Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projekt 2

Tobias Eichler

Entwicklung einer Middleware für das Living Place

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Tobias Eichler

Entwicklung einer Middleware für das Living Place

Eingereicht am: 25. März 2014

Inhaltsverzeichnis

1 Einführung **1**

1.1 Living Place 1

1.2 Probleme der alten Architektur 2

2 Middleware **3**

2.1 Architektur der Middleware 3

2.2 Integration von Complex Event Processing 5

2.3 Framework zur Entwicklerunterstützung 7

2.4 Zugriff von alten Komponenten 9

2.5 Implementation 9

3 Schlussbetrachtung **10**

3.1 Zusammenfassung 10

3.2 Ausblick 10

1 Einführung

In dieser Projektarbeit wird der erste Schritt zur Einführung der neuen Middleware im Living Place Hamburg beschrieben. Es wird erläutert, welche Komponenten und Dienste Teile der Middleware sind und wie diese miteinander interagieren. Der Fokus liegt hierbei zunächst auf der Integration von Complex Event Processing zur Verarbeitung von Kontextinformationen.

Im Folgenden wird zuerst das Living Place zusammen mit der bisherigen Softwarearchitektur beschrieben und auf die Probleme eingegangen, die im Zusammenhang damit stehen. In Kapitel 2 wird anhand der Middleware die neue Architektur vorgestellt. Dabei wird kurz auf die einzelnen Komponenten und Dienste eingegangen. Es wird erläutert, wie die Integration von Complex Event Processing verwirklicht wurde und wie die neue Middleware durch ein Framework von den Entwicklern benutzt werden kann. Außerdem wird auf die Interaktionsmöglichkeiten mit Altsystemen eingegangen. Den Schluss bildet eine Zusammenfassung mit einem kurzen Ausblick.

1.1 Living Place

Das Living Place ist eine Laborumgebung in Form einer 140 m^2 großen Loft-Wohnung, in der Experimente unter Realbedingungen durchgeführt werden können. Dabei geht es unter anderem um Fragestellungen zum Thema Smart-Homes und Urban-Living. Das Labor enthält einen Kontrollraum, der für Usability-Untersuchungen eingesetzt werden kann.

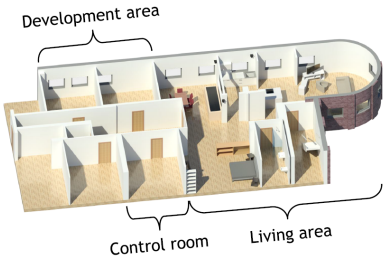

Abbildung 1.1: Das Living Place Hamburg Karstaedt (2012)

In der bisherigen Software-Architektur des Living Place wird ein Blackboard zur Kommunikation verwendet (vgl. Ellenberg u. a., 2011). Alle Komponenten können Nachrichten auf dem Blackboard über Gruppen veröffentlichen und diese abonnieren. Dies wird technisch mit einer zentralen ActiveMQ-Installation umgesetzt. Als Nachrichtenformat wird die JavaScript

Object Notation (JSON) verwendet. Im Laufe der Zeit sind im Living Place eine Vielzahl von Komponenten entstanden, die mit unterschiedlichen Technologien entwickelt wurden, und zusammen ein komplexes System ergeben.

1.2 Probleme der alten Architektur

Alle Komponenten im Living Place sind von ActiveMQ abhängig. Das bedeutet, jeder neue Entwickler muss sich zunächst mit den Details dieser Technologie auseinandersetzen. Die Kommunikationsschnittstelle wurde in fast jeder Komponente unabhängig voneinander implementiert. Dies erhöht die Entwicklungszeit und den Wartungsaufwand der Komponenten. Ein ähnliches Problem gibt es bei der Serialisierung der JSON-Nachrichten.

Die Komponenten sind meistens nicht auskunftsfähig. Dies macht es unmöglich, den Zustand des Systems zu erfassen oder festzustellen, welche Software gerade ausgeführt wird. Es gibt in der Architektur keine vorgesehene Fehlerbehandlung oder Logging-Funktionalität. Fehler bei Systemen führen oft dazu, dass sie und alle davon abhängigen Komponenten ihre Arbeit einstellen, ohne andere darüber zu informieren. Aktoren, die keinen Antwortkanal haben, können eine Aktion nicht bestätigen. Dies ließe sich zwar technisch abbilden, wurde aber in den zur Verfügung stehenden Schnittstellen nicht berücksichtigt. Die einzige Möglichkeit, auf Veränderungen in der Wohnung zu reagieren, ist es die Steuerungsbefehle mitzulesen, was aber keinen Rückschluss über den eigentlichen Zustand der Wohnung zulässt.

In einem Smart-Environment ist die Verarbeitung von Kontext eine Aufgabe, die von vielen Komponenten durchgeführt werden muss. Bisher gibt es keinen Dienst, der dies unterstützt. Jede Komponente muss die Eventströme der Sensoren abonnieren und verarbeiten, um Kontextänderungen erkennen zu können. Damit gibt es keine einheitliche Fehlerbehandlung in den Systemen. Zum Beispiel reagiert jede Komponente potentiell anders auf den Ausfall oder Messfehler eines Sensors.

Im ursprünglichen Systementwurf waren keine Tools zur Unterstützung der Entwickler vorgesehen. Im Rahmen der Masterarbeit von Otto (2013) wurde erstmalig versucht, ein automatisches Build-Tool zusammen mit einer Laufzeitumgebung zu integrieren. Außerdem wurde das Living Place um eine Komponente erweitert, die durch eine regelbasierte Maschine Kontextinterpretationen vornehmen kann. Diese Ergänzungen wurden aber aufgrund mangelnder Interaktionsmöglichkeiten mit alten Systemkomponenten und der erhöhten Komplexität durch die Ergänzung weiterer Technologien nicht adaptiert.

Die Probleme der bisherigen Architektur wurden ausführlich in meiner Projekt 1-Ausarbeitung (Eichler, 2013) untersucht.

2 Middleware

Im Folgenden werden der Aufbau der Middleware und die einzelnen Dienste, mit dem Fokus auf Kontextverarbeitung, beschrieben und es wird darauf eingegangen, wie das Entwickeln neuer Komponenten durch ein Framework vereinfacht werden soll.

2.1 Architektur der Middleware

Die Middleware besteht wie alle anderen Systemteile im Living Place aus Agenten, die in Gruppen zusammenarbeiten, um Dienste bereitzustellen. Der wichtigste Dienst ist hierbei die Gruppenkommunikation, da dieser von allen anderen verwendet wird. Alle veröffentlichten Informationen von Diensten werden über Gruppen zugänglich gemacht. Der Kommunikationsdienst übernimmt außerdem die Verbindung zu anderen Kommunikationsdiensten. Zum Beispiel kann hierüber die Anbindung an ein Altsystem realisiert werden. Im Living Place können dann alle alten Komponenten weiterhin ohne Anpassung über die ActiveMQ miteinander interagieren und zusätzlich mit neuen Komponenten kommunizieren.

Teil der Middleware ist eine Laufzeitumgebung, in der Agenten direkt ausgeführt werden können. Häufige Arten von Agenten sind Sensoren, die Informationen über die Umgebung sammeln, und Aktoren, die die Steuerung von Geräten erlauben. Grundsätzlich können alte und neue Komponenten weiter außerhalb der Middleware auf beliebigen Maschinen ausgeführt werden. Für kleine Agenten ohne weitere Abhängigkeiten kann es allerdings von Vorteil sein, diese in einer von der Middleware kontrollierten Laufzeitumgebung auszuführen.

Dienste werden meistens von Agentengruppen realisiert, die von Nebenläufigkeit profitieren. Um dies zu implementieren, wird auf der Java Virtual Machine (JVM) ein Threadpool verwendet, der die Threads verwaltet, mit denen die Nachrichten nach und nach verarbeitet werden. Im Gegensatz zu einem Agenten sind Threads auf der JVM sehr speicherintensiv und deshalb begrenzt. Alle Agenten in der Laufzeitumgebung teilen sich einen Threadpool und benötigen damit weitaus weniger Arbeitsspeicher. Auf diese Weise ist es möglich, eine extrem große Anzahl von Agenten auf einer Maschine laufen zu lassen und die Prozessorleistung optimal zu nutzen. Sollte mehr Prozessorleistung erforderlich sein, kann eine weitere Laufzeitumgebung auf einer anderen Maschine geschaffen werden. Dies kann die Middleware übernehmen und die Agenten anschließend automatisch verteilen. Ein weiterer Vorteil der Ausführung von Agenten in der Laufzeitumgebung ist, dass keine Serialisierung von Nachrichten notwendig ist, solange die zwei kommunizierenden Agenten in derselben JVM ausgeführt werden.

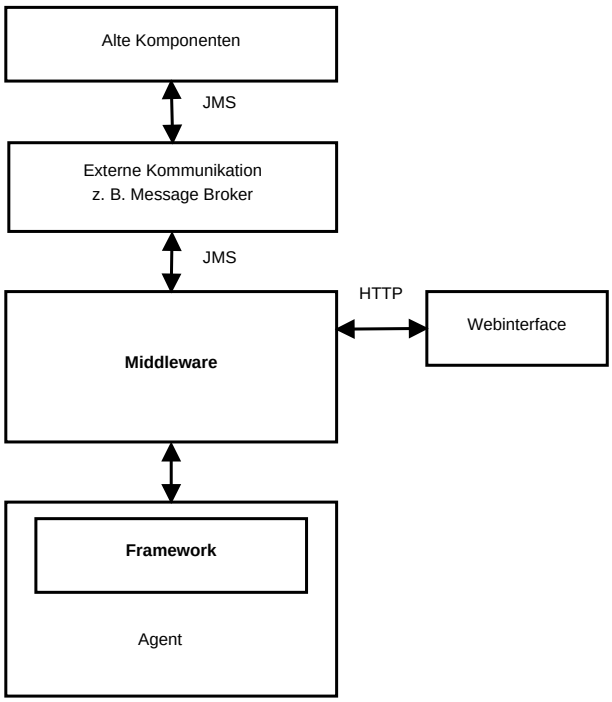

Abbildung 2.1: Interaktion mit der Middleware

Das System hat eine Konfigurationsschnittstelle, über die die Agenten den Systemaufbau verändern können. Damit lassen sich unter anderem kontextabhängige Systemzustände für unterschiedliche Situationen, wie Einzelbenutzer- oder Mehrbenutzerbetrieb, konfigurieren. In einem Labor, wie dem Living Place, gibt es viele Stakeholder mit unterschiedlichen Bedürfnissen. Dazu zählen unter anderem die Informatikstudenten, die die Laborumgebung für Softwaretests von Prototypen einsetzen wollen. Andererseits gibt es aus anderen Fachgebieten, wie z. B. der Medientechnik, Interesse daran, Experimente zur Auswirkung von Licht und Ton auf den Bewohner in einer stabilen Umgebung durchzuführen. Der Konflikt zwischen diesen beiden Bedürfnissen kann durch die dynamischen Konfigurationsmöglichkeiten der Middleware gelöst werden.

Ein verteiltes System kann sehr komplex werden. Deswegen sollten Dienste entwickelt werden, mit deren Hilfe ein Überblick über den Systemzustand geschaffen werden kann. Hierfür bietet die Middleware einen Dienst an. Dieser kann z. B. Auskunft über die aktuell registrierten Agenten und Dienste geben, über gemeldete Fehler im System berichten oder den Zustand von Aktoren aufzeigen. Über eine Schnittstelle kann jede Komponente auf diese Daten zugreifen. Für Entwickler stellt die Middleware zusätzlich einen Webserver bereit, der diese

Informationen geräteunabhängig in einem Browser abrufbar macht. Auf diese Weise sollen Fehler schnell lokalisiert und auch behoben werden können. Jede Komponente kann hier neu gestartet oder, falls diese die Funktion unterstützt, getestet werden. Außerdem können über die Weboberfläche zu Testzwecken beliebige Nachrichten verschickt und empfangen werden.

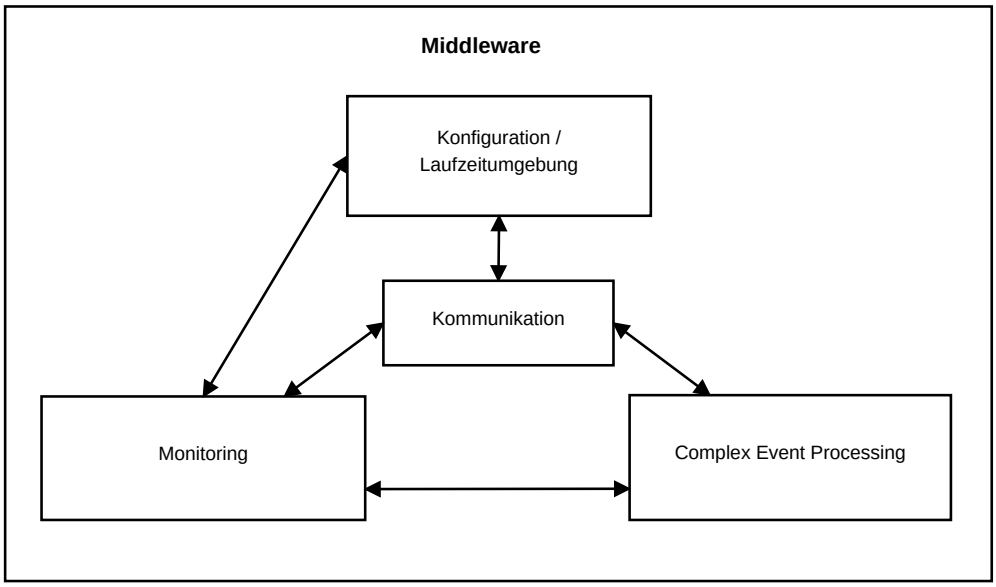

Abbildung 2.2: Interner Aufbau der Middleware

In einem Smart-Environment spielen Kontextinformationen eine wichtige Rolle. Mit ihrer Hilfe kann das System abhängig vom Kontext Entscheidungen treffen und sein Verhalten verändern. Eine Möglichkeit, den Zugriff auf aktuelle Kontextinformationen zu vereinfachen, ist es, einen Dienst anzubieten, der Abfragen auf Kontextveränderungen verarbeitet und Agenten beim Eintritt bestimmter Ereigniskonstellationen informiert. Dieser Dienst ist ebenfalls Teil der Middleware.

2.2 Integration von Complex Event Processing

Mit Hilfe von Complex Event Processing (CEP) lassen sich Abfragen auf komplexe Eventströme realisieren (vgl. Luckham, 2002). Dieses Verfahren kann in Smart-Environments zur Verarbeitung von Kontextinformationen eingesetzt werden. Dabei werden die Eventströme von Sensoren produziert.

Um einen Eventstrom für die CEP-Engine zugänglich zu machen, wird ein Agent erstellt. Dies kann manuell geschehen, wenn z. B. die Events noch weiter verarbeitet werden sollen, oder von der Middleware erledigt werden. Für eine automatische Erstellung des Agenten reicht die

Übergabe der Kommunikationsgruppen und des Nachrichtenformats anhand einer Beispielnachricht.

Im Living Place ist ein Sensorsystem[1] installiert, mit dem Location-Tracking mit RFID-Tags durchgeführt werden kann. Dieser Dienst soll als ein Beispiel für die Integration von CEP im Living Place dienen. Die JSON-Nachrichten des Dienstes können direkt von der CEP-Engine verarbeitet werden. Mit Hilfe des konfigurierten Nachrichtenformats können die Attribute der Nachricht, wie die Koordinaten der aktuellen Position, eingelesen und verarbeitet werden. Das Abfrageergebnis ist dann ebenfalls eine JSON-Nachricht, die direkt oder mit Hilfe des Frameworks von den Clients verarbeitet werden kann.

Neben den Sensoren gibt es im Living Place auch viele verschiedene Aktoren. Diese können unter anderem die Lichtinstallationen, Rollos und Vorhänge in der Wohnung steuern. Der Status der Aktoren beschreibt einen Teil des Zustands der Wohnung. Die Folge von Zustandsänderungen der Aktoren kann ebenfalls als Eventstrom betrachtet und durch die CEP-Engine verarbeitet werden. Damit können andere Komponenten der Wohnung sich nicht nur über aktuelle Statusinformationen benachrichtigen lassen, sondern auch komplexe Abfragen, die mit Sensordaten verbunden sind, starten.

Da die alte Implementierung der Aktoren die Zustandsänderungen nicht betrachtet, muss sie ersetzt werden. Dies dient ebenso als Test für die Entwicklung neuer Komponenten mit Hilfe des Frameworks. Die neuen Aktoren veröffentlichen Zustandsänderungen nun über eine separate Gruppe. Alle Nachrichten beschreiben, im Gegensatz zu den eingehenden Steuerungsnachrichten, einen erfolgreichen Zustandswechsel. Weiterhin wurde eine einheitliche JSON-Schnittstelle für die Status- und Zustandsabfrage entwickelt, die als Basis für weitere Entwicklungen genutzt werden kann.

Bevor ein Event von einer CEP-Engine verarbeitet werden kann, muss die Eventstruktur definiert werden. Oft werden hierfür Klassen benutzt. Ein Event wird dann durch ein einfaches Java-Objekt mit Gettern und Settern beschrieben. Für die Middleware würde das bedeuten, dass alle Events durch Klassen im Classpath der Middleware definiert werden müssen. Dies ist aber nicht möglich, weil neue Sensoren ohne Anpassung der Middleware zum System hinzugefügt werden sollen. Die eingesetzte CEP-Engine erlaubt die Beschreibung der Events durch eine Map-Struktur. Die Keys sind dabei immer Strings und die Values können beliebige Objekte sein. Damit sind also auch Verschachtelungen von Nachrichten möglich. Diese Art der Definition ist sehr ähnlich zum JSON-Format.

[1] Ubisense. http://de.ubisense.net/en/products-and-services/rtls-products.html

Ein Agent registriert einen neuen Eventtyp durch das Verschicken einer in JSON serialisierten Beispielnachricht. Nach Bestätigung der CEP-Engine der Middleware können jetzt Events veröffentlicht werden. Nicht zur Struktur passende Events werden bei der Verarbeitung ignoriert und der Sender wird darüber benachrichtigt. Ein Event bleibt so lange registriert, bis alle Agenten, die diesen Eventtyp angemeldet haben, beendet wurden. Um dies festzustellen, wird der Monitoring-Dienst der Middleware verwendet.

2.3 Framework zur Entwicklerunterstützung

Das Entwickeln neuer Komponenten und die Benutzung der Middleware werden durch ein Framework unterstützt. Das Framework stellt den Rahmen für die Implementierung eines neuen Agenten bereit, kapselt die Interaktion mit der Middleware und kann Aufgaben wie die Serialisierung übernehmen.

Die Interaktion mit der Middleware und anderen Komponenten soll in Form eines Agenten beschrieben werden. Dazu muss angegeben werden, wie die neue Komponente auf eingehende Nachrichten reagiert. Die Nachrichtenformate werden in wiederverwendbaren API-Bibliotheken definiert. Dort wird auch die Serialisierung in JSON festgelegt. Dies stellt außerdem sicher, dass Komponenten, die die API benutzen, vom Service verstanden werden können, da dort ebenfalls die Bibliothek verwendet wird. Alle Nachrichten müssen serialisierbar sein, damit sie über das Netzwerk geschickt werden können. Selbst Agenten, die direkt miteinander kommunizieren, müssen die benötigten Informationen für die Serialisierung bereitstellen, damit sie bei Bedarf verschoben werden können. Die Serialisierung muss vor dem Abschicken einer Nachricht geschehen, deswegen kann diese Aufgabe nicht von der Middleware übernommen werden.

Die Bibliotheken können versioniert werden. Die Middleware kann dann Versionskonflikte zwischen Komponenten automatisch erkennen. Weitere Vorteile der Repräsentation der Nachrichten als Objekte, im Gegensatz zu einfachen JSON-Nachrichten, sind die Typsicherheit und die mögliche Autovervollständigung in Entwicklungsumgebungen. Läuft die Kommunikation nicht über das Netzwerk, kann auf die Serialisierung ganz verzichtet werden und stattdessen können direkt Objekte ausgetauscht werden.

Auf die Kerndienste der Middleware kann direkt vom Framework aus zugegriffen werden. Der Entwickler kann die Gruppen, auf die der Agent hören soll, definieren und das Framework kümmert sich um die Kommunikation mit der Middleware. Außerdem kann angegeben werden, welche Nachrichten aus welchen APIs in Objekte umgewandelt werden sollen. Die vollständige Kontrolle hierüber bleibt jederzeit beim Entwickler. Wenn dies notwendig ist,

kann er direkt auf den JSON-Nachrichten arbeiten. Alle Nachrichten von und an die Middleware werden vom Framework über einen Tunnel geleitet, der eine at-least-once-Semantik sicherstellt. Bei einem Verbindungsabbruch kann das Framework die Verbindung automatisch wiederherstellen und die Agenten nach dem Timeout neu bei der Middleware registrieren. Der Agent erhält dabei Zustandsnachrichten und kann bei Bedarf auf die Änderungen reagieren. Für einfachere Agenten, wie Sensoren und Aktoren, ist dies nicht notwendig.

Das Framework stellt eine weitere Abstraktionsschicht vor der Middleware dar. Dies kann zum Beispiel bei Unit-Tests von großem Vorteil sein. Während der Ausführung der Tests kann das Framework eine Dummy-Middleware benutzen. Diese ist deutlich leichtgewichtiger und kann dazu benutzt werden, die Nachrichten der zu testenden Komponente an die Middleware zu überprüfen. Darüber hinaus können so lokal ausführbare Unit-Tests für bestimmte Anforderungen an alle Komponenten geschrieben werden. Hiermit kann zum Beispiel sichergestellt werden, dass alle Komponenten auf eine bestimmte Nachricht reagieren und ihren Status übermitteln. Außerdem können diese Tests bei Bedarf vor dem Deployment von der Middleware durchgeführt und bei gefundenen Fehlern zurückgewiesen werden.

Durch die dynamischen Konfigurationsmöglichkeiten der Middleware und durch den Einsatz automatischer Build-Tools ist es über das Framework auch möglich, Integrations- oder Systemtests durchzuführen. Dabei wird die zu testende Komponente zusammen mit einer lokalen Instanz der Middleware gestartet. Dabei können beliebige Middleware-Dienste geladen und andere Komponenten eingebunden werden. Da die Kommunikation über ActiveMQ Teil des Gruppenkommunikationsdienstes der Middleware ist, kann auch das Zusammenspiel mit alten Komponenten auf diese Weise getestet werden. Dabei wird automatisch ein lokaler ActiveMQ-Server gestartet.

Das Framework ist so entworfen, dass der Einstieg möglichst leicht ist. Es gibt keine Abhängigkeiten zu den im Hintergrund eingesetzten Technologien, wie ActiveMQ oder der CEP-Engine, und das Ausführen, der Build und das Deployment werden durch Tools unterstützt. Für die häufigsten Arten von Agenten, wie Sensoren, Aktoren und Kontext-Interpreter wird es Beispielprojekte geben. Damit soll der Zeitaufwand, einen Prototypen zu erstellen, reduziert und der Projektstart erleichtert werden.

Das benötigte Wissen über die Tools und das Framework werden im Living Place im Rahmen von Workshops und Coding-Sessions vermittelt (vgl. Kluth und Eichler, 2014). Dabei werden zuerst die Grundlagen in Form von Kurzvorträgen und Beispielen erläutert. Danach sollen die Teilnehmer in Teams einfache Agenten entwickeln. Hierbei sind insbesondere Agenten

interessant, die später von anderen Entwicklern wiederverwendet werden können. Der zusätzliche Arbeitsaufwand kann für die Projektmitglieder weiter reduziert werden, indem die Agenten so ausgewählt werden, dass sie direkt in dem studentischen Projekt benötigt werden. Daraus entsteht ein weiterer Vorteil, da so eine Aufteilung und Wiederverwendbarkeit der Komponenten erzwungen wird.

2.4 Zugriff von alten Komponenten

Für den Zugriff von alten Komponenten auf das System wird eine ActiveMQ-Schnittstelle mit demselben Funktionsumfang bereitgestellt. Dies geschieht automatisch durch die Middleware. Diese übernimmt die Weiterleitung der Nachrichten in Gruppen auf die entsprechenden ActiveMQ-Topics.

Teil der Schnittstelle ist eine Heartbeat-Nachricht, auf die alle Systeme antworten müssen. So kann die Middleware alle angemeldeten Agenten überwachen und angeforderte Dienste beenden, wenn diese nicht mehr benötigt werden. Die Entscheidung, ob eine Komponente erreichbar ist, ist in einem verteilten System nicht trivial. Werden die Heartbeat-Nachrichten eine bestimmte Zeit lang nicht beantwortet, wird davon ausgegangen, dass die Komponente nicht mehr erreichbar ist. Diese Entscheidung kann nur durch die erneute Anmeldung der Komponente bei der Middleware rückgängig gemacht werden. Die Zeitfenster, in denen geantwortet werden muss, müssen hierfür sorgfältig gesetzt werden.

2.5 Implementation

Die Middleware und das Framework werden auf der Basis gut gepflegter Open-Source-Projekte, die auch in der Wirtschaft breite Akzeptanz finden, implementiert. Als Basis für das Framework und die Kommunikation dient das Aktor-Framework Akka[2]. Das Framework enthält unter anderem Funktionen zur transparenten Kommunikation über das Netzwerk und Fehlerbehandlungsstrategien für Aktoren. Als CEP-Engine wird Esper[3] verwendet. Esper ist sehr flexibel bei der Eventpräsentation, was es dem CEP-Dienst erlaubt, Eventypen aus JSON-Beispielnachrichten zu erstellen. Als Abfragesprache wird EPL verwendet, die durch die Ähnlichkeit zu SQL leicht erlernbar ist. Beide Projekte verfügen über eine umfangreiche Online-Dokumentation und es gibt viele verfügbare Beispielprojekte.

[2] J. Bóner et al. „Akka library". http://akka.io
[3] EsperTech Esper. http://esper.codehaus.org

3 Schlussbetrachtung

3.1 Zusammenfassung

Ausgehend von der alten Architektur im Living Place und den damit entstanden Problemen wurde der Entwurf einer neuen Middleware vorgestellt. Die neue Middleware besteht aus lose gekoppelten Diensten, wie zum Beispiel Gruppenkommunikation und Monitoring. Teil der Middleware ist eine Laufzeitumgebung, in der Agenten mit geringem Overhead ausgeführt werden können. Die Interaktion zwischen den Diensten wurde beispielhaft anhand der Integration von Complex Event Processing zur Kontextverarbeitung aufgezeigt. Getestet wurde dies mit einem Sensor für die Position von Personen und den Zustandsänderungen der Aktoren für Licht, Rollos, Vorhänge und Fenster. Dafür wurde eine neue Schnittstelle entwickelt, die auch Status- und Zustandsänderungen kommunizieren kann.

Die Komponenten des Systems müssen nur mit der Middleware kommunizieren und die Agenten haben keine Abhängigkeiten von den in der Middleware eingesetzten Technologien. Die Entwicklung neuer Komponenten und die Kommunikation mit der Middleware werden durch ein bereitgestelltes Framework und durch Entwicklertools vereinfacht. Zudem erfolgt die Kommunikation ausschließlich über Nachrichten, wodurch eine lose Kopplung erreicht wird. Das Testen und Finden von Fehlern wird durch Fehlerbehandlungsstrategien und Monitoringfunktionen der Middleware für Entwickler erleichtert. Die Middleware und alle Komponenten können zur Laufzeit konfiguriert werden. Damit werden kontextabhängige Konfigurationsänderungen des Systems zur Laufzeit möglich.

3.2 Ausblick

Im Weiteren sollen die anderen Dienste der Middleware implementiert und getestet werden. Das entstandene System wird mit allen neuen und alten Komponenten im Living Place evaluiert. Mit der ersten Version des Frameworks und der Middleware starten weitere Workshops mit den Studenten im Living Place. Dies soll zu einer frühen Adaption des neuen Systems und zur Evaluation der Entwicklerunterstützung führen.

In zukünftigen Arbeiten könnten weitere Dienste für die Middleware identifiziert und integriert werden. Dazu könnte eine Speicherkomponente für Kontextinformationen und die Auswertung von Zuständen durch überwachtes Lernen zählen.

Literaturverzeichnis

[Eichler 2013] Eichler, Tobias: *Projekt 1 - Entwicklung einer Middleware für das Living Place.* 2013

[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

[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

[Kluth und Eichler 2014] Kluth, Sascha ; Eichler, Tobias: *Kontextadäquate Anpassung einer agentenbasierten SmartLiving Laborumgebung für agile interdisziplinäre Entwicklungsteams an der HAW Hamburg.* 2014. – URL http://www.hs-bremerhaven.de/studienangebot/bachelorstudiengaenge/informatik/veranstaltungen/nkif2014/programm. – Präsentation am 16.05.2014

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

[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