Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Alexander M. Sowitzki

Integration von Complex Event Processing in ein Multiagentensystem für Smart Environments

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Alexander M. Sowitzki

Integration von Complex Event Processing in ein Multiagentensystem für Smart Environments

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Kai von Luck, M.Sc. Tobias Eichler

Eingereicht am: 17. November 2016

Thema der Arbeit
Integration von Complex Event Processing in ein Multiagentensystem für Smart Environments

Stichworte
Complex Event Processing, Middleware, Smart Environment, Smart Home

Zusammenfassung
In Multiagentensystemen wird eine Vielzahl an Ereignissen erfasst und verarbeitet. Die Verarbeitung geschieht normalerweise mittels eines prozeduralen Ansatzes: Agenten bilden komplexe Ereignisse für sich selbst oder stellvertretend für andere. Um dies fehlerfrei durchzuführen, ist eine aufwändige Entwicklung dieser Verarbeitungslogik nötig, was agiler Softwareentwicklung entgegensteht. In dieser Arbeit wird untersucht, inwieweit sich die deklarative Programmierung der Ereignisverarbeitung mit Complex Event Processing dafür eignet. Dazu wird eine Complex Event Processing Engine in eine bestehende Middleware für Smart Environments integriert. Anhand von Szenarien wird überprüft, ob die Aufgabenangemessenheit dafür gegeben und verbesserte Möglichkeiten für die Entwickler von Agenten geboten sind.

Title of the paper

Integration of Complex Event Processing in a Multi-Agent System for Smart Environments

Keywords

Complex Event Processing, Middleware, Smart Environment, Smart Home

Abstract

A diversity of events is processed in multi-agent systems. The processing is usually performed by a procedural approach. Agents form complex events exclusively for themselves or others. To accomplish this without error, development must be done carefully at the cost of being time-consuming. This stands against agile software development. In this thesis shall be researched how a complex event processing engine with declarative programming can be used to perform better. To accomplish this, a processing engine will be integrated in a smart environment middleware. Different scenarios will be used to verify that this solution is adequate to solve the problem and help agents to be developed faster.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Das Labor Creative Space for Technical Innovations (CSTI) der *HAW Hamburg*, für welches dieses Projekt durchgeführt wird, besteht aus vielen Computersystemen und Elektronik mit Sensoren und Aktoren, die über ein Netzwerk Daten miteinander austauschen, um kooperativ für eine bessere Besuchererfahrung zu sorgen. Dieses System ist aufgrund seines Zwecks ein Smart Environment und funktional ein Multiagentensystem.

Ein Agent innerhalb des Systems ist hier eine Komponente, die eigenständig eine spezielle Aufgabe lösen kann und mit anderen Agenten kooperiert, um höhere Ziele zu erreichen. Diese Kooperation findet durch den Austausch von Informationen in Form von Ereignissen statt, weswegen deren Verarbeitung eine große Rolle einnimmt. Die von Agenten mit Sensoren generierten Messdaten sind ohne Kontext aussagelos.

Erst durch Interpretation und Kontextanreicherung können aus diesen Daten Rückschlüsse auf die Einsatzumgebung gezogen werden. So ist die Information, dass z.B. innerhalb eines Smart Environments ein Magnetschalter geschlossen wurde, an sich nicht weiter relevant. Mit Bezug auf den Kontext lässt sich jedoch feststellen, dass bsw. die Eingangstür geöffnet wurde. Dieses Ereignis kann wiederum mit weiteren Kontexten angereichert werden, um Aktoren zu steuern. So kann die Beleuchtung ein- oder ausgeschaltet werden, da aus der Türöffnung geschlussfolgert werden kann, dass sich ein Benutzer in die Umgebung hinein oder aus ihr heraus bewegt. Das Verarbeiten von Informationen, unter Zuhilfenahme von Kontext und Interpretationsweise, zu Ereignissen sowie deren Gruppierung und Weiterverarbeitung ist als Complex Event Processing (CEP) bekannt.

Ein Vorteil einer universellen Multiagentenumgebung ist die einfache Erweiterbarkeit. Neue Sensoren und Aktoren können separat entwickelt und einfach in die Umgebung integriert werden. Dies ist u.a. dadurch möglich, dass alle Agenten über eine gemeinsame Struktur miteinander kommunizieren, die sogenannte Middleware. Diese stellt alle Funktionen für Agenten bereit, um Nachrichten innerhalb des Systems senden und empfangen zu können.

Jedoch existieren für die Entwicklung von neuen Agenten Hürden, die sich im Bereich der Ereignisverarbeitung befinden. Die Durchführung von CEP wird von Kontextinterpretern innerhalb des Agentensystems übernommen, welche im Normalfall die Agenten sind,

die direkt auf die Ereignisse angewiesen sind. Dies geschieht über prozedurale Programmierung, innerhalb des Agenten werden alle nötigen Arbeitsschritte umgesetzt. Die Umsetzung von neuen Regeln ist zeitaufwändig und fehleranfällig, da neben der Umsetzung der eigentlichen Aufgabe des Agenten für CEP eigens ein Programmteil entwickelt werden muss, dessen Komplexität bei umfangreicheren Regeln die des übrigen Programms weit übersteigen kann und dadurch u.a. schwer wartbar ist. Für Tests muss eine komplette Simulationsumgebung aufgebaut werden. Diese Umstände erschweren letztlich agile Softwareentwicklungsmodelle wie Rapid Prototyping und sind eine Behinderung für die zügige Umsetzung von neuen Laborprojekten.

1.1. Zielsetzung

In dieser Arbeit soll untersucht werden, inwieweit der Einsatz einer CEP Engine angemessen ist, um die dargestellten Probleme zu vereinfachen. CEP Engines werden hauptsächlich im Geschäftsumfeld verwendet, um die Erstellung von Verarbeitungsregeln für Ereignisse über deklarative Domain Specific Languages (DSLs), also spezialisierte Sprachen, zu ermöglichen.

Dazu wird eine solche Engine in die bestehende Middleware des CSTI integriert. Mit Hilfe von Fallstudien soll ermittelt werden, ob die Aufgabenangemessenheit gegeben ist und die Integration der Engine zu einem verbesserten Entwicklungsprozess von Agenten im Labor führen kann.

1.2. Gliederung

Zum Erreichen dieses Ziels werden, einschließlich der Einleitung, fünf Kapitel erstellt.

In der Analyse wird das Grundlagenwissen zum Thema aufgebaut. Es wird darauf eingegangen, worum es sich bei Multiagentensystemen handelt und wie sie aufgebaut sind, was unter Smart Environments verstanden wird, welche Anforderungen an diese gelten und wie sie mit Multiagentensystemen in Zusammenhang stehen. Anschließend wird die CSTI Middleware und das Nutzungsumfeld, das CSTI, beleuchtet. Folgend wird in das eigentliche Grundthema, CEP, eingeführt und ein Vergleich mit dem Bestandssystem aufgestellt. Es wird ermittelt, welche Technologie aus diesem Bereich verwendet werden soll und erörtert, welche Möglichkeiten sich für die Integration bieten. Dann werden verwandte Arbeiten betrachtet, eine Abgrenzung vorgenommen und Anwendungsszenarien aufgestellt, mit welchen überprüft werden soll, ob die Integration einen Vorteil bietet. Die Analyse wird mit einem Fazit beendet.

Im Design wird die Entwicklungsplanung durchgeführt, also u.a. die Ermittlung von benötigten Systemkomponenten zur Umsetzung der Szenarien angestellt, Entwicklungstools und Vorgehensweisen ausgewählt und ein Testplan entwickelt. Des Weiteren wird die Programmarchitektur aufgestellt und Eigenschaften wie die Pluginfähigkeit erläutert. Im letzten Teil des Designs wird die Anbindung an die Middleware beschrieben. Dazu wird ein Application Programming Interface (API) entwickelt, der Ablauf von Anfragen von Agenten an die CEP Engine geplant, die Interaktion der CSTI Middleware mit der Engine, mit Schwerpunkt auf den Fluss von Ereignissen, festgelegt und anschließend auf umgesetzte Maßnahmen zur Sicherstellung von Fehlertoleranz eingegangen.

In der Realisierung wird sich für die zu verwendenden Technologien entschieden. Dies schließt die Auswahl einer Implementierung der CEP Engine sowie zu verwendende Programmbibliotheken ein. Folgend wird die Umsetzung der Anwendungsszenarien exemplarisch beschrieben und ein umgesetzter Agent gezeigt. Im Ausblick werden anschießend Weiterentwicklungsmöglichkeiten des Projekts aufgezeigt und zuletzt ein Fazit gezogen.

Im letzten Kapitel werden die Erkenntnisse der Arbeit zusammengefasst und ein Ausblick gegeben, inwieweit das Thema weiter bearbeitet werden kann.

2. Analyse

In diesem Kapitel werden die Grundlagen, Multiagentensysteme und Smart Environments, beleuchtet. Anschließend wird auf das CSTI und die CSTI Middleware eingegangen. Es folgt die Behandlung von CEP, die Technologieauswahl und die Integrationsplanung. Zuletzt werden verwandte Arbeiten aufgezeigt, die Arbeit abgegrenzt, Szenarien für das Projekt aufgestellt und ein Fazit gezogen.

2.1. Multi-Agent Programming

Ein Smart Environment setzt sich im Normalfall aus mehreren sogenannten Agenten zusammen. Ein Agent ist hier ein Programm, welches in einem bestimmten Rahmen Aufgaben selbstständig ausführen kann. Dazu gehört zum Beispiel das Erfassen von Temperaturmessdaten oder die Steuerung eines Motors. Während diese Aufgaben für sich gut durch einen Agenten lösbar sind, muss für komplexere Aufgaben eine Zusammenarbeit zwischen verschiedenen Agenten erfolgen. Die Mindestvoraussetzung dafür ist eine Kommunikationsmöglichkeit zwischen ihnen. [1, S. 10][2]

So könnte z.B. in einer Lagerhalle die Logistik von automatischen Gabelstaplern übernommen werden, von denen jeder als eigenständiger Agent agiert und mit anderen Gabelstaplern kooperiert, um eine effiziente Verwaltung zu ermöglichen.

2.1.1. Kommunikationsarten

Die Kommunikation zwischen Agenten kann auf verschiedene Arten erfolgen, von denen jede Vor- und Nachteile mit sich bringt.

Remote Procedure Call

Remote Procedure Call (RPC) lässt ein Programm eine Methode eines anderen Programms im Netzwerk aufrufen, als wäre sie programmeigen. Der Aufruf ist üblicherweise synchron, was bedeutet, dass das aufrufende Programm auf die Durchführung der Methode wartet und währenddessen selbst blockiert ist. Diese Art der Kommunikation ist unkompliziert, jedoch

allgemein in der Kritik, da der Programmcode dadurch auf verschiedene Systeme verstreut wird und an Übersichtlichkeit verliert. Ein Aufrufer kann nur eine Methode zurzeit aufrufen. Sollen Gruppen erreicht werden, so sind mehrere Aufrufe abzusetzen, was weiterhin zur Unübersichtlichkeit beiträgt und eine geringere Performance verursacht. [3]

Message Passing

Agenten strukturieren ihre Kommunikation in Nachrichten, welche asynchron an einen anderen Agenten im Netzwerk verschickt werden. Auf eine Versendung erfolgt hier im Gegensatz zu RPC kein Aufruf einer Methode mit definierten Parametern und Rückgabewerten. Anfragen werden über definierte Nachrichtenformate an die Gegenstelle übermittelt. Antworten müssen durch zusätzliche Nachrichten versandt werden. Ist ein Teilnehmer nicht erreichbar, geht die Nachricht verloren. [4]

Soll eine Nachricht an mehrere Teilnehmer gesendet werden, so ist sie durch einen Multicast an eine Gruppe oder für jeden Teilnehmen einzeln mittels Unicast zu verschicken. Ein Unicast hat den Vorteil, dass die Empfänger explizit bestimmt werden können. Das Versenden von Nachrichten an Knoten, die keine Verwendung für diese haben, ist unwahrscheinlich, da es sich um eine Ende-zu-Ende Verbindung handelt. Deshalb kann unkompliziert ein Tunnel für z.B. Verschlüsselung verwendet werden. Eine Multicastverbindung überträgt eine Nachricht an eine Gruppe von Interessenten. Der Sender kann dabei nicht bestimmen, welche Agenten diese empfangen und hat somit keine sicheren Informationen zum Empfänger, was Bestätigungen und Tunnelung der Nachricht erschwert. Von Vorteil ist, dass neue Agenten ohne Modifikation des Senders dem Netzwerk hinzugefügt werden können. [5]

Weitere Formen des Routings sind Anycast und Broadcast, welche jedoch in diesem Anwendungsbereich nicht zweckmäßig sind.

2.1.2. Topologien

Neben der Nutzung verschiedener Kommunikationsarten muss die Menge der Agenten in einer Topologie angeordnet werden. In einem Multiagentensystem kommen hierzu hauptsächlich zwei Varianten in Frage: die direkte Kommunikation aller Teilnehmer untereinander in einem großen Netzwerk oder die Unterteilung der Agenten in Untergruppen, die ihre Kommunikation über Message Broker erledigen.

Direkte Kommunikation (Peer-to-Peer)

Der Austausch von Nachrichten könnte über ein Peer-to-Peer Netzwerk umgesetzt werden. Jeder Agent spricht ohne Umweg über weitere Dienste mit anderen Agenten. Dieser Ansatz ist einfach zu erstellen und benötigt keine zentrale Serverkomponente, was die Wahrscheinlichkeit eines Single-Point-of-Failure reduziert sowie fließende Skalierung und eine geringe Latenz ermöglicht. Selbst größere Peer-to-Peer Netzwerke sind mit gewissem Planungsaufwand realisierbar. Letztlich sind jedoch das Testen und die Überwachung eines solchen Netzes mit höherem Aufwand verbunden, da dazu der Datenverkehr des Netzes an verschiedenen Stellen des Netzes inspiziert werden muss. Fällt ein Agent aus, gibt es zudem keine Zwischenstelle, die die für den Agenten bestimmte Kommunikation puffern kann - speichert der Sender Nachrichten nicht, gehen sie verloren. Bei größeren Netzwerken ist zudem eine umfassende Dokumentation aller Kommunikationswege nötig, um Wartbarkeit zu gewährleisten. [6]

Message Broker

Ein Ansatz, der diese Probleme aufgreift, ist der Einsatz eines Message Brokers. Dies ist eine Serveranwendung, die sich um die Zustellung von Nachrichten zwischen den Agenten kümmert. Sie kann zentral als einzelne Instanz existieren oder in einem Cluster angeordnet werden, um eine hohe Verfügbarkeit sicherzustellen. Agenten können beim Message Broker Nachrichten auf unterschiedlichen Kanälen veröffentlichen oder sich auf Kanälen registrieren, um Nachrichten von diesen zu erhalten. Diese Vorgehensweise ist als Publish Subscribe bekannt und bringt mehrere Vorteile. So kann Zugangskontrolle zu verschiedenen Kanälen eingesetzt werden, um die Authentizität von Nachrichten zu gewährleisten. Zudem kann an einer zentralen Stelle das Gesamtsystem mit der Übersicht über alle Teilnehmer des Netzwerks einfacher überwacht und getestet werden. [7]

Wie bei Multicast wird eine Nachricht von einem Agenten an eine Gruppe ausgegeben. Im Fall des Message Broker ist dieses Ziel abstrakt. Auf eine solche Gruppe können sich Agenten registrieren. Pro Gruppe wird auf Seite des Brokers eine Warteschlange erstellt, in der Nachrichten gepuffert werden. Dies bringt mehrere Vorteile mit sich. Falls die Verarbeitung von Nachrichten länger dauert als deren Generierung, kann bis zu einem gewissen Maß das Verwerfen von Nachrichten vermieden werden. Zudem kann dadurch der Verlust von Nachrichten reduziert werden, wenn Agenten oder Netzwerkkomponenten temporär ausfallen, indem der Agent optional um Persistierung wartender Nachrichten bittet. Weitere optionale Funktionen sind Reliable Delivery, also das Informieren des Senders über die Bearbeitung (Nicht deren Zustellung) einer Nachricht und Vorfiltrierung von Nachrichten anhand von komplexen

Routingbedingungen. Die Parallelisierung von Aufgabenverarbeitungen ist dadurch einfacher zu realisieren. Eine Verbindung von mehreren Standorten mit mehreren unabhängigen Message Brokern ist möglich - Nachrichten können zwischen ihnen geroutet werden. Selbst komplexere Systeme, die wartbar, skalierbar und erweiterbar sind, können so effektiv umgesetzt werden. [8]

Leider ist das Erfüllen dieser Funktionen aufwändig und erfordert einen performant arbeitenden Message Broker, eine Überlastung führt zum Aufstauen des gesamten Nachrichtenverkehrs. Ein System, welches die Verarbeitung nahe der Echtzeit unterstützt, ist nur umständlich zu realisieren. Dies ist von Nachteil, da viele Funktionen in einem Smart Environment nur einfache Nachrichtenübermittlung erfordern, ein hohes Datenvolumen erzeugen und auf eine geringe Latenz angewiesen sind. [9]

Der Einsatz eines Message Brokers vereinfacht die Erstellung von Multiagentensystemen, löst aber eines der Kernprobleme nicht: Die Verarbeitung von Ereignissen. Aus der Kombination verschiedener Ereignisse lassen sich weitere komplexere Ereignisse folgern, die für die Operation eines Smart Environments benötigt werden.

Message Broker verarbeiten grundsätzlich den Inhalt der übermittelten Nachrichten nicht und lassen ihn unverändert - die Nachricht wird als *Black Box* behandelt. Ausschließlich anhand von Metainformationen des Protokolls werden Nachrichten geroutet und vorgefiltert. Viele Message Broker geben daher nur minimale Vorgaben an die Nutzlast der Nachricht. Deshalb können Clientbibliotheken keine weitere Verarbeitung des Inhalts vornehmen. Der Agent erhält die Nachricht als Rohformat und muss selbst Analyse und Konvertierung vornehmen, was zu einer höheren Entwicklungszeit und schlechterer Wartbarkeit führt.

Daher unterstützt kein Produkt mit Open Source Lizenz eine solide Lösung für die Verwaltung und Zustellung von Nachrichten die Bildung dieser Ereignisse. So erhalten Agenten die Nachrichten, konvertieren sie in ein nutzbares Format und bilden aus den einfachen Ereignissen für sich selbst benötigte komplexe Ereignisse. Dieses für jeden Agenten zu realisieren ist umständlich, komplex und fehleranfällig.

2.2. Smart Environments

Ein Smart Environment ist eine physikalische Welt, die sichtbar und unsichtbar mit Computerelementen wie Sensoren, Aktoren und Bildschirmen angereichert ist. Diese befinden sich innerhalb von alltäglichen Objekten und sind in einem Netzwerk verbunden. Die Begrifflichkeit erweitert die Definition von Ubiquitous Computing, welche beschreibt, dass Computer nicht nur als einfache Arbeitsstation erscheinen können, sondern auch in jedem anderen Gerät, Format und Standort. [10]

2.2.1. Zusammenhang mit Multiagentsystemen

Ein Smart Environment kann als eine spezielle Form eines Multiagentensystems angesehen werden: Jedes Computerelement in einem Smart Environment kann die Rolle eines Agenten übernehmen. Eine Smart Environment Komponente kann z.B. an Hardwarekomponenten angebunden werden, um diese zu kontrollieren. Zum einen kann so die Steuerung von Aktoren, wie das Schalten von Beleuchtung oder das Erfassen von Umgebungsdaten, durch das Einlesen von Sensoren, wie Kontaktmeldern bei Schaltern erreicht werden. Diese Funktionen können die Komponenten für sich allein ausführen, um jedoch aus dieser Ansammlung von Elektronik ein smartes System zu erstellen, ist eine Vernetzung zu einem Multiagentsystem nötig. Denkbar wäre auch die Bündelung aller Sensoren und Aktoren an einer computerisierten zentralen Stelle. Jedoch erübrigt sich diese Variante aufgrund von elektronischen Beschränkungen wie Signalreichweiten, Verkabelungs- und Entwicklungsaufwand. [11]

2.2.2. Smart Homes

Ein Smart Home ist eine Sonderform eines Smart Environments. Eine gewöhnliche Wohnumgebung wird mit smarten Komponenten ausgerüstet, um die Wohnqualität zu steigern. So kann Sensorik im Schlafzimmer erkennen, wenn ein Bewohner morgens aufsteht, und in Abhängigkeit davon Toast und frischen Kaffee zubereiten, noch bevor der Bewohner die Küche betritt. Dies kann mit verschiedensten Motivationen umgesetzt werden. So könnte das Alltagsleben von Bewohnern vereinfacht, Pflegefälle einfacher betreut oder eine kindersichere Umgebung geschaffen werden.

2.2.3. Stakeholder

Beteiligte Personen an einem Smart Environment sind folgende:

- **Systemarchitekten** kümmern sich um die Planung und Strukturierung der Umgebung auf Systemebene und liefern entscheidende Arbeit für die komplette Lebensdauer der Einrichtung. Diese Rolle existiert nur für größere Smart Environments. Die Rolle legt Nachrichtenformate sowie den Aufbau des Netzwerks fest.

- **Entwickler** setzen die Planung durch das Programmieren und Bereitstellen von Agenten um. Dies umfasst die interne Konzeption eines Agenten, dessen Programmierung und Ansteuerung von Sensorik und Aktorik.

- **Administratoren** verwalten und warten über einen längeren Zeitraum das System. So helfen sie bei der Integration von neuen Agenten, der Pflege der Hardware und der Aktualisierung der Komponenten.

- **Designer** gestalten als Interaction Designer das Benutzererlebnis ohne starken Implementierungsbezug. Der Funktionsumfang und die Art, wie dieser angeboten wird, entscheidet über die Wirkung der Smart Environments.

- **Benutzer** sind die Zielgruppe des Smart Environments. Sie stellen Anforderungen und sind im Normalfall nicht an der Entwicklung beteiligt.

2.2.4. Anforderungen

Ein Smart Environment stellt Grundanforderungen, welche auch in diesem Projekt gelten. Sie lauten [12]:

- **Ressourcensparsamkeit**: Agenten in einem Smart Environment nehmen eine besondere Rolle ein, da sie sich in unterschiedlichsten Gegenständen wie Kühlschränken, Lampen und Teetassen wiederfinden. Ein größer werdender Teil dieser Geräte ist klein, embedded und mit wenig Rechenleistung ausgestattet, muss jedoch zum Erfüllen der Funktion größere und komplexere Datenmengen wie z.B. Sensordaten verarbeiten. Aus diesem Grund müssen Agenten effizient arbeiten. Ein weiterer Grund dafür ist die zunehmende Komplexität der Aufgaben mit größer werdenden Anforderungen an Smart Environments und steigendem Ressourcenbedarf.

- **Interoperabilität**: Während der Entwicklung eines Smart Environments ist schwer absehbar, welche Agenten im Laufe der Benutzung hinzukommen. Dies setzt eine offene Entwicklungsphilosophie voraus, die schon beim ersten Entwurf eines Agenten berücksichtigt, dass dieser mit unbekannten Teilnehmern des Smart Environments interagieren

muss. Neue Agenten können neue Informationen in unterschiedlichsten Formaten liefern, weswegen Nachrichten in einem universellen Format erstellt werden und für einen möglichst großen Teil von Anwendungsfällen nutzbar sein sollten.

- **Zugänglichkeit**: Ein Smart Environment wird normalerweise während seines Betriebs erweitert. Deswegen muss das System so entworfen werden, dass die Art und Weise wie Komponenten integriert werden können, für Entwickler leicht erschließbar und verständlich ist und möglichst eine Infrastruktur zum Testen bietet.

- **Verarbeitungsgeschwindigkeit**: Um die Anforderungen aller Stakeholder erfüllen zu können, muss die CEP Engine einen hohen Durchsatz mit geringer Latenz aufweisen. Sensoren wie zum Beispiel Positionserkenner produzieren eine große Menge an Ereignissen in kurzer Zeit, die nahezu in Echtzeit verarbeitet werden müssen. Findet die Verarbeitung zu langsam statt, verlieren Daten ihre Gültigkeit.

- **Bedienbarkeit**: Die Kernanforderung an das System ist die einfache Bedienbarkeit, auch durch technikfremde Personen. Dazu muss die Systeminteraktion intuitiv und verständlich sein - selbst bei der Einrichtung des Systems soll, soweit möglich, die Konfiguration selbstständig im Hintergrund ablaufen.

2.3. CSTI Middleware

Das CSTI verwendet eine spezialisierte Middleware, die von Tobias Eichler entwickelt und gewartet wird (Siehe dazu [1]). Die CSTI Middleware wurde initial als Prototyp im Living Place, einem Labor der *HAW Hamburg*, entwickelt und ist mittlerweile im CSTI angesiedelt. Sie orientiert sich an den Bedürfnissen eines Smart Environments [1, S. 22].

Die Middleware arbeitet mit einer Publish Subscribe Architektur. Agenten können Ereignisse in Gruppen veröffentlichen und Gruppen abonnieren, um Ereignisse zu erhalten. Eine Gruppe kann verschiedene Ereignistypen handhaben. Als Serialisierungsformat wird JavaScript Object Notation (JSON) verwendet, was die Nutzung von primitiven Datentypen wie Integer, Gleitkommazahlen und Strings sowie Dictionaries und Listen einschließt. Es wird kein RPC unterstützt. Zudem erreicht die CSTI Middleware eine besonders geringe Latenz bei der Übertragung von Nachrichten. In dem von Eichler aufgestellten Testszenario erreicht die Middleware mit 100 Agenten eine Latenz von $8\,\mathrm{ms}$. Agenten ordnen sich um eine Instanz der CSTI Middleware an, die als Message Broker agiert. Mehrere Instanzen des Message Brokers können dabei als Peer-to-Peer Netzwerk kommunizieren. [1]

Wie für Middlewares üblich ist diese nicht für die Verarbeitung von Daten sondern nur für deren Transport zuständig. Das zu entwickelnde Programm soll daher eine universelle Komponente zur Verarbeitung von Daten werden und die Entwicklung von Agenten durch Studenten vereinfachen.

2.3.1. API-Generator

Ein besonderer Schwerpunkt liegt bei der Benutzerfreundlichkeit. Es soll einfach und unkompliziert möglich sein, neue Agenten zu entwickeln. Die CSTI Middleware beinhaltet einen Generator für die Erstellung von auf die Clienten zugeschnittenen Zugriffsbibliotheken in verschiedenen Programmiersprachen. Diese Bibliotheken serialisieren für den Transport native Objekte und deserialisieren sie beim Ziel, sodass ohne weitere Konvertierung die eigentliche Aufgabe des Agenten umgesetzt werden kann.

Der API-Generator ist ein Gradleplugin, das eine einzelne Konfigurationsdatei erhält und daraus alle benötigten Bibliotheken generiert. Das generierte Artefakt kann direkt in ein Mavenrepository geladen werden.

2.3.2. Akka

Die CSTI Middleware verwendet Akka als Basisplattform. Akka ist ein Framework zur Entwicklung von verteilten Systemen in Java VMs. Es zeichnet sich durch Möglichkeiten zur starken Parallelisierung und Verteilung von Anwendungen aus und verwendet zur Kommunikation zwischen den einzelnen Instanzen Message Passing. [1, S. 53]

2.4. Nutzungsumfeld

Das CSTI ist ein interdisziplinäres Labor der *HAW Hamburg*, welches sich mit Human Computer Interaction (HCI) als Schwerpunkt befasst. Es befindet sich an einer Außenstelle am Steindamm und ist ein umfunktionierter Seminarraum. Das Labor existiert seit Oktober 2015 und ist im Februar 2016, mit dem in Abbildung 2.1 dargestellten Raumplan, fertiggestellt worden. Das Labor beheimatet eine Vielzahl von Projekten aus den Bereichen Virtual Reality, Interaction Design, künstlerische Gestaltung, 3D-Druck und weiteren, für die ein produktives Arbeitsumfeld geschaffen werden soll. Die Vielzahl und Diversität der verschiedenen Bereiche macht dies zu einer fordernden Aufgabe. Im Netzwerk des Labors agiert eine Vielzahl von Agenten, einige im Testbetrieb einige zur langfristigen Bereitstellung von Diensten, mit unterschiedlichen Anforderungen. [13]

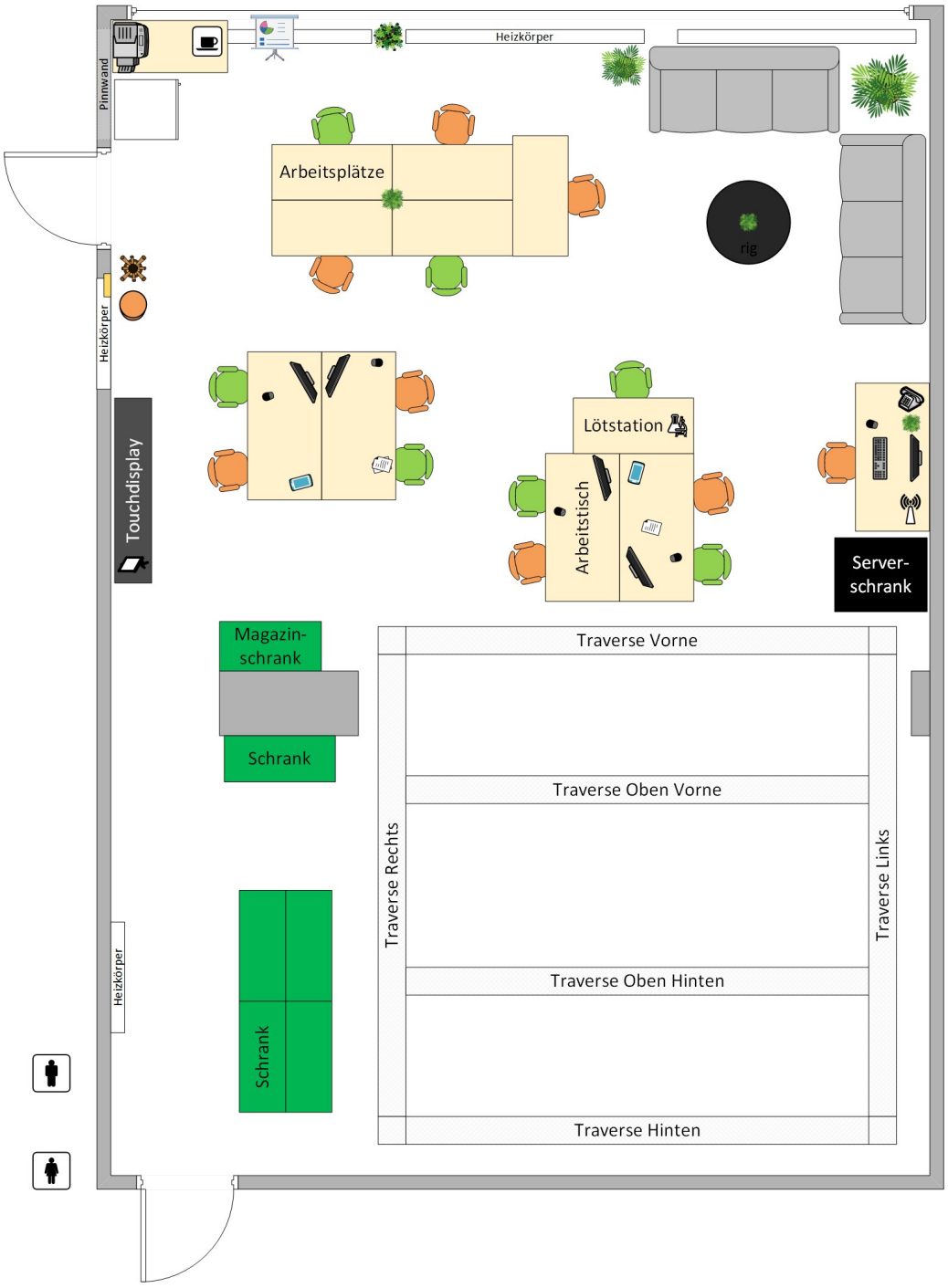

Abbildung 2.1.: Raumplan des CSTI

2.4.1. Spezielle Anforderungen

Eichler befasst sich in seiner Masterarbeit ([1, S. 22]) u.a. mit der Erfassung der Anforderungen von Smart-Home-Laborumgebungen. Diese gelten ebenfalls für dieses Projekt und werden im Folgenden zusammengefasst.

- **Einfindbarkeit für Entwickler**: Das Labor wird von einer größeren Anzahl an Studenten genutzt, die sich in die Nutzung der CEP Engine einarbeiten müssen. Aus diesem Grund soll die Einarbeitungszeit möglichst gering sein. Es wird hierbei vorausgesetzt, dass es sich um Studenten mit Informatikvorkenntnissen handelt, da dies auf den Großteil der im Labor agierenden Entwickler zutrifft. Die Konfiguration der CEP Engine für informatikfremde Studenten anzubieten, läge außerhalb des Zeitrahmens.

- **Interoperabilität**: Während andere Smart Environments oft von zusammenarbeitenden Arbeitsgruppen entworfen werden, treffen hier völlig unabhängige Einzelprojekte aufeinander, die von verschiedenen Studenten entwickelt werden. Um einen Parallelbetrieb oder gar eine Kooperation zwischen Agenten zu ermöglichen, die nicht zwangsläufig auf dasselbe Ziel hinarbeiten, ist besondere Umsicht erforderlich.

- **Wiederverwendbarkeit der Logik**: Obgleich die Agenten unterschiedliche Ziele verfolgen, verwenden sie dieselben Sensoren um das Geschehen im CSTI zu erfassen. Aus diesen Sensordaten können Zustände ermittelt werden, die nicht nur für einen Sensor von Interesse sind. So ist für Anwendungen, die die Position eines Anwenders verwerten, unter anderem relevant, ob sich dieser unterhalb des Traversensystems befindet und nicht außerhalb, wo einige Funktionen des Laborsystems nicht umsetzbar sind. Dies ist mit Kenntnis der Labordimensionen und einem Livedatenstrom der Benutzerposition einfach ermittelbar, führt jedoch zu Redundanzen und einem hohen Änderungsaufwand, sobald sich die Dimensionen der Traverse ändern. Deswegen sollten Agenten Teile ihrer Logik an das System auslagern können.

- **Einfache Betreuung**: Das Labor wird von Mitarbeitern und studentischen Hilfskräften der *HAW Hamburg* betreut. Da die Aufgaben innerhalb des Labors umfangreich sind, kann keine Stelle geschaffen werden, die sich ausschließlich um Betreuung und Administration kümmert. Eine wichtige Anforderung ist deshalb, dass das System wartungsfreundlich ist. Redundanzen sollten vermieden werden, damit der Konfigurationsaufwand nicht unnötig hoch und mit langfristiger Planung ein stabil laufendes System erschaffbar ist.

2.5. Complex Event Processing (CEP)

Der Bereich CEP, ein Themenbereich der Informatik, befasst sich mit der Analyse und Verarbeitung von kontinuierlichen Datenströmen.

2.5.1. Grundlagen

Im Gegensatz zur üblichen Datenverarbeitung, die mit statischen Daten operiert, werden bei CEP Datenströme verarbeitet. Diese Datenströme enthalten in der Regel zeitkritische Daten, wie beispielsweise Börseninformationen oder Temperaturmesswerte, und liefern kontinuierlich Daten. Eine unendliche Mengen von Daten kann nicht gespeichert werden, weswegen die eintreffenden Daten als flüchtig behandelt werden müssen. Daraus und durch die oft hohe Datenfrequenz in der Praxis resultiert die Herausforderung, Daten nahezu in Echtzeit verarbeiten zu müssen. [14, Kapitel 1].

Aus den eintreffenden Strömen von Datensätzen lassen sich, mit Kenntnis über die Interpretationsweise, Ereignisse bilden. Die Definition eines Ereignisses, auch Event genannt, variiert je nach Anwendungsbereich. Im Kontext von Multiagentensystemen wird unter anderem folgende Definition verwendet [15, S. 138]:

> [Ein] Event [ist definiert] als Objekt zur Repräsentierung, Kodierung oder Aufzeichnung in Computersystemen.

Diese Ereignisse formen selbst einen Ereignisstrom. Auf diesem Strom operieren CEP Engines (Siehe 2.2).

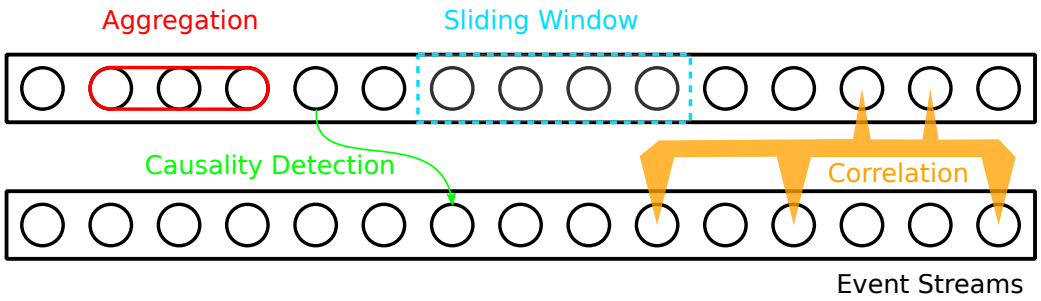

Abbildung 2.2.: Veranschaulichung der Begrifflichkeiten von CEP

Eine Basisfunktionalität der CEP Engine ist die Aggregation von Ereignissen. Bei eintreffenden primitiven, also unbearbeiteten Ereignissen ist nicht immer jedes Ereignis relevant, sondern nur ein gefilterter Ausschnitt. So interessiert zum Beispiel bei Temperaturdaten oft nur der Minimal- und Maximalwert sowie der Durchschnitt innerhalb einer gewissen Zeit. Auch nutzbar ist in diesem Fall die Tendenz der Temperaturverlaufs. Aggregationsfunktionen wie diese werden bei vielen CEP Engines als Grundfunktionen bereitgestellt. [14, Kapitel 3.1]

Während einzelne Ereignisströme für einfache Anwendungsfälle ausreichen, ist die logische Verknüpfung verschiedener Ströme für das Erkennen von komplexen Ereignissen nötig. Dieser Vorgang wird als Korrelation bezeichnet. [14, Kapitel 1]

Da aufgrund von physikalischen Beschränkungen nicht alle Daten eines Ereignisstroms im Speicher gehalten werden können, müssen Ereignisse verworfen werden, sobald sie nicht mehr benötigt werden. Hier werden sogenannte Sliding Windows verwendet, um die Anzahl vorgehaltener Ergebnisse endlich zu modellieren. Die Beschränkung basiert entweder auf der Anzahl der gehaltenen Ereignisse (z.B. die letzten 20 Ereignisse des Ereignisstroms) oder einem Zeitfenster (z.B. Ereignisse der letzten zwei Minuten). Außerhalb dieses Bereiches werden alle Ereignisse verworfen. Die CEP Engine überwacht dafür die Anforderungen aller Anfragen und verwirft Ereignisse, wenn sie von keiner Anfrage mehr benötigt werden. [14, Kapitel 3.1]

In manchen Anwendungsfällen ist die Erkennung von Kausalitäten nötig. So muss erkannt werden, dass ein Ereignis in Folge eines oder mehrerer anderer Ereignisse aus potentiell anderen Datenströmen eingetreten ist. So führt das Abschalten einer Lampe zu geringerer Helligkeit in einem Zimmer. [16, Kapitel 3.1]

Entkopplungsfähigkeit

Anfragen an CEP Engines werden über Event Processing Languages (EPLs) gestellt, Sprachen, die speziell auf die Beschreibung der Verarbeitungsweise von Ereignissen ausgelegt sind. Die Sprache lässt zu, dass Anfragende die Quellen von Informationen, die Struktur der eintreffenden Ereignisse und Struktur der ausgehenden Ereignisse angeben. Für die Hauptaufgabe der Sprache, die Definition der Vorgehensweise für die Behandlung der Ereignisse, werden je nach CEP Engine unterschiedliche Konzepte gewählt. Oft werden diese Sprachen an die Structured Query Language (SQL) angelehnt. Die Beschreibung kann komplett in der EPL erfolgen, je nach Implementierung ist es auch möglich, Funktionen innerhalb der Laufzeitumgebung der CEP Engine auszuführen, um aufwändige Anfragen zu beschleunigen oder zu vereinfachen. Durch die Beschreibung der Verarbeitungslogik des CEP als EPL werden plattformspezifische Implementierungen reduziert. Die Wiederverwertbarkeit von Programmcode steigt und ist auch für verschiedene Programmsprachen gegeben. [17]

Die Ereignisgenerierung kann an eine CEP Engine auf einem separaten und zentralen System übertragen werden, das über genügend Hardwareressourcen verfügt, um komplexere Berechnungen durchzuführen. Endabnehmer der Daten übermitteln mittels der EPL benötigte Berechnungen und können somit über schwächere Hardware verfügen. Sollte durch eine Steigerung der Anzahl der Agenten ein erhöhter Aufwand für die Bereitstellung dieser Berechnungen entstehen, so ist nur die Anpassung des Systems der CEP Engine nötig, die Skalierbarkeit wird vereinfacht. Oft entstehen in umfangreicheren Systemen redundante Berechnungen, die durch eine zentralisierte CEP Engine vermieden werden können. Eine weitere Möglichkeit ist die statische Definition der Regeln durch einen Administrator ohne Einfluss der Abnehmer. [17]

Einordnung

Während bei Datenbanksystemen, z.B. mit SQL, Anfragen üblicherweise einmalig und bedarfsweise abgesetzt werden, sind bei CEP Engines langlebige Abfragen üblich. Eine gestellte Anfrage wird laufend auf dem eintreffenden Datenstrom ausgeführt, der Anfragende erhält kontinuierlich Resultate. Eine Anfrage kann in diesem Fall auch als Verhaltensregel der CEP Engine für eintreffende Ereignisse gedeutet werden. [18, Kapitel 4]

Nutzen

CEP erhöht die Wiederverwendbarkeit von erstellter Logik, da diese problemlos in verschiedenen Programmiersprachen eingesetzt werden kann. Die Geschäftslogik trennt sich von der Programmlogik, der Entwickler kann sich auf den Entwurf der Logik konzentrieren und muss nach der Einarbeitung in die EPL keine zusätzliche Sprache erlernen oder nutzen. Die Bedienung vereinfacht sich somit. Zudem können aufgrund der Trennung Tests einfacher und präziser gestaltet werden. Anfragen können isoliert verifiziert werden. Verschiedene, sonst aufwändige Grundprobleme, wie das Erkennen von fehlenden Ereignissen können durch Zuhilfenahme von Grundfunktionalitäten der CEP Engine gelöst werden.

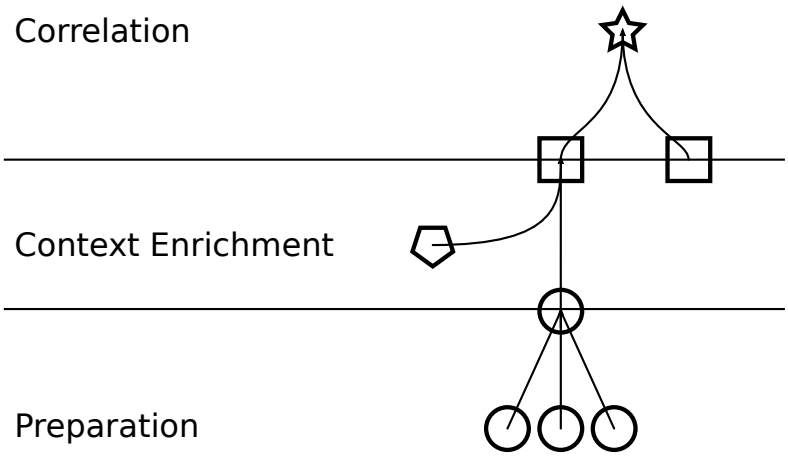

Abbildung 2.3.: Veranschaulichung der Verarbeitung von Ereignissen in CEP

Ereignisse werden durch die CEP Engine in mehreren Schritten angereichert (Siehe Abbildung 2.3). Initial werden primitive Ereignisse z.B. mit der Anwendung von Aggregation für den weiteren Verlauf vorbereitet. Anschließend werden Ereignisse mit Kontext versehen. So können bsw. Sensordaten mit Kenntnis von Kontextinformationen interpretiert werden, um Schlüsse auf die gemessene Umgebung zu ziehen. Eine gemessene kurzzeitige Verdunkelung eines Helligkeitssensors ist uninteressant, wenn die Raumhelligkeit gemessen wird, jedoch kritisch wenn es sich um eine Lichtschranke handelt. Darauffolgend wird ein Ereignis mit anderen korreliert, um sich die Stärken von CEP, wie z.B. die Erkennung von Mustern, zunutze zu machen. Ereignisse jedes Zwischenschrittes können dabei für die Erstellung anderer Ereignisse verwendet werden. Dadurch werden Redundanzen reduziert und das Regelwerk klein gehalten.

Rapid Prototyping vereinfacht sich durch die Nutzung von CEP. Anfragen können schnell und ohne gesonderte Berücksichtigung der Programmarchitektur umgestellt, angepasst und erweitert werden.

Anwendungsbeispiel

Als Beispiel gelte ein aktives Drainagesystem in einem größeren Haus. An allen vier Ecken des Hauses befindet sich eine Pumpe, die Bodenwasser zu einer Versickerungsgrube abpumpt. Jede Pumpe liefert dem CEP-System ihre aktuelle Betriebslast und den Wasserdurchfluss. Ein Niederschlagssensor überwacht zudem die Wetterbedingungen. Während die Informationen

der Datenströme geringen Nutzen haben (Ist eine Pumpe eingeschaltet, fließt Wasser durch die Pumpe?), können durch die Korrelation der einzelnen Ströme komplexe Ereignisse abgelesen werden. So kann erkannt werden, ob eine Pumpe nicht funktionstüchtig ist, eine andere Wasserbelastung als die durch Regen auftritt oder ein Rohr potentiell verstopft ist. [14, Kapitel 1][15, Kapitel 6.3.1]

2.5.2. Vergleich mit Bestandssystem

Um die Komplexitätsreduktion von CEP im Gegensatz zur Eigenimplementierung der Ereignisgenerierung zu zeigen, sei folgendes Beispiel gegeben. Ein Nutzer erstellt einen Ereignisstrom, der in unregelmäßigem Abstand Ereignisse vom Typ EVENT liefert. Jedes dieser Ereignisse hat das numerische Attribut VALUE. Der Nutzer möchte den Durchschnittswert dieses Attributs der letzten 60 s, also ein Sliding Window, erhalten.

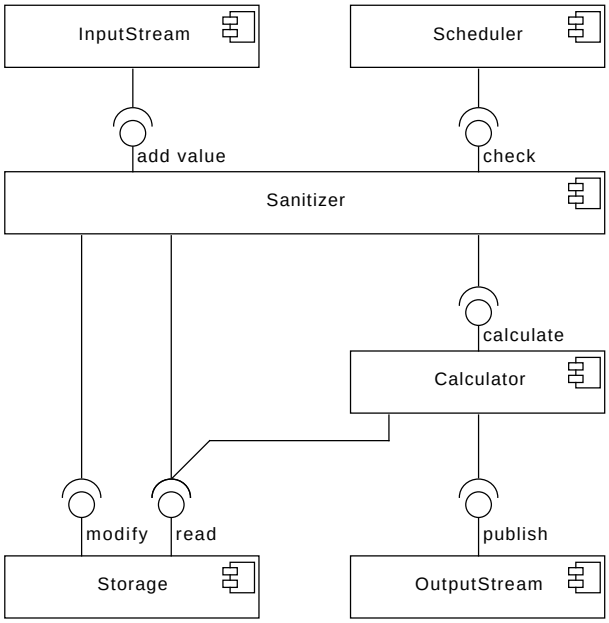

Abbildung 2.4.: Komponentendiagramm eines beispielhaften Agenten für die dedizierte Berechnung eines Durchschnitts

Wird diese Berechnung als eigenständiger Agent realisiert, so ist eine komplette Publish Subscribe Architektur abzubilden (Siehe Abbildung 2.4). Jedes eintreffende Ereignis wird über einen Eingabestrom empfangen und muss in einem Speicher gepuffert werden. Ein Sanitizer hat sicherzustellen, dass diese Ereignisse entfernt werden, sobald sie außerhalb des Zeitfensters

liegen und zur Berechnung nicht mehr nötig sind. Dazu müssen Ergebnisse effizient sortiert werden (Ereignisse treffen nicht zwangsläufig in korrekter Reihenfolge ein). Es ist zudem ein Scheduler nötig, der diesen Aufräumvorgang zum richtigen Zeitpunkt anstößt. Bei jedem Hinzufügen oder Entfernen eines Ereignisses ist der Durchschnitt neu zu berechnen, wofür alle vorgehaltenen Werte eingelesen werden müssen. Das Ergebnis ist an den Ausgabekanal zu propagieren.

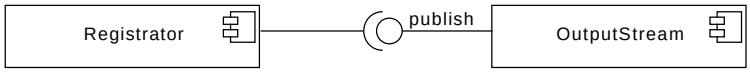

Abbildung 2.5.: Komponentendiagramm eines beispielhaften Agenten für die Berechnung eines Durchschnitts mittels CEP Engine

Wenn dieses Beispiel mit einer CEP Engine wie *Esper* umgesetzt werden soll, muss vorab die Anfrage für sie entworfen werden (Siehe Listing 2.1). In der Implementierung sind zwei Komponenten nötig: Ein Ausgabekanal, über den die Anfrage an die CEP Engine abgesetzt wird und ein Dispatcher welcher die Anfrage in eine Nachricht verpackt und über den Kanal versendet (Siehe Abbildung 2.5).

```
1 select avg(value) as averageValue from Event.win:time(60 sec)
```

Listing 2.1: Anfrage an eine CEP Engine zur Berechnung des Durchschnitts eines Wertes

Dieses Beispiel ist bewusst einfach gehalten, um anschaulich zu bleiben. Komplexere Beispiele, wie die Aggregation von Ereignissen, das Erkennen des Ausbleibens von Ereignissen und Rückspeisung von Ereignissen können beim Javabeispiel schnell Komplexität und Arbeitsaufwand erhöhen. Es wären zusätzliche Eingabekanäle nötig, was das Kombinieren von Ereignissen verschiedener Ströme erfordert und ein komplizierteres Mapping voraussetzt. Das Anwendungsbeispiel mit *Esper* würde sich in seiner Komplexität nur minimal ändern. Im einfachsten Fall muss nur die Anfrage verändert werden. In komplizierteren Situationen werden anstatt einer Anfrage mehrere registriert.

2.6. Technologieauswahl

Es existieren bereits mehrere CEP Engines, auf welche für dieses Projekt zugegriffen werden kann. Da mehrere Kandidaten verfügbar sind, wird ein Vergleich angestellt. In diesem werden die Produkte *Axibase TSD*, *Drools Fusion*, *Esper*, *MS StreamInsight*, *openPDC*, *SQL-Stream* und *WSO2 CEP* einbezogen.

Kriterien

Folgende Kriterien wurden für den Vergleich ermittelt:

- **Lizenzmodell**: Es ist ein Produkt mit Open Source Lizenz zu finden. Dies unterstützt eine freie Weiterentwicklung des Projekts und fördert den wissenschaftlichen Austausch. Zudem ermöglichen freie Produkte das Entstehen von größeren Communities. Diese produzieren oft hilfreiche Dokumentation in Form von Forenbeiträgen und Wikiseiten, was die Entwicklung beschleunigt.
- **Ereignisformat**: Es soll verglichen werden, welche Formatierung der Ereignisse die CEP Engine unterstützt. Die CSTI Middleware liefert im Normalfall Ereignisse als Javaobjekte an den Agenten. Für die Übertragung der Ereignisse wird JSON eingesetzt. Mit geringem Aufwand könnten diese Daten ebenfalls genutzt werden.
- **Programmiersprache**: Die Programmiersprache der Implementierung der CEP Engine ist ebenfalls von Relevanz. Sollte in Folgeprojekten eine Anpassung der Engine erforderlich sein, so sollte diese Sprache für die Nutzer des Labors bekannt oder zumindest einfach zu erlernen sein. Des Weiteren muss die CEP Engine auf einer Plattform des CSTI betrieben werden. Ein Großteil dieser Plattformen läuft aus verschiedenen Gründen mit dem Betriebssystem Linux. Es ist also eine Sprache zu bevorzugen, deren Laufzeitumgebung unter Linux lauffähig ist.
- **Dokumentation**: Für eine schnelle Einarbeitungszeit ist die Dokumentation der EPL kritisch. Deswegen sollte diese den Einstieg in CEP und das Herangehen an die Erstellung von Anfragen beinhalten. Zudem sollen mit Zuhilfenahme der Dokumentation komplexere Anfragen möglich werden. Eine komplette Dokumentation der CEP Engine sollte bereits vorhanden sein, da das Verfassen dieser außerhalb des Arbeitsrahmens läge.
- **Integrationsfähigkeit**: Die CEP Engine muss in die CSTI Middleware integriert werden. Dies erfordert eine enge Anbindung, die Erstellung eines eigenen Adapters für die Ein- und Ausgabe von Ereignissen sollte möglich sein. Dies muss von der API unterstützt werden.
- **Laufzeitregeln**: Es muss möglich sein, Anfragen zur Laufzeit zu konfigurieren. In einem dynamischen Smart Environment ist davon auszugehen, dass auch nach längerer Betriebszeit neue Agenten dem System beitreten und neue Anfragen an die CEP Engine stellen. Diese Anfragen können sich auch je nach Nutzung des Smart Environments

schnell und regelmäßig ändern. Ein Neustart der CEP Engine bei jeder neuen Anfrage ist dabei nicht praktikabel. Regeln müssen während des Programmbetriebs hinzugefügt und gelöscht werden können.

Vergleich

Die Resultate des Vergleichs sind folgende:

- **Axibase TSD**: *Axibase TSD* ist unter Apache Lizenz veröffentlicht und ermöglicht das Verändern von Regeln zur Laufzeit, Ereignisse können als einfacher Text, CSV oder JSON beschrieben werden was vorteilhaft für die Integration wäre. Die CEP Engine ist darauf ausgelegt über HTTP, TCP oder UDP angesprochen zu werden. Es sind Clientbibliotheken für Python, Java, Ruby, PHP und Go verfügbar. Leider umfasst die Dokumentation nur die Verwendung der API, die Verwendung der EPL wird nicht weiter beschrieben. Ebenso gibt es keine Dokumentation zu der Erweiterung der CEP Engine mit einem Netzwerkadapter.
- **Drools Fusion**: *Drools Fusion* ist ebenfalls unter Apache Lizenz veröffentlicht und verfügt über eine umfangreiche Dokumentation. Es gibt keine vorgegebenen Möglichkeiten die CEP Engine an Netzwerksysteme anzubinden, die einfache Erstellung eigener Adapter ist jedoch vorgesehen. Die CEP Engine ist in Java programmiert. Leider sind Änderungen des Regelwerks zur Laufzeit nur schwer möglich [19].
- **Esper CEP**: *Esper CEP* ist unter Gnu Public Licence veröffentlicht, es gibt zudem die Möglichkeit eine proprietäre Variante mit vergrößertem Funktionsumfang kommerziell zu erstehen. Änderungen des Regelwerks zur Laufzeit werden unterstützt, Ereignisse können als Javaobjekte, XML, CSV oder eine Verschachtelung von Map-/Arraykombinationen formatiert werden. Die CEP Engine ist in Java programmiert und verfügt über integrierte Adapter an TCP, HTTP und Advanced Message Queuing Protocol (AMQP). Das Erstellen von Adaptern ist durch eine umfangreiche und gut dokumentierte Schnittstelle einfach möglich. Es wird zur Beschreibung der Logik eine eigene EPL verwendet. [20]
- **Microsoft StreamInsight**: Dieses Produkt ist ausschließlich kommerziell erhältlich. Es ist in der Lizenz des SQL Servers von Microsoft enthalten. Dabei gibt es verschiedene Preiskategorien, welche die Leistungsfähigkeit und Skalierbarkeit bestimmen. Der Preis dieser Lizenz liegt mit 6000 $ jährlich weit außerhalb des finanziellen Rahmens. Die öffentliche Dokumentation ist spärlich. Die CEP Engine ist in C# programmiert und Closed Source. Es existieren Clientbibliotheken für .NET und C#. Dynamische Einstellungen sind möglich, Ereignisse können als Text, MIME und MHTML formatiert werden. Die

Verwendung dieses Produkts wäre unvorteilhaft, da erforderliche Leistungen vor dem Kauf exakt bestimmt werden müssten, um die korrekte Lizenz auswählen zu können. Zum anderen ist es nicht möglich, die CEP Engine mit einem Adapter zu erweitern. Sie müsste als eigenständiges Programm existieren und über zusätzliche Protokolle mit einem Agenten kommunizieren, der die Anbindung an die CSTI Middleware übernimmt.

- **OpenPDC**: *OpenPDC* wird unter MIT Lizenz vertrieben und verfügt nur über eine minimale öffentliche Dokumentation. Die Konfiguration des Regelwerks scheint nur händisch über eine Weboberfläche möglich zu sein. Die CEP Engine ist in C# programmiert, sie kann über IEEE C37.118, IEEE 1344, BPA PDCstream, FNET, SEL Fast Message und Macrodyne angebunden werden. Welche Formate Ereignisse annehmen können, ist nicht dokumentiert. Aufgrund der unzureichenden Dokumentation ist *OpenPDC* für den Anwendungsfall weniger geeignet.

- **SQL-Stream**: Dieses Produkt wird kommerziell und als Freeware angeboten. Die Freewareversion erlaubt lediglich die Verarbeitung von 1 GB Datenvolumen, was diese Variante ausschließt. Der Preis der kommerziellen Variante ist stark von der verwendeten Hardware abhängig. So würde die Verwendung eines einzelnen CPU-Kern einmalig 15 000 $ und 5000 $ jährlich kosten. Die Konfigurationsmöglichkeiten des Regelwerks sind nicht öffentlich dokumentiert, Ereignisse können als Rohtext, JSON oder XML zugeführt werden. Die Serverkomponente wird als Closed Source ausgeliefert und ist über TCP und UDP ansteuerbar.

- **WSO2 CEP**: *WSO2 CEP* wird in einer freien Variante unter Apache Lizenz und kommerziell vertrieben. Die Änderung des Regelwerks zur Laufzeit ist möglich, Ereignisse können als XML, JSON, Rohtext oder als verschachtelte Map/Array-Kombination formatiert werden. Die CEP Engine ist in Java programmiert und verfügt über integrierte Adapter für HTTP, JMS, SOAP, REST, Email, Apache Thrift, Websockets und MQTT. Die Dokumentation ist umfangreich.

Ergebnis

Die Ergebnisse des Vergleichs sind in Tabelle 2.1 zusammengefasst. *Drools Fusion* sowie *openPDC* ermöglichen keine Regeländerungen zur Laufzeit und sind daher nicht geeignet. *Microsoft StreamInsight* und *SQL-Stream* liegen außerhalb des preislichen Rahmens und sind nicht quelloffen, weswegen sie sich nicht für den Anwendungsbereich eignen. *Axibase TSD* ist nicht ausreichend dokumentiert und würde daher die Umsetzung umständlich machen.

	Axibase TSD	Drools Fusion	Esper CEP	MS Stream-Insight	open-PDC	SQL-Stream	WSO2 CEP
Laufzeitregeln	+	-	+	+	-	?	+
Lizenzmodell	+	+	+	-	+	-	+
Ereignisformate	+	-	+	-	?	+	+
Programmsprache	+	+	+	-	-	?	+
Anbindbarkeit	-	-	+	-	-	-	+
Dokumentation	-	+	+	-	-	-	+

Tabelle 2.1.: Vergleichsmatrix für die Auswahl der CEP Engine

Esper CEP und *WSO2 CEP* qualifizieren sich in allen Punkten. Beide haben einen guten Funktionsumfang, umfangreiche Dokumentation sowie einfache Anwendbarkeit. Die Auswahl zwischen diesen wird im Design festgelegt.

2.7. Integration

Die CEP Engine ist in ein bestehendes System zu integrieren. Dies kann auf mehrere Arten geschehen, welche jeweils Vor- und Nachteile mit sich bringen.

2.7.1. Integrationsweise

Die CEP Engine könnte in einen vorhandenen Agenten integriert werden, der sowohl seine eigentliche Funktion als auch die einer CEP Engine übernehmen würde. Dadurch erübrigt sich die Entwicklung eines neuen Programms für das Smart Environment. Dieser Ansatz würde jedoch aufgrund der Multifunktionalität bei Änderung eines Programmteils zu einer verminderten Verfügbarkeit führen, was diese Herangehensweise ausschließt.

Eine weitere Möglichkeit wäre, die CEP Engine in der Laufzeitumgebung der CSTI Middleware selbst unterzubringen. Die Engine wäre dadurch ein Teil der Middlewareinstanz und

wäre somit sinnvollerweise, da sie die zentrale Stelle zur Datenverarbeitung werden soll, im Kern des Systems angesiedelt. Da dies jedoch zu einer künstlichen Kopplung zweier Systemkomponenten führen und die Wartung und Erweiterung erschweren würde, ist dieser Ansatz ebenfalls zu verwerfen.

Die dritte Option ist, die Komponente als eigenständigen Agenten zu entwerfen. Die CEP Engine kann so unabhängig von anderen Agenten und der CSTI Middleware betreut und erweitert werden, ohne andere Funktionalitäten zu beeinträchtigen. Dies führt zu einer losen Kopplung und entspricht dem Prinzip eines Multiagentenframeworks und somit dem der CSTI Middleware.

2.7.2. Systemkomponenten

In dem zu betrachtenden System sind also vier Komponenten angesiedelt. Siehe dazu Abbildung 2.6.

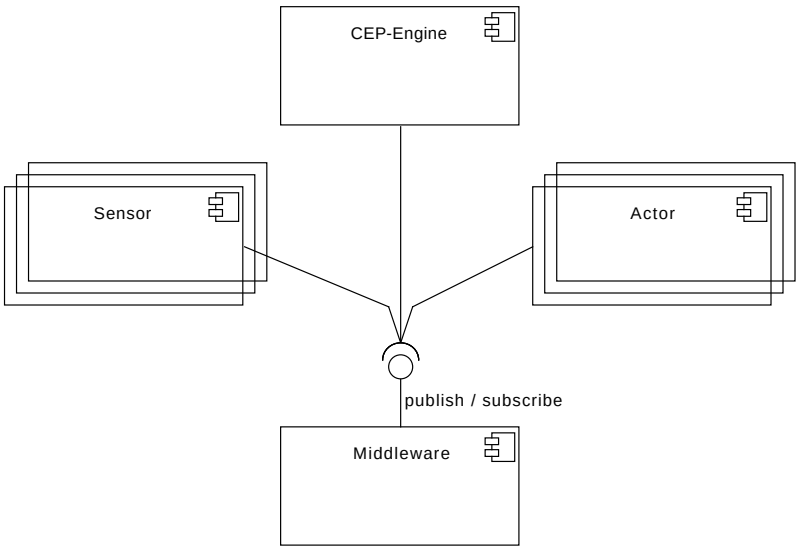

Abbildung 2.6.: Systemkomponenten

- **Middleware**: Die CSTI Middleware übernimmt im System die Rolle des Message Brokers und bietet ihre Dienste mittels Publish Subscribe an. Sämtliche Kommunikation im System läuft ausschließlich über diese Komponente ab.

- **Sensor**: Im System befinden sich mehrere Sensoren, welche Informationen und Ereignisse in das System einführen. Diese werden auf von ihnen gewählten Kanälen/Gruppen über die CSTI Middleware dem System bekannt gemacht.

- **Actor**: Die Aktoren innerhalb des Systems verwenden die durch die CSTI Middleware weitergegebenen Informationen, um außerhalb des Systems wirken zu können. Bevor ein Aktor Informationen nutzen kann, muss er bei der CEP Engine eine oder mehrere Anfragen stellen, die die erforderlichen Ereignisse in einer definierten Struktur generieren.

- **CEP Engine**: Die CEP Engine erhält von Aktoren Anfragen. Diese Anfragen enthalten eine Beschreibung, welche aussagt, aus welchen Kanälen Informationen verwendet werden sollen, wie diese zu verarbeiten sind und in welchen Kanal generierte Ereignisse zu emittieren sind. Die CEP Engine übernimmt also die Rolle eines Kontextinterpreters.

2.8. Verwandte Arbeiten

Zwei Arbeiten in direkter Nähe zu diesem Projekt werden im Folgenden dargestellt. Beide beschäftigen sich mit der Integration von CEP in eine neue Umgebung. Während die erste Arbeit eine Integration in einen Wohnraum vornimmt, beschäftigt sich die zweite Arbeit mit der Integration in ein Krankenhaus.

2.8.1. Complex Event Processing in altersgerechten Smart Homes

Augusto und Nugent befassen sich in “The Use of Temporal Reasoning and Management of Complex Events in Smart Homes” mit der Anwendung von CEP in altersgerechten Smart Homes. Im konkreten Anwendungsfall handelt es sich um ein Seniorenwohnheim mit mehreren separaten Apartments, die mit einer Vielzahl unterschiedlicher Sensoren ausgestattet sind. Sämtliche Sensorik wird von einer Überwachungszentrale verarbeitet. Das Ziel ist, verschiedene potenziell gefährliche Situationen (z.B.: Herd in der Küche ist eingeschaltet, das Waschbecken im Badezimmer ist seit längerem in Benutzung) zu erkennen, zu analysieren und letztlich auf sie zu reagieren. Dazu wird CEP angewendet.

Die Autoren kommen zu dem Schluss, dass die Kombination von Smart Home und CEP eine solide Lösung zur Unterstützung von Senioren ist und betreutes Wohnen signifikant vereinfacht.[21]

2.8.2. Complex Event Processing in Krankenhäusern

In Krankenhäusern entsteht durch eine Vielzahl Sensoren eine große Menge an zu verarbeitenden Daten. Patientenmonitore, Logistiksysteme und Positionsmelder liefern verschiedenste Daten, die erst durch CEP zielführend genutzt werden können. Yao, Chu und Zang beschreiben in “Leveraging complex event processing for smart hospitals using RFID” einen Ansatz zur Anwendung von CEP in diesem Umfeld in Kombination mit Radio Frequency Identification (RFID). So können der Standort von Patienten sowie ihr Zustand zentral erfasst und die Verteilung von Medikamenten zwischen verschiedenen Ausgabestellen vereinfacht werden. Sie beschäftigen sich zudem mit den Besonderheiten, die die Integration eines RFID-Systems in diesem Kontext aufkommen lässt.

Die Schlussfolgerung ist, dass die Konfiguration des Regelwerks für CEP kompliziert und langwierig ist, da viel medizinisches Fachwissen einzubeziehen ist und zu erstellende Regeln einen hohen Unschärfeanteil beinhalten. Abseits dieser Punkte stellt CEP eine nutzbare und effiziente Möglichkeit zur Verbesserung der Patientenbetreuung und Sicherheit in einem Krankenhausumfeld dar. [22]

2.9. Abgrenzung

Das Projekt soll nicht die Programmierung der Logik kapseln. Entwickler müssen die Beschreibung der gewünschten Resultate selbst in der EPL der CEP Engine realisieren. Die Entwicklung einer clientseitigen Bibliothek zur Generierung der EPL kann in Folgearbeiten durchgeführt werden.

Die Administration der CEP Engine soll ohne eine Benutzeroberfläche durchgeführt werden. Ziel ist es die Implementierung weitestgehend wartungsfrei zu halten. Sollte dennoch Interaktion nötig sein, so erfolgt diese auf textuellem Wege.

Es wird davon ausgegangen, dass die CSTI Middleware sicherstellt, dass nur berechtigte Agenten Zugang zum System erhalten. Diesen Agenten wird uneingeschränkt vertraut, eine Beschränkung von Zugriff oder Ressourcen wird nicht vorgenommen.

2.10. Anwendungsszenarien

Anhand von drei Szenarien soll gezeigt werden, dass CEP die gegebene Problemstellung angeht und einen signifikanten Vorteil gegenüber der jetzigen Herangehensweise bietet.

2.10.1. Fernsehersteuerung

Es soll der Betriebszustand eines Fernsehers im Smart Environment gesteuert werden. Der Fernseher soll eingeschaltet werden, wenn der Benutzer auf dem Sofa sitzt, was durch Standorterfassung des Nutzers festgestellt werden soll. Zudem soll erkannt werden, ob der Benutzer ungestört ist. Dies ist der Fall, wenn kein Gast die Türklingel betätigt und vor der Tür wartet, kein Anruf über das Telefon eingeht und der Nutzer auf dem hauseigenen Tablet kein Video anschaut. Ist er beschäftigt, so ist der Fernseher stummzuschalten.

2.10.2. Helligkeitsregelung

Ein Raum des Smart Environments verfügt über eine dimmbare Lampe, ein Rollo sowie einen Helligkeitssensor. Anhand der Aktivität des Nutzers soll ermittelt werden, welche Helligkeit am vorteilhaftesten ist. Diese Helligkeit soll durch Steuerung der Lampe und des Rollos erreicht werden. Um höhere Helligkeiten zu erreichen, soll zuerst das Rollo hochgefahren werden. Ist dies vollständig geschehen, die gewünschte Helligkeit jedoch nicht erreicht, so soll die Lampe aufgedimmt werden bis der Sollzustand erreicht ist. Beim Abdunkeln wird zuerst die künstliche Beleuchtung heruntergeregelt, anschließend das Rollo gesenkt. Um Schwankungen bei der Helligkeitsmessung zu dämpfen, soll ein Durchschnittswert gebildet werden.

2.10.3. Nachrichtenauswahl

Der Benutzer verwendet das Tablet, um Nachrichten zu lesen. Es soll für ihn eine Vorfilterung übernommen werden, die Nachrichten für die aktuelle Lesestimmung aussucht. So sollen Nachrichten kürzer ausfallen, wenn der Benutzer nebenbei fernsieht, es werktags abends ist oder innerhalb der nächsten Stunde ein Termin ansteht. Nachrichten und Termine sollen dabei von einer SQL-Datenbank bezogen werden.

2.11. Fazit

Die Analyse legt nahe, dass sich die Verwendung von CEP in Smart Environments als eine sinnvolle Alternative zur bestehenden Lösung erweisen könnte.

Die Umsetzung einer CEP Engine-Integration mit den vorgegebenen Eigenschaften ist im Rahmen dieser Arbeit erreichbar.

Es ist mit einer Aufwandsersparnis bei der Entwicklung neuer Agenten zu rechnen. Stabilere, einfacher zu testende und von der Hardware entkoppelte Agentenlogik kann auf Kosten eines geringen Einarbeitungsaufwandes erreicht werden. Die Logik der Datenverarbeitung ist

unabhängig von der Programmiersprache des Agenten beschrieben. Der Wartungsaufwand kann somit durch Entkopplung und Vereinfachung des Programmcodes reduziert werden. Kleinere Agenten im Hinblick auf die Codelänge sind möglich.

Wie auch bei der bestehenden Lösung bleibt eine Einarbeitung in die EPL der CEP Engine Voraussetzung. Aufgrund des benötigten Verständnisses sollten nur Entwickler mit Programmiererfahrung den Dienst benutzen. Die Arbeit hat nicht zum Ziel, das Design von Geschäftslogik zu übernehmen - vielmehr soll die eigentliche Umsetzung dieser vereinfacht werden.

3. Design

Das Design beschäftigt sich mit der Planung der Umsetzung. Dazu wird zuerst ermittelt, über welche Software- und Hardwaresysteme das Labor verfügt, welche davon genutzt werden sollen und welche Entwicklungsstrategie zu verfolgen ist. Drauffolgend wird die Programmarchitektur ermittelt, Nachrichtenformate sowie API festgelegt und dargestellt, wie die Interaktion mit der CEP Engine abläuft. Zuletzt wird auf Besonderheiten des Konzepts eingegangen.

3.1. Entwicklungsplanung

Vor der eigentlichen Konzeption des Programms wird ermittelt, welche Elemente der Nutzungsumgebung und welche Entwicklungstools zu nutzen sind.

3.1.1. Nutzungsumgebung

Das CSTI verfügt als Smart Environment über Dienste, die für dieses Projekt verwendet werden können.

Middlewareinstanz

Eine permanent laufende Instanz der CSTI Middleware ist Bestandteil des CSTI. Über diese kommunizieren alle Agenten des Labors miteinander. Es besteht zudem die Möglichkeit, andere Labore wie das Living Place über diese Instanz zu erreichen. Agenten im Netzwerk erhalten über ein Broadcastprotokoll die Information, unter welchem Host und Port die Middleware erreichbar ist. Die Instanz läuft auf einer virtuellen Maschine im Serversystem des Labors, welche sich direkt in den Räumlichkeiten befindet.

Tracker

Die Umsetzung der Szenarien setzt voraus, dass die Position des Benutzers ermittelt werden kann. Dazu können die folgenden Trackingsysteme des Labors verwendet werden. Beide sind an die Plattform des CSTI angebunden, bereits bestehende Agenten speisen die Trackingdaten in das System ein.

- **ArtTrack**: ArtTrack ist ein auf visueller Erfassung basiertes System, welches seit 1999 von der Firma ART entwickelt wird. Dieses System erfasst Trackingobjekte innerhalb eines von Infrarotkameras ausgeleuchteten Bereichs millimetergenau und kann sowohl die Höhe des Objektes als auch dessen Ausrichtung bestimmen. Dieses System würde sich zur Positionsbestimmung eignen, jedoch ist dafür das Tragen eines gut erkennbaren Markers nötig, der von mehreren Kameras erfasst werden muss. Durch den Einsatz von Möbeln im Erfassungsbereich wird dazu eine erhöhte Anzahl an Kameras benötigt.
- **Ubisense**: Das CSTI verfügt über ein Trackingsystem der Firma Ubisense, welches funkbasiert über RFID arbeitet. Der Benutzer muss dazu einen Transponder bei sich tragen und kann dadurch zentimetergenau geortet werden. Zudem ist auch hier die Ausrichtung und Höhe des Transponders bestimmbar. Das System benötigt an den Ecken des Erfassungsbereichs je eine Transceivereinheit und kommuniziert über diese mit dem Transponder. Der Transponder muss nicht offen getragen werden.

3.1.2. Entwicklungstools

Versionskontrolle

Zur Verwaltung des Quellcodes wird das Versionskontrollsystem Git verwendet. Dieses ermöglicht die Versionierung, Synchronisation und Zusammenführung des Quellcodes sowie die einfachere Erfassung und Dokumentation von Änderungen.

Für jedes Projekt ist dabei ein Repository anzulegen, welches alle vom Projekt direkt benötigten Dateien enthält. Dieses unterstützt die parallele Entwicklung von mehreren Entwicklungszweigen in sogenannten Branches. Dadurch können Funktionen einzeln dem Programm hinzugefügt und getestet werden, ohne dass diese Arbeit mit anderen Aufgaben in Konflikt kommt. Es könnte ein Entwicklungszweig für die stabile im Labor eingesetzte Version verwendet werden und ein Zweig für aktuelle Entwicklungen. Ist eine Änderung des Programms in einem eigenen Zweig umgesetzt und getestet worden, so kann sie ohne größeren Aufwand in den Entwicklungszweig integriert werden. Ist das Labor bereit für eine Aktualisierung, so wird der Entwicklungszweig mit seinen Änderungen in den Hauptzweig integriert. Durch dieses Flussmodell von Änderungen kann der Entwicklungsprozess qualitativ hochwertig strukturiert werden.

Buildmanager

Für das Projekt wird der Buildmanager Gradle verwendet. Die Aufgaben eines Buildmanagers sind das Beziehen von Abhängigkeiten, das Bauen des Projekts sowie das Testen und Hochladen

der Resultate. Dazu ist die Konfiguration eines Buildscripts nötig. Das Script dieses Projektes wurde dahingehend erweitert, dass mittels des Versionskontrollsystems die aktuelle Version des Projektes ermittelt und anschließend für das gebaute Resultat übernommen wird. Dadurch entfällt die manuelle Spezifikation einer Versionsnummer. [23]

Artefaktrepository

Das Labor verfügt über ein Artefaktrepository auf Basis von Maven. In ihm werden alle Buildresultate des Labors gespeichert. So ist das mehrmalige Bauen eines Projekts unnötig und verschiedene Versionen können über längere Zeit eingelagert werden. [24]

Repositorymanager

Das eben genannte Repository wird bei der laboreigenen Instanz von Gitlab, einem Repositorymanager, hinterlegt. Über Gitlab ist es möglich interessierten Parteien Zugriff auf ein Wiki, einen Issuetracker und ein Buildsystem zu geben. [25]

In dem Wiki werden die Bedienungsanleitung sowie verschiedene Nutzungsbeispiele an die Benutzer ausgegeben. Dadurch ist die Dokumentation allen Entwicklern direkt zugänglich und kann bei Bedarf direkt bearbeitet werden.

Sollten sich im Laufe des Betriebs Fehler bemerkbar machen, so können diese über den Issuetracker festgehalten werden. So kann eine Fehlerbeschreibung, die verwendete Version und der Fortschritt der Behebung einfach eingesehen werden.

An Gitlab wurde ein Buildserver angebunden, der das automatisierte Bauen von Projekten ermöglicht. In einer Konfigurationsdatei des Repositories wird dem Buildsystem beschrieben, wie jedes Projekt zu bauen ist, und was mit den Resultaten zu tun ist. Das Bauen kann automatisch nach jedem Hochladen von Änderungen getrennt nach Entwicklungszweigen erfolgen.

Integrated Development Environment

Zur eigentlichen Umsetzung des Programms wird Eclipse verwendet. Eclipse ist eine Entwicklungsumgebung, welche durch zahlreiche Erweiterungen an die Bedürfnisse des Entwicklers angepasst werden kann und zudem über zahlreiche Funktionen wie Autovervollständigung und Formatierung des Quellcodes sowie Fehlererkennung verfügt. [26]

3.1.3. Testverfahren

Das zu entwickelnde Programm soll langfristig in den Laboren eingesetzt werden und Studenten eine stabile Plattform zum Entwickeln bieten. Aus diesem Grund sind umfangreiche Tests nötig, um die Qualität des Resultats verifizieren zu können. Test-driven development ist als Vorbild anzusehen.

Modultests

Jedes Modul wird einzeln auf korrekte Schnittstellenimplementierung getestet. Eine Testabdeckung von über 90 % soll erreicht werden. Diese erhöht die Wahrscheinlichkeit, dass Fehler während der Entwicklung und nicht nach Produktivitätsbeginn erkannt werden. Da das Programm in mehrere Module geteilt ist, kann somit sichergestellt werden, dass spätere Änderungen einfach möglich sind. So könnte die CSTI Middleware durch einen AMQP Clienten oder die CEP Engine durch eine andere Implementierung ersetzt werden.

Zur Ausführung der Modultests wird JUnit verwendet, ein bekanntes Framework zum Durchführen dieser Tests unter Java. Gradle verfügt über ein Plugin welches diese Tests automatisch nach jedem Build durchführt. So kann während der Entwicklung laufend überprüft werden, ob neue Fehler eingeführt wurden. [27]

Die Testabdeckung wird mit einem Plugin für Gradle erfasst. Dieses verwendet die Bibliothek JaCoCo, welche unter anderem auch in Plugins für Eclipse Verwendung findet. Das Plugin generiert einen Report in HTML, aus dem ersichtlich ist, welche Module welche Testabdeckungsrate aufweisen und welche Ausführungszweige im Quellcode nicht abgedeckt wurden. [28]

Funktionstest

Die in Abschnitt 2.10 definierten Szenarien sollen vor der Integration in das System auf Umsetzbarkeit überprüft werden. Dazu wird eine Testumgebung erstellt, welche mittels einer einfachen Beschreibung konfiguriert wird und ein einzelnes Szenario rein virtuell ohne Anbindung an die CSTI Middleware oder reale Sensoren simulieren kann. So können Designfehler rechtzeitig erkannt und behoben werden.

Integrationstests

Es soll ein automatischer Test entworfen werden, der die korrekte Anbindung an die CSTI Middleware prüft. Dieser soll beinhalten, dass sich das Programm an einer lokalen Instanz

der CSTI Middleware anmeldet. Das Testprogramm sendet über diese Instanz Anfragen und überprüft eintreffende Ergebnisse.

Praxistest

Nach der Integration in das Labor soll das zu entwickelnde Programm einem Praxistest unterzogen werden. Dazu werden die Szenarien, soweit innerhalb des Labors möglich, ausführlich nachgestellt und überprüft.

3.2. Programmarchitektur

Das Programm besteht aus drei Komponenten: Der Schnittstelle zur CSTI Middleware, der Schnittstelle zur CEP Engine und einer Zwischenkomponente, die sich um die allgemeine Verwaltung kümmert (Siehe Listing 3.1).

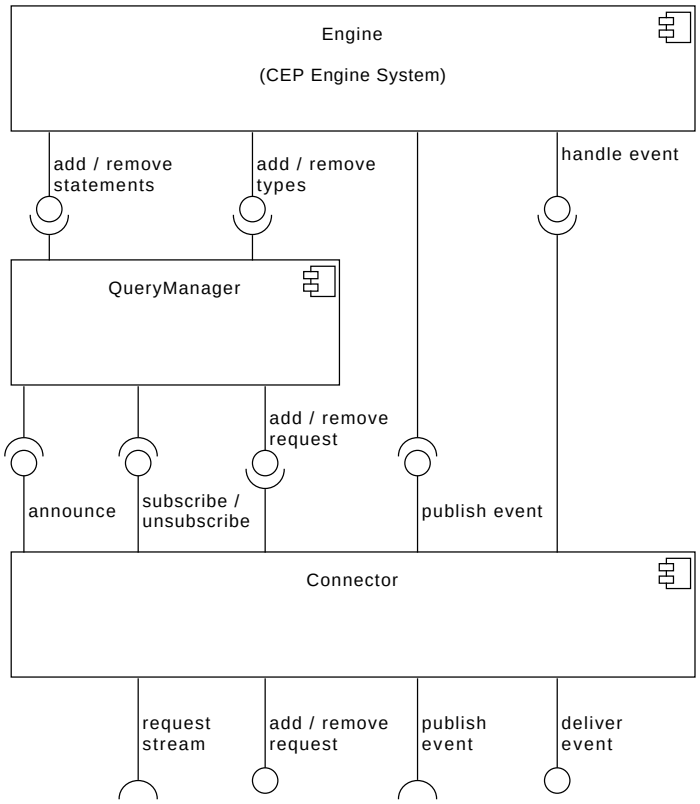

Abbildung 3.1.: Programmmodule

Connector

Die Kommunikation von Agenten mit der CEP Engine soll ausschließlich über Gruppen der CSTI Middleware durchgeführt werden. Dabei verhält sich die CEP Engine gegenüber der Middleware wie ein Agent. Eine Komponente des Programms realisiert dies, indem die Clientbibliothek der CSTI Middleware verwendet wird, um benötigte Gruppen zu abonnieren und deren Nachrichten programmintern weiterzuleiten sowie ausgehende Nachrichten zu veröffentlichen.

Abonniert werden müssen der Kanal für Anfragen sowie alle Ereigniskanäle deren Ereignisse verarbeitet werden sollen. Während Anfragen von der Bibliothek als Javaobjekte empfangen werden können, lässt sich dieses nicht für Ereignisse realisieren. Für die Konvertierung der eingehenden JSON Nachrichten muss ein Deserialisierer angegeben werden, dem die Objekttypen bekannt sein müssen. Da das Programm gegenüber Ereignistypen generisch gehalten werden muss, kann diese Angabe nicht erfolgen. Zur Lösung dieses Problems wird ein spezieller Deserialiserer geschrieben, der alle eintreffenden Ereignisse in eine Java-Map umwandelt. Diese enthält die Attribute der Nachricht als primitive Datentypen oder Listen und weitere Maps, in welchen eine Verschachtelung möglich ist. So erhaltene Ereignisse werden zur Verarbeitung an andere Komponenten weitergeleitet.

Sollen Announcements oder Rejections der CEP Engine veröffentlicht werden, so kann auch hier der generierte Serialisierer verwendet werden. Ausgehende Ereignisse werden von der CEP Engine ebenfalls als Map mit Elementen generiert. Ein spezieller Serialisierer setzt diese in JSON um.

QueryManager

Eingehende Anfragen werden von dieser Komponente verarbeitet. Jede Anfrage kann mehrere Eingangskanäle nutzen und muss einen Ausgabekanal angeben. Für alle Kanäle müssen Typendefinitionen aufgestellt werden.

Ein Eingabekanal kann von mehreren Anfragen verwendet werden, wobei der Ereignistyp immer gleich sein muss. Mehrere Anfragen können im gleichen Ausgabekanal veröffentlichen, sofern auch hier der Ereignistyp gleich ist. Dies wird vom QueryManager sichergestellt. Ist ein Typ neu, wird er bei der CEP Engine bekannt gemacht.

Anschließend kann die eigentliche Beschreibung der Anfrage (auch Statement genannt) bei der CEP Engine angemeldet werden. Ergibt sich ein Fehler, so generiert die Komponente eine Fehlermeldung und veröffentlicht diese.

Außerdem kümmert sich QueryManager um das Entfernen von nicht mehr benötigten Anfragen. Findet ein Timeout statt, so wird das Statement entfernt. Anschließend wird überprüft, welche Ereignistypen von keinen anderen Anfragen mehr benötigt werden. Diese werden deregistriert und anschließend allen Agenten bekannt gemacht, dass die Anfrage nicht mehr bedient wird.

Engine

Engine ist die Schnittstelle zur Implementierung der CEP Engine. Statements und Typoperationen werden hier für Esper umgewandelt und anschließend weitergereicht. Alle Eingangsereignisse werden von Connector direkt hier eingegeben und in die Runtime von Esper eingespeist. Für jeden Ausgabekanal muss ein Listener bei der Implementierung registriert werden. Dieser Listener erhält von der Implementierung generierte Ereignisse und veröffentlicht sie über Connector.

3.2.1. Andere Umsetzungsmöglichkeiten

Es besteht die Möglichkeit, die CEP Engine direkt als einen Aktor in Akka umzusetzen, um eine einfachere Integration in die CSTI Middleware zu realisieren. Dazu könnte ein bereits bestehendes Framework verwendet werden, welches diese Funktionalität zur Verfügung stellt [29]. Es wurde sich dagegen entschieden, da ein möglichst generischer Ansatz gewünscht ist - ein leichtgewichtiger Ansatz der *Akka*, nur innerhalb der Connector-Komponente verwendet, kommt dem entgegen.

3.2.2. Pluginfunktionen

Aufgrund der Universalität der EPL von *Esper* wird erwartet, dass ein Großteil der Anfragen vollständig in dieser umgesetzt werden können. Sollte ein Sonderfall jedoch so nicht realisierbar sein, so ist es möglich, Berechnungen in Anfragen an Javamethoden auszulagern. Dazu kann aus Anfragen jede sichtbare statische Methode innerhalb der Laufzeitumgebung aufgerufen werden. Dies schließt u.a. die mathematischen Hilfsfunktionen ein, wodurch es z.B. auf einfache Weise möglich ist, mit einer Anfrage den Sinus eines Wertes zu berechnen. Sollten selbst geschriebene Methoden aufgerufen werden, so können diese als Plugins in das Programm integriert werden. Dazu sind diese wie für Java üblich in JAR-Dateien bereitzustellen und dem Classpath des Programms hinzuzufügen. Alternativ kann diese JAR als Mavenartefakt in einem Repository bereitgestellt werden und als Abhängigkeit in der Gradlekonfiguration des Programms definiert

werden. Das Buildsystem kümmert sich dann eigenständig um das Beziehen und Einbinden des Plugins.

3.3. Systemanbindung

Folgend wird erörtert, wie der Kommunikationsablauf zwischen einem Agenten, der eine Anfrage an die CEP Engine stellen möchte, und dem zu entwickelnden Programm abläuft.

3.3.1. API

Es werden für den Kommunikationsablauf zwei Nachrichtentypen verwendet. Um Nachrichten dieses Typs zu erstellen, wird der API-Generator der CSTI Middleware verwendet.

Query

Ein Nachrichtentyp dient zur Beschreibung einer Anfrage. Die Signatur der Nachricht ist in Listing 3.1 veranschaulicht. Die Einträge bedeuten:

```
{
  "identificator": "lightSetting",
  "statement": "select case when brightness.currentValue < 50.0
    then true else false end as on
    from ‘Brightness__sensor1.std:lastevent()‘ as brightness",
  "inputs": {
    "Brightness__sensor1": { "currentValue": "java.lang.Float" }
  },
  "output": {
    "identificator": "LightSetting__light1",
    "definition": { "on": "java.lang.Boolean" },
    "feedback": false
  }
}
```

Listing 3.1: Beispielhafte Anfrage

- **Identificator**: Der eindeutige Identifikator der Anfrage. Anfragen mit gleichem Identifikator werden gleichgesetzt.
- **Statement**: Die eigentliche Anweisung an die CEP Engine als String beschreibt mittels einer EPL, wie das gewünschte Ereignis zu erstellen ist.

- **Eingabekanalidentifikator**: Identifikator der Ereignisströme, deren Ereignisse für die Anfrage benötigt werden. Die CSTI Middleware ermöglicht das Versenden von mehreren Nachrichtentypen über eine Gruppe, weswegen ein kombinierter Identifikator nötig ist. Er setzt sich aus dem Typ der Nachricht, zwei Unterstrichen und dem Namen der Gruppe, über welche die Ereignisse empfangen werden, zusammen.
- **Eingabetypdefinitionen**: Eine Formatbeschreibung des zu erwartenden Formats des über den Eingabekanal eintreffenden Ereignisses. Diese wird als Kombination von Strings sowie verschachtelten Containern dargestellt. Enthält ein Container als Wert einen String, so wird dieser als Klassenname des zu erwartenden Datentyps verwendet. Im Beispiel bedeutet dies, dass Ereignisse aus dem Strom BrightnessInfo__sensor1 einen Wert mit dem Namen currentValue beinhalten, der vom Typ float ist.
- **Ausgabekanalidentifikator**: Der Identifikator des Kanals, über den von dieser Anfrage generierte Ereignisse veröffentlicht werden. Dessen Struktur ist identisch mit denen der Eingabekanäle.
- **Ausgabetypdefinition**: Eine Map, die für den Ausgabekanal eine Typbeschreibung derselben Struktur wie die der Eingabetypdefinition enthält.
- **Ausgaberückkopplung**: Ist dieses Flag gesetzt, werden generierte Ereignisse nur innerhalb der CEP Engine verwendet und nicht in die CSTI Middleware exportiert.

Request

Ein Request dient dazu, die Bedienung einer Query von der CEP Engine zu fordern. Dazu enthält diese Nachricht die betreffende Anfrage.

Rejection

Ist eine Anfrage ungültig oder wird sie aus anderen Gründen abgelehnt, so erfolgt als Antwort eine Nachricht dieses Typs. Sie beinhaltet die ursprüngliche Anfrage sowie eine Beschreibung des Fehlers.

Acception

Ist die Anfrage gültig und wird sie von der CEP Engine angenommen, wird mit dieser Nachricht geantwortet. Sie enthält die betreffende Anfrage.

Suspension

Wird eine angenommene Nachricht nicht mehr bedient, so wird dies mit dieser Nachricht bekannt gemacht. Auch hier ist die Anfrage enthalten.

3.3.2. Anfrageablauf

Zur Verdeutlichung des Folgenden wird im Sequenzdiagramm 3.2 die direkte Kommunikation zwischen einem Aktor und der CEP Engine für die Registrierung und Deregistrierung von Anfragen dargestellt. Die Interaktion beginnt damit, dass ein Aktor eine Anfrage erstellt. Diese wird der CEP Engine übermittelt und von ihr überprüft. Ist die Anfrage gültig, teilt sie allen Gruppen mit, dass die Anfrage bedient wird. Eine Ablehnung im Falle einer ungültigen Anfrage wird ebenfalls mitgeteilt.

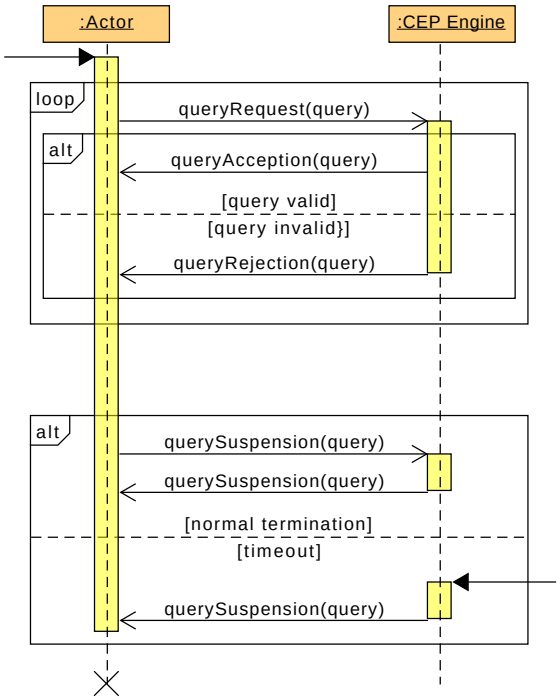

Abbildung 3.2.: Registrierung einer Anfrage

Garbagecollection

Da die CEP Engine darauf ausgelegt ist, über einen längeren Zeitraum betrieben zu werden, müssen nicht mehr benötigte Anfragen erkannt und entfernt werden.

Ein Agent kann von der CEP Engine fordern, dass eine Anfrage gelöscht wird. Dies sollte allerdings nur in Ausnahmefällen, wie beim Testen neuer Anfragen, durchgeführt werden, da andere Agenten auf diese Anfrage angewiesen sein könnten.

Anfragen werden nur für eine bestimmte Zeit, die sogenannte *Lease Time*, bedient und anschließend automatisch von der CEP Engine deregistriert. Dies kann von Aktoren verhindert werden, indem die gleiche Anfrage vor Ablauf der Frist, üblicherweise nach der Hälfte der *Lease Time*, erneut gestellt wird. Wird eine Anfrage entfernt, so wird dies allen Aktoren mitgeteilt.

Unterscheidung von Anfragen

Grundsätzlich sind zwei Anfragen gleich, wenn Statement sowie Ein- und Ausgabekanäle und deren Definitionen gleich sind. Da dies jedoch unsicher zu erkennen ist, gleiche Statements können unterschiedlich formuliert werden, wird jede Anfrage zusätzlich mit einem Namen versehen.

Die Definition des Nachrichtenformats eines Kanals muss bei allen Anfragen widerspruchsfrei sein. So ist es nicht möglich, bei einer Anfrage für ein Feld den Typ String zu deklarieren, während eine andere Anfrage diesen bereits als Integer festgelegt hat. Sollte ein Kanal beispielsweise Felder einer Nachricht nicht benötigen, so müssen diese nicht angegeben werden. Eine andere Anfrage darf diese Felder definieren.

Mehrere Anfragen dürfen für denselben Ausgabekanal Ereignisse generieren, wenn sich auch hier die Ereignisdefinition nicht widerspricht. Wieder können Felder in der Definition ausgespart werden.

Bereitstellung von Konstanten

Es soll vermieden werden Konstanten bzw. sogenannte *Magic Numbers* in Anfragen zu integrieren. Wenn beispielsweise mehrere Agenten Beleuchtungselemente steuern, so wäre es denkbar, dass diese eine gemeinsame Anfrage absetzen, welche die Agenten beim Unterschreiten eines Helligkeitswertes informiert. Möchte der Entwickler diesen Wert ändern, so müsste dies in allen Agenten erfolgen, was nicht praktikabel wäre.

Um dieses Problem zu lösen, sollten die Konstanten als Ereignisse dargestellt werden, welche von einer einzelnen Stelle generiert und für solche Berechnungen verwendet werden. Dies

könnte ein Agent sein, welcher über die Präferenzen des Benutzers und die Kenntnis der räumlichen Gegebenheiten verfügt.

Direkte Rückkopplung

Wie oben beschrieben, gibt es die Möglichkeit, Ereignisse nicht an die CSTI Middleware zu exportieren, sondern direkt in die CEP Engine zurückzuspeisen. Von der Engine generierte Ereignisse können im Standardfall nicht direkt von anderen Statements verwendet werden, da sie nicht automatisch wiedereingeführt werden. Es gibt dafür zwei Möglichkeiten.

Entweder abonniert der Manager der CEP Engine den exportierten Kanal und erhält somit exportierte Ereignisse von der CEP über den Umweg über eine Middlewaregruppe zurück. Dies hat den Vorteil, dass Agenten zusätzliche Ereignisse in dieser Gruppe veröffentlichen können und auch von außerhalb der Engine ersichtlich ist, welche Ereignisse versendet werden. Von Nachteil ist, dass die Middleware zusätzlich mit Ereignissen belastet wird und eine höhere Latenz verursacht.

Die andere Möglichkeit ist, bestimmte Ereignisse nur innerhalb der Engine zu veröffentlichen. Andere Statements erhalten somit nahezu sofort nach Generierung eines Ereignisses Zugriff auf dieses, die Übermittlung über das Netzwerk entfällt. Dies eignet sich gut für Zwischenergebnisse, Agenten erhalten diese Ereignisse jedoch nicht und können auch über diese Gruppe keine Ereignisse für die Engine veröffentlichen.

Um duplizierte Eingangsergebnisse zu vermeiden, kann nur eine der beiden Varianten angewendet werden. Ein gleichzeitiges Abonnieren einer Eingangsgruppe und die Einrichtung einer Rückkopplung wird ausgeschlossen.

3.3.3. Betriebsablauf

Die Kommunikation über die Middleware ist in Abbildung 3.3 dargestellt. Zwecks Übersicht wurde hierbei auf das Eintragen der wiederholten Registrierung und Behandlung von fehlerhaften Anfragen verzichtet. Siehe dazu Abschnitt 3.3.2.

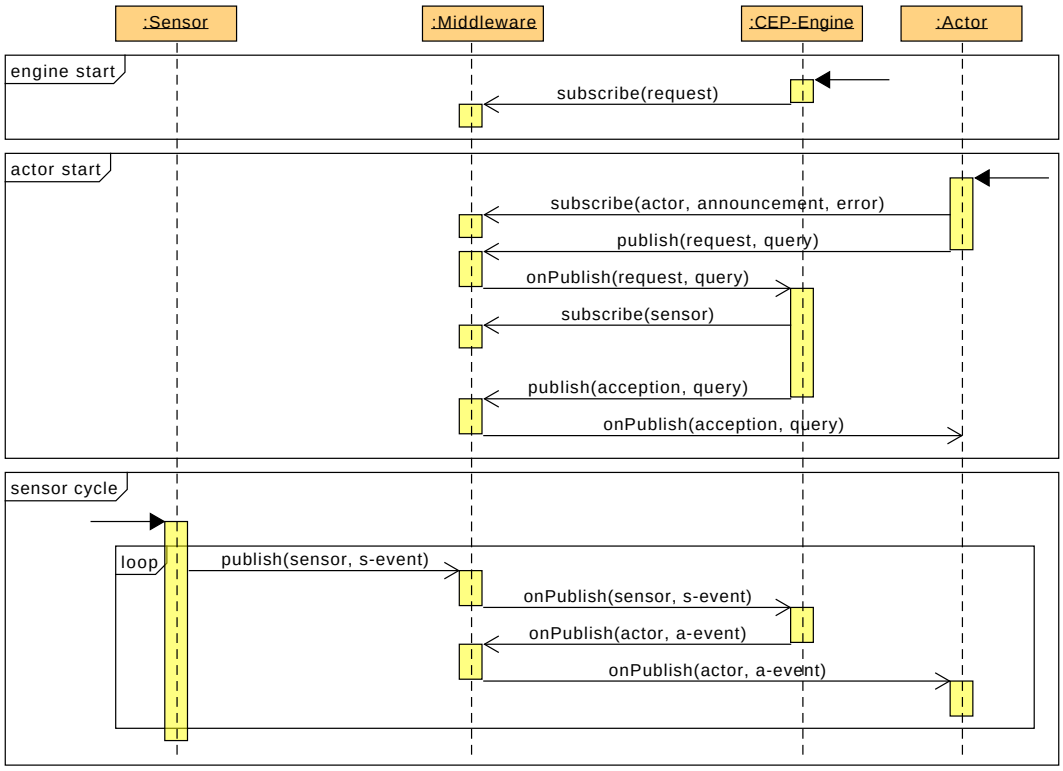

Abbildung 3.3.: Interaktion mit der CSTI Middleware

Beim Systemstart abonniert die Engine den Kanal für Anfragen (request) und wartet auf eben jene. Ein Aktor, der eine Anfrage stellen soll, muss drei Kanäle abonnieren: Den Bekanntmachungskanal (announcement), über welchen die CEP Engine bekannt gibt, dass eine Anfrage bedient wird, den Kanal für Fehlermeldungen (error), über welchen fehlerhafte Anfragen bekannt gegeben werden und den Kanal über welchen der Aktor Ereignisse empfangen möchte (actor). Anschließend bereitet der Aktor die Anfrage vor und veröffentlicht sie auf dem Anfragekanal. Die CSTI Middleware leitet die Anfrage zur CEP Engine weiter. Diese überprüft die Anfrage und abonniert alle Kanäle, die für die Verarbeitung nötig sind (hier nur Sensor). Nach Fertigstellung teilt sie allen Aktoren über den Announcementkanal mit, dass die Anfrage

bedient wird. Sobald der Sensor des Systems aktiv wird, veröffentlicht er unter seinem Kanal (Sensor) Ereignisse (Event). Die CSTI Middleware leitet diese zur CEP Engine weiter, da diese den Kanal abonniert hat. Die CEP Engine verarbeitet das Ereignis und generiert daraus ein neues Ereignis (A-Event). Dieses veröffentlicht sie auf dem in der Anfrage konfigurierten Kanal. Die CSTI Middleware gibt das Ereignis an den Aktor weiter.

3.3.4. Fehlertoleranz

Für einen Langzeitbetrieb ist es notwendig, dass die CEP Engine robust gegenüber Fehlern ist. Dazu wurden folgende Situationen bedacht.

Regenerierung

Die CEP Engine agiert als Kontextinterpreter. Dies bedeutet, dass Ausfälle und Versagen von Komponenten besonders beachtet werden müssen. Das zu entwickelnde Programm verfügt über keine Persistierung, weswegen nach einem Neustart alle Anfragen und Zustände verloren gehen. Das System sollte also über eine Möglichkeit zur Selbstwiederherstellung verfügen.

Die oben vorgestellte Lösung sieht vor, dass nach einem Neustart eine Registrierungsphase erfolgt. Aufgrund des verwendeten Leasepatterns dauert diese maximal die volle Dauer der *Leasetime*. Nach dieser müssen alle konformen Agenten ihre Anfrage erneuert haben. Da in den Verlängerungsanfragen alle nötigen Informationen enthalten sind, kann die Anfrage sofort intern registriert werden.

Nachfolgend müssen die Ereigniszustände wiederhergestellt werden. Dies geschieht dadurch, dass Sensoren regulär ihren Zustand veröffentlichen. Diese können erst nach Abschluss der Registrierungsphase sicher verarbeitet werden, da Ereignisse, die von keiner Anfrage benötigt werden, verworfen werden.

Ungültige Anfrage

Die CEP Engine überprüft jede eingereichte Anfrage und verweigert die Registrierung von Statements, die syntaktisch fehlerhaft sind. Sie erkennt zudem gewisse semantische Fehler. So werden Statements abgelehnt, die auf Felder eines Ereignisses zugreifen, welche für dieses nicht definiert wurden. QUERYMANAGER überprüft die Ereignistypen und verweigert die Registrierung, falls hier Fehler wie die oben genannten vorhanden sind.

Ressourcenhungrige Anfrage

Ressourcen für Anfragen werden nicht künstlich beschränkt. Benötigt eine Anfrage besonders viele Ressourcen, was zum Beispiel der Fall ist, wenn eine sehr große Anzahl von Ereignissen für die Anfrage vorgehalten werden muss, so wird nach *Best Efford* versucht, diese zu bedienen.

4. Realisierung

In diesem Kapitel wird darauf eingegangen, wie das Projekt umgesetzt wurde. Dazu werden zunächst die verwendeten Technologien neben der CEP Engine aufgezeigt. Anschließend wird die Umsetzung der Szenarien anhand von exemplarischen Problemlösungen beschrieben. Folgend wird ein Ausblick für das Projekt aufgestellt und ein Fazit gezogen.

4.1. Technologien

4.1.1. CEP-Engine

In Abschnitt 2.6 wurde festgestellt, dass sich die Produkte *Esper CEP* und *WSO2 CEP* zur Umsetzung des Projekts eignen. Es muss nun entschieden werden, welches Produkt sich zur Umsetzung besser eignet.

WSO2 hat einen direkten Vergleich mit *Esper* aufgestellt und erzielt dabei laut eigenen Angaben einen höheren Nachrichtendurchsatz[30]. Beide Produkte können Architekturen mit mehreren Prozessorkernen voll nutzen. Während Esper nur in der kostenpflichtigen Variante Möglichkeiten zum Clustering anbietet, unterstützt die freie Version von *WSO2* dies. Jedoch ist die Ansteuerung der CEP Engine über HTTP dazu Voraussetzung. Da dies nicht erfüllt ist, ist Clustering mit beiden Produkten nicht realisierbar.

Vor dem Herunterladen des Programmcodes vom *WSO2* sollte ein Formular mit vollem Namen, Telefonnummer und Betriebszugehörigkeit ausgefüllt werden. Nach korrekter Eingabe der Daten und mehreren Versuchen war es jedoch nicht möglich, die erforderliche Datei zu erhalten. Anfragen beim Kundendienst von *WSO2* zur Klärung dieses Problems blieben bis auf eine automatische Bestätigung unbeantwortet. *Esper CEP* befindet sich in den zentralen Repositories von Maven und lässt sich somit problemlos mit einer Vielzahl von Clienten beziehen.

Recherchen zeigen, dass mehrere Arbeiten *Esper* als eine ausgereifte, weit genutzte und universelle Lösung identifiziert haben [31, S. 1715][32, S. 44].

Da die schlechten Bezugsmöglichkeiten von *WSO2* voraussichtlich wiederkehrend für Schwierigkeiten sorgen würden und beide Produkte ansonsten gleich gut geeignet sind, wird für dieses Projekt *Esper CEP* verwendet.

4.1.2. Drittbibliotheken

Zur Umsetzung des Ziels werden mehrere Hilfsbibliotheken verwendet.

Google Guava

Googles Guava wird verwendet um die Standardbibliotheken von Java zu erweitern. So verfügt Java über keine bidirektionalen Maps oder eine kompakte Initialisierungweise für Container, welche für die Aufstellung der Testfälle von Vorteil sind. Guava bietet dieses und einen großen Umfang von Verbesserungen, welche auch für zukünftige Weiterentwicklungen von Vorteil sind. [33]

SnakeYaml

Für die Einstellung der Testumgebung werden Konfigurationsdateien verwendet. Der Einfachheit halber könnte als Format JSON verwendet werden, für welches Java einen integrierten Parser bereitstellt. Dieses Format erlaubt jedoch keine Inlinekommentare, was zwar für die Nachrichtenübermittlung von Vorteil ist, da so die Interpretierung einfacher und die Nachrichtengröße kompakter ist. Bei Konfigurationen ist jedoch eine Dokumentationsmöglichkeit wünschenswert. Aus diesem Grund wird YAML als Beschreibungssprache verwendet, da der JSON Sprachstandard eine Untermenge dieser und somit kompatibel ist. Die Bibliothek SnakeYaml kann YAML kodierte Daten einlesen und stellt sich als Java Map bereit, welche verschachtelt weitere Maps, Listen oder primitive Datentypen enthalten kann. Dadurch ist das Verwenden der Konfiguration schneller umzusetzen. [34]

Log4j

Als Dienst sollte das zu entwickelnde Programm über eine solide Möglichkeit zum Protokollieren des Betriebsablaufs verfügen. Dafür wird die Bibliothek Log4j verwendet. Diese ermöglicht auf einfache Weise das Klassifizieren von Protokollierungen, das Ausgeben dieser in mehrere Ausgänge wie die Standardausgabe oder Dateien sowie das einfache Formatieren und Anreichern der Ausgaben mit Zeitstempel und Quellenangaben. [35]

Argparse4j

Der Dienst soll Portable Operating System Interface (POSIX) konforme Kommandozeilenargumente unterstützen. POSIX ist eine Sammlung von Standards, welche die Bestrebung hat Kompatibilität zwischen verschiedenen Betriebssystemen und deren Komponenten herzustellen. Die Bibliothek Argparse4j soll verwendet werden, um die Kommandozeilenargumente einzulesen und dem Programm bereitzustellen. Dazu wird programmatisch definiert, welche Argumente und in welcher Kombination diese erlaubt sind. Zudem wird aus dieser Definition ein Hilfstext für eine bessere Bedienbarkeit generiert. [36]

4.2. Anwendungsszenarien

Zur Umsetzung der Szenarien sind mehrere Aufgabenschritte nötig. Für jedes Szenario müssen Sensorereignisse und zu generierende Aktorereignisse definiert werden. Während es möglich wäre, mit nur einer Anfrage ohne Zwischenschritte und folglich einem Ereignistyp ein Szenario zu bedienen, so wäre das Resultat eine sehr lange und unübersichtliche Beschreibung in der EPL von *Esper* und würde die Wiederverwendung von Teilergebnissen unmöglich machen. Aus diesem Grund müssen Zwischenereignisse definiert und in einer Hierarchie angeordnet werden. Diese definiert aus welchen Sensorinformationen Zwischenereignisse erstellt werden und wie diese letztlich zur Generierung von Aktorereignissen beitragen. Hier ist zu beachten, dass Abhängigkeiten zwischen den Szenarien bestehen. Dies ist in Bezug auf das CSTI realitätsnah, da einzelne Aufbauten im Labor miteinander interagieren können. Zuletzt werden die Anfragen für *Esper* erstellt.

4.2.1. Ereignisse

Alle benötigten Ereignistypen wurden zur Umsetzung in ein Oberszenario strukturiert. Dieses ist in Abbildung 4.1 veranschaulicht. Rote Ereignisse dienen zur Steuerung von Aktoren, sind also als Endresultate zu verstehen. Grün sind Zwischenereignisse die von Anfragen für Steuerungsereignisse verwendet werden. Gelbe Ereignisse stammen von Sensoren und sind Eingangsereignisse. Blau sind Konstanten.

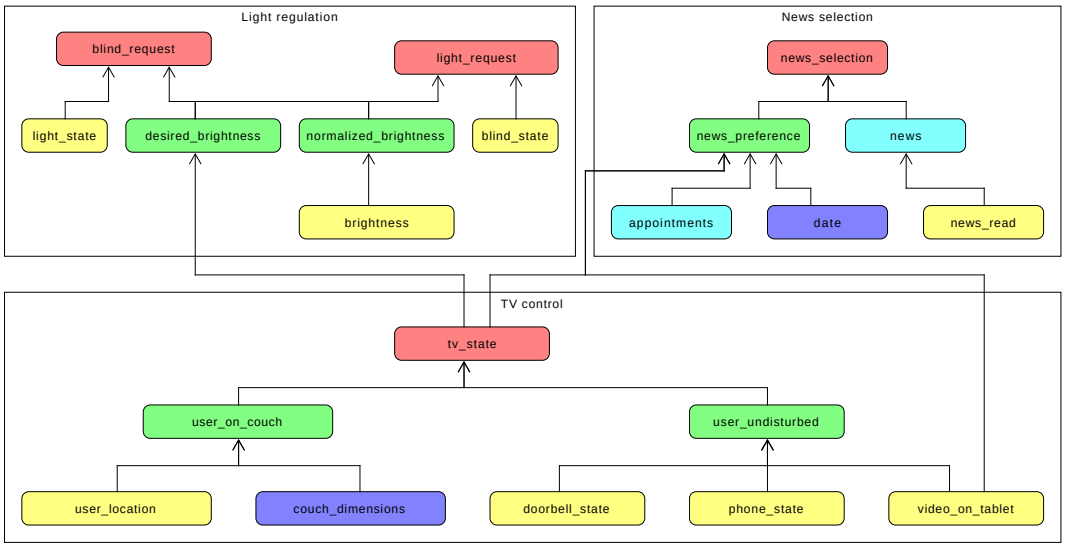

Abbildung 4.1.: Ereignishierarchie der Szenarien

Fernsehersteuerung

Um die Fernsehersteuerung zu realisieren, sind mehrere Ereignistypen nötig. Es ist zum einen zu ermitteln, ob sich der Benutzer auf dem Sofa befindet. Dazu muss die aktuelle Position des Benutzers bekannt sein. Des Weiteren muss bekannt sein, wo sich das Sofa befindet und welche Dimensionen es hat. Zum anderen muss erkannt werden, ob der Benutzer ungestört ist. Im Szenario fließen dazu ein, ob die Klingel in einem gewissen Zeitraum betätigt wurde, das Telefon klingelt oder für ein Telefonat benutzt wird und die Information ob der Benutzer ein Video auf dem Tablet sieht. Treten eine oder mehrere dieser Bedingungen ein, so ist der Benutzer gestört. Aus diesen Informationen wird geschlossen, ob der Fernseher aus, ein oder stummgeschaltet sein soll.

Helligkeitsregelung

Für die Regelung der Helligkeit ist zu erfassen, welche Soll-Helligkeit zu erreichen ist. Dies wird in Form eines Ereignisses von Ereignisquellen außerhalb des Szenarios definiert. Als Beispiel könnte die Helligkeit reduziert werden, wenn der Fernseher aus dem vorherigen Beispiel läuft. Tut er dies nicht, ist die maximal mögliche Helligkeit zum Arbeiten herzustellen. Die Werte des Helligkeitssensors liefern eine Momentaufnahme der Zimmerhelligkeit. Jedoch wäre dieser Messwert anfällig für Schwankungen, wie z.B. das Vorbeiziehen einer Wolke vor der Sonne oder das Vorbeigehen einer Person an der Lampe. Deswegen sollte der Durchschnittswert der Helligkeit über einen gewissen Zeitraum gebildet werden, um zuverlässigere, wenn auch trägere Messwerte zu erhalten. Der Zustand des Lichtes und des Rollos sind von diesen beiden Strömen abhängig. Zudem beeinflussen sie sich wechselseitig. Solange das Rollo nicht vollständig hochgefahren ist, bleibt das Licht aus, solange das Licht an ist, fährt das Rollo nicht herunter. Unter diesen Bedingungen versuchen beiden Einheiten die Helligkeit anzupassen.

Nachrichtenauswahl

Zur Auswahl der Nachrichten muss bestimmt werden, ob der Benutzer eine Präferenz für kurze oder lange Artikel hat. Hinweise auf die Präferenz für kurze Artikel sind ein laufender Fernseher, Termine, die unmittelbar bevorstehen, oder eine späte Uhrzeit mit darauffolgendem Werktag. Der Terminkalender des Benutzers kann nur schwer als Ereignisstrom dargestellt werden, es handelt sich im Kontext von Esper um sogenannte historische Daten. Diese können beispielsweise in einer SQL-Datenbank abgelegt werden. Bei *Esper* können in normalen Anfragen auch SQL-Statements verwendet werden, um auf diese Daten zugreifen zu können. Neben der Präferenz ist der Zugriff auf einen Satz Nachrichten erforderlich. Dies könnte ebenfalls mit einem Zugriff auf eine SQL-Datenbank erfolgen, die eine nach Länge sortiere Nachrichtenauswahl bereitstellt.

4.2.2. Ereignisstrukturen

Von insgesamt 17 verschiedenen Ereignistypen sind in Listing 4.1 exemplarisch zwei davon dargestellt.

```
# user_location
{"x": 10.0, "y": 20.0}
# blind_state
{"extent": 10, "max_extent": 100,
 "fully_extended": false, "fully_retracted": false}
```

Listing 4.1: Ereignisstrukturen

Die Position des Benutzers wird in USER_LOCATION als zweidimensionales Koordinatenpaar dargestellt. Eine Angabe der Höhe wird vernachlässigt, da sie in diesem Beispiel nicht benötigt wird.

Der Zustand des Rollos wird in BLIND_STATE dargestellt. Es ist nötig anzugeben, wie weit das Rollo momentan ausgefahren ist (extent) und wie es maximal sein könnte (max_extent). Zur Vereinfachung folgender Anfragen wird in jedem Ereignis vermerkt, ob das Rollo vollständig ein- oder ausgefahren ist.

4.2.3. Anfragen

Für jedes komplexe Ereignis ist eine Anfrage zu erstellen. Auch hier ist aus neun Anfragen ein exemplarischer Ausschnitt dargestellt.

Sofabenutzungserkennung

Zur Erstellung des Ereignisses USER_ON_COUCH soll eine Anfrage unter der Zuhilfenahme von USER_LOCATION und COUCH_DIMENSION erkennen, ob sich der Nutzer in der BOUNDING BOX befindet.

Es gibt zwei Möglichkeiten COUCH_DIMENSION darzustellen. Die Werte könnten als Konstanten in die Anfrage integriert werden. Diese Variante hätte jedoch zur Folge, dass bei einer Positionsänderung des Sofas diese in allen Agenten geändert werden müssten. Hier wäre sinnvoll, die Erstellung der Anfrage einen einzelnen Agenten übernehmen zu lassen. Weitere Agenten würden auf den Kanal zugreifen ohne eine Abfrage abzusetzen. Dies wäre jedoch fehleranfällig. Die zweite Möglichkeit ist die Darstellung der Konstante als Ereignis. Ein Agent im System verfügt über die benötigten Informationen, um das Ereignis zu bilden, und sendet in regelmäßigem Abstand diese Information an die Engine. Die Engine speichert wie in Listing 4.2 dargestellt den aktuellsten Wert zwischen und verwendet ihn für die Berechnung.

```
select
  case
    when
      user.x >= couch.x_min and
      user.x <= couch.x_max and
      user.y >= couch.y_min and
      user.y <= couch.y_max
    then true
    else false
  end as value
  from
    ‘Location__user_location‘.std:lastevent() as user,
    ‘Dimensions__couch_dimensions‘.std:lastevent() as couch
```

Listing 4.2: Anfrage für user_on_couch in Esper EPL

Zudem wird das aktuellste Ereignis aus dem Strom USER_LOCATION verwendet und überprüft, ob sich die gegebenen Koordinaten innerhalb der *Bounding Box* von COUCH_DIMENSION befinden. Ist dies der Fall, erhält das Feld VALUE von USER_ON_COUCH einen positiven Wahrheitswert, ansonsten einen negativen. Dies wird für jedes eintreffende Ereignis durchgeführt. Im CSTI existiert bereits ein Agent, welcher die Positionsdaten von *ArtTrack* als Ereignisse in die CSTI Middleware einspeist und für dieses Szenario verwendet wird. Sollte das Sofa in zukünftigen Aufbauten gedreht werden, so könnte eine andere Funktion zur Erkennung verwendet werden.

Helligkeitsregelung

Die Generierung von NORMALIZED_BRIGHTNESS wird durch zwei Anfragen vorgenommen.

NORMALIZED_BRIGHTNESS soll über einen gewichteten Durchschnitt der Helligkeit der letzten halben Minute erstellt werden. Aus dieser halben Minute werden zehn gleichmäßig verteilte Werte nach Aktualität aufsteigend gewichtet. Um dies zu realisieren, muss sichergestellt werden, dass pro drei Sekunden, also mit 20 Hz, ein Ereignis eingeht. Während dies am sinnvollsten auf Sensorseite konfiguriert werden kann, gibt es die Möglichkeit, dies über eine Anfrage zu realisieren. Die Anfrage für STABILIZED_BRIGHTNESS (Siehe Listing 4.3) sorgt dafür, dass bei einer zu hohen Eingabefrequenz innerhalb eines drei Sekunden Fensters alle Ereignisse bis auf das letzte verworfen werden und bei einer zu geringen Frequenz das letzte Ereignis wiederholt wird.

```
select * from ‘Brightness__brightness‘ output last every 3 seconds
```

Listing 4.3: Anfrage für stabilized_brightness in Esper EPL

Dadurch ist die Berechnung des Durchschnitts für NORMALIZED_BRIGHTNESS (Siehe Listing 4.4) eleganter zu bewerkstelligen. Dazu wird jeder neue Wert zehnfach gewichtet und seine neun Vorgänger mit fallender Gewichtung einbezogen.

```
select
 (  prior(1,value) + 9*prior(2,value) + 8*prior(3,value) +
  7*prior(4,value) + 6*prior(5,value) + 5*prior(6,value) +
  4*prior(7,value) + 3*prior(8,value) + 2*prior(9,value) +
  1*prior(10, value)) / (10+9+8+7+6+5+4+3+2+1) as value
from ‘Brightness__stabilized_brightness‘.win:time(30 sec)
```

Listing 4.4: Anfrage für normalized_brightness in Esper EPL

Es sind drei Anfragen an die Rollosteuerung über das Ereignis BLIND_REQUEST möglich: STOP lässt das Rollo die aktuelle Position halten, CLOSE schließt das Rollo, OPEN öffnet es. Die Anfrage ist in Listing 4.5 zu sehen. Das Rollo wird nur bewegt, wenn zwischen Soll- und Istzustand der Helligkeit mindestens eine absolute Differenz von 5.0 Einheiten liegt.

```
select
  case
    when
      desired.value < normalized.value-5.0 and
      light_state.fully_off=true
    then "close"
    when
      desired.value > normalized.value+5.0
    then "open"
    else "stop"
  end
  as request
from
  ‘Brightness__desired_brightness‘.std:lastevent() as desired,
  ‘Brightness__normalized_brightness‘.std:lastevent() as normalized
```

Listing 4.5: Anfrage für blind_request in Esper EPL

Fernsehabschaltung

Um seine Lebensdauer zu verlängern, soll verhindert werden, dass sich der Fernseher unnötig an- und abschaltet. Wenn der Benutzer beispielsweise das Sofa kurzfristig verlässt, um etwas aus der Küche zu holen, so kann ein Ausschalten vermieden werden, indem erst abgeschaltet

wird, wenn das Sofa über einen längeren vorgesehenen Zeitraum nicht benutzt wird. Setzt sich der Benutzer erstmals aufs Sofa, so ist jedoch der Fernseher unmittelbar einzuschalten, da der Benutzer direkt mit einer Reaktion rechnet. Zur Visualisierung siehe Abbildung 4.2.

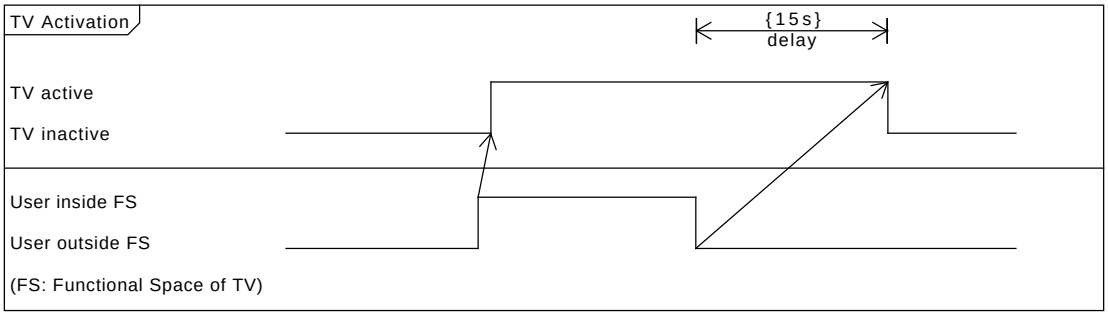

Abbildung 4.2.: Timing des TV-Standby-Szenarios

Nachrichtenauswahl

Das Tablet und potentiell andere Smart Objects sollen Nachrichten anzeigen, die einen angemessenen Umfang für den Bedarf des Benutzers haben und über die Middleware verteilt werden. Im aktuellen Kontext wird die Nutzerstimmung in NEWS_PREFERENCE zum einen daraus ermittelt, ob der Fernseher eingeschaltet ist oder ein Video auf dem Tablet angezeigt wird. Ist dies der Fall, werden kürzere Nachrichten angezeigt. Zum anderen wird einbezogen, ob der Nutzer innerhalb der nächsten Stunde zu einem Termin aufbrechen muss. Dazu wird auf eine SQL Datenbank zugegriffen, in welcher alle Termine gespeichert sind. Die aktuelle Zeit wird einbezogen und Termine abgerufen, die innerhalb des betreffenden Zeitrahmens liegen. Wird hierbei ein Termin gefunden, so werden ebenfalls kurze Nachrichten ausgewählt.

Die eigentliche Auswahl der Nachrichten wird in der Abfrage für NEWS_SELECTION (in Listing 4.6) vorgenommen.

```
1 select news.*
2 from
3   pattern[every timer:interval(30 sec)],
4   ‘Preference__news_preference‘.std:lastevent(),
5   sql:NewsDB
6     [’select * from News where type = ${news_preference.type}’]
7     as news
```

Listing 4.6: Anfrage für news_selection

Hier werden Ereignisse aus zwei Quellen, NEWS_PREFERENCE und einer eingebetteten SQL Abfrage gebildet. Die Abfrage greift auf die vorher angebundene Datenbank NEWSDB zu und selektiert alle Einträge aus der Tabelle NEWS, deren Typ laut NEWS_PREFERENCE dem aktuell gewünschten Nachrichtentypen entspricht. Jede Nachricht wird dabei als eigenes Ereignis verschickt. Diese Anfrage wird bei einer Änderung von NEWS_PREFERENCE und alle 30 s durchgeführt.

Ist eine Nachricht auf einem Gerät gelesen worden, so teilt dieses Gerät dies der CEP Engine über das Ereignis NEWS_READ mit. Diese entfernt daraufhin die Nachricht mit einer SQL Anfrage.

4.2.4. Agentenaufbau

Zur Veranschaulichung der Umsetzung der Szenarien ist in Listing 4.7 eine Agentenimplementierung zu einem vereinfachten Problem dargestellt.

```
1  public class ClientAgent extends Agent {
2    private final Group requestGroup = createGroup("cep-request");
3    private final Group indicatorGroup = createGroup("indicator");
4    private final CEPQuery query;
5
6    public ClientAgent() {
7      MapValue inputs = new MapValue(yaml.load(
8          "{BrightnessValue__sensor0: {value: java.lang.Integer}}"));
9      String statement = "select case when brightness.value < 20 " +
10       "then true else false end as state " +
11       "from BrightnessValue__sensor.std:lastevent() as brightness";
12     MapValue outputDefinition
13       = new MapValue(yaml.load("{state: java.lang.Boolean}"));
14     OutputConfiguration output
15       = new OutputConfiguration("IndicatorSetting__indicator",
16         outputDefinition, false);
17     this.query = new CEPQuery("indicator-setter", inputs, statement, output);
18   }
19   public void preStart()  {
20     subscribe(indicatorGroup, IndicatorSerialization.indicatorDeserializer());
21     requestGroup.tell(this.query, CepSerialization.CEPQueryFormat(), getAgent());
22   }
23   public void onReceive(Object message) throws Exception {
24     if (message instanceof IndicatorSetting) {
25       Indicator.getInstance().set(((IndicatorSetting) message).state());
26     }
27   }
28 }
```

Listing 4.7: Implementierung eines beispielhaften Agenten in Java

Eine Anzeige soll aufleuchten, wenn die gemessene Helligkeit eines Sensors unter 20 Einheiten fällt und wieder ausgehen, sobald der Wert überschritten wird. Dazu bereitet der Agent in Zeile 2-3 zwei Gruppen vor: CEP-REQUEST für das Absetzen der Anfrage und INDICATOR für die erwarteten Ereignisse.

Im Konstruktor des Agenten (Zeile 6) wird die Anfrage vorbereitet. Da das Erstellen von komplexeren Datenstrukturen in Java platzeinnehmend ist, wird *SnakeYaml* verwendet, um aus definierten Strings Objekte zu bauen. Auf diese Weise wird in Zeile 7-8 die Signatur des Eingabestroms BRIGHTNESSVALUE__SENSOR0, also aller Ereignisse vom Typ BRIGHTNESSVALUE aus der Gruppe SENSOR0, definiert. Das Statement wird als einfache Bedingung in Zeile 9-11 realisiert. Zeile 12-13 definieren die Signatur des Ausgabeereignisses, in Zeile 14-16 wird diese mit den Informationen, dass das Ereignis in der Gruppe INDICATOR als Typ INDICATORSETTING ohne Rückkopplung eingefügt werden soll, zur Ausgabekonfiguration zusammengesetzt. In Zeile 17 werden diese Informationen zur eigentlichen Anfrage zusammengesetzt.

Die Funktion PRESTART in Zeile 20 wird vom bereitgestellten Agentenframework aufgerufen, sobald die Verbindung zur CSTI Middleware aufgebaut ist. Hier wird in Zeile 21 die Gruppe der gewünschten Ereignisse abonniert und die Anfrage in Zeile 22 an die CEP Engine abgesetzt. Die Methode ONRECEIVE wird aufgerufen, wenn der Agent eine Nachricht erhält. Hier wird der Sicherheit halber der Typ der Nachricht überprüft und anschließend die Instanz der Anzeige aufgerufen, die über eine Settermethode den im Ereignis gespeicherten Zustand erhält.

4.3. Ausblick

Aus der hier durchgeführten Bearbeitung des Themas haben sich weitere Funktionalitäten ergeben, die hier nur als Ausblick dargestellt werden, da sie den Umfang dieser Arbeit bei weitem übersteigen würden. Sie können im weiteren Arbeiten umgesetzt werden.

4.3.1. Vereinfachung des Nachrichtenformats

Momentan müssen neben dem eigentlichen Statement in einer Anfrage alle verwendeten Ereignistypen und deren Gruppennamen zusätzlich definiert werden. Um diesen umständlichen Vorgang zu vereinfachen, sollte das Statement vom Programm interpretiert werden. Durch Konvention könnten somit benötigte Eingabekanäle ausgelesen werden.

Typendefinitionen sind bei *Esper* optional, wenn in Statements sogenanntes *Duck Typing*, also schwach typisiertes Arbeiten mit Objekten, angewandt wird. Das Arbeiten ohne starke Typisierung kann bei einem komplexen Smart Environment zu schwer nachvollziehbaren Fehlern führen und sollte vermieden werden.

Eine Möglichkeit wäre die Erstellung eines Interpreters, der in der Lage wäre, den kompletten Sprachumfang zu verstehen. Durch geschickte Analyse des Statements könnte dann eine Teildefinition der Typen abgeleitet werden. Es wären zudem neben dem Statement keine Zusatzangaben mehr nötig. *Esper* verfügt über eine Pluginarchitektur die ermöglicht, die Aufgabe dieses Interpreters mit einem Plugin zu realisieren.

Die CSTI Middleware überwacht intern übermittelte Nachrichtentypen und verfügt über Schnittstellen, die genutzt werden könnten, um die Definition der Typen dynamisch zu erhalten. Dies könnte verwendet werden, um die manuelle Definition dieser obsolet zu machen.

4.3.2. Erstellung einer Domain Specific Language

Eine neue DSL könnte erstellt werden, die alle benötigten Informationen enthält, um die für *Esper* spezifische EPL zu generieren. Dadurch kann eine Anweisung komplett in dieser Sprache dargestellt werden. Die Sprache müsste nicht den kompletten Funktionsumfang von *Esper* unterstützen und könnte dadurch einfach strukturiert werden. Dies würde die Einarbeitungszeit und Fehlerquellen weiter reduzieren.

4.3.3. Entwicklung einer Clientbibliothek

Die gefundene Lösung erfordert momentan, dass Entwickler sich genauer mit der Bedienung der CEP Engine beschäftigen. Ein Agent muss Kanäle eigenständig abonnieren und eine Anfragenachricht erstellen. Dies kann durch die Entwicklung einer Clientbibliothek für das Framework der Middleware vereinfacht werden. Der Entwickler könnte über einfache Methodenaufrufe seine Statements absetzen. Durch oben genannte Typisierungsverbesserungen würde nur noch die Definition des Statements nötig sein.

4.3.4. Umsetzen von Clustering

Esper bietet in der verwendeten Lizenz keine eingebaute Möglichkeit für Clustering. Das bedeutet, dass das Produkt eigenständig nur als einzelne Instanz auf einem System betrieben werden kann. Jedoch ermöglicht die durchgeführte Umsetzung der Integration, dass *Esper* mithilfe der CSTI Middleware einfach auf Clustering umgestellt werden kann.

Die vollständige Implementierung einer Clusterlösung liegt außerhalb des Rahmens dieser Arbeit. Jedoch ist eine der Grundvoraussetzungen für die Option, mehrere Instanzen voneinander unabhängig im Netzwerk betreiben zu können, als Vorarbeit bereits umgesetzt. Dies wird dadurch ermöglicht, dass über Programmparameter einstellbar ist, welche Gruppennamen

die Engine zur Kommunikation mit den Agenten verwendet. Dadurch kann Clustering als Erweiterung einfach hinzugefügt werden.

Eine einfache Möglichkeit zur Umsetzung von Clustering wäre das Betreiben mehrerer Instanzen des zu entwickelnden Programms. Agenten wählen nach bestimmen Kriterien die zu verwendende Instanz der Engine aus. Dies erfordert jedoch, dass jeder Agent Kenntnis über die Struktur des Clusters hat, was Wartungsprobleme aufwirft. Das Clustering könnte auch über einen transparenten *Load Balancer* durchgeführt werden, welcher sämtliche Anfragen von Agenten entgegennimmt und an die Instanzen des Clusters verteilt, was dazu führt, dass nur diese Komponente den Cluster kennen und verwalten muss. Eine weitere Möglichkeit wäre ein selbstverwaltetes System, in welchem Engineinstanzen eigenständig eintreffende Anfragen untereinander verteilen.

4.3.5. Entwicklung einer eigenen CEP Engine

Alle verfügbaren CEP Engines sind für die Verwendung im Geschäftsbereich entwickelt worden. Die Entwicklung einer eigenen CEP Engine würde die Abhängigkeit zu Drittquellen für Software reduzieren und ermöglichen, eine CEP Engine speziell für Smart Environments zu entwickeln, wodurch die Aufgabenangemessenheit weiter gesteigert werden kann.

4.4. Fazit

In der Realisierung wurde ermittelt, welche Technologien eingesetzt werden sollen und ob sich die Szenarien mit der ermittelten Lösung verwirklichen lassen.

Die aufgestellten Szenarien konnten mit dem Einsatz der CEP Engine umgesetzt werden. Wie oben gezeigt, sind die deklarierten Instruktionen für die Engine kurz und übersichtlich. Die verwendete EPL weist eine umfassende Sammlung an Elementen auf, um Anfragen zu strukturieren. Eine gute Onlinedokumentation (Siehe [37]) ermöglicht den schnellen Einstieg in die Sprache. Während die Szenarien auch anders umgesetzt werden können, ist dieser Ansatz angemessen für die Verwendung im CSTI.

5. Schluss

Im letzten Kapitel wird die Arbeit zusammengefasst und anschließend die Zukunft des Themas erörtert.

5.1. Zusammenfassung

In dieser Arbeit ist eine CEP Engine in die Labormiddleware integriert worden. Es konnte gezeigt werden, dass dadurch eine einfache Einwicklung von ereignisverarbeitenden Agenten möglich ist.

Dazu wurde gezeigt, was ein Multiagenten System ist und welche Anforderungen und Probleme dieses aufweist. Es zeigte sich, dass in diesem System keine Komponente vorgesehen ist, die für die Verarbeitung von Daten vorrangig verwendet wird. Smart Environments weisen eine große Menge von Stakeholdern auf, die ihrerseits unterschiedliche Anforderungen haben.

Die Engine muss von den Entwicklern einfach zu bedienen sein und eine hohe Integrationsfähigkeit besitzen. Zudem muss sie universell anwendbar sein, um alle Anwendungsszenarien zu bedienen. Es ist relevant, dass Komponenten untereinander eine hohe Interoperabilität aufweisen und von Benutzern intuitiv verwendet werden können.

Die Nutzungsumgebung, das CSTI, weist die Besonderheit auf, dass unter Entwicklern eine hohe Fluktuation besteht und eine einfache Einarbeitung möglich sein muss. Da die EPL an SQL angelehnt ist und die Entwickler größtenteils aus Informatikstudiengängen kommen, welche Pflichtmodule für SQL beinhalten, ist dies gegeben. Zudem ist eine gute Wartbarkeit der CEP Engine erforderlich. Zur Integration zeigte sich, dass die Engine am sinnvollsten als ein separater Agent zu implementieren war.

Es wurde gezeigt, dass CEP kein triviales Problemfeld ist. Um alle Anforderungen des Smart Environments zu berücksichtigen, musste eine einfache und effiziente Möglichkeit geschaffen werden, CEP besser in die Laborumgebung zu integrieren. Im Gegensatz zu der vorhergehenden Lösung, CEP prozedural in Agenten integriert umzusetzen, profitiert das Labor durch den Einsatz einer CEP Engine mit deklarativen Anweisungen von Entkopplung und Universalität. In der Technologieauswahl wurde *Esper CEP* als erfolgversprechendster Kandidat für die Implementierung erkannt.

Da die gegebene Middleware bereits kanal- bzw. gruppenbasiert arbeitet, konnte ein direkter Ansatz zum Abonnieren der für die Engine relevanten Gruppen erstellt werden. Die hierfür benötigten Informationen werden in einer einzelnen Anfragenachricht untergebracht. Garbage Collection wurde umgesetzt, um den Langzeitbetrieb zu ermöglichen. Eine modulare Programmstruktur ermöglicht es, auf einfache Weise die Implementierung der Engine oder die Systemanbindung auszuwechseln. Dank eines kompakten Kommunikationsprotokolls ist es einfach, als Entwickler von Agenten Anfragen an die CEP Engine zu definieren und über die CSTI Middleware abzusetzen. Für den reibungslosen Betrieb ist bei der Entwicklung ein besonderes Augenmerk auf Fehlertoleranz gelegt worden.

In der Realisierung wird festgestellt, dass der beschriebe Ansatz eine einfache Umsetzung der Szenarien ermöglicht und für diese Aufgabenstellung angemessen ist.

5.2. Ausblick

Der Einsatz einer CEP Engine in Smart Environments erweist sich als sehr vielversprechend. Während diese Arbeit mit der Aufarbeitung dieses Themas beginnt, kann es in Folgearbeiten weiter vertieft werden. Dafür ist eine Masterarbeit geplant. Die Erkennung von Tendenzen und komplexeren Analysen kann zu einer erweiterten Funktionalität beitragen. Dazu stellen CEP Engines Möglichkeiten zur Erkennung von sogenannten Patterns bereits. Damit können Kausalitäten erkannt und Voraussagen getroffen werden. Zudem kann weiter untersucht werden, wie auf der Seite von Agenten Verbesserungen vorgenommen werden können, um CEP stärker zugänglich zu machen.

Es gibt bereits Bemühungen, diese Integration in anderen Arbeiten zu verwenden. Im Projekt *How Will We Breathe Tomorrow* von Jessica Broscheit verarbeiten Smart Objects und Wearables laufend unter anderem die Feinstaubbelastung der Stadt Peking und visualisieren diese. Hier eignet sich CEP um Veränderungen und Tendenzen zu erkennen. Die Webseite des Projekts ist unter [38] zu finden.

A. Anhang

A.1. Diagramme

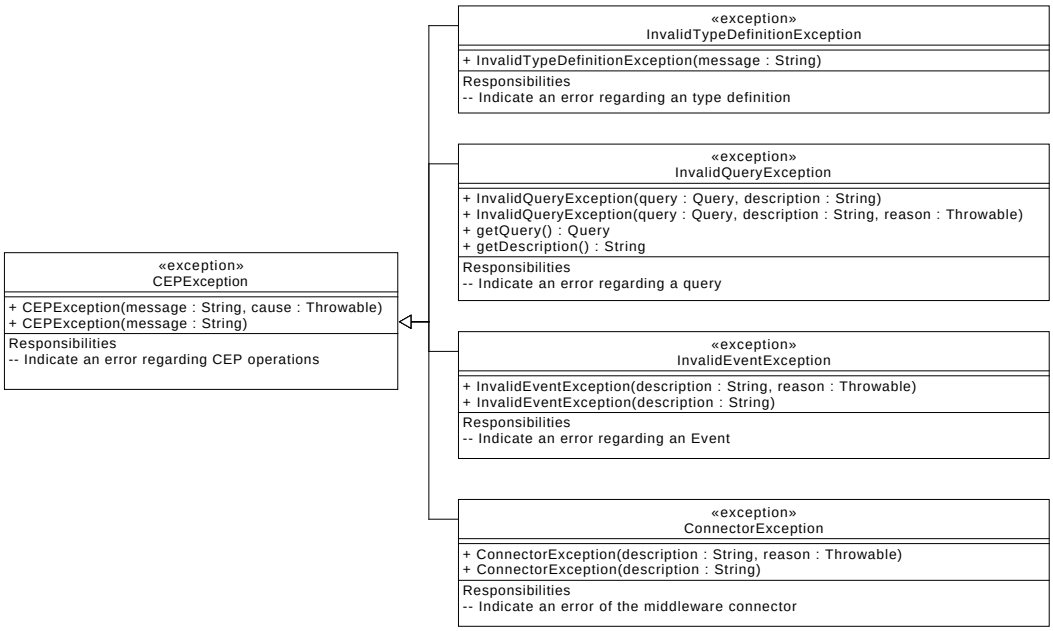

Abbildung A.1.: Klassendiagramm zu Exceptions im Programm

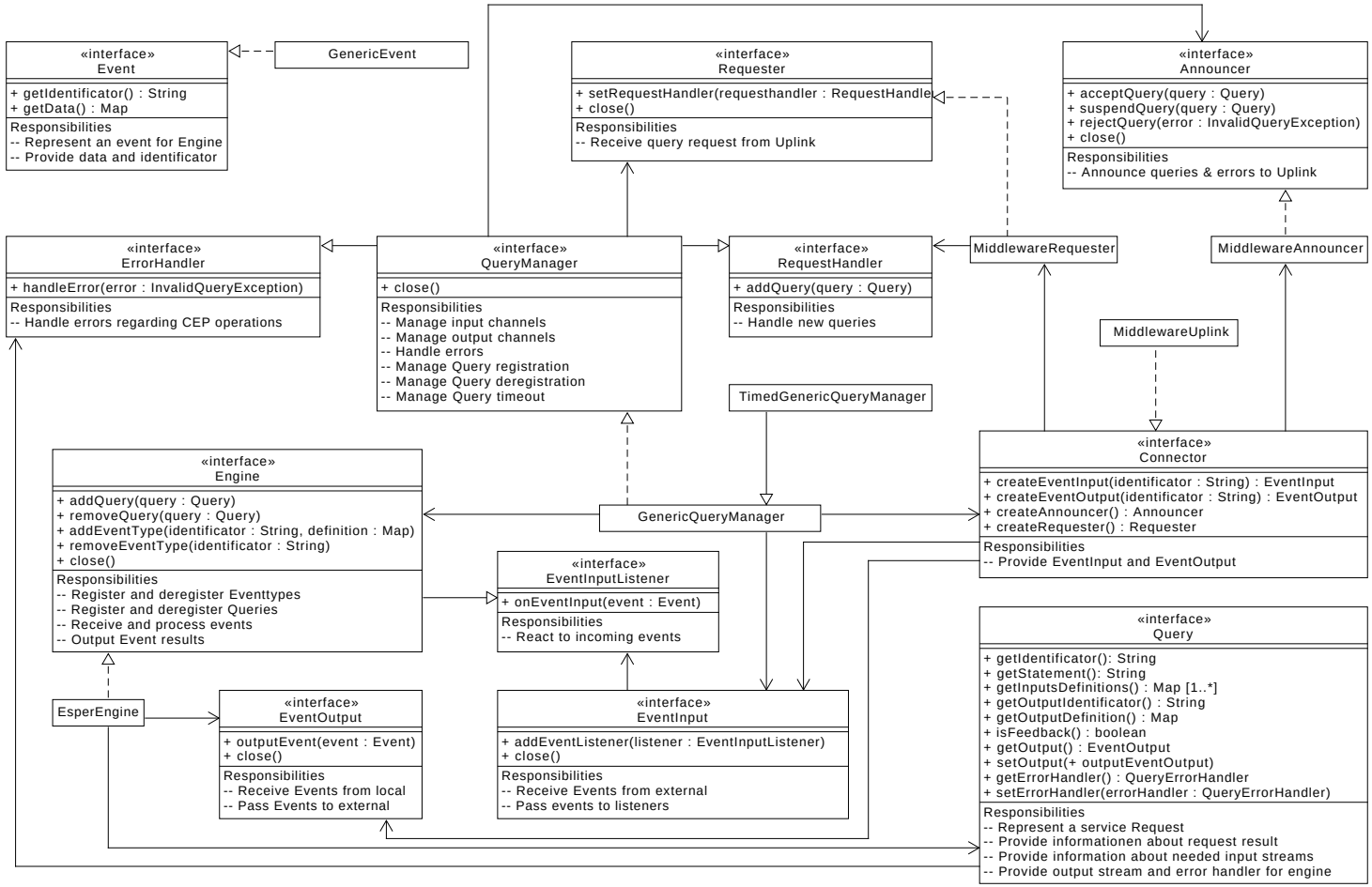

Abbildung A.2.: Klassendiagramm zum Programmaufbau

Literatur

[1] Tobias Eichler. "Agentenbasierte Middleware zur Entwicklerunterstützung in einem Smart-Home-Labor". 2. Okt. 2014. URL: https://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/eichler.pdf (besucht am 22. 05. 2016).

[2] *VDE/VDI 2653. Agentensysteme in der Automatisierungstechnik.* Juni 2010.

[3] *RPC: Remote Procedure Call Protocol Specification Version 2.* Englisch. RFC 1057. IETF, Juni 1988. URL: https://tools.ietf.org/html/rfc1057.

[4] *Message Passing.* Englisch. URL: https://www.defit.org/message-passing (besucht am 17. 08. 2016).

[5] *Unicast, Multicast and Anycast - Types of communication in IPv6.* Englisch. URL: http://www.omnisecu.com/tcpip/ipv6/unicast-multicast-anycast-types-of-network-communication-in-ipv6.php (besucht am 17. 08. 2016).

[6] *Peer-to-Peer (P2P) Architecture:Definition, Taxonomies, Examples, and Applicability.* Englisch. RFC 5694. IETF, Juni 1988. URL: https://tools.ietf.org/html/rfc5694.

[7] *Message Broker.* URL: https://www.informatik.uni-augsburg.de/lehrstuehle/swt/vs/lehre/archiv/WS_05_06/VerteilteSysteme_Uebung/downloads/MessageBroker.pdf (besucht am 20. 08. 2016).

[8] *AMQP 0-9-1 Model Explained.* Englisch. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html (besucht am 20. 08. 2016).

[9] *RabbitMQ Performance Measurements, part 2.* Englisch. URL: https://www.rabbitmq.com/blog/2012/04/25/rabbitmq-performance-measurements-part-2 (besucht am 20. 08. 2016).

[10] M. Weiser, R. Gold und J. S. Brown. "The origins of ubiquitous computing research at PARC in the late 1980s". Englisch. In: *IBM Systems Journal* 38.4 (1999), S. 693–696.

[11] J. Diane Cook. "Multi-agent smart environments". Englisch. In: *Journal of Ambient Intelligence and Smart Environments* 1 (2009), S. 47–51. URL: http://eecs.wsu.edu/~cook/pubs/jaise09.pdf.

[12] James Kusznir und J. Diane Cook. "Designing Lightweight Software Architectures for Smart Environments". Englisch. In: *Intelligent Environments (IE), 2010 Sixth International Conference on*. IEEE, 19. Juli 2010. URL: http://www.eecs.wsu.edu/~cook/pubs/ie10p1.pdf (besucht am 20. 08. 2016).

[13] *Creative Space for Technical Innovations*. URL: http://creative-space.haw-hamburg.de (besucht am 22. 07. 2016).

[14] Ralf Bruns und Jürgen Dunkel. *Complex Event Processing*. Springer, 2015.

[15] Daniel Jobst. *Service- und Ereignisorientierung im Contact-Center*. 2010.

[16] Ralf Bruns und Jürgen Dunkel. *Event-Driven Architecture*. Springer, 2010.

[17] *Complex Event Processing: Auswertung von Datenströmen*. URL: http://www.heise.de/ix/artikel/Kontinuierliche-Kontrolle-905334.html (besucht am 20. 08. 2016).

[18] David Luckham. *The Power of Events*. Englisch. 2002.

[19] Kjell Otto. "Aktuelle Entwicklungskonzepte zur Projektintegration in einem Smart Home anhand von Maven, OSGi und Drools Fusion". 24. Apr. 2013. URL: https://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/master/otto.pdf (besucht am 09. 06. 2016).

[20] *Esper CEP - Technical Specifications*. Englisch. URL: http://www.espertech.com/products/technicalspec.php (besucht am 10. 06. 2016).

[21] Juan C. Augusto und Chris D. Nugent. "The Use of Temporal Reasoning and Management of Complex Events in Smart Homes". Englisch. In: *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004*. Aug. 2004. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.65.5192&rep=rep1&type=pdf (besucht am 22. 05. 2016).

[22] Wen Yao, Chao-Hsien Chu und Li Zang. "Leveraging complex event processing for smart hospitals using RFID". Englisch. In: *Journal of Network and Computer Applications*. Hrsg. von Quan Z. Sheng, Sherali Zeadally und Aikaterini Mitrokotsa. Bd. 34. 3. Elsevier, Mai 2011.

[23] *Gradle Build Tool*. Englisch. URL: https://gradle.org (besucht am 24. 08. 2016).

[24] *Maven*. Englisch. URL: https://maven.apache.org (besucht am 24. 08. 2016).

[25] *Gitlab*. Englisch. URL: https://about.gitlab.com (besucht am 24. 08. 2016).

[26] *Eclipse*. Englisch. URL: https://eclipse.org (besucht am 24. 08. 2016).

[27] *JUnit*. Englisch. URL: http://junit.org/junit4 (besucht am 24. 08. 2016).

[28] Englisch. URL: http://www.jacoco.org/jacoco (besucht am 07. 08. 2016).

[29] *CEP with Akka and Esper or Streams*. Englisch. URL: http://www.lightbend.com/activator/template/akka-with-esper (besucht am 30. 07. 2016).

[30] *CEP Performance: Processing 100k to millions of events per second using WSO2 Complex Event Processing*. Englisch. URL: http://wso2.com/library/blog-post/2013/08/cep-performance-processing-100k-to-millions-of-events-per-second-using-wso2-complex-event (besucht am 10. 06. 2016).

[31] Gianpaolo Cugola und Alessandro Margara. "Complex event processing with T-REX". Englisch. In: *Journal of Systems and Software*. Bd. 85. 8. Elsevier, Aug. 2012.

[32] Gianpaolo Cugola und Alessandro Margara. "Processing flows of information: From data stream to complex event processing". Englisch. In: *ACM Computing Surveys*. Bd. 44. 3. ACM, Juni 2012.

[33] *Guava*. Englisch. URL: https://github.com/google/guava (besucht am 24. 08. 2016).

[34] *Snakeyaml*. Englisch. URL: https://bitbucket.org/asomov/snakeyaml (besucht am 24. 08. 2016).

[35] *Log4j*. Englisch. URL: https://logging.apache.org/log4j/2.x (besucht am 24. 08. 2016).

[36] *Argparse4j*. Englisch. URL: https://argparse4j.github.io (besucht am 24. 08. 2016).

[37] *Esper Reference*. Englisch. URL: http://www.espertech.com/esper/release-5.3.0/esper-reference/html/index.html (besucht am 14. 08. 2016).

[38] *How Will We Breathe Tomorrow*. URL: http://howwillwebreathetomorrow.com (besucht am 15. 08. 2016).

Glossar

Agent "Ein Agent ist ein Computersystem, das sich in einer bestimmten Umgebung befindet und welches fähig ist, eigenständige Aktionen in dieser Umgebung durchzuführen, um seine (vorgegebenen) Ziele zu erreichen."[2]. 1–3, 5–12, 14, 17, 19, 21, 23, 27, 29, 31, 36–38, 41, 42, 44, 52, 53, 57, 59, 61, 62, 70

Aktor Ein Smart Object welches sein Umfeld beeinflussen kann. Bsp: Rollosteuerung. 1, 26, 37, 40, 41, 43, 44

AMQP Ein Zugriffsprotokoll für nachrichtenbasierte Middlewares. 22, 34

API Eine Schnittstelle eines Programms, welche die Anbindung an das System und andere Programme ermöglicht. 3, 21, 31

CEP Behandelt die Analyse, Erkennung, Gruppierung und Verarbeitung von zusammenhängenden Ereignissen. ix, 1, 2, 5, 15–19, 21, 26–28, 61, 62, 69, 71

CEP Engine Ein Programm welches sich um die Durchführung von CEP kümmert. ix, xi, 2, 3, 11, 14–18, 20–29, 31, 34–44, 47, 56–59, 61, 62

Clustering Eine Technik, die mehrere separate Instanzen einer Ressource nutzt, um Leistungssteigerung und Ausfallsicherheit zu erreichen. 58, 59

CSTI Ein neues Labor der *HAW Hamburg*. ix, 1, 2, 5, 11–14, 21, 31, 32, 49, 53, 59, 61, 69

CSTI Middleware Ein von Tobias Eichler entwickelter Message Broker, der im CSTI und Living Place eingesetzt wird. ix, 2, 3, 5, 11, 12, 21, 23–27, 31, 34–39, 42–44, 53, 57, 58, 62

DSL Eine für einen speziellen Anwendungsfall verwendete Programmiersprache. 2, 58, 69, 71

EPL Eine Untergruppen von DSLs, die dazu dient deklarativ die Verarbeitung von Ereignissen zu beschreiben. 16, 17, 21, 22, 27, 29, 37, 38, 49, 58, 59, 61

JSON Ein leichtgewichtiges Datenformat, welches von Menschen leicht lesbar und von Maschinen leicht verarbeitbar sein soll. 11, 21–23, 36, 48, 71

Living Place Ein Smart Home-Labor der *HAW Hamburg*. 11, 31, 69

Maven Ein Buildautomatisierungstool der Apache Foundation. Es ermöglicht das automatische Bauen von Programmen mittels Konfigurationsdateien. Zudem verfügt es über die Möglichkeit, Abhängigkeiten zu definieren und zu beziehen. Der Zweite Aspekt wird von anderen Tools mitverwendet. 47

Message Broker Ein Programm dessen Aufgabe es ist, Nachrichten zwischen verschiedenen Programmen über unterschiedliche Protokolle auszutauschen. Eine Unterart der Middleware. 6–8, 11, 25, 69

Middleware Ein universelles Verbindungsprogramm zwischen verschiedenen Anwendungen. 1–3, 11, 12, 42, 43, 69, 70

Multiagentensystem Eine Umgebung, welche aus mehreren Agenten besteht, die kooperativ ein gemeinsames Problem zu lösen versuchen. iii, 1, 2, 5, 6, 8, 9, 15

Peer-to-Peer Die direkte Kommunikation von Netzwerkteilnehmern untereinander ohne eine zentrale Zwischenkomponente. 7, 11

POSIX Ein Standard zur Beschreibung der Schnittstelle zwischen Betriebssystem und Anwendungsprogramm. 49

Publish Subscribe Ein Softwarepattern für den Nachrichtenaustausch zwischen mehreren Teilnehmern. Nachrichten werden von Quellen nicht direkt an Senken gesendet, sondern in bestimmten Kanälen eines Dienstes veröffentlicht. Teilnehmer, die entsprechende Nachrichten erhalten sollen, abonnieren entsprechende Kanäle. 7, 11, 19, 25

Rapid Prototyping Ein Softwareentwicklungsmodell, welches auf die Entwicklung von Prototypen als Zwischenschritte zum Erreichen des Ziels setzt. 18

RFID Technologie zum Austausch von Daten über Funk. 27, 32

RPC Prozedur zum Aufrufen von Funktionen eines Programms durch andere. 5, 6, 11

Sensor Ein Objekt, welches eine Umgebung messen und quantifizieren kann. 1, 26

Sliding Window Ein Fenster, welches die Anzahl der zu berücksichtigenden Ereignisse für CEP beschränkt. 16, 19

Smart Environment Eine Umgebung mit vernetzten automatisierbaren Geräten und Computern. 1, 2, 5, 8–11, 14, 21, 24, 28, 31, 57, 59, 61, 62, 71

Smart Home Ein Haushalt mit Smart Environment Fähigkeiten. 9, 26, 70

Smart Object Ein Objekt welches die Fähigkeit besitzt mit anderen Smart-Objekts zu kommunizieren und mit seiner Umgebung zu interagieren. 55, 62, 69, 71

SQL Eine standardisierte DSL zur Ansteuerung von Datenbanken. 16, 17, 51, 55, 56, 61

Stakeholder Personen, die berechtigtes Interesse an einem Projekt haben, da es sie direkt betrifft. 11, 61

Ubiquitous Computing Der Einsatz von Computern in allen Gegenständen im Gegensatz zum traditionellen Ansatz, Computer als separate Geräte anzusehen. 9

Wearable Ein Kleidungsstück, welches ein Smart Object ist. 62

YAML Eine robuste Datenbeschreibungssprache, die eine Obermenge zu JSON darstellt. 48

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

Hamburg, 17. November 2016 Alexander M. Sowitzki