eine Reihe von wichtigen Designentscheidungen getroffen werden. Hierzu gehören die Wahl des Kommunikationsformats, die einzusetzenden Technologien für die Kontextverarbeitung und die Laufzeitumgebungen. Auf Basis dieser Entscheidungen wird im weiteren Verlauf des Kapitels das Design der Middleware und des Frameworks im Detail erläutert.

3.1.1 Auswahl des Nachrichtenformates

In einem agentenbasierten System, in dem die gesamte Kommunikation über Nachrichten abgewickelt wird, ist die Wahl des Nachrichtenformats besonders wichtig. Die Auswahl wird durch die Anforderungen nach Entwicklerunterstützung beeinflusst. Während der Analyse wurden mit FIPA-ACL (s. 2.2.1) und JSON (s. 2.4.1) bereits zwei unterschiedliche Nachrichtenformate vorgestellt.

Die Verwendung von JSON-Nachrichten hat wie ACL den Vorteil, dass das Format von Menschen direkt gelesen werden kann. Dies ist besonders in Entwicklungs- und Laborumgebungen von Vorteil, wo die Kommunikation zwischen Agenten von Entwicklern bei der Fehlersuche

untersucht werden muss. Im Vergleich zu ACL ist der Aufbau von JSON sehr viel freier und leichter zu erlernen. Hinzu kommt, dass die meisten Studenten im Rahmen von Webentwicklung oder anderen Systemen schon einmal JSON benutzt haben und das Format kennen. JSON wird im Living Place seit Projektstart erfolgreich eingesetzt und alle vorhandenen Komponenten verwenden JSON-Nachrichten zur Kommunikation.

Mit der Definition von Pflichtfeldern, wie zum Beispiel Sender- oder Zeitinformationen, und weil JSON als Textnachrichten verschickt wird, entsteht ebenfalls ein Overhead. JSON ist jedoch ein Nachrichtenformat mit einem einfachen Aufbau, welches sich leicht in andere Formate überführen lässt und für das es bereits viele Performance-optimierte Bibliotheken zur Serialisierung gibt. Sollten die Latenzanforderungen der Systeme an bestimmten Stellen nicht eingehalten werden können, kann hier auf ein anderes komprimiertes Format umgestiegen werden.

Aus diesen Gründen wird für das Design der Middleware JSON als Nachrichtenformat verwendet. Außerdem soll das Format an weiteren dafür geeigneten Stellen, wie zum Beispiel in Konfigurationsdateien, eingesetzt werden, um die Komplexität für die Entwickler möglichst gering zu halten.

3.1.2 Kontextverarbeitung durch Complex-Event-Processing

Eine wichtige Anforderungen an das System ist die Unterstützung von Kontextverarbeitung. Diese soll durch die Integration von Complex-Event-Processing (CEP) erreicht werden.

Complex-Event-Processing ist ein Verfahren zur Analyse von Eventströmen (vgl. Luckham, 2002). Mit Hilfe dieser Technik lassen sich Abfragen auf komplexen Eventströmen realisieren. Dies lässt sich unter anderem gut zur Kontextverarbeitung in Context-Aware-Systems einsetzen. Dabei werden die Eventströme von den Sensoren produziert und die Sensordaten als einzelne Events interpretiert.

Es gibt mehrere Implementierungen von CEP-Engines, die unterschiedliche Abfragesprachen verwenden. Sie erlauben die Definition von Zeitfenstern mit Bedingungen. Die Zeitfenster können dabei über mehrere Eventströme reichen. Zusammen mit Konstrukten, die Bedingungen, verschiedene Zeitfenster und Eventströme miteinander kombinieren können, ergibt sich die Möglichkeit, auch komplexe Ereigniskonstellationen abzufragen. Oft verwendet werden SQL-ähnliche Abfragesprachen, die sich anbieten, weil sie von Entwicklern, die SQL bereits

kennen, leicht erlernt werden können.

CEP-Abfragen können selbst Events generieren. Dies kann eingesetzt werden, um bestimmte Event-Konstellationen zu einem Event zusammenzufassen und damit zum Beispiel Situationen zu markieren. Eine CEP-Engine kann also auch als Kontext-Interpreter eingesetzt werden.

Ein Vorteil des Verfahrens ist die Leistungsfähigkeit und Skalierbarkeit. Es ist für eine extrem große Menge von Events und für die Eventanalyse in Echtzeit ausgelegt. Viele Implementierungen unterstützen das Hinzufügen weiterer Verarbeitungsknoten, um das System zu skalieren. Außerdem passt das eventbasierte Design sehr gut zu den auf Nachrichtenaustausch aufbauenden, agentenbasierten Systemen.

Ein Nachteil bei der Verwendung von CEP-Engines in Context-Aware-Systems ist, dass die Events nur innerhalb des definierten Zeitfensters betrachtet werden können. Eine Speicherung über größere Zeiträume ist nicht vorgesehen, kann aber für eine Analyse von Daten über einen längeren Zeitraum notwendig sein. Ein Beispiel hierfür ist die Erkennung von Gewohnheiten oder Bewegungsmustern des Benutzers. Für diese Art von Eventverarbeitung sollten andere Verfahren eingesetzt werden, die unter anderem eine Datenbank für Kontextinformationen und eine Abfragemöglichkeit dieser persistierten Daten bereitstellen.

Mit Drools Fusion steht bereits die Implementierung einer CEP-Engine im Living Place zur Verfügung (vgl. Otto, 2013). Bei Drools[1] handelt es sich um eine regelbasierte Maschine, die es ermöglicht mit Hilfe von Regeln, Entscheidungen abhängig von einer Wissensbasis zu treffen. Durch die Erweiterung der regelbasierten Maschine mit temporalen Operatoren können komplexe Eventströme verarbeitet werden (vgl. Walzer u. a., 2007). Drools Fusion ist ein Modul, welches diese Erweiterungen implementiert.

Die vorhandene Integration der Engine im Living Place erlaubt es unter anderem, Regeln in der Regel-Sprache von Drools zu schreiben und so neues Verhalten innerhalb der Wohnung zu definieren. Die Events werden durch die Registrierung von Sensoren und deren Informationen beim System definiert. Eine Änderungen von Regeln zur Laufzeit ist dabei nicht möglich. Außerdem gibt es aktuell keine Möglichkeit, Abfragen an die Engine zu stellen, um bestimmte Events zu erhalten oder festzustellen, ob gerade eine bestimmte Event-Konstellation vorherrscht. Eine Reaktion auf Kontextveränderungen ist bei diesem System also nur über die

[1] JBoss Drools - www.drools.org; letzter Zugriff: 26.09.2014

festgelegten Regeln möglich.

Aus diesen Gründen soll die Implementierung durch eine neue in die Middleware integrierte Komponente ersetzt werden, um die Kontextverarbeitung und Entscheidungsfindung auf Basis von Kontextänderungen in Agenten zu unterstützen.

3.1.3 Laufzeitumgebungen

Weitere aus der Analyse hervorgehende Anforderungen an das System sind die einfache Installation und Konfiguration von Agenten. Hierfür soll eine Laufzeitumgebung eingesetzt werden, die es erlaubt, Agenten zur Laufzeit automatisiert nachzuladen. Die Laufzeitumgebung soll der Middleware eine Schnittstelle anbieten, über die die Ausführung von Agenten kontrolliert werden kann. Da die Implementierungen der Middleware und der Agenten innerhalb der Java-Virtual-Machine ausgeführt werden sollen, steht mit OSGi bereits ein etablierter Standard zur Umsetzung dieser Anforderung zur Verfügung. Änderungen zur Laufzeit kombiniert mit einem Versions- und Abhängigkeitsmanagement können eine komplexe Aufgabe darstellen. Deswegen bietet es sich zur Fehlervermeidung an, hierfür eine umfangreich getestete Lösung wie OSGi einzusetzen.

OSGi

Die OSGi-Spezifikation wurde von der OSGi-Alliance, einem Industriekonsortium, welches sich aus mehreren größeren Firmen zusammensetzt ist, entwickelt (vgl. Hall u. a., 2011). OSGi ist ein Standard zur Verbesserung der Modularisierung von JVM-Anwendungen. OSGi steht für Open Services Gateway initiative und wird mittlerweile als fester Begriff im Zusammenhang mit den Spezifikationen und der OSGi-Service-Plattform verwendet.

Die Spezifikation umfasst die OSGi-Service-Plattform, welche auf der JVM aufbaut. Die Plattform stellt ein Framework bereit, auf das die Basisdienste und die Anwendungen aufbauen. Es gibt verschiedene Implementierungen des OSGi-Frameworks, die aufgrund der Standardisierung der Schnittstellen austauschbar sind.

Software, die auf einer OSGi-Plattform ausgeführt werden soll, wird in Bundles bereitgestellt. Bundles sind normale Jar-Dateien, die zusätzliche Konfiguration enthalten, die die Eigenschaften des Bundles wie den Namen, die Version und die Abhängigkeiten festlegen. Abhängigkeiten werden unter anderem über Import- und Exportlisten auf Basis von Java-Package-Namen beschrieben. Jedes Bundle kann Services definieren, die anderen Bundles zur Verfügung gestellt

werden. Außerdem kann über die Konfiguration eine Activator-Klasse festgelegt werden, die Methoden enthält, die beim Starten oder Stoppen des Bundles ausgeführt werden sollen.

Ein großer Vorteil bei der Verwendung von OSGi ist die Unterstützung von Versionierung. Alle exportierten und importierten Pakete können mit Versionnummern oder Versionbereichen versehen werden. Das Framework stellt dann sicher, dass alle Abhängigkeiten im System korrekt aufgelöst werden können. Benötigen zwei Komponenten die gleiche Bibliothek in unterschiedlichen Versionen, werden beide Versionen geladen und getrennt bereitgestellt. Dabei werden die unterschiedlichen Versionen durch Classloader-Isolation voneinander abgegrenzt, um Konflikte auszuschließen. Komponenten können nur miteinander über eine Schnittstelle kommunizieren, wenn die definierten Versionsbereiche sich in mindestens einer Version überschneiden.

Ein Nachteil von OSGi ist die zusätzliche Konfiguration, die zur Umsetzung der genannten Features notwendig ist. Es existieren mehrere Tools zur Konfiguration von Bundles. Dazu gehören unter anderem einige Build-Tools und Plugins für Entwicklungsumgebungen. Trotzdem entsteht für den Entwickler ein erheblicher Mehraufwand. Die Konfigurationsmöglichkeiten und Formate müssen erlernt werden, was insbesondere in Umgebungen mit häufig wechselnden Entwicklern ein Problem darstellt. Außerdem kann OSGi nur mit Bundles umgehen, andere Abhängigkeiten, die nicht als Bundles verfügbar sind, müssen umgewandelt oder in das Bundle integriert werden.

Es können Declarative Services verwendet werden, um die Komplexität von Bundles und den Overhead beim Starten zu reduzieren (vgl. Hall u. a., 2011). Dabei wird die notwendige Konfiguration in einer Manifest-Datei innerhalb des Bundles definiert, die auf weitere XML-Dateien verweisen kann. Durch die Verwendung dieser Konfigurationsdateien kann zum Beispiel die Verwaltung von referenzierten Abhängigkeiten in der Implementierung vereinfacht werden.

Der OSGi-Standard beinhaltet auch das Einbetten von OSGi-Frameworks in eine Anwendung. Deshalb gibt es hierfür eine Framework übergreifende Schnittstelle. Das Einbetten ist besonders dann sinnvoll, wenn nicht alle Module einer Anwendung in OSGi-Bundles überführt werden können oder wenn die Laufzeitumgebung von der Anwendung selbst kontrolliert werden soll. Ein häufiger Anwendungsfall hierfür ist die Implementierung eines Plugin-Systems mit Hilfe einer eingebetteten Laufzeitumgebung. Dabei müssen nur die Plugins als Bundles vorliegen.

Innerhalb des Living Places wird OSGi bereits teilweise verwendet. Es wurde zusammen mit der Integration von Complex-Event-Processing eingeführt (vgl. Otto, 2013) und wird momentan unter anderem in den Komponenten für die Berechtigungsverwaltung eingesetzt (vgl. Bornemann, 2013). Der Entwurf sieht vor, dass alle Sensoren und Aktoren als OSGi-Bundles implementiert werden und sich bei zwei zentralen Diensten der Information- und Action-Registry registrieren. Abbildung 3.1 zeigt diesen Vorgang. Jeder Student, der eine neue Komponente entwickeln möchte, muss sich zuerst mit OSGi, den notwendigen Konfigurationsdateien und dem Build-Vorgang von Bundles vertraut machen. Dies ist einer der Gründe dafür, dass wenige neue Komponenten innerhalb dieser Architektur entstanden sind. Außerdem ist momentan keine vollständige Integration des Altsystems vorhanden.

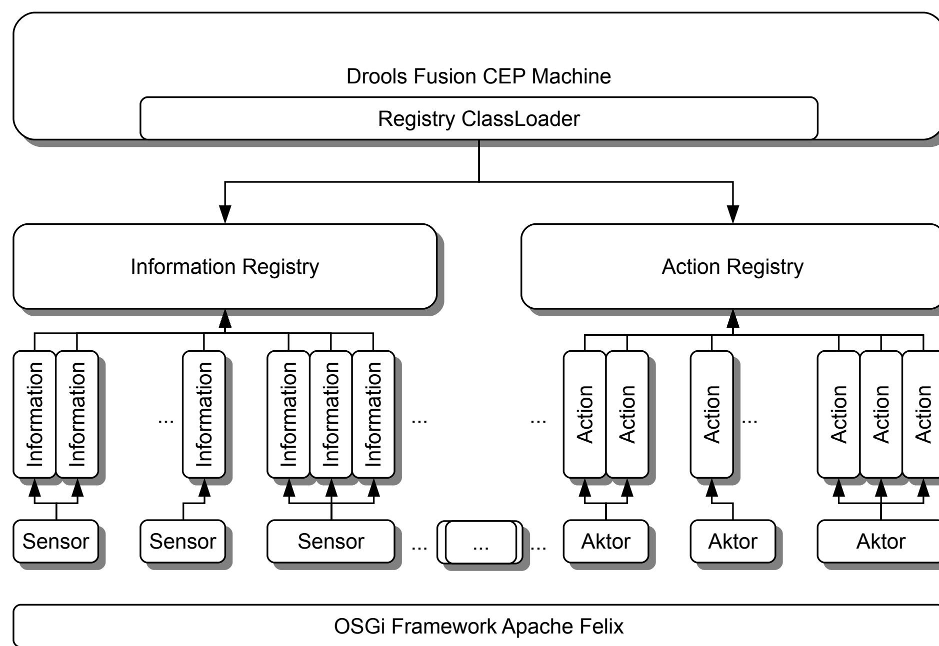


Abbildung 3.1: OSGi-Integration im Living Place. Otto (2013)

Auch in anderen Projekten im Kontext von Smart-Environments wird OSGi als Teil der Middleware verwendet. Hierzu zählt zum Beispiel die Service-orientierte Architektur für Smart-Homes von Wu, Liao und Fu (2007), die Mobile-Agent-Technologie mit OSGi verknüpft. Dabei entsteht ein System aus miteinander in einem Peer-to-Peer-Netzwerk verbundenen OSGi-Plattformen. Ziel dabei ist es, die Probleme zu umgehen, die bei der Verwendung von OSGi und der damit oft verbundenen Architektur mit einem zentralem Server entstehen.

In dieser Arbeit soll ebenfalls OSGi verwendet werden. Um die Entwickler nicht zu belasten, wird hier aber ein anderer Ansatz gewählt. Das Design wird nicht wie für OSGi-Applikationen üblich vollständig angepasst und alle Artefakte werden als Bundles bereitgestellt, sondern die OSGi-Laufzeitumgebung soll nur eine Komponente des Systems werden, die Agenten ausführen kann und von der Middleware verwaltet wird. Dadurch kann das System von Grund auf als agentenbasiertes System konzipiert werden. Der zusätzliche Konfigurations- und Implementierungsaufwand soll weitestmöglich vom Entwickler ferngehalten werden. Dies kann durch den Einsatz von Declarative Services in Verbindung mit angepassten Build-Tools erreicht werden. Die Konfiguration wird dann zu großen Teilen generiert und kann bei Bedarf vor dem Deployment durch erfahrene Entwickler angepasst werden.

Die Konfiguration der Bundles kann hier gut automatisiert werden, weil sie für die meisten Komponenten wie Sensoren und Aktoren einfach ist. Schnittstellenbibliotheken, die die Nachrichtenformate definieren, exportieren ihre gesamte Schnittstelle und importieren in der Regel nur das Paket für die Serialisierung. Agenten importieren die verwendeten Schnittstellen und haben meist keine weiteren Exporte. Die Importe für das Framework und die verwendeten Bibliotheken sind für die meisten Agenten gleich und müssen somit nicht angepasst werden. Verwendet ein Agent besondere Importe von Bibliotheken, die von anderen Komponenten nicht gebraucht werden oder die nicht als Bundle verfügbar sind, können diese direkt in das Bundle integriert werden.

Das System wird so entworfen, dass grundsätzlich alle Komponenten auch außerhalb der Laufzeitumgebung lauffähig sind. Damit wird sichergestellt, dass Komponenten, die nicht als Bundle erstellt werden können, trotzdem im System installiert werden können. Hierzu zählen auch externe Systeme und Komponenten mit vielen Abhängigkeiten, die nicht als Bundles vorliegen. Diese werden extern ausgeführt, können aber trotzdem an die Middleware angebunden werden, alle Dienste benutzen und vom System überwacht werden.

3.2 Middleware

Die Middleware bildet die Kernkomponente der neuen Systemarchitektur. Im Folgenden werden der Aufbau, die einzelnen Komponenten und Dienste der Middleware im Detail erläutert.

Die wichtigste Middleware-Komponente ist zuständig für die Gruppenkommunikation. Sie

realisiert einen Publish-Subscribe-Dienst, der von allen Agenten zur Kommunikation mit anderen benutzt wird. Weitere Komponenten der Middleware sind für die Verwaltung und Überwachung von Agenten, die Kontrolle der Laufzeitumgebung und die Integration von Complex-Event-Processing zuständig.

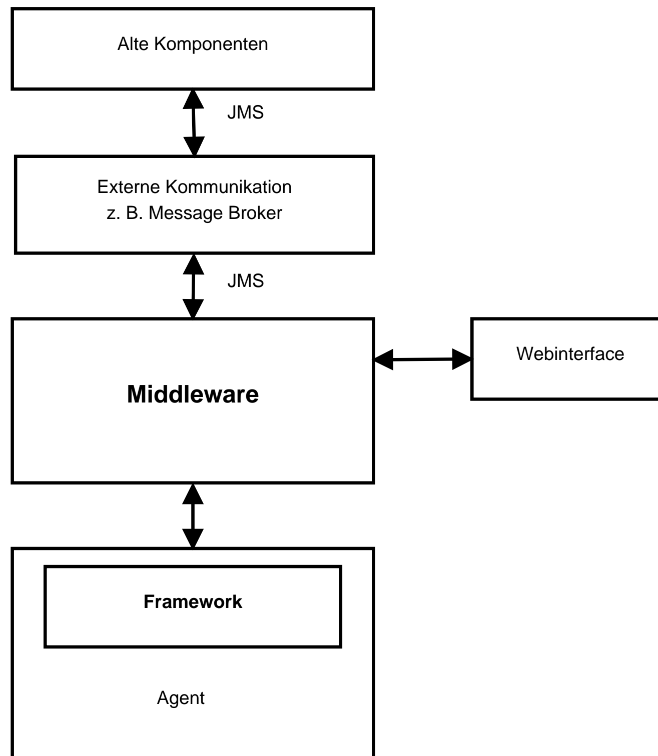


Abbildung 3.2: Blackbox-Ansicht der Middleware.

Abbildung 3.2 zeigt eine Blackbox-Ansicht der Middleware. Die externen Komponenten greifen alle über die Schnittstellen der Middleware auf die einzelnen Dienste zu. Das Webinterface bietet Entwicklern eine graphische Benutzeroberfläche zu diesen Schnittstellen. Alle mit der Middleware verbundenen Komponenten sind Agenten. Die Kommunikation mit der Middleware wird mit dem Framework gekapselt. Mit speziellen Agenten kann die Gruppenkommunikationskomponente der Middleware an andere Kommunikationstechnologien angeschlossen werden. Hier ist als Beispiel die Anbindung eines Message-Brokers über den Java Messaging Service (JMS) eingezeichnet.

3.2.1 Schnittstellen von Agenten

Die Schnittstellen der einzelnen Agenten basieren auf einer Reihe von JSON-Nachrichten, die die Anfrage- und gegebenenfalls die Antwortformate definieren. Ein Grund für die Definition aller Schnittstellen in JSON ist eine lose Kopplung zwischen den Agenten. JSON-Nachrichten können einfach ohne weitere Abhängigkeiten über das Netzwerk verschickt werden. Ein weiterer Grund ist die aus der Anforderungsanalyse hervorgegangene Unterstützung von Heterogenität. Für fast alle Programmiersprachen existieren JSON-Bibliotheken, über die die Anbindung an das System umgesetzt werden kann.

Aus der Sicht des Programmierers ist es allerdings schwieriger, mit JSON-Nachrichten umzugehen, als mit Objekten der Programmiersprache. Fehler in JSON-Nachrichten können erst zur Laufzeit entdeckt werden. Häufig werden die Nachrichten erst in dem Agent entdeckt, der die Schnittstelle anbietet. Dadurch werden Fehler sehr spät erkannt. Deswegen wird jede Schnittstelle durch eine Bibliothek definiert, die eine Reihe von Objekten für Anfragen und die dazugehörigen Antworten enthält. Teil der Schnittstelle sind die Serialisierungsfunktionen für die Objekte, die die Umwandlung zwischen den Objekten und JSON-Nachrichten definieren.

Die vollständige Definition der Schnittstelle mit Objekten führt dazu, dass sie bei der Programmierung einfach verwendet werden kann. Es verhindert, dass Parameter falsch benannt oder aus Versehen weggelassen werden. Außerdem werden die Parametertypen vom Compiler überprüft. Weitere Prüfungen der Wertebereiche können innerhalb der Objekte dazu eingesetzt werden, Fehler direkt bei der Objekterstellung zu erkennen.

Die Schnittstellenbibliotheken werden wie alle anderen Artefakte mit einer eindeutigen Versionsnummer versehen. Lässt es sich nicht vermeiden, dass eine Schnittstellenänderung nicht abwärtskompatibel zur alten Version ist, kann mit Hilfe dieser Versionsnummer sichergestellt werden, dass keine Probleme auftreten. Das System kann beim Deployment der Agenten die Versionsnummern prüfen und inkompatible Versionen melden, bevor sie geladen werden. Die erforderlichen Versionsnummern können von den Entwicklern in Intervallen angegeben werden, um eine größtmögliche Kompatibilität zwischen den Komponenten zu erreichen.

Die Dokumentation der Schnittstellen kann mit JavaDoc oder ScalaDoc[2] innerhalb des Quelltextes erfolgen. Dies macht die Dokumentation in vielen Entwicklungsumgebungen direkt während der Programmierung verfügbar. Externe Formate für die Dokumentation können

[2] ScalaDoc Style Guide - http://docs.scala-lang.org/style/scaladoc.html; letzter Zugriff: 26.09.2014

teilweilse automatisch generiert werden, was die Dokumentation der Schnittstellen weiter beschleunigen kann.

In Scala ist der Implementierungsaufwand von Schnittstellen auf diese Weise sehr gering. Einzelne Nachrichten können durch Case-Classes definiert werden. Dafür sind lediglich die Angaben des Objektnamens und der Parameter mit Typinformation notwendig. Eine Gleichheitsrelation und eine Hash-Methode werden automatisch generiert (vgl. Odersky u. a., 2011). Die Definition der Serialisierung kann automatisch aus der Case-Class-Definition generiert werden. Für andere Objekte kann die Serialisierung manuell definiert werden. Die automatische Generierung wird durch Scala-Macros realisiert und wird vom Compiler ausgeführt. Dadurch werden Fehler direkt bemerkt und es entsteht kein zusätzlicher Aufwand beim Ausführen. Um dem Entwickler die vollständige Kontrolle darüber zu geben, welche Objekte deserialisiert werden können, kann er dies über eine Liste von Klassennamen steuern.

```
0  /** Befehl zur Steuerung einer einzelnen nicht farbigen Lampe.
1    *
2    * @param id ID der Lampe
3    * @param intensity Intensität der Lampe (0 - 255)
4    */
5  case class ControlLight(id: Int, intensity: Int)
6
7  /** Befehl zur Abfrage des Status einer einzelnen nicht
8    * farbigen Lampe. Die Antwort wird an die Antwortgruppe in Form
9    * einer StateOfLight Nachricht gesendet.
10   *
11   * @param id Die ID der Lampe
12   * @see StateOfLight
13   */
14 case class GetStateOfLight(id: Int)
15
16 /** Antwort auf die Abfrage des Status einer einzelnen
17   * nicht farbigen Lampe.
18   *
19   * @param id ID der Lampe
20   * @param intensity Intensität der Lampe (0 - 255)
21   */
22 case class StateOfLight(id: Int, intensity: Int)
23
```

Abbildung 3.3: Ein Teil der Schnittstellendefinition zur Lichtsteuerung im Living Place.

Abbildung 3.3 zeigt ein Beispiel für eine Schnittstellendefinition der Lichtsteuerung im Living Place. Es werden Klassen für die Nachrichten definiert, die die Steuerung und Statusabfrage von Einzellampen ermöglichen. Alle Nachrichten sind typsicher und können über ScalaDoc mit weiteren Informationen versehen werden.

3.2.2 Kommunikation

Das System basiert auf einer von der Middleware verwalteten Komponente zur Gruppenkommunikation. Ein Agent kann Gruppen abonnieren und Nachrichten auf beliebigen Gruppen veröffentlichen. Eine Gruppe kann eindeutig über ihren Namen identifiziert werden und existiert genau dann, wenn sie mindestens einen Abonnenten für Nachrichten hat. Über die Gruppen können ausschließlich Strings oder Objekte zusammen mit einem Serialisierer verschickt werden. Das Nachrichtenformat ist nur vom verwendeten Serialisierungsmechanismus abhängig und kann damit ausgetauscht werden, ohne die Kommunikationsdienste anpassen zu müssen.

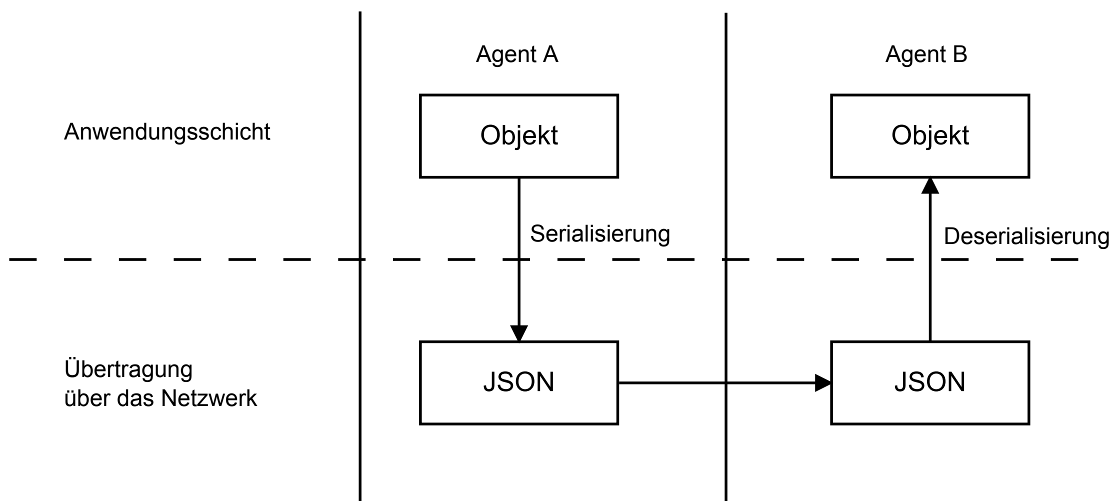


Abbildung 3.4: Protokollstack mit Serialisierung von JSON-Nachrichten.

Abbildung 3.4 zeigt den Ablauf beim Versenden einer Nachricht zwischen zwei Agenten. Der Ablauf kann mit Hilfe eines einfachen Protokollstacks beschrieben werden. Sender und Empfänger befinden sich in der Anwendungsschicht. Wird eine Nachricht verschickt, wird diese mit Hilfe des definierten Serialisierers in diesem Beispiel in das JSON-Format überführt und über das Netzwerk verschickt. Auf der Seite des Empfängers wird ein Deserialisierer

eingesetzt, um die JSON-Nachricht wieder in ein Objekt umzuwandeln. Beide Agenten müssen hierfür die Schnittstellenbibliothek importieren, die die Objektbeschreibung enthält. Die Entscheidung, welches Objekt für eine neue Nachricht instanziiert werden muss, wird auf Basis einer Typinformation in der Nachricht getroffen, die vom Serialisierer angelegt wird. Mit diesem Mechanismus können vom Framework auch weitere Informationen, wie zum Beispiel der Sender oder eine Antwortgruppe, in eine Art Header vor die Nachricht geschrieben werden.

Beim Veröffentlichen einer Nachricht werden keine Garantien gegeben. Nachrichten können durch Netzwerkfehler oder Systemausfälle verloren gehen. Außerdem wird die Reihenfolge der Nachrichten nicht kontrolliert. Dies sollte bei der Implementierung von Agenten berücksichtigt werden. Wird für einen Dienst eine zuverlässige Kommunikation benötigt, kann diese auf Basis der vorhandenen Infrastruktur implementiert werden. Dafür kann zum Beispiel ein Acknowledgment jeder Nachricht oder eine Sequenznummer eingesetzt werden. Solche Implementierungen werden vom Framework unterstützt. Viele Nachrichten, wie zum Beispiel Sensordaten, die periodisch veröffentlicht werden, sind nicht systemkritisch und benötigen keinen zuverlässigen Kommunikationskanal. Dies spart Ressourcen, erhöht die Systemperformance und vereinfacht den Systemaufbau sowie die Implementierung. Die Reduzierung von Antwortzeiten ist wichtig, um die Anforderung der Reaktionsfreudigkeit umsetzen zu können.

Die Verwendung von Gruppenkommunikation macht das System sehr offen. Andere Komponenten können einfach Schnittstellen benutzen und weitere Sensordaten abonnieren. Dies macht das System leicht erweiterbar und die Komponenten lokationsunabhängig. Das ist eine Voraussetzung für die Umsetzung von Mobilität. Außerdem können Entwickler zu jeder Zeit die Kommunikation im System überwachen, was besonders bei der Fehlersuche hilfreich ist.

3.2.3 Kontextverarbeitung

Die Analyse ergab als eine wichtige Anforderung die Unterstützung der Kontextverarbeitung. Auf der Seite der Middleware wird die Kontextverarbeitung in dem System durch die Integration von Complex-Event-Processing (CEP) unterstützt. Die CEP-Engine wird durch einen Agenten gekapselt, der anderen Komponenten die Möglichkeit bietet, Abfragen an die CEP-Engine zu stellen. Die Abfrageergebnisse sind wieder Eventströme, die über dafür vorgesehene Gruppen an den Agenten weitergeleitet werden. Da alle beteiligten Systemteile einfache Agenten sind, wird hierfür die normale Gruppenkommunikation des Systems benutzt. Der Ablauf wird in Abbildung 3.5 verdeutlicht.

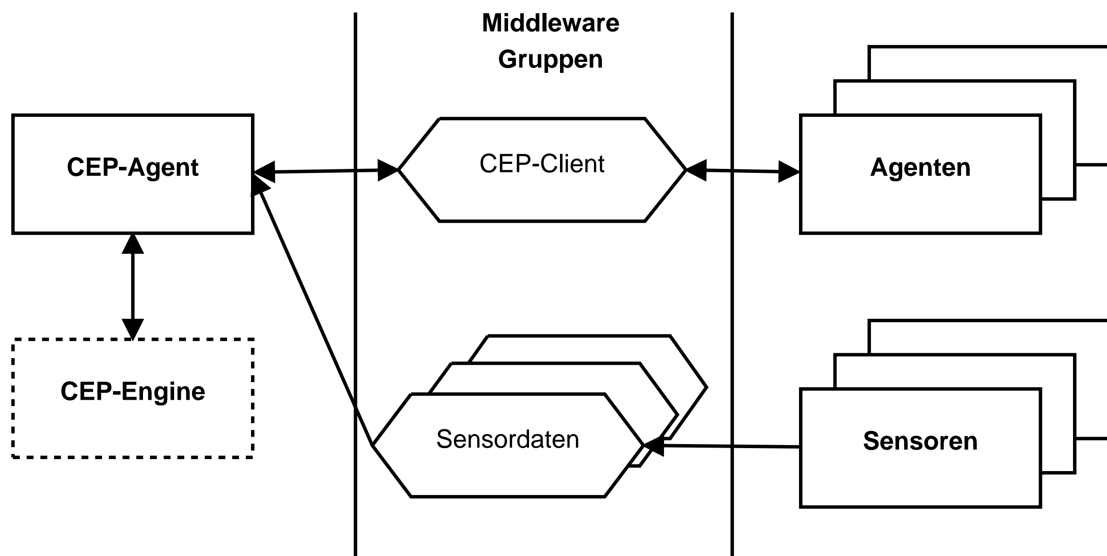


Abbildung 3.5: Integration von Complex-Event-Processing in das System.

Viele CEP-Engines basieren auf nativen Objekten. Das hat die Vorteile, dass sie direkt aus dem Programm heraus benutzt werden können und zur Definition des Objekt-Modells kein zusätzliches Wissen benötigt wird. In einem verteilten System würde dies aber bedeuten, dass die Komponente, in der die CEP-Engine ausgeführt wird, alle Objekte, die verarbeitet werden sollen, kennen muss, was dem restlichen Systemdesign widerspricht und zu einer starken Kopplung führt. Das System könnte so nur mit größerem Aufwand um weitere Objekttypen erweitert werden.

In dieser Implementierung wird deswegen Esper[3], eine auf Java basierende Open-Source-Implementierung für Complex-Event-Processing, verwendet. Objekte können in Esper durch eine Liste von Schlüssel-Wert-Paaren definiert werden. Zusätzlich ist dabei allerdings eine Metabeschreibung des Objektes erforderlich, die angibt, welche Attribute ein Objekt besitzt. Diese Repräsentation passt sehr gut zu JSON-Nachrichten, weil das JSON-Format ebenfalls aus verschachtelten Schlüssel-Wert-Paaren besteht.

Die von der CEP-Engine zu verarbeitenden Daten stammen meist von den Sensoren. Wird ein neuer Sensor-Agent im System gestartet, überprüft er vor der Veröffentlichung der ersten Daten die Registrierung des Eventtyps im CEP-Agenten. Dieser Schritt wird vom Framework vollständig abgedeckt. Zur Registrierung ist lediglich eine Beispielnachricht mit allen Attri-

[3]EsperTech - http://esper.codehaus.org; letzter Zugriff: 26.09.2014

buten und die Angabe der Gruppe, auf der die Daten gesendet werden, erforderlich. Diese Angaben können über die Konfiguration des Sensor-Agenten abgedeckt werden. Danach werden alle Nachrichten, die auf der angegebenen Gruppe versendet werden, von der CEP-Engine verarbeitet.

Die CEP-Engine ist für die Verarbeitung großer Mengen von Events entwickelt worden und kann bei Bedarf auf einem externen Rechner ausgeführt werden, um eine Beeinflussung der restlichen Systemperformance auszuschließen. Eine Skalierung der CEP-Engine kann zu späterer Zeit hinzugefügt werden, da Complex-Event-Processing die Verarbeitung auf mehreren Knoten zulässt (vgl. Luckham, 2002). Die Aufteilung der Events auf die Knoten kann dabei von der CEP-Engine umgesetzt werden oder von CEP-Agenten übernommen werden.

Der CEP-Agent kann gut dazu benutzt werden, Kontext-Interpreter zu entwickeln. Hierzu muss lediglich ein neuer Agent gebaut werden, der eine Abfrage an die CEP-Engine erstellt, die Ergebnisse bei Bedarf weiterverarbeitet und als Sensor wieder veröffentlicht. Durch Angaben in der Konfigurationsdatei können die durch Interpretation gewonnenen Daten wieder von der CEP-Engine verarbeitet werden.

3.2.4 Laufzeitumgebungen

Ein wichtiger Teil der Middleware sind die Laufzeitumgebungen. Einer der Gründe für den Einsatz von Laufzeitumgebungen ist die Erleichterung von Installation und Konfiguration von Komponenten im System. Dies ist eine der in der Analyse identifizierten Anforderungen. Innerhalb einer Laufzeitumgebung können Agenten in einer vom System kontrollierten Umgebung ausgeführt werden. Das vereinfacht das Deployment und macht Veränderungen des Systems zur Laufzeit möglich.

Eine Laufzeitumgebung ist eine Komponente der Middleware, die eine Schnittstelle besitzt, mit der Informationen über die enthaltenen Agenten eingeholt werden können. Außerdem können die Agenten in der Laufzeitumgebung über die Schnittstelle gestoppt oder gestartet und neue Agenten können hinzugefügt werden. Die Middleware kann mehrere Laufzeitumgebungen verwalten und damit bei wachsender Anzahl von Agenten skalieren. Außerdem können so alternative Implementierungen für Laufzeitumgebungen zum System hinzugefügt werden, wenn sie die Schnittstelle zur Middleware implementieren. Jede Laufzeitumgebung stellt Dienste ausschließlich für die Agenten bereit, die sie kontrolliert.

Die Implementierung der hier verwendeten Laufzeitumgebungen basiert auf OSGi. Das bedeutet, dass alle Komponenten und ihre Abhängigkeiten als OSGi-Bundles vorliegen müssen. Die Installation von Agenten erfolgt über eine URL, die auf einen Ort im Dateisystem oder Netzwerk zeigen kann. Vollständig automatisiert läuft das Deployment bei einer Installation aus einem Repository. Das System bietet Unterstützung für die Auflösung von Adressen verschiedener Artefakt-Repositories. Abhängigkeiten der Komponenten können durch ein OSGi-Bundle-Respository aufgelöst werden.

Ein Agent kann gestartet werden, sobald alle Artefakte, die er benötigt, verfügbar sind. Dies wird von der OSGi-Plattform sichergestellt. Ein Agent kann nur zwei Zustände besitzen: Entweder er wird ausgeführt oder nicht. Für die Migration von Agenten ist eine Unterbrechung der Ausführung notwendig. Bundles können nur entfernt werden, wenn alle abhängigen Agenten beendet wurden. So ergibt sich ein Lebenszyklus, der dem eines OSGi-Bundles sehr ähnlich ist. Der Unterschied besteht darin, dass das Starten von Agenten nicht von der OSGi-Plattform verwaltet, sondern durch die Middleware kontrolliert wird.

Ein Koordinator nimmt die Steuerungsbefehle für Komponenten entgegen und leitet sie an die registrierten Laufzeitumgebungen weiter. Beim Starten von Agenten kann angegeben werden, dass diese überwacht werden sollen. Stoppt der Agent aufgrund eines internen Fehlers oder weil seine Laufzeitumgebung nicht mehr erreichbar ist, veranlasst der Koordinator den erneuten Start des Agenten. Dies ist ein Mechanismus, um eine hohe Fehlertoleranz im System zu erreichen. Über den gleichen Mechanismus können beim Start der Middleware Agenten konfiguriert werden, die automatisch gestartet werden sollen. Weiterhin kann der Koordinator den Umzug von Agenten zwischen Laufzeitumgebungen veranlassen, um eine Lastverteilung oder eine Optimierung der Kommunikationswege zu erreichen.

Die Agenten innerhalb einer Laufzeitumgebung werden von einem weiteren Agenten kontrolliert, der zusammen mit dieser gestartet wird. Der Agent führt die Änderungen an den Komponenten durch und kann Agenten starten und stoppen. Alle Agenten laufen in derselben JVM und benutzen Ressourcen eines gemeinsamen Threadpools. So ist es möglich, eine große Anzahl von Agenten gleichzeitig auszuführen. Da eine Serialierung bei der Kommunikation innerhalb derselben JVM nicht notwendig ist, können die Nachrichten direkt zugestellt werden, was die Latenz und das Risiko von Nachrichtenverlust deutlich reduziert. Nachrichten, die an andere Knoten weitergeleitet werden müssen, werden weiterhin serialisiert. Dieser Vorgang kann aber parallel zur internen Zustellung der Nachrichten ausgeführt werden.

Alle Komponenten können weiterhin außerhalb der Laufzeitumgebungen ausgeführt werden. Dabei hat die Middleware allerdings nur eine eingeschränkte Kontrolle über die Agenten.

3.2.5 Speicherung von Zuständen durch Event-Sourcing

Viele Agenten verwalten einen internen Zustand, um ihre Aufgabe zu erfüllen. Ein Beispiel hierfür ist die Lichtsteuerung im Living Place, die den Status aller Lampen verwalten muss. Bei Auftreten eines Fehlers kann es notwendig sein, einen Agenten neu zu starten. Um zu verhindern, dass bei diesem Vorgang der Zustand verloren geht, muss dieser persistiert werden.

Die Migration eines Agenten ist ähnlich zu dem Vorgehen im Fehlerfall. Soll ein Agent migriert werden, wird er beendet, nachdem sein Zustand eingefroren wurde. Im Anschluss wird er an der gewünschten Stelle neu gestartet und dazu veranlasst, seinen internen Zustand neu zu laden. Die gespeicherten Daten müssen also systemweit zur Verfügung stehen.

In einem auf Nachrichten basierenden System bietet es sich an, Persistenz durch das Mitschreiben von Events zu erreichen. Dieses Vorgehen wird auch als Event-Sourcing bezeichnet (vgl. Fowler, 2005). Ein Agent generiert vor der Änderung seines Zustandes ein Event, das zuerst abgespeichert wird. Sollte der Zustand des Agenten verloren gehen, kann dieser durch erneutes Abspielen der gespeicherten Events wiederhergestellt werden. Damit die Datenmenge der gespeicherten Events nicht zu groß wird und damit die Wiederherstellung sehr lange dauert, können Snapshots des Zustandes angelegt werden. Bei der Wiederherstellung kann so direkt zu den angelegten Snapshots gesprungen werden und es müssen nur Events, die nach der Erstellung des Snapshots eingetroffen sind, erneut betrachtet werden. Abbildung 3.6 zeigt die Migration eines Agenten im Fehlerfall mit Hilfe von Event-Sourcing.

Alle Agenten, die einen Zustand haben, der nicht verloren gehen darf, sollten die Persistierung von Events über die entsprechenden Methoden vom Framework vornehmen. Die Middleware ist so konfiguriert, dass alle so gespeicherten Events und Snapshots in Cassandra[4], einer verteilten Datenbank abgelegt werden und somit innerhalb des gesamten Systems zugänglich sind. Zusätzlich kann ein Entwickler eigene Mechanismen implementieren, die beim Start oder Neustart eines Agenten ausgeführt werden sollen.

[4]Apache Cassandra - http://cassandra.apache.org/; letzter Zugriff: 26.09.2014

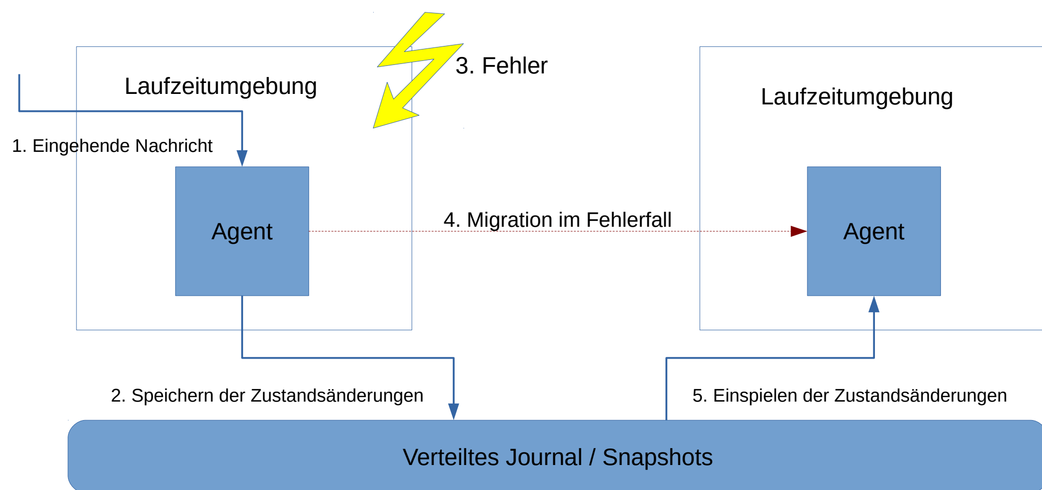


Abbildung 3.6: Fehlertoleranz durch Event-Sourcing.

Das Mitschneiden von Nachrichten und Events kann auch zur Überwachung des Systems eingesetzt werden. Im Fehlerfall können so nicht nur die Protokollausgaben, sondern auch die eigentlichen Nachrichten durchsucht werden. Gespeicherte Events oder Snapshots eines Agenten können dazu benutzt werden, seinen genauen Zustand zu einem bestimmten Zeitpunkt zu analysieren. Bei einem vollständigen Mitschnitt aller Nachrichten wäre es möglich durch die erneute Verarbeitung der gespeicherten Nachrichten, einen Fehlerzustand im gesamten System zu reproduzieren, um ihn genauer zu untersuchen (vgl. Fowler, 2005).

3.2.6 Monitoring von Agenten

Das Monitoring von Agenten soll dazu eingesetzt werden, die Fehlertoleranz des Systems zu erhöhen. Das System soll in der Lage sein, Fehler selbst zu erkennen und darauf zu reagieren. Die Überwachungsaufgaben werden innerhalb des Systems auf die Agenten verteilt. Dieser Vorgang wird auch als Self-Healing bezeichnet (vgl. Dashofy u. a., 2002).

Jeder Agent im System kann andere Agenten oder Agentengruppen überwachen und sich über Veränderungen im Lebenszyklus informieren lassen. Überwacht ein Agent die Agenten, von denen seine Funktionalität abhängt, kann er sofort reagieren, wenn einige davon nicht mehr erreichbar sind. Das Monitoring anderer Agenten kann aber auch eingesetzt werden, um Ressourcen für einen Dienst freizugeben, wenn dieser nicht mehr benötigt wird. Außerdem

wird das Monitoring eingesetzt, um Entwicklern oder Agenten einen aktuellen Überblick über die laufenden Agenten im System zu verschaffen. Dies ist notwendig für die Umsetzung der Anforderung „Übersicht und Kontrolle".

Agenten müssen sich während des Starts bei der Middleware registrieren. Dieser Vorgang wird durch das Framework automatisiert, sodass dafür kein Entwicklungsaufwand entsteht. Nach der Registrierung sendet die Middleware regelmäßige Heartbeat-Nachrichten an den Agenten, die automatisch beantwortet werden. Der Monitoring-Dienst bietet eine Schnittstelle, über die die Beobachtung anderer Agenten in Auftrag gegeben werden kann. Nach einem Aufruf wird der Agent über alle Statusänderungen der zu überwachenden Agenten informiert.

Die Überwachung von Agenten würde sehr viele Nachrichten produzieren, wenn jeder Agent die Nachrichten einzeln über das Netzwerk beantworten müsste. Dieses Problem besteht aber nur für externe Agenten. Interne Agenten inklusive der Agenten, die in einer der Laufzeitumgebungen ausgeführt werden, laufen in derselben JVM wie einer der Monitor-Agenten und können damit ohne Probleme überwacht werden. Die externen Agenten werden in Gruppen organisiert, um den Ressourcenbedarf zu reduzieren. Dies ist besonders dann sinnvoll, wenn eine Agentengruppe gemeinsam einen Dienst bereitstellt. Innerhalb der Gruppe gibt es nur einen Agenten, der sich bei der Middleware registriert und überwacht wird. Dieser übernimmt die Überwachungsaufgaben für die anderen Agenten in der Gruppe und kann bei Bedarf Informationen an die Middleware weitergeben. Das System kann durch das Hinzufügen weiterer Monitor-Agenten in den einzelnen Knoten beliebig skaliert werden.

Eine weitere Funktionalität des Monitoring-Dienstes sind Plugins. Agenten können die Middleware beauftragen, sie über bestimmte Arten von neuen Agenten zu informieren, um darauf zu reagieren. Hierüber kann die modulare Erweiterung von Diensten realisiert werden. Die Kommunikation zwischen den Agenten und den Plugins läuft über dafür zugewiesene Gruppen. Damit können bei Bedarf auch externe Komponenten zur Erweiterung der Funktionalität eingesetzt werden. Zusammen mit der Möglichkeit, Agenten über eine Programmierschnittstelle zur Laufzeit nachzuladen, ergibt sich hieraus ein dynamisches und modulares System.

3.2.7 Verteilte Ausführung

Die Middleware ist so implementiert, dass ihre Dienste auf mehrere Knoten verteilt werden. Die Middleware und insbesondere der Kommunikationsdienst sind kritische Bestandteile des Systems. Eine auf mehrere Knoten verteilte Architektur verhindert den Ausfall des Gesamtsys-

tems im Fehlerfall. Solange einer der Knoten erreichbar ist, kann das System weiterarbeiten und versuchen, sich wieder herzustellen. Außerdem können Anfragen an die Middleware zwischen den Knoten aufgeteilt werden. Diese Redundanz und Lastverteilung zwischen den Middleware-Knoten soll die Anforderungen nach Fehlertoleranz und Skalierbarkeit umsetzen.

Die einzelnen Middleware-Knoten bilden einen Cluster, indem sie miteinander über ein Gossip-Protokoll (vgl. Khambatti u. a., 2003) kommunizieren. Sie tauschen untereinander periodisch die bekannten Knoten aus, sodass sie ein einheitliches Bild vom Cluster bekommen. Unter den Knoten wird ein Anführer bestimmt, der Entscheidungen wie das Entfernen eines nicht erreichbaren Knotens trifft. Die Implementierung der Cluster-Funktionalitäten ist im verwendeten Aktor-Framework enthalten. Der Aufbau eines Middleware-Clusters mit mehreren Agenten und einer Laufzeitumgebung ist in Abbildung 3.7 zu erkennen.

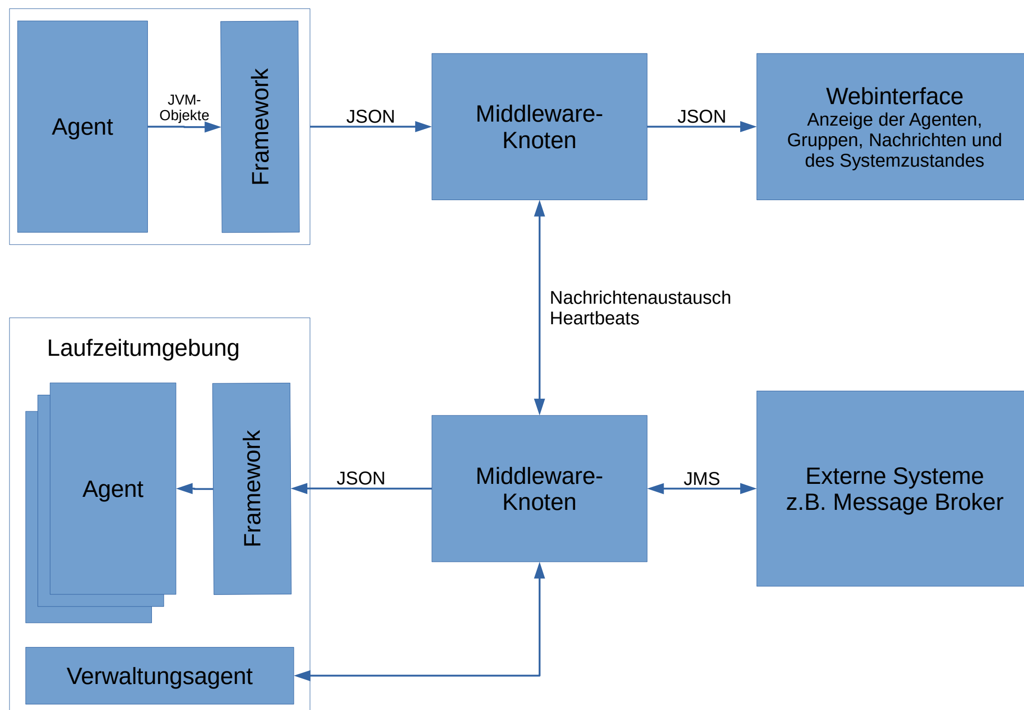


Abbildung 3.7: Aufbau eines Middleware-Clusters mit zwei Knoten und einer Laufzeitumgebung.

Die Implementierung der Gruppenkommunikation ist über alle Middleware-Knoten verteilt. Jeder Agent interagiert mit einem dieser Knoten, kann aber bei Bedarf wechseln. Dies ist zum

Beispiel bei Nichterreichbarkeit des ausgewählten Knotens notwendig. Die Kommunikation und alle anderen verteilten Dienste funktionieren, solange einer der Knoten erreichbar ist. Bei einem Ausfall eines Knotens können allerdings Nachrichten verloren gehen. Jeder Knoten ist dafür verantwortlich, alle Nachrichten an die angemeldeten Agenten weiterzuleiten. Das System versucht, die Zuordnung der Agenten auf die Knoten möglichst gut zu verteilen und kann dabei sowohl die Auslastung der Knoten als auch die Latenz betrachten.

Einige Dienste lassen sich nur mit großen Aufwand auf mehrere Knoten verteilten. Hierzu zählt zum Beispiel der Teil des Monitoring-Dienstes, der eine Liste aller Agenten im System verwaltet. Eine Möglichkeit, dies zu implementieren, wären verteilte Transaktionen oder das Einführen von Eventual Consistency (vgl. Tanenbaum und van Steen, 2008). Dies würde aber die Systemperformance belasten oder kann dazu führen, dass zeitlich begrenzt einige Agenten ein abweichendes Bild vom System haben. Deswegen wird hierfür ein einfacherer Ansatz gewählt, bei dem die Funktionalität von einem einzigen Agenten übernommen wird. Die notwendige Fehlertoleranz wird erreicht, indem der Cluster dafür sorgt, dass immer eine Instanz des Agenten läuft und der Zustand des Agenten über Event-Sourcing gespeichert wird. Ein Nachteil dieses Ansatzes ist es, dass dieser Dienst nicht skaliert werden kann. Bei der Verwaltung einer Agentenliste kann dies aber vernachlässigt werden, weil das Starten und Stoppen von Agenten im Vergleich zu anderen Vorgängen im System selten auftritt. Sollten zu einem späteren Zeitpunkt weitere Anforderungen hinzukommen, kann dieser Teil des Monitoring-Dienstes immer noch ausgetauscht werden.

3.3 Framework zur Entwicklerunterstützung

Das Framework ist eine Softwarebibliothek, die die Implementierung von Agenten vereinfachen soll. Die Bibliothek enthält häufig verwendete Funktionen für Agenten und kapselt die Kommunikation mit der Middleware.

3.3.1 Vom Aktor zum Agenten

Das Aktor-Modell ist ein theoretisches Konzept, welches für die Untersuchung von Nebenläufigkeit und für die Implementierung von verteilten Systemen eingesetzt wird (vgl. Hewitt u. a., 1973; Hewitt, 2014). Ein Aktor kann auf Nachrichten, die er empfängt, reagieren. Er kann Nachrichten an andere ihm bekannte Aktoren schicken und neue Aktoren erstellen.

Jeder Aktor besitzt eine Mailbox, in der alle Nachrichten in der Reihenfolge, in der sie eintref-

fen, abgelegt werden. Der Aktor arbeitet diese Nachrichten sequenziell ab und wartet, wenn alle Nachrichten bearbeitet wurden. Durch diesen Ablauf kann innerhalb eines Aktors ohne besondere Rücksicht auf eine verteilte Ausführung programmiert werden. Änderungen eines im Aktor gekapselten Zustands müssen nicht geschützt werden. Die Nebenläufigkeit im System entsteht durch die voneinander unabhängige Abarbeitung von Nachrichten in den einzelnen Aktoren.

Die Probleme bei der Arbeit mit gemeinsamen Variablen in nebenläufigen Systemen werden mit der Kapselung aller Zustände durch Aktoren umgangen. Um diese Kapselung beizubehalten, sollten auf keinen Fall Zustände über veränderbare Nachrichten ausgetauscht werden. Der Einsatz von nicht veränderlichen Nachrichtenobjekten spielt sehr gut mit Ansätzen aus der funktionalen Programmierung zusammen. Zusätzlich zur Komplexitätsreduzierung durch den Wegfall von Synchronisationsmechanismen wie Mutexen und der damit verbundenen Fehleranfälligkeiten kann dies durch die asynchrone Verarbeitung zu einem Performance-Gewinn führen. Das System ist zu keiner Zeit blockiert, weil es auf geteilte Ressourcen warten muss.

Das Framework basiert auf Akka[5], einem Aktor-Framework für die JVM-Sprachen Java und Scala. Ein Aktor ist ein einfaches Objekt, welches eine Definition für das Verhalten bei eingehenden Nachrichten enthält. Aktoren werden innerhalb von Aktor-Systemen ausgeführt, welche einen Threadpool beinhalten. Die Ausführungszeit der Threads wird mit Hilfe eines Dispatchers auf die Aktoren verteilt. Aktoren sind damit ein sehr leichtgewichtiges Konstrukt im Vergleich zu einem Thread, weil für die Erstellung kaum mehr als ein Objekt initialisiert werden muss. Außerdem ist dadurch der Speicherbedarf von Aktoren sehr gering, was es Aktor-Systemen ermöglicht, enorme Mengen von Aktoren zu verwalten. Diese leichtgewichtige Prozess-Abstraktion wird in einer früheren Entwicklung namens Scala-Actors von Haller und Odersky (2009) beschrieben.

Ein Agent kann als ein um weitere Funktionen erweiterter Aktor betrachtet werden. Diese Funktionen umfassen zum Beispiel einen einfachen Zugriff auf die Gruppenkommunikation und die Interaktionsmöglichkeiten mit der Middleware. Im Allgemeinen ist der Übergang zwischen einem Aktor und einem Agenten fließend, da viele Fähigkeiten, die Agenten zugeschrieben werden, durch das verwendete Aktor-Framework umgesetzt werden können. Zum Beispiel können Aktoren durch Zugriff auf Kontextinformation und Interpretation kontextabhängige Entscheidungen treffen oder ihr Verhalten auf Grund von Erfahrungen verändern.

[5]Webseite des Akka-Projekts - http://akka.io; letzter Zugriff: 26.09.2014

3.3.2 Type-Classes

Type-Classes sind ein Programmierkonstrukt, welches als eine Alternative für Ad-hoc-Polymorphismus eingesetzt werden kann. Type-Classes wurden zuerst für Haskell entwickelt, werden aber mittlerweile auch in anderen Sprachen wie zum Beispiel Scala benutzt (vgl. Oliveira u. a., 2010). Sie dienen dem nachträglichen kontrollierten Überladen von Funktionen (vgl. Wadler und Blott, 1989).

Anwendungsbeispiele für Type-Classes sind Serialisierung und Ordnungs- und Vergleichsoperationen. Die Verwendung von Type-Classes ermöglicht das nachträgliche Erweitern der Relationen ohne Anpassung der einzelnen Objekte. Abbildung 3.8 zeigt ein Beispiel für ein Scala-Programm, welches formatierte Ausgaben durch Type-Classes definiert.

```
0  //Definition
1  //Type-Class
2  trait Formatter[T] {
3    def format(obj: T): String
4  }
5
6  //Implementierung für Integer
7  implicit val intFormatter = new Formatter[Int] {
8    def format(i: Int) = i.toString
9  }
10
11 //Methode mit implizitem Parameter
12 def print[T](obj: T)(implicit formatter: Formatter[T]) =
13   println(formatter.format(obj))
14
15 //Benutzung
16 def main(args: Array[String]): Unit = {
17   //implizite Übergabe
18   print(1)
19   //explizite Übergabe
20   print(2)(intFormatter)
21 }
22
```

Abbildung 3.8: Formatierung als Beispiel für die Verwendung von Type-Classes in Scala.

Ein Mechanismus in Scala, der sehr gut mit Type-Classes zusammenspielt, sind Implicits (vgl. Oliveira u. a., 2010). Das Implicit-Keyword kann für Variablen, Methoden und Klassen verwendet werden und veranlasst den Compiler dazu, für als implizit markierte Parameter

automatisch passende Objekte zu suchen. Werden mehrere Implicits mit passendem Typ für ein Parameter gefunden, wird ein Fehler bei der Kompilierung ausgegeben, weil alle automatischen Auflösungen eindeutig sein müssen. Werden alle Type-Classes und die dazugehörigen Parameter als implizit markiert, braucht der Programmierer die Type-Classes nicht mehr manuell übergeben. Durch den Import verschiedener Mengen von Type-Classes lässt sich das Verhalten von Programmen steuern.

3.3.3 Funktionen

Eine der wichtigsten Funktionen des Framework ist die Kapselung der Kommunikation mit der Middleware. Das Framework baut einen verlässlichen Kommunikationstunnel auf, über den alle Nachrichten an die Middleware verschickt werden. Bricht die Verbindung zur Middleware ab, wird versucht, den Tunnel wiederherzustellen. Dabei werden die Nachrichten zwischengespeichert und später zugestellt. Das Framework speichert eine Liste aller bekannten Middleware-Knoten, um im Falle eines Verbindungsabbruches auf einen anderen Knoten wechseln zu können. Die Kapselung ist mit einem einfachen Acknowledgement realisiert. Nicht bestätigte Nachrichten werden erneut zugestellt und doppelte Nachrichten werden anhand einer Sequenznummer auf der Empfängerseite herausgefiltert. Mehrere Agenten können, um Ressourcen zu sparen, dieselbe Verbindung zur Middleware benutzen, wenn sie in derselben Laufzeitumgebung ausgeführt werden. Dieses Verhalten kann über die Konfiguration des Agenten gesteuert werden und ist für die Implementierung vollständig transparent.

Das Framework übernimmt beim Versenden von Nachrichten die Serialisierung der Objekte. Die Methode zum Senden einer Nachricht enthält eine Type-Class. Damit muss vor dem Senden einer Nachricht die Definition der Serialisierung aus der entsprechenden Schnittstellenbibliothek importiert werden. So ist es ausgeschlossen, dass eine Nachricht verschickt wird, für die keine Serialisierung definiert ist. Bei Bedarf kann der Entwickler die Serialisierung mit der expliziten Übergabe der Type-Class kontrollieren.

Ein Agent kann sich bei beliebig vielen Gruppen anmelden. Nach der Anmeldung werden ihm alle Nachrichten aus dem gesamten System, die auf der Gruppe veröffentlicht werden, zugestellt. Bei der Anmeldung wird eine Klasse übergeben, die für die Deserialisierung der Nachrichten zuständig ist. Damit kann gesteuert werden, welche Nachrichten einer Gruppe deserialisiert werden sollen. Die Deserialisierung wird ebenfalls aus den Schnittstellenbibliotheken importiert. So ist eine Zuweisung von Schnittstellen auf Gruppen möglich. Realisiert ein Agent über eine Gruppe verschiedene Schnittstellen, können mehrere Deserialisierer kombiniert

werden. Kann eine Nachricht nicht deserialisiert werden, wird sie unverändert zugestellt und eine Warnung ausgegeben. Der Agent kann auf dieses Ereignis wie auf normale Nachrichten reagieren.

Sollte es notwendig sein, die Serialisierung oder Deserialisierung abzuschalten, können dafür spezielle Type-Class-Implementierungen verwendet werden, die die Nachrichten als JSON-String weiterleiten. Diese Implementierung kann zum Beispiel beim Testen hilfreich sein.

Für häufig benötigte Funktionen der Middleware, wie zum Beispiel das Anmelden bei einer Gruppe, das Verschicken einer Nachricht oder das Überwachen anderer Agenten, stellt das Framework Methoden bereit, die dann die Nachrichten an die Middleware verschicken. Dadurch wird die Interaktion mit der Middleware weiter vereinfacht. Außerdem können für die Interaktionen die Autovervollständigung der Entwicklungsumgebungen benutzt werden.

Bei der Implementierung von Agenten kommt es oft vor, dass Schnittstellen anderer Dienste aufgerufen werden und das Ergebnis abgewartet werden muss. Dieses Muster wird direkt vom Framework durch eigene Methoden, die asynchrone Rückgabewerte haben, unterstützt. Jeder Agent besitzt vom Framework verwaltete Gruppen, über die die Antworten auf solche Abfragen geschickt werden. Die Informationen über die Antwortgruppe und eine für die Zuordnung benötigte eindeutige Nachrichten-ID werden in einem optionalen Nachrichten-Header übertragen, der vom Framework abgefangen wird. Durch die Verwendung von asynchroner Programmierung ist zu keinem Zeitpunkt eine Blockierung des Programmflusses notwendig.

Asynchrone Aufrufe werden mit Hilfe von Futures realisiert. Futures sind anonyme asynchrone Funktionen, die als Teil der Scala-Bibliothek implementiert sind. Anstatt auf eine Berechnung zu warten, um daraus einen Rückgabewert zu erstellen, kann ein Future generiert werden. Bei jedem Future können beliebig viele Callbacks registriert werden, die ausgeführt werden, sobald ein Ergebnis vorliegt. Damit lässt sich das Warten auf Ressourcen oder Ereignisse vollständig vermeiden. Um auszuschließen, dass Futures nicht terminieren, können Timeouts angegeben werden. Wird die angegebene Zeit überschritten, ist das Ergebnis des Futures ein Fehler. Dies trifft auch zu, wenn während der Ausführung eine Exception geworfen wird. Deswegen sollte der Fehlerfall ebenfalls durch die Callbacks abgedeckt sein.

3.3.4 Komponenten- und Integrationstests

Eine wichtige Anforderung an das System ist die Testbarkeit. Das automatisierte Testen von Komponenten kann die Entwicklung deutlich beschleunigen, weil dadurch schon früh im Entwicklungsprozess Fehler gefunden werden können. Dies trifft im Besonderen bei der Weiterentwicklung oder Überarbeitung einer Komponente zu, wenn die Tests bereits vorhanden sind. Bei der Entwicklung nach einem iterativen Prozess, in dem früh Prototypen vorhanden sein sollen, ist dies oft der Fall. Deswegen bieten das Framework und die Systemarchitektur besondere Unterstützung für Komponenten- und Integrationstests.

Mit Komponententests wird versucht, Fehler in der Implementierung einer Komponente zu finden. Externe Schnittstellen werden hier oft durch Mock-Objekte ersetzt, um vollständige Kontrolle über den Testgegenstand zu erhalten (vgl. Mackinnon u. a., 2001). Die Mock-Objekte können dazu benutzt werden, bestimmte Schnittstelleninteraktionen sicherzustellen und Rückgabewerte an die Komponente zu liefern. In einem nachrichtenbasierten System ist es aufgrund der losen Kopplung besonders einfach, die Implementierung von Schnittstellen auszutauschen.

Das eingesetzte Aktor-Framework Akka enthält bereits Unterstützung für Tests von Aktoren. Diese Unterstützung wird vom Framework für Agenten erweitert. Innerhalb von Tests können einfache Mock-Agenten geschrieben werden, die Schnittstellen aus importierten Bibliotheken realisieren. Die Serialisierung kann hierbei deaktiviert werden. Die Testklasse selber ist ein Agent mit zusätzlicher Funktionalität. Es können Erwartungen über Nachrichten auf Gruppen formuliert werden, die dann vom Framework sichergestellt werden. Dabei können auch Erwartungen formuliert werden, die zeitbasiert sind. Dies ist zum Beispiel für die Prüfung von Timeouts bei asynchronen Aktionen hilfreich. Bei den Komponententests wird die Verbindung zur Middleware vom Framework durch einen Dummy ersetzt, um dem Entwickler vollständigen Zugriff auf die Interaktion mit der Middleware zu geben.

Nach den Tests der einzelnen Komponenten sind Integrationstests sinnvoll, um die Zusammenarbeit und Interaktion zwischen den Komponenten zu testen. Dies stellt in einem verteilten System eine zusätzliche Herausforderung dar, weil die einzelnen Komponenten auf verschiedenen Maschinen ausgeführt werden können. Um dies zu testen, müssen auch die Integrationstests innerhalb mehrerer Java Virtual Machines ausgeführt werden. Hierfür bietet das verwendete Build-Tool über ein Plugin Unterstützung an. Es können mehrere Knoten konfiguriert werden, die unterschiedliche Teile des Systems ausführen. Die Integrationstests für die einzelnen Knoten werden wie die Komponententests implementiert und können dadurch

dieselben Funktionen des Frameworks nutzen. Auf diese Weise kann ein Agent zusammen mit der Middleware getestet werden. Dabei können vor der Ausführung der Tests über die Middleware beliebige Systemkomponenten nachgeladen werden, was dem Entwickler die Kontrolle über die Testumgebung gibt.

Bei der Formulierung solcher verteilten Tests ist es oft notwendig, die Knoten solange warten zu lassen bis sie alle auf einem Stand sind und zum Beispiel bestimmte Dienste geladen wurden. Dies kann durch Barriers realisiert werden, die vom Akka-Framework bereitgestellt werden. Erreicht ein Knoten eine Barrier, wartet er bis alle anderen Knoten eine Barrier mit demselben Namen erreicht haben. Aber auch bei der verteilten Ausführung der Tests wird die spätere Umgebung noch nicht vollständig simuliert. Zum Beispiel können bei einer über das Netzwerk verteilten Ausführung jederzeit Nachrichten verloren gehen. Deswegen bietet das Framework auch hierfür Funktionen an, um Nachrichten abzufangen oder zu verzögern.

3.3.5 Konfigurationsmanagement

Die Konfiguration der Agenten kann in einer JSON-Datei vorgenommen werden, die Teil des Projektes ist. Hier werden neben den projektspezifischen Einstellungen auch Werte für das Framework definiert. Hierzu gehören zum Beispiel die Verbindungsdaten zur Middleware. Beim Build wird die JSON-Datei in das Jar-Artefakt integriert.

Die JSON-Datei besteht aus einem Top-Level-Objekt mit beliebig verschachtelten Unter-Objekten, die die Konfigurationswerte enthalten. Jeder Wert kann über einen eindeutigen Pfad referenziert werden. Einzelne Werte der Konfiguration können so beim Start mit Laufzeitargumenten überschrieben werden. Damit lassen sich zum Beispiel mehrere Startskripte schreiben, die unterschiedlich konfigurierte Agenten starten. Wird ein Agent innerhalb einer Laufzeitumgebung ausgeführt, können ebenfalls Werte überschrieben werden. Diese Anpassungen lassen sich über das Webinterface einstellen.

Wichtige Gruppennamen sollten ebenfalls für jeden Agenten in der Konfigurationsdatei definiert werden. So kann beim Deployment sichergestellt werden, dass Agenten, die miteinander kommunizieren wollen, auch die richtigen Gruppennamen haben. Die Zuweisung der Gruppen wird außerdem vom Framework gekapselt, um eine nachträgliche Anpassung der Namen zu ermöglichen. Damit können Agenten zur Laufzeit die Gruppen wechseln. Andere Konfigurationsänderungen erfordern, wenn keine weitere Anpassung vorgenommen wurde, einen Neustart des Agenten. In den meisten Fällen ist dies aber kein Problem, weil der Neustart von

zustandslosen Agenten oder Agenten mit gesicherten Zuständen vorgenommen werden kann, ohne das restliche System zu beeinflussen.

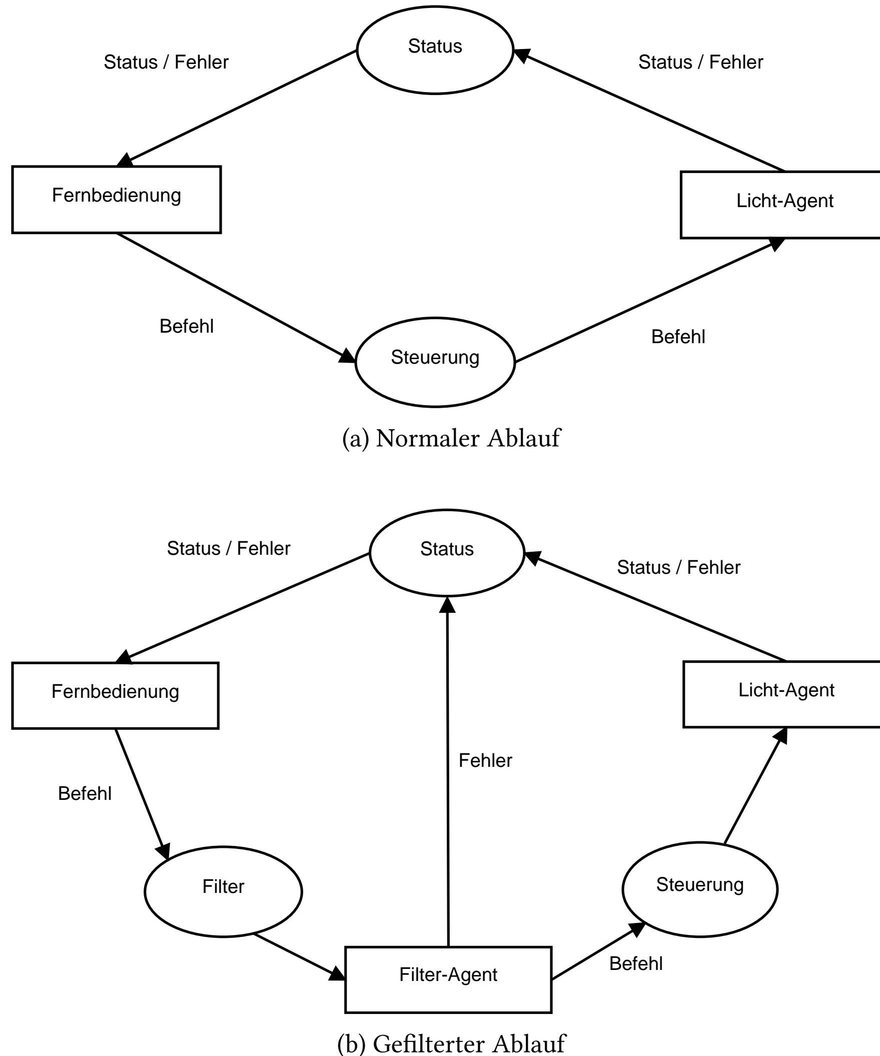


Abbildung 3.9: Einsatz eines Agenten zur Nachrichtenfilterung.

Das Wechseln von Gruppen zur Laufzeit kann dazu benutzt werden, das System den Anforderungen des aktuellen Kontexts anzupassen. Dabei können Änderungen sowohl von Entwicklern, den Benutzern oder auch dem System selbst veranlasst werden. Ein Beispiel hierfür wäre ein Agent, welcher als Nachrichtenfilter eingesetzt wird. Er bekommt eine Eingabe- und eine Ausgabegruppe über die Konfiguration zugewiesen und leitet nur erwünschte Nachrichten

zwischen den Gruppen weiter. Dabei wäre es auch möglich, Meldungen über herausgefilterte Nachrichten zu generieren, damit das System darauf reagieren kann. Durch die dynamische Konfiguration kann dieser Filter-Agent zur Laufzeit zwischen die Kommunikation anderer Agenten geschaltet werden.

Ein Anwendungsfall im Living Place hierfür ist die Fenstersteuerung im Zusammenhang mit den Rollos (s. Abbildung 3.9). Das Öffnen von Fenstern kann bei Wind dazu führen, dass geschlossene Rollos beschädigt werden. Ein Filter-Agent kann alle Steuerungsbefehle herausfiltern, die an die Fenstersteuerung gesendet werden, während das Rollo geschlossen ist. Der aufrufende Agent, wie zum Beispiel die Fernbedienung, wird informiert und kann für den Benutzer eine Fehlermeldung anzeigen. Würde diese Funktionalität in der Fenstersteuerung selber implementiert werden, wäre sie nicht wiederverwendbar und die Komplexität des Fenster-Agenten würde erhöht werden, weil er nicht mehr nur eine genau abgegrenzte Aufgabe erfüllt. Außerdem lässt sich das erreichte Verhalten ohne weitere Anpassungen über die Konfiguration des Systems kontextabhängig ein- und ausschalten.

3.4 Tools zur Entwicklerunterstützung

Im Rahmen dieser Arbeit wurden eine Reihe von Tools für die Entwickler im Living Place bereitgestellt, um die Anforderungen zur Entwicklerunterstützung zu erfüllen. Diese Tools sollen den gesamten Entwicklungsprozess von der Quelltextverwaltung bis hin zum Deployment abdecken. Ein Ziel dabei ist es, dass die einzelnen Systeme möglichst gut zusammenarbeiten. Weiterhin soll die Middleware auf die Tools zugreifen können, um den ganzen Prozess weitestgehend zu automatisieren.

Durch die automatische Erstellung und das automatische Zusammenfügen der Softwarekomponenten kann eine kontinuierliche Integration umgesetzt werden, bei der Builds täglich oder sogar bei jeder Änderung ausgeführt werden können (vgl. Fowler, 2006). Eine kontinuierliche Integration hat den Vorteil, dass Fehler bei der Integration früh bemerkt werden und die häufige Ausführung der automatisierten Tests erzwungen wird. Zusammen mit einem automatisierten Deployment kann so eine Arbeitserleichterung für die Entwickler entstehen.

Die Benutzung der Tools und der Entwicklungsprozess im Living Place sind für die Entwickler innerhalb eines internen Blogs dokumentiert. Die Dokumentation kann von allen überarbei-

tet und erweitert werden. Dies soll dazu führen, dass Fehler und schlechter verständliche Abschnitte schnell überarbeitet werden und die Dokumentation immer aktuell bleibt.

3.4.1 Versionsverwaltung

Als Tool zur Versionsverwaltung wird Git[6] verwendet. Die Verwaltung der Git-Repositories und der Zugangsberechtigungen wird von einer zentralen Gitlab-Installation[7] verwaltet. Diese bietet ein Webinterface, über das Repositories eingesehen und verwaltet werden können.

Gitlab unterstützt das Team beim gemeinsamen Arbeiten am Quelltext. Jedes Repository kann in mehreren Branches bearbeitet werden. Dies ermöglicht es mehreren Entwicklern, den Quelltext gleichzeitig zu bearbeiten, ohne sich gegenseitig zu stören. Die Branches können von Git später wieder zusammengeführt werden, wobei der Entwickler die vollständige Kontrolle darüber behält, welche Änderungen übernommen werden. Hat ein Entwickler die Arbeit an einem Branch abgeschlossen, kann er einen Merge-Request auf den Haupt-Branch starten. Der Entwickler, der die Kontrolle über das Repository hat, kann diesen Request kommentieren, verändern und später, wenn er möchte, übernehmen.

Im Living Place werden diese Funktionen von Git und Gitlab dazu benutzt, die Arbeitsprozesse der Studenten zu unterstützen. Jeder Student hat die vollständige Kontrolle über seine eigenen Projekte, die er zum Beispiel für Projektarbeiten realisiert. Gemeinsame Projekte werden von den Studenten gemeinsam kontrolliert. Damit können einzelne Merge-Requests zuerst im Team diskutiert und getestet werden, bevor die Änderungen übernommen werden. Die Sichtbarkeit des Quelltextes innerhalb des Projektteams fördert die Zusammenarbeit und ermöglicht es, schneller Hilfestellung zu geben, wenn Probleme bei der Implementierung auftauchen.

Innerhalb der Versionsverwaltung können auch Teile der Dokumentation abgelegt werden. Hierzu gehören zum Beispiel Projekt- und Schnittstellenbeschreibungen, die dann zusätzlich zum Quelltext im Gitlab angezeigt werden können. Beispielprojekte können dort zusammen mit Anleitungen veröffentlicht werden, um eine Vorlage für den Projektstart zu geben, welche direkt über Git heruntergeladen werden und danach ausgeführt werden kann. Dies wird benutzt, um die Anforderungen „Zugriff auf Entwickler-Ressourcen“ und „Dokumentation“ zu realisieren.

[6] Git - http://git-scm.com; letzter Zugriff: 26.09.2014
[7] Gitlab - https://about.gitlab.com; letzter Zugriff: 26.09.2014

3.4.2 Build-Tool

Build-Tools können das Erstellen von Softwareartefakten aus dem Quelltext automatisieren. Außerdem enthalten sie oft Mechanismen zum automatischen Auflösen von Abhängigkeiten.

Als Build-Tool wird das Simple Build Tool[8] (SBT) verwendet. SBT ist für die Scala-Programmierung entwickelt worden, kann aber auch für Java-Projekte verwendet werden. Die Abhängigkeitsauflösung erfolgt über einen ähnlichen Mechanismus wie bei Maven[9], welches ein häufig für Java-Projekte verwendetes Build-Tool ist. Damit sind alle Artefakte aus Maven-Repositories zugänglich. Das zentrale Repository Maven-Central[10] enthält zum Zeitpunkt der Erstellung dieser Arbeit über 80.000 verschiedene Artefakte in unterschiedlichen Versionen (vgl. The Central Maven Repository, 2014).

Im Gegensatz zu Maven, wo die Konfiguration im XML-Format geschrieben wird, kann SBT die Build-Anweisungen aus einer Scala-Datei lesen. Die Konfiguration durch ein Programm hat den Vorteil, dass viele Fehler schon bei der Kompilierung bemerkt werden. Zusätzlich dazu ist die Konfiguration über ein eigenes Textformat möglich. Dieses eignet sich besonders gut zur schnellen Definition von wenigen Werten für einfache Builds.

Build-Tools können Artefakte in Repositories veröffentlichen, um sie dort für andere Builds verfügbar zu machen. Zusätzlich zu den Artefakten werden die Build-Konfigurationen mit in das Repository geladen. So werden die Informationen über die Abhängigkeiten jedes Artefakts mit gespeichert. Während eines Builds versucht Maven, alle Abhängigkeiten des Build-Gegenstands über die definierten Repositories auszulösen. Anschließend werden alle transitiven Abhängigkeiten in ein lokales Repository heruntergeladen und stehen damit während des Builds zur Verfügung.

SBT ist über Plugins erweiterbar, die sowohl Konfiguration als auch Build-Tasks hinzufügen können. Für die Entwicklung neuer Agenten kann so ein Plugin benutzt werden, um die Konfiguration der Agenten weiter zu vereinfachen und häufig verwendete Tasks zu automatisieren. Die Konfiguration wird hauptsächlich durch die Bereitstellung passender Standardwerte vereinfacht. Ein Beispiel hierfür sind die Standard-Imports für alle Agenten, die für OSGi definiert werden müssen. Die Imports sind als Collection vordefiniert, was die Verwendung

[8] Simple Build Tool - http://www.scala-sbt.org; letzter Zugriff: 26.09.2014
[9] Apache Maven - http://maven.apache.org; letzter Zugriff: 26.09.2014
[10] Maven Central - http://search.maven.org; letzter Zugriff: 26.09.2014

mit Erweiterung um eigene Imports sehr einfach macht. Außerdem können über das Plugin häufige Vorgänge automatisiert werden. Hierzu zählt das Ausführen des Agenten, ohne dass der manuelle Start der Middleware erforderlich ist.

Weiterhin kann das Build-Tool zur Qualitätssicherung der Komponenten eingesetzt werden. Über Plugins kann die Formatierung des Quelltextes nach vordefinierten Regeln automatisiert werden. Eine einheitliche Formatierung der Quelltexte aller Projekte erleichtert die Lesbarkeit. In einer Umgebung wie dem Living Place muss sichergestellt werden, dass die definierten Vorgaben für Studentenprojekte umgesetzt werden. Dazu zählt zum Beispiel eine einheitliche Projektstruktur. Diese Vorgaben können ebenfalls durch das Build-Tool geprüft werden. Dabei sollten bei Abweichungen eine hilfreiche Fehlermeldung und ein Hinweis auf die Dokumentation ausgegeben werden. So können häufige Fehler vermieden werden, um die Entwicklung weiter zu beschleunigen.

3.4.3 Build-Server

Ein Build-Server kann mit Hilfe einer Build-Definition Artefakte automatisiert aus dem Quelltext erstellen und veröffentlichen. Build-Tools wie SBT können auch auf dem Server eingesetzt werden, was die Wiederverwendung der Build-Definition erlaubt. Der Quelltext kann in der benötigten Version direkt aus der Versionsverwaltung heruntergeladen werden.

Im Living Place wird Jenkins[11] zur Verwaltung der Build-Server eingesetzt. Für die Erstellung der Artefakte mit SBT kontrolliert Jenkins einen Linux-Server, der nur für diese Aufgabe erstellt wurde. Die benötigten Abhängigkeiten werden während des Builds aus einem Artefakt-Repository direkt auf den Build-Server heruntergeladen. Bei jeder Ausführung des Builds werden die im Quelltext definierten Tests ausgeführt und die Ergebnisse mitgeschrieben.

Jenkins bietet die Möglichkeit, die Build-Ergebnisse zu archivieren und zu veröffentlichen. Es wäre möglich, die Artefakte über Skripte direkt zu installieren. Alle fehlerfreien Artefakte werden in einem Repository veröffentlicht, aus dem die Middleware sie direkt herunterladen kann.

Über das Webinterface von Jenkins können die Entwickler den Status oder die eventuellen Fehlermeldungen des Builds und der ausgeführten Tests einsehen. Außerdem können bei Bedarf Builds manuell ausgelöst werden. Jenkins kann so konfiguriert werden, dass alle

[11]Jenkins - http://jenkins-ci.org; letzter Zugriff: 26.09.2014

Änderungen am Quelltext auf dem Server der Versionsverwaltung einen Build auslösen. Damit steht der aktuellste Snapshot der Software immer bereit und Fehler beim Testen werden direkt bemerkt. Die automatische Erstellung der Snapshots erleichtert die Zusammenarbeit mit Entwicklerversionen. Außerdem können Builds für Merge-Requests erstellt werden, um ein direktes automatisches Feedback über die vorgeschlagenen Änderungen zu erhalten. Der Build-Server kann die Benutzer über bestimmte Ereignisse, wie zum Beispiel fehlgeschlagene Builds, per E-Mail informieren oder sie in einem RSS-Feed veröffentlichen.

3.4.4 Artefakt-Repositories

Alle erstellten Artefakte werden in einem Artefakt-Respository hochgeladen und stehen dort für alle Projektmitglieder zur Verfügung. Hierfür wurde Sonatype Nexus[12], ein Repository-Server, der mit Maven kompatibel ist, installiert.

Zusätzlich zu den Bibliotheken können über das Repository auch integrierte Dokumentationen wie JavaDoc oder der Quelltext veröffentlicht werden. Diese zusätzlichen Artefakte können durch Plugins in den Entwicklungsumgebungen automatisch bei der Auflösung der Abhängigkeiten mit eingebunden werden. So stehen dem Entwickler die Dokumentation und auch der Quelltext der Bibliotheken zur Verfügung, was besonders bei der Fehlersuche in Projekten helfen kann.

Die Nexus-Installation dient außerdem als Proxy für externe Maven-Repositories. Alle Artefakte, die von Builds abgefragt werden, werden vom Proxy-Server gespeichert. Damit wird ausgeschlossen, dass ein Build zu späterer Zeit nicht mehr ausgeführt werden kann, weil Artefakte auf einem externen Repository gelöscht oder verändert wurden. Außerdem kann durch die Zwischenspeicherung von Artefakten auf dem Proxy-Server die benötigte Zeit für die Abhängigkeitsauflösung auf dem Build-Server und die Ausführung von Builds in den Netzen der Hochschule deutlich reduziert werden.

Nexus bietet über ein Webinterface direkten Zugriff auf alle Artefakte und zeigt dabei auch die Abhängigkeitsgraphen an. Dies ist hilfreich für Entwickler, um Probleme bei der Abhängigkeitsauflösung, insbesondere mit transitiven Abhängigkeiten, zu finden.

Zum einfachen Deployment von OSGi-Bundles können spezielle Repositories eingesetzt werden. Ein OSGi-Bundle-Repository (OBR) ist ein erweitertes Maven-Repository. Der Unterschied

[12] Sonatype Nexus Repository Manager - http://www.sonatype.org/nexus; letzter Zugriff: 26.09.2014

besteht in einer repository.xml Datei. Diese enthält zusätzliche Informationen zu den OSGi-Bundles im Repository, wie zum Beispiel Versionsnummern, Abhängigkeiten und Schnittstellendeklarationen. Die meisten OSGi-Frameworks enthalten Schnittstellen und Kommandozeilen-Befehle zum Installieren von Bundles aus einem OBR. Dabei kann nicht nur nach dem Artefaktnamen, sondern auch nach OSGi-Services und Schnittstellendeklarationen gesucht werden. Mit Hilfe eines solchen Repositories können Bundles über einen einzigen Befehl zusammen mit allen transitiven Abhängigkeiten in einer Laufzeitumgebung der Middleware installiert werden.

3.4.5 Webinterface der Middleware

Das Webinterface der Middleware ist ebenfalls ein Tool zur Entwicklerunterstützung, das neben anderen Maßnahmen zur Umsetzung der Anforderung „Übersicht und Kontrolle“ bereitgestellt wird. Es bietet verschiedene Funktionen, um die Systemübersicht zu verbessern und das System zu kontrollieren. Das Webinterface stellt eine graphische Benutzeroberfläche zu vielen Schnittstellen der Middleware-Dienste zur Verfügung. Das Webinterface ist ebenfalls ein Agent, welcher außerhalb der Middleware ausgeführt wird. Damit wird sichergestellt, dass alle Funktionen, die das Webinterface ausführen kann, auch von jedem Agenten im System ausgelöst werden können.

Zu den wichtigsten Funktionen zählen eine Übersicht über die momentan ausgeführten Agenten. Es können neue Agenten mit bei Bedarf angepasster Konfiguration gestartet werden. Außerdem können weitere Informationen zu Agenten angezeigt werden. Hierzu können Fehler, Warnungen oder der aktuelle interne Zustand zählen. Weiterhin können die Kommunikationsgruppen und deren Abonnenten eingesehen werden. Entwickler können direkt über die Weboberfläche Gruppen öffnen und die aktuellen Nachrichten verfolgen oder selbst welche versenden.

3.5 Integration in vorhandene Systeme

Eine Anforderung an die Architektur ist die einfache Integration in vorhandene Systeme. Wie auch beim Living Place Hamburg kommt es selten vor, dass eine Softwarearchitektur direkt beim Projektstart eingeführt wird. Bei der Migration zu einer neuen Softwarearchitektur sollten die alten Komponenten weiterhin lauffähig bleiben, weil sonst alle Funktionen wegfallen würden oder zum Zeitpunkt der Umstellung alle Komponenten angepasst werden müssten.

Die neue Systemarchitektur wurde so entworfen, dass sie sich gut in lose gekoppelte Systeme integrieren lässt. Das Living Place Hamburg ist ein Beispiel hierfür. Alle Komponenten kommunizieren ausschließlich über Nachrichten miteinander, die über eine zentrale Gruppenkommunikationskomponente geleitet werden. Die Integration der neuen Architektur kann hier über die Erweiterung der Gruppenkommunikation umgesetzt werden. Es wird eine Brücke zwischen alter und neuer Kommunikationskomponente eingerichtet, sodass alle Nachrichten immer in beiden System veröffentlicht werden. Im Living Place ist diese Kommunikationsbrücke besonders einfach umzusetzen, da die Nachrichtenformate und das Format der Gruppennamen der Systeme identisch sind. Ist das Nachrichtenformat unterschiedlich, müssen die Brücken die Formate umwandeln und somit zwischen den Systemen vermitteln.

Mit Kommunikationsbrücken können unterschiedliche Technologien an die Middleware angebunden werden. Ein weiteres Beispiel ist eine Erweiterung für TCP-Socket-Kommunikation. TCP-Sockets können mit fast allen Technologien inklusive hardwarenahen Implementierungssprachen verwendet werden. Damit kann die Middleware auch in sehr heterogenen Systemen eingebunden werden.

Die Anbindung der Middleware an das Altsystem führt dazu, dass viele Funktionen dort automatisch zur Verfügung stehen. Dies betrifft zum Beispiel die Überwachung der Kommunikation und die Möglichkeit, manuell Nachrichten über ein Webinterface zu verschicken. Andere Funktionen, wie zum Beispiel die Übersicht aller Komponenten, können nicht automatisch erweitert werden. Hierfür ist eine Anmeldung der Komponenten bei der Middleware notwendig. In einem System, das diesen Vorgang bereits vorsieht, kann ein Adapter für die Middleware geschrieben werden, um den alten Registrierungsvorgang abzufangen.

Alle Dienste der Middleware sind durch Schnittstellen über die Gruppenkommunikation und damit über die Kommunikationsbrücken erreichbar. Die Anpassung alter Komponenten kann also direkt erfolgen, ohne sie aus der alten Umgebung entfernen zu müssen. Ist eine Anpassung notwendig, aber zu zeitaufwendig, kann für die Komponente ein Adapter geschrieben werden. Dabei wird die alte Komponente durch einen Agenten gekapselt, indem alle Kommunikationskanäle über ihn umgeleitet werden. Der Adapter kann dann zum Beispiel die für die Registrierung und das Monitoring notwendige Kommunikation übernehmen. Für die Implementierung der Adapter kann das Framework verwendet werden.

3.6 Mögliche Weiterentwicklungen

Die Verarbeitung und Speicherung von Kontextinformationen ist für die Entwicklung von Komponenten in einem Smart-Environment sehr wichtig. In dieser Arbeit wurde mit der Integration von Complex-Event-Processing eine Möglichkeit zur eventbasierten Verarbeitung von Kontextinformationen bereitgestellt. Eine Erweiterung der Middleware um eine Datenbank zur längeren Speicherung und der Abfrage von Kontextinformationen könnte lohnenswert sein.

Das Design des Frameworks könnte bei Bedarf auch in anderen Sprachen implementiert werden, um die Entwicklung von Projekten mit anderen Technologien zu erleichtern. Innerhalb des Living Places würde sich hier eine Implementierung für Microsoft .NET anbieten, weil C# hinter Java im Living Place die am zweithäufigsten verwendete Programmiersprache ist. Mit Akka.NET[13] wird eine Portierung von Akka in C# bereits entwickelt und könnte für die Implementierung verwendet werden. Besonders für hardwarenahe Projekte könnte auch die Entwicklung eines Frameworks auf Basis von libcppa, einem Aktor-Framework für C++ (vgl. Charousset, 2012), interessant sein.

Ein weiteres interessantes Thema könnte die Integration von maschinellen Lernverfahren sein. Diese können insbesondere zur Situationserkennung auf Basis von Kontextinformationen eingesetzt werden. Lernende Agenten können ihr Verhalten aufgrund von gesammelten Erfahrungen anpassen. Lernende Agenten könnten das Verhalten des System verbessern, sie würden aber auch die Komplexität des Systems erhöhen.

[13] Akka.NET Projekt - https://github.com/akkadotnet/akka.net; letzter Zugriff: 26.09.2014

4 Evaluation

Das Framework und die Middleware sollen unter verschiedenen Gesichtspunkten evaluiert werden. Zuerst werden einige Projekte vorgestellt, die im Rahmen dieser Arbeit und meiner vorangegangen Projektarbeiten im Living Place mit Hilfe des Frameworks erstellt wurden. Für einige Anwendungsfälle, wie zum Beispiel die Benutzerinteraktion, ist es interessant, die Latenzzeiten im System zu betrachten. Deswegen werden hierzu anhand von Beispielszenarien Messungen vorgenommen. Dabei wird auch die Skalierbarkeit des Systems im Hinblick auf große Anzahlen von Agenten und Gruppen untersucht.

Im zweiten Teil dieses Kapitels werden die Erfahrungen im Team nach der Einführung der neuen Konzepte ausgewertet. Dies geschieht vor allem am Beispiel des Second-Screen-Frameworks, eines Projektes, welches während der Erstellung dieser Arbeit umgesetzt wurde.

4.1 Fallstudien

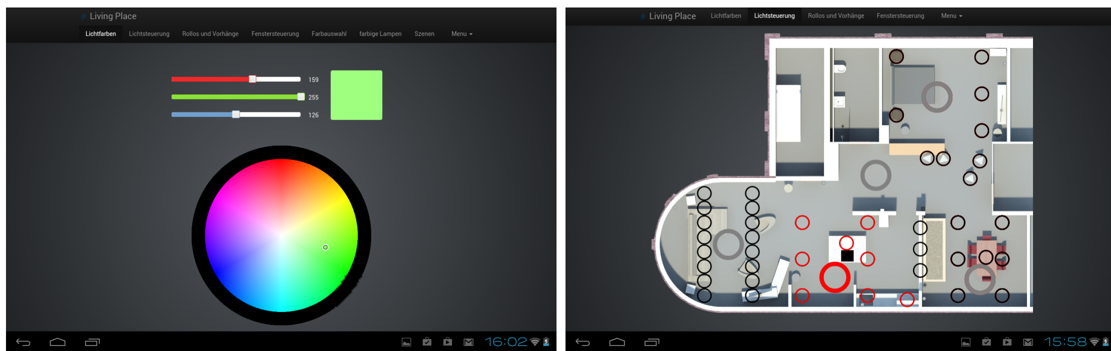
Alle Implementierungen stehen für das Projektteam als Beispiel bereit und wurden so gewählt, dass sie einen möglichst breiten Teil der Funktionalitäten abdecken. Dazu gehört unter anderem die Weiterleitung von Befehlen über verschiedene Schnittstellen und Protokolle.

4.1.1 Fernbedienung

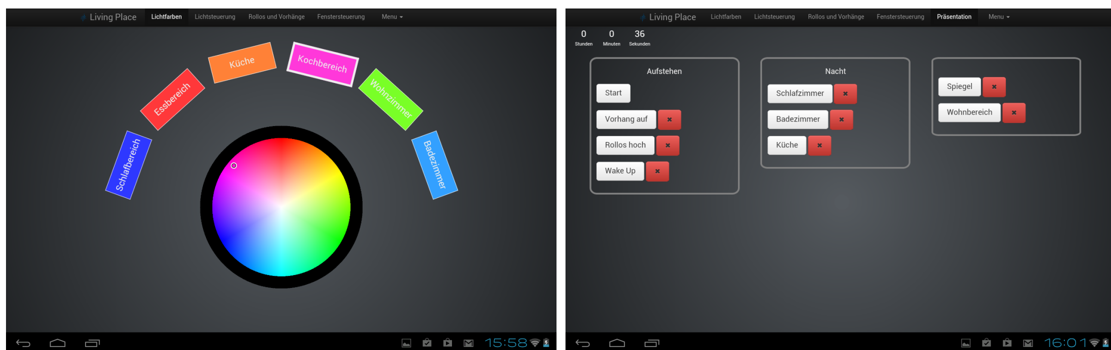
Die Fernbedienung ist aktuell die umfangreichste Implementierung einer Agentengruppe außerhalb der Middleware. Über die Fernbedienung lassen sich sämtliche Aktoren im Living Place steuern und außerdem Sensordaten einsehen.

Der Funktionsumfang der Fernbedienung wird durch eine Gruppe von Agenten realisiert, die sich eine vom Framework kontrollierte Verbindung zur Middleware teilen. Die einzelnen Agenten sind für die Interaktion mit verschiedenen Aktoren zuständig und haben die dafür wichtigen Gruppen abonniert.

Wie auch das Webinterface der Middleware basiert die Implementierung der Fernbedienung auf Webtechnologien. Die Oberfläche der Fernbedienung kann auf beliebig vielen Endgeräten innerhalb der Wohnung aufgerufen werden. Dies können zum Beispiel Smartphones, Tablets und die größeren Multitouchflächen wie der Küchentresen oder der Couchtisch sein. Weiterhin sind auch die Fernseher mögliche Anzeigegeräte, wenn nur der Status der Wohnung gezeigt werden soll. Jede Instanz der Fernbedienung hält einen Websocket zum Server offen, über den alle Statusänderungen kommuniziert werden. Auf diese Weise werden alle Instanzen synchron gehalten.



(a) Farbauswahl (b) Lampen-Einzelsteuerung



(c) Raumfarben (d) Präsentationsansicht

Abbildung 4.1: Benutzeroberfläche der Fernbedienung des Living Place.

In der Laborumgebung wird die Fernbedienung oft zur Fehlersuche verwendet. Zum Beispiel können einzelne Lampen auf Funktionsfähigkeit getestet werden. Sämtliche Statusänderungen an den angebundenen Aktoren im Living Place sind über das Webinterface sichtbar. Damit lässt sich die Interaktion mit Aktoren von neuen Agenten visualisieren. Außerdem kann die Fernbedienung so als Dummy eingesetzt werden, um einen Test außerhalb der Wohnung zu

ermöglichen. Die Visualisierung von Sensordaten hilft Entwicklern, die Kontextinformationen aus der Wohnung beim Testen besser zu verstehen.

Die Fernbedienung enthält eine Präsentationsansicht, über die Vorführungen im Living Place gesteuert werden können. Die einzelnen Felder dieser Ansicht lassen sich leicht über Javascript konfigurieren und mit beliebig komplexen Aktionskombinationen belegen. Die Präsentationsansicht kann für spezielle Bedürfnisse, wie zum Beispiel Usability-Experimente im Labor, angepasst werden.

4.1.2 Aktoren

Lichtsteuerung

Die Lichtsteuerung ist einer der wichtigsten und umfangreichsten Aktoren im Living Place. Ursprünglich wurde die Steuerungssoftware in einer Komponente zusammen mit der Rollo- und Vorhangssteuerung umgesetzt. Im Zuge der Umstellung des Systems auf die neue Middleware war es sinnvoll, die Lichtsteuerung neu zu implementieren und damit weitere Funktionen zu ergänzen. Hierzu zählt vor allem eine Auskunftsfähigkeit über den Status der Beleuchtung der Wohnung und der Komponente.

Die ankommenden Steuerungsbefehle beim Licht-Agenten werden über eine TCP-Verbindung an einen Hardware-Controller im Living Place weitergeleitet. Da Akka Aktor-Implementierungen für die asynchrone Kommunikation über Netzwerkprotokolle enthält, ist die Implementierung eine einfache Weiterleitung an einen beim Start erstellten TCP-Aktor. Da der Hardware-Controller nicht auskunftsfähig ist, muss der Licht-Agent sich den aktuellen Zustand aus der Menge der Änderungen merken. Um dabei die Fehleranfälligkeit zu reduzieren, wird der Status bei speziellen Aktionen zurückgesetzt. Eine dieser Aktionen ist der Befehl für „Alle Lampen aus“.

Alle Statusänderungen werden auf einer dafür festgelegten Gruppe veröffentlicht. Zusätzlich dazu kann jeder Agent im System Anfragen über den aktuellen Status der Beleuchtung der ganzen Wohnung oder den Status bestimmter Lampen stellen.

4.1.3 Sensoren

Temperatursensor

Die Implementierung des Temperatursensor-Agenten dient als Beispiel für einen typischen Sensor. Die Sensordaten werden durch eine Hardware gemessen, die in der Wohnung installiert wird. Der Zugriff erfolgt über eine Webschnittstelle, die in XML formatierte Daten liefert.

In der Konfiguration des Agenten wird festgelegt, in welchen Zeitabständen und in welche Gruppe die Sensorinformationen veröffentlicht werden sollen. Der Agent ruft in dem definierten Intervall die Webschnittstelle auf und versendet das Ergebnis über die Middleware. Dabei wird eine asynchrone HTTP-Bibliothek verwendet, sodass der Agent zu keiner Zeit blockiert. Andere Agenten können durch Abfragen außerhalb des Intervalls zusätzliche Werte anfordern. Sowohl die Nachrichten, die die Messergebnisse transportieren, als auch die Abfragen werden über eine Schnittstellenbibliothek definiert.

Durch eine weitere Einstellung in der Konfiguration meldet das Framework den Sensor-Agenten bei dem CEP-Agenten an. Dies führt dazu, dass alle Sensordaten von der CEP-Engine verarbeitet werden und auch dort abrufbar sind. Der CEP-Agent überwacht alle angemeldeten Sensoren über den Monitoring-Dienst der Middleware, um nicht erreichbare Sensoren von der CEP-Engine abzumelden.

Positionserkennung - Ubisense-Adapter

Ubisense ist ein Positionstracking-Dienst, der im Living Place installiert ist und der Benutzer oder Gegenständen innerhalb der Wohnung mit Hilfe von Tags lokalisieren kann. Die mitgelieferten Bibliotheken zum Zugriff auf die Sensorinformationen basieren auf .Net. Dieser Sensor ist also ein gutes Beispiel für eine externe Technologie, die in das System integriert wird.

Für die Integration von Ubisense kann ein in C# implementierter Dienst erstellt werden, der die Sensordaten ausliest und für einen Adapter-Agenten bereitstellt. Die Kommunikation kann hier wieder über eine Webschnittstelle oder direkt über TCP oder UDP realisiert werden. Im Living Place wurde dieser Dienst bereits implementiert und die Daten werden über einen Message-Broker veröffentlicht. Da der Message-Broker mit der Middleware verbunden ist, muss hier also nur noch der Adapter-Agent implementiert werden. Diese Aufgabe gleicht der Implementierung des Temperatursensors sehr, weil hierfür das Framework verwendet werden kann.

Ist dieser Zwischenschritt nicht gewünscht, könnte von der C#-Implementierung direkt auf die Middleware zugegriffen werden. Dies kann entweder über eine TCP-Verbindung oder direkt über die ActiveMQ-Installation geschehen. Dabei kann die Implementierung allerdings nicht durch das Framework unterstützt werden. Die Interaktion mit der Middleware müsste über den gewählten Kommunikationskanal selbst implementiert werden.

4.2 Latenzzeiten und Skalierbarkeit des Systems

In der Analyse wurden mehrere Anforderungen an die Middleware und das Framework gestellt. Dazu gehören unter anderem die Forderungen nach Skalierbarkeit und Fehlertoleranz, die im Folgenden mit Hilfe eines Testaufbaus anhand der Implementierung untersucht werden sollen.

Zuerst soll getestet werden, ob das System in der Lage ist, eine große Anzahl von Agenten auszuführen, ohne dass die Latenzen der Nachrichten zu groß für bestimmte Anwendungen werden. Im Living Place gibt es mehrere Projekte, die auf Benutzereingaben reagieren müssen. Dazu gehören zum Beispiel die Agenten, die über die Fernbedienung gesteuert werden können. Wird die Zeit zwischen Steuerungsbefehl und Ausführung der Aktion zu groß, gehen viele Benutzer von einem Fehlerverhalten aus. Weiterhin gibt es Projekte, die Steuerungsbefehle in Form von Gesten über Sensoren wie die Kinect[1] oder Leap Motion[2] entgegennehmen. Ein Beispiel hierfür ist der Badezimmerspiegel im Living Place, der mit der alten im Labor vorhandenen Gruppenkommunikation implementiert wurde (vgl. Ghose, 2014). Auf dem Spiegel wird als Interaktionshilfe eine kleine Hand dargestellt, die direkt über Gesten gesteuert werden kann, um Elemente der Benutzeroberfläche zu manipulieren. Dadurch kann der Benutzer Verzögerungen schnell wahrnehmen.

Für die direkte Interaktion mit einer Benutzeroberfläche sollten die Antwortzeiten unter 0,1 Sekunden bleiben, damit der Benutzer das Gefühl bekommt, das System würde unmittelbar reagieren. Längere Antwortzeiten von bis zu einer Sekunde werden vom Benutzer bemerkt, aber sie werden zum Beispiel für die Ausführung eines Befehls oder das Wechseln einer Webseite nicht als störend empfunden. Ohne Fortschrittsanzeige können Wartezeiten von mehr als 10 Sekunden dazu führen, dass die Aufmerksamkeit des Benutzers verloren geht und eine Störung vermutet wird (vgl. Nielsen, 2014).

[1] Microsoft Kinect - http://www.microsoft.com/en-us/kinectforwindows; letzter Zugriff: 26.09.2014
[2] Leap Motion - https://www.leapmotion.com; letzter Zugriff: 26.09.2014

Im Living Place wird ein Großteil der Software auf virtuellen Maschinen ausgeführt, die zusammen auf wenigen Host-Rechnern laufen. Das bedeutet, die virtuellen Maschinen könnten sich bei einem Test gegenseitig beeinflussen und die Ergebnisse verfälschen. Deswegen wurden die Messungen in einem anderen Labor durchgeführt, in dem acht baugleiche Rechner zur Verfügung stehen. Die Rechner sind über ein LAN-Netzwerk miteinander verbunden.

Einer der Rechner wird während des Tests als Seed-Node und Steuerungsknoten verwendet. Außerdem wird der Datenbankserver auf diesem Rechner ausgeführt. Startet der erste Knoten, werden dort die Koordinatoren für das Monitoring und die Laufzeitumgebungen gestartet. Danach können für den Test eine beliebige Anzahl weiterer Knoten gestartet werden. Jeder weitere Knoten wird zusammen mit einer Laufzeitumgebung auf einem separaten Rechner ausgeführt.

Für das Testszenario werden zwei Arten von Agenten verwendet, die über Konfigurationsdaten angepasst werden. Ein Anfrage-Agent schickt alle x Sekunden eine Anfrage an eine vorkonfigurierte Gruppe und erwartet die Antwort auf einer anderen. Die Anfrage enthält einen zufälligen String, um eine eindeutige Zuordnung zu ermöglichen. Ein Antwort-Agent nimmt die Anfrage entgegen und schickt eine Antwort-Nachricht mit dem kopierten Nachrichteninhalt auf der ihm zugewiesenen Gruppe. Für die Tests werden jeweils Paare von Agenten gestartet, die auf zwei exklusiv ihnen zugewiesenen Gruppen miteinander kommunizieren. Die Anzahl der Gruppen ist also doppelt so groß wie die Anzahl der Agenten. Damit muss die Middleware nicht nur mit der Agentenanzahl, sondern auch der Anzahl der Gruppen skalieren. Bei allen Tests wird vom Anfrage-Agent die Zeit zwischen Abschicken einer Nachricht und dem Empfangen der Antwort gemessen. Dabei werden die Serialisierung und Deserialisierung mit betrachtet, weil diese ein wichtiger Teil des Designs sind und auch in den beschriebenen Anwendungsfällen eine Rolle spielen. Die Messwerte werden über eine weitere Gruppe an einen Agenten außerhalb der Testumgebung übertragen und dort mitgeschrieben.

In Abbildung 4.2 werden die Ergebnisse des ersten Tests gezeigt. Während des Tests verschickt jeder Anfrage-Agent alle vier bis fünf Sekunden eine Nachricht. Die durchschnittliche Antwortzeit der Testagenten lag mit 120.000 Agenten und 7 Knoten bei unter 12 ms. Dies zeigt, dass das System mit einer sehr großen Anzahl von Agenten zurechtkommt. Im Vergleich dazu werden im Living Place momentan weit unter 1.000 Softwarekomponenten über die Gruppenkommunikation miteinander verbunden. Agenten und auch Gruppen sind in dem

System somit keine kritische Ressource, was bedeutet, dass während der Entwicklung nicht über die Anzahl der Agenten nachgedacht werden muss.

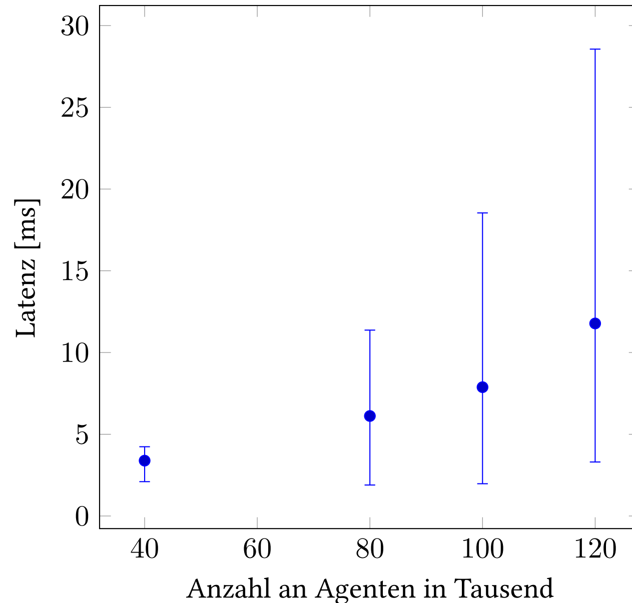


Abbildung 4.2: Test mit acht Middleware-Knoten und sieben Laufzeitumgebungen mit unterschiedlicher Anzahl von Agenten.

Bei der getesteten Nachrichtenrate ergeben sich durchschnittlich etwa 27.000 Nachrichten pro Sekunde zwischen den Testagenten. Dazu kommen für jede Antwort eine Nachricht für die Messdaten und die Systemnachrichten wie zum Beispiel Heartbeats zur Überwachung der Erreichbarkeit der Agenten und Middleware-Knoten. Dies ist ebenfalls eine sehr große Zahl für ein Smart-Environment. Aktoren verarbeiten meist eine Nachricht pro Steuerungsbefehl, die im Vergleich sehr selten ausgeführt werden. Häufige und regelmäßige Nachrichten sind nur von den Sensoren und Kontext-Interpretern im System zu erwarten, wobei die Nachrichtenrate stark abhängig vom Sensor ist. Sensoren, die Werte wie Temperatur oder Luftfeuchtigkeit überwachen, müssen nicht alle paar Sekunden senden, da sich die Werte erwartungsgemäß langsam verändern. Zu den schneller sendenden Sensoren gehören zum Beispiel die Positions- oder Gestenerkennung.

Zu den einzelnen Messungen wurden jeweils die Perzentile P_{10} und P_{90} berechnet und im Graphen eingetragen. Der Wert von P_{90} gibt an, unter welchem Wert 90% der Latenzen liegen. An den Werten in Abbildung 4.2 lässt sich eine mit zunehmender Systembelastung wachsende Schwankung der Latenzen ablesen. Aber selbst bei einer Anzahl von 120.000 Agenten werden

90% der Nachrichten innerhalb von ca. 29 ms zugestellt. Dieser Wert sollte sich immer noch für die meisten Arten von Benutzerinteraktionen eignen. Es ist zu vermuten, dass die Schwankungen durch das Scheduling auf den einzelnen Knoten, das Zustellen von Systemnachrichten und die Garbage-Collection der JVM auf den einzelnen Knoten in Verbindung mit einer hohen Systemauslastung verursacht werden.

Evaluation der Skalierbarkeit des Systems

Ein weiterer Test soll zeigen, wie sich das System verhält, wenn extrem viele Nachrichten verschickt werden. Dies soll ausschließen, dass das System von schnell sendenden Agenten so beeinflusst wird, dass andere Agenten beeinträchtigt werden. Außerdem soll die Skalierbarkeit der Middleware getestet werden. Das Hinzufügen weiterer Middleware-Knoten soll das System entlasten und die Latenzen verringern.

Bei dem Test wurden zuerst 5.000 und dann 10.000 Agenten gestartet, die alle 300 bis 400 Millisekunden eine Testnachricht verschicken. An den Ergebnissen in den Abbildungen 4.3 und 4.4 lässt sich erkennen, dass diese große Nachrichtenanzahl bei einer kleinen Anzahl von Middleware-Knoten hohe Latenzen verursachen kann. Dies ist ein Zeichen dafür, dass das System ausgelastet ist. Bei dem Test mit 10.000 Agenten und zwei Laufzeitumgebungen wird eine Latenz von durchschnittlich ca. 60 ms erreicht. Durch das Hinzufügen weiterer Knoten lässt sich diese Latenz deutlich reduzieren. Bei sechs Laufzeitumgebungen ist sie mit durchschnittlich ca. 5,5 ms schon sehr gering. Außerdem reduzieren sich mit zunehmender Anzahl von Middleware-Knoten und Laufzeitumgebungen die Schwankungen zwischen den Latenzen deutlich.

Das Design des Frameworks macht es mit Hilfe des Akka-Frameworks möglich, die Arbeit auf alle vorhandenen Prozessorkerne zu verteilen. Bei den Tests ist zu erkennen, dass dies auch bei hoher Last geschieht und die Ressourcen der Rechner optimal ausgenutzt werden. Eine weitere Möglichkeit, das System zu skalieren, besteht somit darin, weitere Kerne hinzuzufügen, was zum Beispiel in einer virtuellen Maschine, wie sie im Living Place verwendet wird, einfach möglich ist.

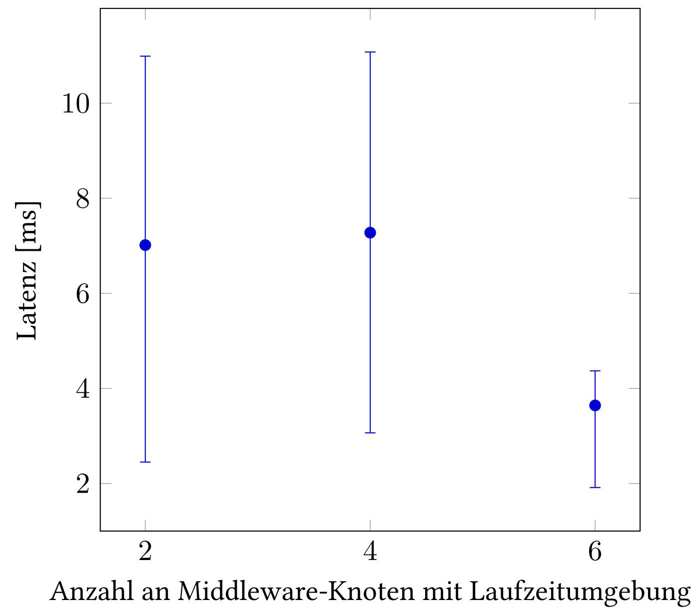


Abbildung 4.3: Test mit 5.000 Agenten und unterschiedlicher Anzahl von Middleware-Knoten mit Laufzeitumgebung.

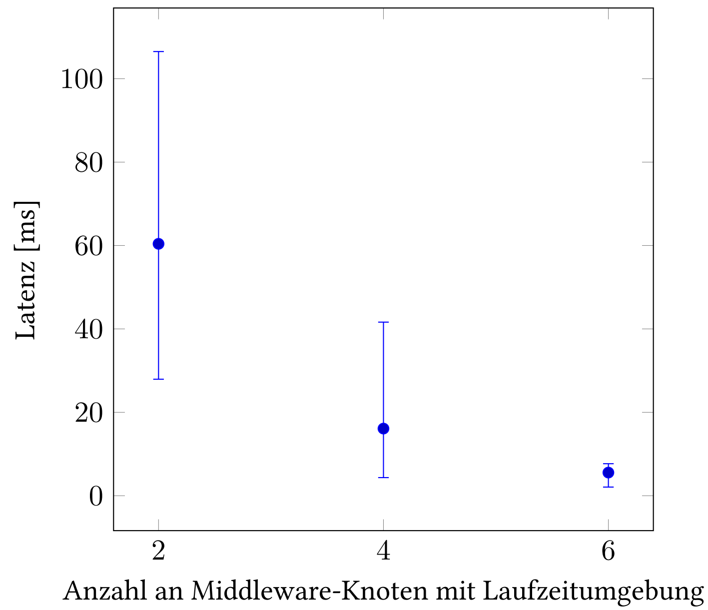


Abbildung 4.4: Test mit 10.000 Agenten und unterschiedlicher Anzahl von Middleware-Knoten mit Laufzeitumgebung.

Fazit

Die Testergebnisse zeigen, dass die Middleware und das Framework auch mit einer für das Anwendungsfeld sehr großen Anzahl von Agenten und Gruppen verwendet werden können. Außerdem wurde gezeigt, dass das System mit dem Hinzufügen weiterer Knoten mit Laufzeitumgebungen skaliert und sich dadurch die Latenzen und Latenzschwankungen reduzieren lassen. Die Messungen zeigen, dass das System selbst mit einer großen Anzahl von Agenten und mit hohem Nachrichtenaufkommen auch für direkte Benutzerinteraktion verwendet werden kann. Insbesondere die Tests mit einer ausreichenden Anzahl an Knoten oder einer realistischen Anzahl von Agenten zeigen dafür sehr gute Latenzen von unter 10 ms.

4.2.1 Mögliche Optimierungen

Das System wurde hauptsächlich für eine gute Entwicklerunterstützung und weniger für Systemperformance optimiert. Diese Optimierung kann bei Bedarf in zukünftigen Arbeiten weiter vorangetrieben werden.

Für besonders Performance-intensive Agenten oder Teilsysteme können eigene dedizierte Middleware-Knoten und Laufzeitumgebungen bereitgestellt werden, um die Beeinflussung des Restsystems zu verringern. Ein weiterer daraus resultierender Vorteil ist, dass diese Systemteile leicht vom Rest des Systems getrennt und in anderen Systemen wiederverwendet werden können. Laufzeitumgebungen und Middleware-Knoten können auf getrennten Maschinen ausgeführt werden, um auszuschließen, dass Agenten die Dienste der Middleware beeinträchtigen.

Bei der Gruppenkommunikation werden alle Nachrichten über mindestens einen der Middleware-Knoten verschickt. Für Nachrichten, die nur zwischen zwei Agenten ausgetauscht werden sollen, bietet es sich deswegen an, eine direkte Kommunikation aufzubauen. Dies ist über das Framework und Akka ohne weitere Anpassungen möglich. Sollte die Adresse eines Agenten nicht bekannt sein, kann sie über die Gruppenkommunikation abgefragt werden. Eine direkte Kommunikation sollte die Latenzzeiten und die Schwankungen deutlich reduzieren. Sie sollte aber nur bei Bedarf eingesetzt werden, weil sie die Überwachungstools der Middleware umgeht und somit die Systemübersicht erschwert. Es ist auch möglich, Nachrichten parallel direkt und über die Gruppenkommunikation zu verschicken, um die Vorteile beider Wege zu verbinden. Dieser Ansatz wäre zum Beispiel für die Implementierung des Badezimmerspiegels im Living Place zu empfehlen, um die Latenz der Events bei der Steuerung durch die Kinect zu minimieren, aber weiterhin die Möglichkeit zu bieten, die Steuerung an anderer Stelle im System abzugreifen.

In der aktuellen Version der Middleware und des Frameworks werden alle Nachrichten im Klartext übertragen. Dies ist eine sehr ineffiziente Methode, die mit wenig Aufwand durch eine Komprimierung der Nachrichten verbessert werden kann. Die Nachrichten werden im Klartext übertragen, damit sie von den Entwicklern überall und jederzeit eingesehen werden können. Durch eine Komprimierung wäre dieser Vorteil nur leicht eingeschränkt, weil Anzeigetools, wie zum Beispiel das Webinterface, die Nachrichten weiter vollständig anzeigen können. Sollte eine weitere Optimierung der Nachrichtengrößen notwendig sein, können hierfür auch Tools wie Protobuf[3] eingesetzt werden. Mit Protobuf lassen sich Nachrichten in einem Performance-optimierten Binärformat verschicken. Hierfür müsste nur die Serialisierungskomponente des Frameworks ausgetauscht werden, eine Anpassung der Agenten ist nicht erforderlich.

Alle Messungen beinhalten die Zeit für die Serialisierung und die Deserialisierung der Nachrichten. Diese Zeit könnte nicht nur durch den Austausch oder die Optimierung des Nachrichtenformates, sondern auch durch Caching reduziert werden. Das Zwischenspeichern von häufig verschickten Nachrichten ist besonders innerhalb der Laufzeitumgebungen sinnvoll, wo eine Framework-Instanz für viele gleiche Agenten zuständig ist. Caching sollte vor allem das Verschicken von Sensorinformationen beschleunigen und die Prozessorlast reduzieren, da hier oft dieselben oder sehr ähnliche Nachrichten verschickt werden, die sich nur in den Messwerten unterscheiden.

4.3 Erfahrungen im Team

Während der Erstellung dieser Arbeit wurde bereits ein erstes studentisches Projekt mit Hilfe der Middleware und des Frameworks durchgeführt. Ziel des Projektes ist es, ein Framework bereitzustellen, welches die Zuweisung von Inhalten auf beliebigen Anzeigegeräten steuert (vgl. Kirchner, 2014). Diese Funktionalität macht das Framework besonders interessant für Second-Screen-Anwendungen.

Die Auswertung des Projektes bezüglich der Vor- und Nachteile der Verwendung der im Rahmen dieser Arbeit entwickelten Software geschah in Form eines Interviews. Das Projekt wurde auf Basis der jeweils aktuellen Version der Middleware und des Frameworks durchgeführt und im Laufe des Projektes aktualisiert. Auf diese Weise konnten Rückmeldungen schon früh in die Entwicklung mit einfließen, um Fehler zu beheben und die Entwicklerunterstützung

[3] Google Protobuf - https://github.com/google/protobuf; letzter Zugriff: 26.09.2014

zu verbessern.

Es wurde positiv angemerkt, dass das Framework auf Akka und Scala und den damit verbundenen Design-Philosophien basiert. Auf diese Weise kann die sehr umfangreiche Dokumentation von Akka vollständig verwendet werden. Diese kann auch zusätzliche Hinweise zum Design eines neuen Projektes geben. Akka und Scala sind zwei Open-Source-Projekte, die große Communities aufweisen und auch in der Wirtschaft häufig verwendet werden. Deswegen kann ein Entwickler eines neuen Projektes bei Problemen auf viele Online-Ressourcen zugreifen, die ihm Hilfestellung geben können.

Zusätzlich dazu könnte der Entwicklungsprozess aber noch weiter verbessert werden, indem weitere Beispielentwurfsmuster für das Framework bereitgestellt werden. Da alle im Living Place erstellten Projekte von den Studenten einsehbar sind, werden mit der Zeit weitere Beispiele hinzukommen. Um für neue Studenten diese Projekte nützlich zu machen, sollte die Dokumentation explizit auf Positivbeispiele in verfügbaren Projekten hinweisen und häufige Entwurfsmuster erklären.

Die Verwendung von Aktor-Programmierung im Framework und das eventbasierte Design wurden positiv hervorgehoben und als sehr passend für den Anwendungsfall beschrieben. Allerdings kann die Art der Programmierung ungewohnt für neue Entwickler sein. Dies hat zur Folge, dass sich die Entwicklung neuer Projekte verzögern kann, wenn neue Konzepte zuerst erlernt werden müssen.

Probleme, die während der Entwicklung des Projektes auftraten, resultierten hauptsächlich aus dem grundlegenden Systemaufbau. Zum Beispiel kann das Finden eines Fehlers, bei dem eine gesendete Nachricht nicht zugestellt wird, sich in einem verteilten nachrichtenbasierten System als schwierig herausstellen. Da jeder Aktor Nachrichten jedes Typs akzeptiert, kann diese Art von Fehler oft nicht bei der Kompilierung des Projektes gefunden werden. Hier ist eine umfangreiche Fehlerbehandlung mit Protokollausgaben und eine gute Testabdeckung sehr hilfreich. Außerdem können die bereitgestellten Tools der Middleware helfen zu analysieren, welche Nachrichten wohin verschickt wurden.

Durch die Verwendung des Frameworks konnte trotz der notwendigen Lernphase deutlich Zeit eingespart werden. Hauptsächlich weil die Grundbausteine für das Design einer neuen Komponenten bereits durch das Framework vorgegeben sind. Außerdem wird die Interaktion

mit anderen Komponenten durch das Framework erleichtert. Dabei bleiben durch die Vorgaben die Schnittstellen einheitlich und können von allen anderen Komponenten verwendet werden.

4.4 Fazit

Während der Evaluation wurde das System in Hinblick auf die in der Analyse ausgearbeiteten Anforderungen untersucht. Dazu wurden mehrere Fallstudien innerhalb des Living Places durchgeführt und ein Testszenario ausgewertet, um das Verhalten des Systems unter Last zu untersuchen. Außerdem wurden Rückmeldungen von Studenten analysiert, die bereits mit der entwickelten Software gearbeitet haben.

Anhand der vorgestellten Beispielprojekte lässt sich die Unterstützung von Heterogenität erkennen. Altsysteme und Komponenten, die mit anderen Technologien entwickelt wurden, können ohne Probleme oder größeren Aufwand an die Middleware angeschlossen werden.

Mit den ausgeführten Tests wurden die Belastbarkeit des Systems, die Skalierbarkeit und Reaktionsfreudigkeit untersucht. Das System blieb auch mit einer sehr großen Anzahl von Agenten und einer hohen Nachrichtenlast stabil und wies selbst für Benutzerinteraktionen noch gute Latenzzeiten auf. Weiterhin zeigte der Test, dass sich das System durch das Hinzufügen weiterer Knoten und Laufzeitumgebungen skalieren lässt. Der Test wurde mit für die Laborumgebung sehr hohen Werten ausgeführt, sodass auch eine stark zunehmende Anzahl von Projekten kein Problem darstellen sollte.

Die Installation der Middleware und der entwickelten Agenten im Living Place dient als weiterer Test. Hier wurden das Deployment und die Fehlertoleranz der Agenten innerhalb der Laborumgebung getestet. Dabei wurden verschiedene Szenarien mit unterschiedlichen Konfigurationen durchgeführt. Die Fehlertoleranz des Systems wurde durch das vorübergehende Ausschalten von Middleware-Knoten geprüft. Danach wurde beobachtet, ob das System in der Lage ist, sich selbst zu reparieren. Dabei muss das System Agenten verschieben und gespeicherte Zustände laden. Dadurch wurde auch die Mobilität der Agenten innerhalb des Systems getestet.

Es wurde bereits ein studentisches Projekt mit Hilfe der aus dieser Arbeit entstandenen Software erfolgreich abgeschlossen. Befragungen während und nach der Durchführung des Projektes haben gezeigt, dass das Framework und die Systemarchitektur die Entwicklung beschleunigen können. Außerdem kann die Software Hilfestellung bei der Suche nach Fehlern und beim

Erstellen eines Systemüberblicks geben. Die installierten Plattformen, wie zum Beispiel die Versionsverwaltung, der Build-Server und der Artefakt-Server, werden bereits aktiv von den Studenten im Living Place benutzt. Damit ist der Zugriff auf die Entwickler-Ressourcen für alle Teilnehmer sichergestellt.

Für zukünftige Projekte stehen die Middleware und das Framework im Living Place zur Verfügung. Mehrere studentische Projekte planen bereits den Einsatz des Frameworks. Außerdem soll die Software für anstehende Workshops und Veranstaltungen im Living Place Hamburg genutzt werden.

5 Schluss

Im Folgenden wird diese Arbeit durch eine kurze Zusammenfassung der vorangegangenen Kapitel und einen Ausblick auf weitere interessante Fragestellungen abgeschlossen.

5.1 Zusammenfassung

In dieser Arbeit wurde eine Middleware zur Entwicklerunterstützung in einem Smart-Home-Labor entworfen. Die Entwicklerunterstützung wird zum einem durch das Systemdesign und zum anderen durch die Bereitstellung eines Frameworks erreicht.

Zuerst wurden in der Analyse (s. Kapitel 2) eine Reihe von Anforderungen an das System identifiziert. Dazu gehören allgemeine Anforderungen wie Fehlertoleranz, Skalierbarkeit und Reaktionsfreudigkeit, aber auch für das Anwendungsfeld Smart-Environments spezifische Anforderungen wie die Unterstützung von Heterogenität und Kontextverarbeitung. Entwickler sollen sich ohne großen Aufwand einen Überblick über das System und die Kommunikation verschaffen und bei Bedarf direkt eingreifen können.

Ausgehend von den Anforderungen wurden eine verteilte Middleware und ein Framework zur Entwicklerunterstützung entworfen (s. Kapitel 3). Der wichtigste Dienst der Middleware ist die Gruppenkommunikation, über die alle Systemkomponenten miteinander kommunizieren. Die Middleware überwacht wichtige Teile des Systems und kann somit auf Fehler von Agenten reagieren. Alle Agenten des Systems können außerhalb der Middleware oder innerhalb einer der von der Middleware kontrollierten Laufzeitumgebungen ausgeführt werden. Weitere Dienste, wie zum Beispiel die Integration von Complex-Event-Processing zur Kontextverarbeitung, werden durch die Agenten bereitgestellt. Das Framework zur Entwicklung neuer Agenten basiert auf einer Bibliothek für Actor-Programmierung auf der JVM und erlaubt die einfache Implementierung von Verhalten, abhängig von eingehenden Nachrichten. Außerdem werden durch das Framework häufig benötigte Funktionen, wie zum Beispiel die Anbindung an die Middleware oder die Serialisierung von Nachrichten, übernommen.

Die Evaluation (s. Kapitel 4) der Middleware erfolgte durch Tests im Living Place Hamburg und durch Messungen von Latenzzeiten in einem Testszenario. Dabei wurde gezeigt, dass die Middleware in der Lage ist, mit einer sehr großen Anzahl von Agenten, Nachrichten und Gruppen zurechtzukommen. Die gemessenen Werte zeigen, dass sich das System aufgrund geringer Latenzzeiten auch für Anwendungen mit Benutzerinteraktion eignet. Durch das Hinzufügen weiterer Middleware-Knoten und Laufzeitumgebungen lässt sich das System weiter skalieren. Das Framework wurde durch eine Reihe von Fallstudien und der Auswertung eines studentischen Projektes, welches bereits während der Erstellung dieser Arbeit abgeschlossen wurde, evaluiert. Dabei wurde gezeigt, dass das Framework die Entwicklung neuer Agenten vereinfachen und beschleunigen kann.

5.2 Ausblick

Die Ergebnisse dieser Arbeit stehen für zukünftige Projekte im Living Place Hamburg zur Verfügung. In naher Zukunft planen weitere studentische Projekte die Verwendung des Frameworks und es wird ein Workshop stattfinden, in dem die Software verwendet werden soll. Außerdem wird eine Veranstaltung an der Hochschule die Entwicklung neuer Komponenten im Living Place in Form eines Projektes thematisieren und ebenfalls auf die Ergebnisse dieser Arbeit zurückgreifen. Die Konzepte und Implementierungen des Frameworks und der Middleware sollen in Zukunft aktiv weiterentwickelt und im Rahmen weiterer Arbeiten fortgeführt werden.

Die Verwendung des Frameworks fördert ein einheitliches Design der einzelnen Projekte und ermöglicht es Entwicklern, sich auf die Kernfragestellungen ihrer Arbeiten zu konzentrieren. Da im Verlauf dieser Arbeit noch keine langfristige Evaluation vorgenommen werden konnte, muss die Zukunft zeigen, ob die Verwendung des System und die Entwicklerunterstützung im Living Place Hamburg tatsächlich zu einer langfristigen Verbesserung der Wartbarkeit und Testbarkeit des Gesamtsystems führen. Weiterhin ist zu beobachten, ob die Middleware durch die Förderung der Zusammenarbeit zwischen Agenten langfristig das Verhalten des Gesamtsystems verbessert und ein intelligenteres Verhalten erreicht werden kann.

Es ist zu erwarten, dass sich durch neue Erkenntnisse im Bereich von Smart-Environments und die technologische Entwicklung weitere Anforderungen ergeben, die eine Erweiterung oder Anpassung des Systems erfordern. Nachfolgende Arbeiten könnten versuchen, die Entwicklerunterstützung in Smart-Environments weiter zu verbessern oder noch fehlende Systemkomponenten zu ergänzen. Eine mögliche Ergänzung wäre eine Datenbank für Kontextinformationen

zur Untersuchung und Abfrage von Kontextänderungen über längere Zeiträume.

Das System wurde für den Einsatz in unterschiedlichen Smart-Environments entworfen und sollte sich durch den flexiblen Aufbau und die lose Kopplung des Systems ohne großen Aufwand in andere Umgebungen integrieren lassen. Es ist zu untersuchen, wie sich die Middleware in anderen Laborumgebungen verhält und ob auch dort der Entwicklungsprozess neuer Projekte durch das Framework erfolgreich unterstützt werden kann. Das hier entworfene System wurde für Laborumgebungen optimiert. Der Einsatz in Produktivumgebungen und die Bedienung durch Menschen ohne technischen Hintergrund wurde nicht betrachtet. Es ist noch offen, welche Anpassungen für einen solchen Einsatz notwendig sein werden.

Teile des Systemdesigns könnten auch für andere verwandte Problemstellungen interessant sein. Ein Beispiel hierfür wäre die Vernetzung von Komponenten aus mehreren Smart-Environments über das Internet, wie es für die Umsetzung eines „Internets der Dinge“ notwendig wäre. Die Middleware scheint aufgrund des verteilten Entwurfs gut für solche und ähnliche Ansätze geeignet.

Literaturverzeichnis

[Abowd u. a. 1999] Abowd, Gregory D. ; Dey, Anind K. ; Brown, Peter J. ; Davies, Nigel ; Smith, Mark ; Steggles, Pete: Towards a Better Understanding of Context and Context-Awareness. In: *Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing*. London, UK, UK : Springer-Verlag, 1999 (HUC '99), S. 304–307. – URL http://dl.acm.org/citation.cfm?id=647985.743843. – ISBN 3-540-66550-1

[Augusto u. a. 2010] Augusto, Juan C. ; Nakashima, Hideyuki ; Aghajan, Hamid: Ambient Intelligence and Smart Environments: A State of the Art. In: *Handbook of Ambient Intelligence and Smart Environments*. 2010, S. 3–31

[Bellifemine u. a. 1999] Bellifemine, Fabio ; Poggi, Agostino ; Rimassa, Giovanni: JADE - A FIPA-compliant agent framework. In: *Proceedings of the Practical Applications of Intelligent Agents*, 1999

[Bellifemine u. a. 2001] Bellifemine, Fabio ; Poggi, Agostino ; Rimassa, Giovanni: JADE: a FIPA2000 compliant agent development environment. In: *Proceedings of the fifth international conference on Autonomous agents*. New York, NY, USA : ACM, 2001 (AGENTS '01), S. 216–217. – URL http://doi.acm.org/10.1145/375735.376120. – ISBN 1-58113-326-X

[Bornemann 2013] Bornemann, Sven B.: *Entwicklung eines kontextsensitiven Berechtigungssystems für Smart Homes*, HAW Hamburg, Masterarbeit, 2013. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/bornemann.pdf

[Charousset 2012] Charousset, Dominik: *libcppa – An actor library for C++ with transparent and extensible group semantic*, HAW Hamburg, Masterarbeit, 2012. – URL http://inet.cpt.haw-hamburg.de/thesis/completed/dominik_charousset.pdf

[Dashofy u. a. 2002] Dashofy, Eric M. ; Hoek, André van der ; Taylor, Richard N.: Towards Architecture-based Self-healing Systems. In: *Proceedings of the First Workshop on Self-*

healing Systems. New York, NY, USA : ACM, 2002 (WOSS '02), S. 21–26. – URL http://doi.acm.org/10.1145/582128.582133. – ISBN 1-58113-609-9

[Ecma International 2013] ECMA INTERNATIONAL: *The JSON Data Interchange Format*. Oktober 2013. – URL http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf. – 1st Edition

[Ellenberg u. a. 2011] ELLENBERG, Jens ; KARSTAEDT, Bastian ; VOSKUHL, Sören ; LUCK, Kai von ; WENDHOLT, Birgit: An Environment for Context-Aware Applications in Smart Homes. In: *International Conference on Indoor Positioning and Indoor Navigation (IPIN)*, 2011

[Fortino u. a. 2013] FORTINO, Giancarlo ; GUERRIERI, Antonio ; LACOPO, Michelangelo ; LUCIA, Matteo ; RUSSO, Wilma: An agent-based middleware for cooperating smart objects. In: *Highlights on Practical Applications of Agents and Multi-Agent Systems*. Springer, 2013, S. 387–398

[Foundation for Intelligent Physical Agents 2002a] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS: *FIPA Abstract Architecture Specification*. 2002. – URL http://www.fipa.org/specs/fipa00001

[Foundation for Intelligent Physical Agents 2002b] FOUNDATION FOR INTELLIGENT PHYSICAL AGENTS: *FIPA ACL Message Structure Specification*. 2002. – URL http://www.fipa.org/specs/fipa00061/

[Fowler 2005] FOWLER, Martin: *Event Sourcing - Capture all changes to an application state as a sequence of events*. Online. Dezember 2005. – URL http://martinfowler.com/eaaDev/EventSourcing.html. – Letzter Zugriff: 22.07.14

[Fowler 2006] FOWLER, Martin: *Continuous Integration*. Online. Mai 2006. – URL http://www.martinfowler.com/articles/continuousIntegration.html. – Letzter Zugriff: 24.07.2014

[Ghose 2014] GHOSE, Sobin: *Konzeption und Evaluation eines interaktiven Badezimmerspiegels*, HAW Hamburg, Bachelorarbeit, 2014. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/bachelor/ghose.pdf

[Hall u. a. 2011] HALL, Richard S. ; PAULS, Karl ; MCCULLOCH, Stuart ; SAVAGE, David: *OSGi in Action*. Manning, 2011

[Haller und Odersky 2009] Haller, Philipp ; Odersky, Martin: Scala Actors: Unifying Thread-based and Event-based Programming. In: *Theor. Comput. Sci.* 410 (2009), Februar, Nr. 2-3, S. 202–220. – URL http://dx.doi.org/10.1016/j.tcs.2008.09.019. – ISSN 0304-3975

[Harroud und Karmouch 2005] Harroud, H. ; Karmouch, A.: A policy based context-aware agent framework to support users mobility. In: *Telecommunications, 2005. advanced industrial conference on telecommunications/service assurance with partial and intermittent resources conference/e-learning on telecommunications workshop. aict/sapir/elete 2005. proceedings*, 2005, S. 177–182

[Henricksen u. a. 2005] Henricksen, Karen ; Indulska, Jadwiga ; McFadden, Ted: Middleware for Distributed Context-Aware Systems. In: *Proceedings of the 2005 Confederated international conference on On the Move to Meaningful Internet Systems - Volume \>Part I*, URL http://dx.doi.org/10.1007/11575771_53, 2005, S. 846–863

[Hewitt 2014] Hewitt, Carl: Actor Model of Computation: Scalable Robust Information Systems. URL http://arxiv.org/abs/1008.1459v32, 2014 (v32). – Forschungsbericht

[Hewitt u. a. 1973] Hewitt, Carl ; Bishop, Peter ; Steiger, Richard: A Universal Modular ACTOR Formalism for Artificial Intelligence. In: *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1973 (IJCAI'73), S. 235–245. – URL http://dl.acm.org/citation.cfm?id=1624775.1624804

[Jennings 2000] Jennings, Nicholas R.: On Agent-based Software Engineering. In: *Artif. Intell.* 117 (2000), März, Nr. 2, S. 277–296. – URL http://dx.doi.org/10.1016/S0004-3702(99)00107-1. – ISSN 0004-3702

[Karstaedt 2012] Karstaedt, Bastian: *Kontextinterpretation in Smart Homes auf Basis semantischer 3D Gebäudemodelle*, HAW Hamburg, Masterarbeit, 2012. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/karstaedt.pdf

[Khambatti u. a. 2003] Khambatti, Mujtaba ; Ryu, Kyung ; Dasgupta, Partha: Push-Pull Gossiping for Information Sharing in Peer-to-Peer Communities. In: *In Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA), Las Vegas*, CSREA Press, 2003, S. 1393–1399

[Kirchner 2014] KIRCHNER, Christian: Entwicklung eines kontextsensitiven Agenten zur Visualisierung von Inhalten im Living Place / HAW Hamburg. URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master2014-proj/kirchner.pdf, 2014. – Projektbericht 1

[López u. a. 2011] LÓPEZ, Tomás S. ; RANASINGHE, Damith C. ; PATKAI, Bela ; MCFARLANE, Duncan: Taxonomy, Technology and Applications of Smart Objects. In: *Information Systems Frontiers* 13 (2011), April, Nr. 2, S. 281–300. – URL http://dx.doi.org/10.1007/s10796-009-9218-4. – ISSN 1387-3326

[von Luck u. a. 2010] LUCK, Prof. Dr. K. von ; KLEMKE, Prof. Dr. G. ; GREGOR, Sebastian ; RAHIMI, Mohammad A. ; VOGT, Matthias: Living Place Hamburg – A place for concepts of IT based modern living / Hamburg University of Applied Sciences. URL http://livingplace.informatik.haw-hamburg.de/content/LivingPlaceHamburg_en.pdf, Mai 2010. – Forschungsbericht

[Luckham 2002] LUCKHAM, David C.: *The Power of Events: An Introduction Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley, 2002

[Mackinnon u. a. 2001] MACKINNON, Tim ; FREEMAN, Steve ; CRAIG, Philip: Extreme Programming Examined. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001, Kap. Endo-testing: Unit Testing with Mock Objects, S. 287–301. – URL http://dl.acm.org/citation.cfm?id=377517.377534. – ISBN 0-201-71040-4

[Meyer und Rakotonirainy 2003] MEYER, Sven ; RAKOTONIRAINY, Andry: A Survey of Research on Context-aware Homes. In: *Proceedings of the Australasian Information Security Workshop Conference on ACSW Frontiers 2003 - Volume 21*. Darlinghurst, Australia, Australia : Australian Computer Society, Inc., 2003 (ACSW Frontiers '03), S. 159–168. – URL http://dl.acm.org/citation.cfm?id=827987.828005. – ISBN 1-920682-00-7

[Nielsen 2014] NIELSEN, Jakob: *Response Times: The 3 Important Limits - Web-Based Application Response Time*. Online. 2014. – URL http://www.nngroup.com/articles/response-times-3-important-limits/. – Letzer Zugriff: 24.08.14

[Odersky u. a. 2011] ODERSKY, Martin ; SPOON, Lex ; VENNERS, Bill: *Programming in Scala: A Comprehensive Step-by-Step Guide, 2Nd Edition*. 2nd. USA : Artima Incorporation, 2011. – ISBN 0981531644, 9780981531649

[Oliveira u. a. 2010] OLIVEIRA, Bruno C. ; MOORS, Adriaan ; ODERSKY, Martin: Type Classes As Objects and Implicits. In: *Proceedings of the ACM International Conference on Object*

Oriented Programming Systems Languages and Applications. New York, NY, USA : ACM, 2010 (OOPSLA '10), S. 341–360. – URL http://doi.acm.org/10.1145/1869459.1869489. – ISBN 978-1-4503-0203-6

[Otto 2013] Otto, Kjell: *Aktuelle Entwicklungskonzepte zur Projektintegration in einem Smart Home anhand von Maven, OSGi und Drools Fusion*, HAW Hamburg, Masterarbeit, 2013. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/otto.pdf

[Poslad 2007] Poslad, Stefan: Specifying Protocols for Multi-agent Systems Interaction. In: *ACM Trans. Auton. Adapt. Syst.* 2 (2007), November, Nr. 4. – URL http://doi.acm.org/10.1145/1293731.1293735. – ISSN 1556-4665

[Robles und Kim 2010] Robles, Rosslin J. ; Kim, Tai-Hoon: Review: Context Aware Tools for Smart Home Development. In: *International Journal of Smart Home* Bd. 4, Januar 2010

[Sahli 2008] Sahli, Nabil: Survey: Agent-based Middlewares for Context Awareness. In: *Electronic Communications of the EASST- ECEASST / Volume 11 (2008): Context-aware Adaption Mechanisms for Pervasive and Ubiquitous Services* 11/03 (2008). – URL http://opus.kobv.de/tuberlin/volltexte/2008/2007/

[Schilit u. a. 1994] Schilit, B. ; Adams, N. ; Want, R.: Context-Aware Computing Applications. In: *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*. Washington, DC, USA : IEEE Computer Society, 1994 (WMCSA '94), S. 85–90. – URL http://dx.doi.org/10.1109/WMCSA.1994.16. – ISBN 978-0-7695-3451-0

[SFB 62] SFB 62: *Sonderforschungsbereich Transregio 62*. – URL http://www.sfb-trr-62.de/. – Letzter Zugriff: 26.09.14

[Soldatos u. a. 2007] Soldatos, John ; Pandis, Ippokratis ; Stamatis, Kostas ; Polymenakos, Lazaros ; Crowley, James L.: Agent based middleware infrastructure for autonomous context-aware ubiquitous computing services. In: *Comput. Commun.* 30 (2007), Februar, Nr. 3, S. 577–591. – URL http://dx.doi.org/10.1016/j.comcom.2005.11.018. – ISSN 0140-3664

[Strese u. a. 2010] Strese, Hartmut ; Seidel, Uwe ; Knape, Thorsten ; Botthof, Alfons: Smart Home in Deutschland / Institut für Innovation und Technik (iit) in der VDI/VDE-IT. URL http://www.vdivde-it.de/publikationen/studien/smart-home-in-deutschland-untersuchung-im-

rahmen-der-wissenschaftlichen-begleitung-zum-programm-next-generation-media-ngm-des-bundesministeriums-fuer-wirtschaft-und-technologie/at_download/pdf, 2010. – Forschungsbericht

[Takahashi u. a. 2005] Takahashi, H. ; Suganuma, T. ; Shiratori, N.: AMUSE: an agent-based middleware for context-aware ubiquitous services. In: *Parallel and Distributed Systems, 2005. Proceedings. 11th International Conference on* Bd. 1, 2005, S. 743–749 Vol. 1. – ISSN 1521-9097

[Tanenbaum und van Steen 2008] Tanenbaum, Andrew S. ; Steen, Maarten van: *Verteilte Systeme - Prinzipien und Paradigmen*. Bd. 2., aktualisierte Auflage. Pearson Studium, 2008

[The Central Maven Repository 2014] The Central Maven Repository: *Quick Statistics for The Central Repository*. Online. Juli 2014. – URL http://search.maven.org/#stats. – Letzer Zugriff: 24.07.14

[Uckelmann u. a. 2011] Uckelmann, Dieter ; Harrison, Mark ; Michahelles, Florian: An Architectural Approach Towards the Future Internet of Things. In: *Architecting the Internet of Things*. Springer Berlin Heidelberg, 2011, S. 1–24

[Voskuhl 2011] Voskuhl, Sören: *Modellunabhängige Kontextinterpretation in einer Smart Home Umgebung*, HAW Hamburg, Masterarbeit, 2011. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/voskuhl.pdf

[Wadler und Blott 1989] Wadler, P. ; Blott, S.: How to Make Ad-hoc Polymorphism Less Ad Hoc. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : ACM, 1989 (POPL '89), S. 60–76. – URL http://doi.acm.org/10.1145/75277.75283. – ISBN 0-89791-294-2

[Walzer u. a. 2007] Walzer, Karen ; Schill, Alexander ; Löser, Alexander: Temporal Constraints for Rule-based Event Processing. In: *Proceedings of the ACM First Ph.D. Workshop in CIKM*. New York, NY, USA : ACM, 2007 (PIKM '07), S. 93–100. – URL http://doi.acm.org/10.1145/1316874.1316890. – ISBN 978-1-59593-832-9

[Weiser 1999] Weiser, Mark: The Computer for the 21st Century. In: *SIGMOBILE Mob. Comput. Commun. Rev.* 3 (1999), Juli, Nr. 3, S. 3–11. – URL http://doi.acm.org/10.1145/329124.329126. – ISSN 1559-1662

[Wooldridge 2002] WOOLDRIDGE, Michael: Intelligent Agents: The Key Concepts. In: *Proceedings of the 9th ECCAI-ACAI/EASSS 2001, AEMAS 2001, HoloMAS 2001 on Multi-Agent-Systems and Applications II-Selected Revised Papers.* London, UK, UK : Springer-Verlag, 2002, S. 3–43. – URL http://dl.acm.org/citation.cfm?id=645699.665762. – ISBN 3-540-43377-5

[Wu u. a. 2007] WU, Chao-Lin ; LIAO, Chun-Feng ; FU, Li-Chen: Service-Oriented Smart-Home Architecture Based on OSGi and Mobile-Agent Technology. In: *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on* 37 (2007), Nr. 2, S. 193–205. – ISSN 1094-6977

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 2. Oktober 2014 Tobias Eichler