Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Projektbericht Sommersemester 2010

Kjell Otto, Sören Voskuhl

Entwicklung einer Architektur für den Living Place Hamburg

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Inhaltsverzeichnis

1 Einleitung

> „Designing interactive computer systems to be efficient and easy to use is important so that people in our society may realize the potential benefits of computer-based tools. [...]“ (Card u. a., 1983)

Die in diesem Zitat erwähnte Effizienz und Benutzbarkeit von Computersystemen spielen bis heute eine tragende Rolle bei der Entwicklung von Informationssystemen. Diese Systeme werden immer kleiner und leistungsfähiger, sodass der Mensch in nahezu allen Lebenslagen von Computern umgeben ist. Die damit einhergehende allgegenwärtige Rechenleistung wird von Mark Weiser als „Ubiquitous Computing“ beschrieben (Weiser, 1991).
Ausgehend von dieser Entwicklung in der Computerforschung wird an der HAW Hamburg seit über drei Jahren an einem Projekt gearbeitet, welches untersucht, inwiefern dem Menschen das alltägliche Leben in seinem Wohnbereich erleichtert werden kann, in dem die Computer ihn bei alltäglichen Aufgaben unterstützen oder sich seinen persönlichen Präferenzen anpassen. Inspiriert von dieser Vision, wird an der HAW mit dem „Living Place Hamburg“ eine Wohnung aufgebaut, in der die in verschiedenen Projekten entwickelten Systeme und Interaktionstechniken unter realen Bedingungen getestet und bewertet werden können. Ein Grundriss dieses Wohnbereichs wird in Abbildung 1.1 präsentiert.
Mit dem Ziel diese Anwendungen fundiert zu bewerten, wird ein breites Spektrum an Sensoren eingesetzt, deren Daten aufbereitet und interpretiert werden, um den aktuellen Kontext in die Reaktion der Computer einfließen zu lassen. Für den Einsatz dieser „Context-Aware-Systeme“ ist es erforderlich, dass die Computer Informationen über ihre Umwelt sammeln und diese den verschiedenen Teilnehmern zur Verfügung stellen. Bei der Bereitstellung dieser Daten muss sichergestellt werden, dass alle beteiligten Geräte eine identische Sicht auf die Kontextdaten haben, sodass jede Reaktion eines Teilsystems zum aktuellen Gesamtkontext passt.
Aus diesem Grund wird sich diese Projektausarbeitung intensiv mit der Bereitstellung von Informationen in einer intelligenten Umgebung beschäftigen. Die Ziele dieser Arbeit werden im folgenden Abschnitt genauer betrachtet.

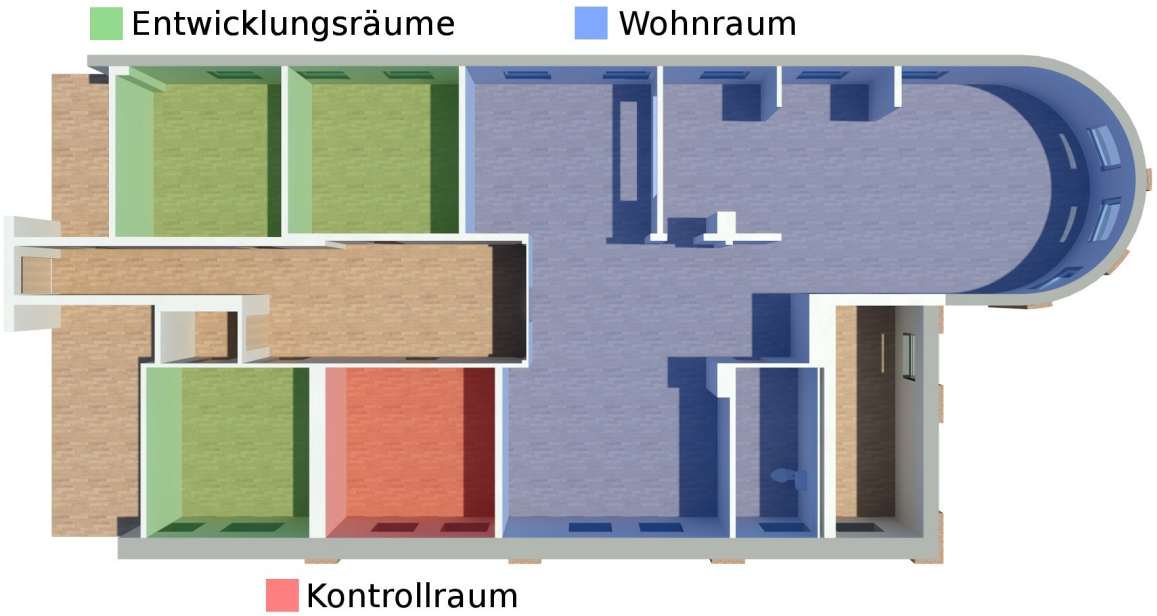

Abbildung 1.1: Grundriss des Living Place Hamburg

Ziele

In diesem Projekt soll eine Basis für die Architektur des Living Place Hamburg geschaffen werden. Hierbei liegt der Fokus im ersten Schritt in der Bereitstellung einer Kommunikationsschnittstelle, über welche die einzelnen Teilsysteme ihre Daten austauschen können.
Da die Anzahl der Systeme im Laufe der Zeit stetig wachsen wird, ist es essentiell, dass diese Schnittstelle eine hohe Anzahl an Daten verwalten und zuverlässig ausliefern kann. Aus diesem Grund soll in diesem Projekt die bisherige Kommunikationsbasis des Living Place bewertet und ggf. ersetzt werden.
In einem weiteren Schritt soll das mögliche Erscheinungsbild der Gesamtarchitektur ermittelt werden. Hierzu werden verschiedene Gespräche notwendig sein, damit die Anforderungen an eine Architektur eines intelligenten Wohnbereichs identifiziert und frühzeitig in die Planung integriert werden können. Grundlage dieser Anforderungen sollen die bisher existierenden Systeme sowie die Erkenntnisse der Studierenden im Laufe des Projektes sein.

2 Projektarbeit

In diesem Kapitel sollen die erzielten Arbeitsergebnisse des Projektes vorgestellt werden. Dabei wird im ersten Teil die Kommunikationsschnittelle präsentiert, während sich der zweite Abschnitt mit der Gesamtarchitektur des Living Place beschäftigt.

2.1 IROS Eventheap

Zu Beginn des Projektes wurde der IROS Eventheap (Johanson und Fox, 2002) als Kommunikationsbasis für die verschiedenen Applikationen innerhalb des Living Place verwendet (Stegelmeier u. a., 2009). Der Eventheap ermöglicht mittels einer Blackboard-Architektur das Verwalten auftretender Events heterogener Geräte, mit dem Ziel diese interessierten Applikationen zur Verfügung zu stellen. Hierzu abonnieren die Clients für sie relevante Events, woraufhin sie vom Eventheap benachrichtigt werden, sobald ein System ein entsprechendes Ereignis auf das Blackboard schreibt. Die einzelnen Events werden hierbei als Tupel implementiert, welche jeweils ein Schlüssel-Werte Paar darstellen.
Da der Eventheap vollständig Java-basiert ist, gestaltete sich die Integration von Applikationen anderer Programmiersprachen als schwierig. Hierzu mussten beispielsweise verschiedene Adapter geschrieben werden, über welche die Kommunikation mit dem Blackboard mit Hilfe von Sockets realisiert werden konnte. Aufgrund der verschiedenartigen Anwendungen im Living Place, stellte diese Eigenschaft des Eventheaps und der damit verbundene Programmieraufwand ein großes Problem dar. Außerdem hat die bereits seit einigen Jahren eingestellte Weiterentwicklung dieses Systems (letzte verfügbare Version stammt vom 26.3.2004) uns dazu veranlasst, einige Performance-Tests mit dem Eventheap durchzuführen. Mit der Absicht eine verlässliche Aussage darüber treffen zu können, ob der IROS Eventheap den Anforderungen dieses Projektes gerecht wird, haben wir verschiedene Testszenarien entwickelt. Die Ergebnisse dieser Tests sollen im Folgenden aufgezeigt und diskutiert werden.

Performance Test

Da die Anzahl entwickelter Systeme innerhalb des Living Place Hamburg stetig wächst

Time-To-Live	Sender	Empfänger	Datenverlust
TTL = ∞	60000 Events	3600 Events	94%
TTL = 500ms	60000 Events	57300 Events	4,5%

Tabelle 2.1: Gesendete und empfangene Events

und diese neu integrierten Teilsysteme mit verschiedenen, heterogenen Sensoren arbeiten, ist davon auszugehen, dass zukünftig die Zahl der Sensordaten, welche vom Blackboard verwaltet werden müssen, weiterhin steigt. Aus diesem Grund sollte der hier vorgestellte Performance-Test Aufschluss darüber geben, wie viele Events innerhalb eines Zeitintervalls zuverlässig vom IROS Eventheap an die Abonnenten ausgeliefert werden.

Im ersten Schritt sollte analysiert werden, wie viele Events vom Eventheap pro Sekunde entgegengenommen werden können bzw. wie viele an die interessierten Systeme versendet werden. Zu diesem Zweck wurde ein Java-Programm entwickelt, das dauerhaft Events versendet und ein weiteres Programm, welches dieses Ereignis abonniert.
Im Laufe dieses Tests konnte beobachtet werden, dass die Anzahl an entgegengenommenen Events die Zahl der versendeten Nachrichten erheblich übersteigt. Während der Eventheap mit ca. 1800 angenommenen Nachrichten pro Sekunde ein zufriedenstellendes Ergebnis lieferte, ergab der Test mit ca. 30 versendeten Nachrichten pro Sekunde ein unzureichendes Ergebnis.
Das Resultat dieses Tests ließ darauf schließen, dass bei einer so hohen Differenz zwischen Senden und Empfangen auf dem Blackboard, die Empfänger die meisten Nachrichten viel zu spät übermittelt bekommen würden. Dies würde insbesondere dann gelten, wenn Sensoren sehr viele Daten generieren und auf den Eventheap schreiben. Bei der Architektur für einen intelligenten Wohnbereich ist es essentiell, dass einem Teilsystem das Auftreten eines Ereignisses möglichst ohne Zeitverzögerung mitgeteilt wird, damit es direkt auf den aktuellen Kontext reagieren kann. Umso kleiner der zeitliche Abstand zwischen dem Auftreten eines Events und der Reaktion des Systems ist, desto flüssiger gestaltet sich die Interaktion zwischen dem Benutzer und dem System, was eine gute Benutzbarkeit zur Folge hat.
Zur Lösung dieses Problems haben wir uns dazu entschieden, den einzelnen Events eine Time-To-Live (TTL) zu geben. Dies bedeutet, dass Nachrichten, die nach einem festgelegten Zeitraum noch nicht ausgeliefert wurden, nicht weiter betrachtet und somit gelöscht werden.
In der Tabelle 2.1 wird ein weiterer Testdurchlauf dargestellt. In diesem Test wurde beobachtet, wie zuverlässig Events bei hoher auftretender Anzahl ausgeliefert werden. Dabei wird deutlich, dass ohne eine TTL ein erheblich größerer Anteil an Ereignissen verloren geht als bei einer TTL von 500ms. Der hohe Datenverlust ohne eine TTL ist darauf zurückzuführen, dass bei einer hohen Belastung des Systems neue Nachrichten nicht beachtet werden.

Bewertung

Aufgrund der durchgeführten Tests lässt sich darauf schließen, dass der IROS Eventheap den Anforderungen nicht genügt, sobald große Datenmengen an die Clients übermittelt werden müssen. Da im Living Place Hamburg sehr viele Daten zwischen den Kommunikationspartnern ausgetauscht werden müssen, würde ein Einsatz des Eventheaps zu Einschränkungen der Performance führen. Zwar lässt sich der Datenverlust durch die Verwendung einer Time-to-live minimieren, allerdings sind wir nach mehreren Gesprächen zu dem Entschluss gekommen, dass die Verantwortung bei dieser Vorgehensweise für den einzelnen Entwickler sehr hoch ist. Sollte die Belegung einer TTL für ein Event vergessen werden, könnte das schwerwiegende Auswirkungen auf alle beteiligten Teilsysteme haben, da evtl. ein großer Teil der Events nicht mehr bei dem Empfänger ankommt. Des Weiteren ist es in einem intelligenten Wohnbereich elementar, dass keine Events im Netzwerk verloren gehen, weil beispielsweise sicherheitskritische Teilsysteme nicht reagieren können, wenn sie durch den Datenverlust den Kontextwechsel nicht wahrnehmen.
Als Konsequenz dieser Aspekte haben wir entschieden, dass wir eine umfassende Recherche bezüglich der auf dem Markt befindlichen Blackboard-Systeme betreiben sollten. Die Ergebnisse und Anforderungen unserer Suche werden im Folgenden Abschnitt vorgestellt.

2.2 Alternative Blackboard-Architekturen

Bevor wir konkret in die Recherche nach geeigneten Systemen eingestiegen sind haben wir unsere Anforderungen an ein solches System, welche aus verschiedenen Diskussionen und den Erfahrungen mit dem IROS Eventheap stammen, zusammengetragen.

Anforderungen

Die Anforderungen an eine Kommunikationsschnittstelle sind zum großen Teil durch die Anwendungen, die in diesem Umfeld erstellt werden, vorgegeben. Da viele Sensoren, die permanent Messwerte publizieren, zum Einsatz kommen, ist eine Anforderung die schnelle und einfache Kommunikation von Datenpaketen. Da die Sensordaten zu einem möglichst realistischen Abbild der Umwelt auf das System beitragen sollen, muss die Übertragung dieser echtzeitnah sein. Dieser Anspruch auf eine echtzeitnahe Übermittlung, ist in diesem Fall als „soft“ zu betrachten. Die zweite Anforderung stammt aus der Kommunikation zwischen einzelnen Komponenten im System. Die Kommunikationspartner müssen die Möglichkeit haben, sich möglichst infrastrukturunabhängig zu finden und eine Kommunikation aufzubauen. Des Weiteren müssen auch spezielle Nachrichtenformate definierbar sein, um eine individuelle Kommunikation zu ermöglichen. Die vorher verwendete Kommunikationsschnittstelle,

der Eventheap, hat zur Realisierung der meisten Anforderungen wenig beigetragen, siehe 2.1. In der Diskussion über die Anforderungen an die Kommunikationsschnittstelle wurde auch die Dokumentation genannt, um einen einfachen Einstieg in diese zu ermöglichen.

Recherche zu verschiedenen Publisher/Subscriber Systemen

Bei unserer Recherche nach einem geeigneten Nachrichten-Paradigma für den Living Place sind wir schnell auf den Begriff Publisher/Subscriber-Modell gestoßen. Dieser Begriff bezeichnet die Vorgehensweise, wie Nachrichten in einer Blackboard-Architektur übermittelt werden. Zusammengefasst kann dieses Modell so beschrieben werden, dass ein „Producer“ Informationen auf einem Software-Bus (Event-Manger) veröffentlicht und ein Consumer die Informationen, die er von diesem Bus erhalten möchte, abonniert (Eugster u. a., 2003). Die betrachteten Systeme dieser Art sollen im Folgenden auf ihre Einsetzbarkeit überprüft werden.
Message Broker Systeme dienen der Vermittlung von Nachrichten zwischen Kommunikationspartnern. Sie ermöglichen verschiedene Kommunikationsmuster, wie z.B. Publisher/Subscriber und Producer/Consumer. Dennoch wurde auch ein Data Distribution Service (DDS) in Betracht gezogen. Wir haben folgende Systeme in ihren Eigenschaften gegeneinander abgewogen:

1. **RTI-DDS**
   Der Data Distribution Service (DDS) von Real Time Innovations (RTI)[1] ist eine Implementation des Object Management Group (OMG) DDS Standards. Dieser Standard wurde zur datenzentrierten Kommunikation entwickelt und legt ein einheitliches Datenformat mit der Beschreibung durch Inter Domain Language (IDL) Dateien fest. Die Implementation von RTI ist in C++ realisiert und bietet dem Entwickler Java-Bindings an. Die Dokumentation ist mit einem ausführlichen Tutorial versehen und aktuell. Gegen diese Lösung spricht der hohe Kostenaufwand bei der Anschaffung der Software und die geschlossenen Quelltexte.

2. **RabbitMQ**
   Der RabbitMQ[2] stammt von der Rabbit Technologies Ltd. und basiert auf dem Advanced Message Queuing Protocol (AMQP). Dieses Protokoll wurde entwickelt, um eine robuste und schnelle Kommunikation für das Message Passing zu gewährleisten. RabbitMQ ist ein Message-Broker System, das mit Erlang implementiert wurde. RabbitMQ bietet ein breites Spektrum an Programmiersprachen für den clientseitigen Zugriff, wie zum Beispiel: Ruby, Python, Java und .Net. Die Quelltexte sind offen und zugänglich. Das größte Argument gegen diesen Message-Broker war die Implementation in

[1] http://www.rti.com/
[2] http://www.rabbitmq.com/

Erlang, einer Sprache die keiner von uns beherrscht und uns somit die Möglichkeit nimmt, tief in das System einzugreifen.

3. **ActiveMQ**
   Der ActiveMQ[3] ist ein Message-Broker der Apache Foundation. Dieses System wurde in Java implementiert und bietet neben einer ausführlichen Dokumentation auch die benötigte Geschwindigkeit, siehe 2.3. Der Zugriff auf den ActiveMQ kann mittels Ruby, Python, C, C++, C#, Java oder anderer Sprachen realisiert werden. Er unterstützt Java Message Service (JMS) in der Version 1.1 sowie den J2EE 1.4. Der ActiveMQ unterstützt verschiedene Transportprotokolle wie z.B. in-VIM, TCP, SSL, NIO, UDP, Multicast und andere. Man kann auch über ein Webinterface auf den ActiveMQ zugreifen und ihn mittels Representational State Transfer (REST) über URIs ansprechen.
   Die Entscheidung fiel auf den ActiveMQ Message-Broker, weil die Dokumentation umfangreich, die Geschwindigkeit hoch und die Implementation in Java ist. Wir sind durch die offenen Quellen in der Lage, das System an einer beliebigen Stelle den Ansprüchen des Living Place Hamburg anzupassen.

2.3 ActiveMQ

Installation des ActiveMQ

Die Installation des ActiveMQ ist nach dem herunterladen und entpacken abgeschlossen. Die entpackten Binaries können direkt ausgeführt werden. Man kann zusätzlich Autostartskripts erstellen und unter /etc/init.d/ in das Serversystem einhängen. Die Einstellung des ActiveMQ wird mit einer Beans-XML-Konfiguration realisiert. Eine Konfigurationsdatei mit der man alle Standardeinstellungen vorgegeben bekommt, ist bereits in der Installation vorhanden. Es sind auch Beispielkonfigurationen für Skalierbarkeit, Sicherheit und die Anbindung von externen Datenbanken vorhanden. Für erste Tests übergibt man dem Startskript keine Parameter und die systemweite Konfiguration wird verwendet. Um eine individuelle Konfigurationsdatei zu verwenden, wird der absolute Pfad am Ende des Kommandozeilenaufrufs übergeben. Über diese Datei kann man auch indirekt gekoppelte Instanzen des Systems konfigurieren. Die Persistenz des ActiveMQ wird mit einer Datenbank realisiert, der sog. KahaDB[4]. Diese Datenbank wurde speziell für die Anwendung im ActiveMQ entwickelt und wird auch über die XML-Datei eingestellt und parametrisiert.
Generell werden für den ActiveMQ einige Einstellungsdateien angeboten, um verschiedene Schwerpunkte hervorzuheben. Die Standardkonfigurationen sind im conf/ Ordner und zeigen exemplarisch Konfigurationen für:

[3] http://activemq.apache.org/
[4] http://activemq.apache.org/kahadb.html

Time-To-Live	Sender	Empfänger	Datenverlust
TTL = ∞	60000 Events	60000 Events	0%

Tabelle 2.2: Gesendete und empfangene Events über den ActiveMQ

- Datenbanken über JDBC-Adapter um z.B. eine MySQL-Datenbank zur Persistenz zu verwenden
- Authentifizierungsmechanismen über Benutzerrollen und Gruppen
- Skalierbarkeit durch das Ausschalten von optimiertem Dispatching von Nachrichten

Erste Tests mit ActiveMQ

Nach dem Entpacken, Einstellen und Ausführen des ActiveMQ sollte dieser getestet werden. Zu diesem Zweck liefert er Tools mit, die man für Tests verwenden kann und ist direkt über das Webinterface erreichbar. Zunächst haben wir mit dem Webinterface einzelne Topics erstellt und Nachrichten versendet. Sollte man ggf. eine Testnachricht verschicken wollen, ohne einen eigenen Client zu programmieren, ist diese Möglichkeit ideal. Der ActiveMQ sollte einen wesentlichen Geschwindigkeitsvorteil gegenüber dem Eventheap bringen und dazu haben wir den Lasttest vom Eventheap auf dem ActiveMQ wiederholt. Dabei haben wir keine TTL einstellen müssen, um bessere Ergebnisse als bei dem Eventheap zu erzielen, wie die Tabelle 2.2 zeigt.

Der ActiveMQ sollte als zentrale Komponente des Living Place Hamburg überwacht werden. Sobald der ActiveMQ ausfallen sollte, wird die Kommunikation, ohne dass wir weitere Vorkehrungen getroffen haben, zusammenbrechen. Zum Thema Skalierbarkeit des ActiveMQ nehmen wir im Ausblick Stellung. Zur Überwachung des ActiveMQ haben wir verschiedene Monitoring Programme untersucht:

AMon

Der AMon ist ein „third party“- Monitoring Programm der Firma Total Transaction Management, das dem Systemadministrator einen Überblick über den ActiveMQ verschafft, vgl. (TTM, 2009). Das AMon-Monitoringsystem hat zwei Mechanismen, mit denen der Message Broker überwacht werden kann. Zum einen den sog. „Monlet-Container“ und zum anderen den SNMP-Agent. Der Monlet-Container ist eine programmierbare Schnittstelle zum Anzeigen von z.B. Queue-Füllständen oder durchschnittlichem Nachrichtendurchsatz in Topics, aufgrund von Schwellwerten oder programmierbarer Ereignisse. Dabei wird der

ActiveMQ über EventMessages mit den Monlets verbunden und die Nachrichten in einen Ereigniskontext gesetzt, um speziell angepasste „Monitore“ zu entwickeln.

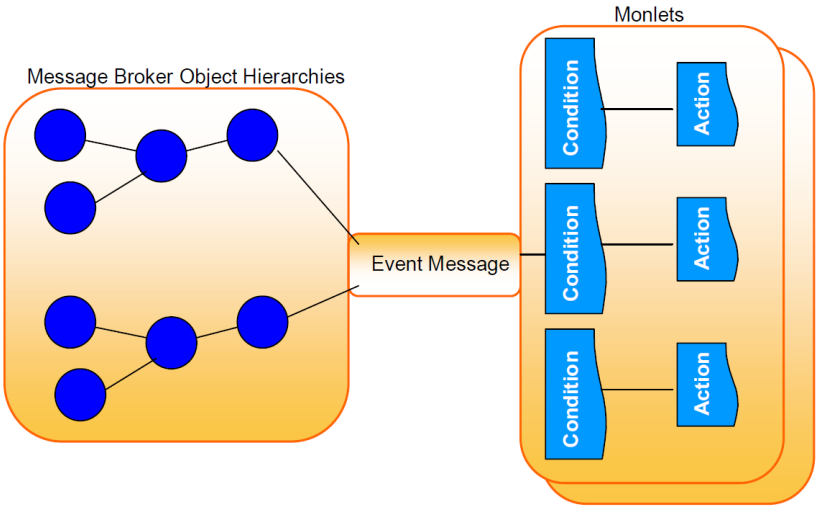

Abbildung 2.1: AMon Monlets und Message Event (TTM, 2009)

Der SNMP-Agent dient dem Überwachen der Systemressourcen des Brokers und kann eingesetzt werden, um den Message-Broker an ein SNMP-fähiges externes Monitoring System anzuschließen. Dieses Tool ist eine kommerzielle Lösung und wir haben noch keine Testlizenz angefordert. Dennoch ist dieses Tool eventuell interessant, um die Integration in eine vollständige Systemübersicht zu realisieren.

JMeter

Das JMeter[5] ist ein weiteres Projekt der Apache Foundation und dient dem Testen von Softwarekomponenten. Dieses Werkzeug kann eingesetzt werden, um automatische Testszenarien zu generieren und den Systemstatus unter Last zu prüfen. Die Installation des JMeter verlief nach Anleitung reibungslos, nur waren wir nicht in der Lage, das Tool mit unserer ActiveMQ-Instanz interagieren zu lassen. Nach mehreren Anläufen bezüglich der Einrichtung des JMeter, haben wir dieses Tool nicht weiter in Betracht gezogen.

JConsole

Die JConsole[6] ist Bestandteil der J2SE Plattform und stellt das Standardwerkzeug zum

[5] http://jakarta.apache.org/jmeter/
[6] http://java.sun.com/developer/technicalArticles/J2SE/jconsole.html

Monitoren von Java Applikationen dar. Es dient der Überwachung aller Ressourcen einer Java Virtual Machine (JVM) und bietet die Möglichkeit diese auch remote, über Java Management Extensions (JMX), zu überwachen. Die JConsole wurde im Living Place getestet und bietet eine detaillierte Übersicht der JVM, auf dem der ActiveMQ läuft. Dennoch eignet sich die JConsole nicht ohne Modifikation zur Integration in eine Komplettlösung, denn sie bietet keine SNMP Anbindung.

Einarbeitung in ActiveMQ

Für einen tieferen Einblick in den ActiveMQ muss man dessen API selbst benutzen und eigene Clients programmieren. Wir haben mit Java begonnen und ein Minimalbeispiel direkt aus der Dokumentation entwickelt. Der Teil der für Publisher und Subscriber identisch bleibt, ist hier gekennzeichnet:

```
0  ActiveMQConnectionFactory connectionFactory = new ActiveMQConnectionFactory("tcp://192.168.14.2:61616");
   TopicConnection connection;
   boolean transacted;

   try {
5      connection = connectionFactory.createTopicConnection();
       connection.start();
       System.out.println("Connected." + format.format(new Date(System.currentTimeMillis())));

       TopicSession newSession = connection.createTopicSession(transacted, Session.AUTO_ACKNOWLEDGE);
10     Topic topic = newSession.createTopic("PerformanceTestTopic");
       ...
```

Ab dem Erstellen eines Topics mit der Session, unterscheiden sich Publisher und Subscriber in der Art, dass die Instanzen unterschiedlich von der Factory geliefert werden. Hat man seine TopicPublisher bzw. TopicSubscriber Instanz definiert, kann man beginnen, Nachrichten zu versenden. Es gibt verschiedene Arten von Nachrichten, unter anderem die TextMessage und die ObjectMessage. Besonders die TextMessage wird im Living Place eingesetzt, da der Nachrichtenaustausch im JSON-Format eine Serialisierung voraussetzt und den Kommunikationsteilnehmern die Sprachunabhängigkeit bietet. Das JSON-Format der Nachrichten wurde wie folgt definiert[7], um Applikationsversionen verwaltbar und eventuell nötige Änderungen an vorhandenen Applikationen unnötig zu machen:

Ein Request für einen „Publisher“, wie er sie über seine „Pull“-Queue empfängt, hat mindestens eine Versionsnummer und eine eindeutige ID der Applikation, welche dieser Request erstellt hat. Die Version soll verwendet werden, um alte „Publisher“-Versionen weiterhin verwenden zu können ohne den „Subscriber“, der eventuell nicht geändert wurde, weiter betreiben zu können. Ein „Publisher“ muss also bei einer Aktualisierung des Nachrichtenformats darauf achten, dass er seine älteren Nachrichtenformate auch weiterhin unterstützt, um die

[7] http://livingplace.informatik.haw-hamburg.de/wiki/index.php/Nachrichtenformat

Funktion von älteren „Subscribern“ nicht zu beeinträchtigen. Die ID wird von der Clientseite verwendet, um eine Nachricht die dem „Subscriber“ über das Topic zugesandt wird, als Antwort auf seinen Request zu identifizieren. Der „Publisher“, der eine Response Nachricht erstellt, liefert diese über das Topic an alle Subscriber aus und muss wenigstens die Felder des Requests unverändert übernehmen und ein Statusfeld mitliefern. In diesem Statusfeld wird eine Auskunft über die Antwort gegeben.

Wir haben an dieser Stelle die Verantwortung über die Pflege des Nachrichtenformats an die Applikationsentwickler weitergegeben, weil es keine Möglichkeit gibt, den Quelltext von allen Applikationen mit einer zentralen Instanz zu warten. Dennoch wird von Entwicklern erwartet, dass in der Wikiseite ihrer Applikation eine ausführliche Beschreibung zu der Nachrichtenversion und dem zugehörigen Format erstellt wird.

Um die Interoperabilität zu gewährleisten, haben wir auch mit C# und Python Clients programmiert, um zu sehen, wie der ActiveMQ mit verschiedenen Programmiersprachen angesprochen werden kann. Die Snippets für Minimalclients sind auf der Wikiseite für die Kommunikationsschnittstelle zu finden[8].

Architektur des ActiveMQ

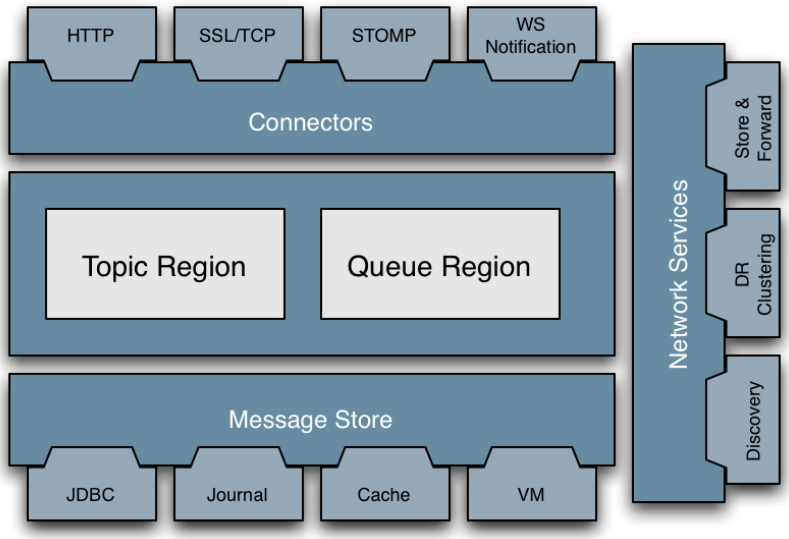

Abbildung 2.2: Architektur des ActiveMQ (The Apache Software Foundation, 2010)

[8] http://livingplace.informatik.haw-hamburg.de/wiki/index.php/Kommunikationsschnittstelle

Der ActiveMQ Message-Broker bietet zwei Mechanismen, die auf unterschiedliche Art und Weise zur Kommunikation eingesetzt werden können. Die „Connectors“ dienen als Zugriffsschnittstelle auf diese zwei Mechanismen und bieten Zugriff über verschiedene Plattformen, vgl. Abbildung 2.2. Der „Message Store“ stellt die Persistenzschicht dar und kann neben der KahaDB als Standarddatenbank auch MySQL und andere Datenbanken bedienen. Die Network Services bieten Skalierbarkeit auf Serverebene und abstrahieren diese, sodass sich die Funktionalität des Brokers für Clients nicht verändert. Die Kommunikationsmechanismen, „Topic Region“ und „Queue Region“ werden im Folgenden in ihrer Funktion vorgestellt:

- **Message-Queues**
  Eine Message-Queue aus dem Kontext des ActiveMQ funktioniert so wie die bekannten Queues auch, nur dass beim ActiveMQ mehrere Kommunikationsteilnehmer beide Enden der Queue verwenden können. Es können also mehrere „Producer“ (Nachrichten produzierende Kommunikationsteilnehmer) Messages an die Queues senden und mehrere „Consumer“ (Nachrichten konsumierende Kommunikationsteilnehmer) Messages aus der Queue konsumieren. Der ActiveMQ speichert Nachrichten, die zwar von einem „Producer“ an die Queue gesendet wurden, aber noch von keinem „Consumer“ abgeholt wurden zwischen. Konsumieren mehrere „Consumer“ Nachrichten aus der Queue, so werden die Nachrichten nur einmal ausgeliefert, d.h. eine Nachricht aus einer Queue wird immer nur ein „Consumer“ erhalten. Somit ist dieser Mechanismus als „n-zu-1“-Mechanismus zu verwenden.

- **Topics**
  Die Topics im ActiveMQ Kontext sind der Mechanismus zum automatischen Verteilen von Nachrichten an mehr als einen Kommunikationsteilnehmer. Topics fungieren dabei als eindeutige Namen für Nachrichten, an denen sich mehrere Kommunikationsteilnehmer beteiligen können. „Publisher“ sind die Kommunikationsteilnehmer, die Nachrichten an das Topic senden und „Subscriber“ sind die Kommunikationsteilnehmer, die diese Nachrichten vom ActiveMQ automatisch zugesandt bekommen. Hierbei muss beiden Seiten, „Publisher“ und „Subscriber“ die IP des ActiveMQ bekannt sein und der Name des Topics. Im Gegensatz zur Message-Queue, wird mit diesem Mechanismus jeder der „Subscriber“ eines Topics die Nachrichten aller „Publisher“ zugesandt bekommen. Somit ist dieser Mechanismus als „n-zu-m“-Mechanismus zu verwenden.

Im Living Place wird es Dienste geben, die nicht wie z.B. Sensoren, periodisch Nachrichten versenden müssen, aber dennoch Informationen für viele Interessenten publizieren. Ein Beispiel dafür wäre z.B. der KalenderAgent[9], welcher für viele Applikationen wichtige Informationen über den Bewohner des Living Place bereitstellt, jedoch diese nicht periodisch zur Verfügung stellen sollte.
Der Mechanismus den wir zur Lösung dieser, in den Mechanismen des ActiveMQ nicht

[9] http://livingplace.informatik.haw-hamburg.de/wiki/index.php/Kalender-Agent

vorgesehenen Mechanik entwickelt haben, ermöglicht den „Subscribern“, „Publisher“ nach Nachrichten auf einem Topic zu fragen. Dazu erstellt der „Publisher“ sich eine sogenannte „RequestQueue“, in die „Subscriber“ Anfragen sendet, die den „Publisher“ dazu veranlassen, auf das von ihm zur Verfügung gestellte Topic Antworten zu senden. Dieser Mechanismus entspricht einem Remote Procedure Call aus Java. Im Kontext des Living Place haben wir diesen Mechanismus auch im Wiki[10] als „Pull“-Mechanismus beschrieben.

Migration bisheriger Anwendungen auf den ActiveMQ

Nachdem die anfänglichen Tests mit dem ActiveMQ erfolgreich beendet wurden, sollte im nächsten Schritt der IROS Eventheap vollständig abgelöst werden. Hierzu war es erforderlich, die implementierten Szenarien, welche den Eventheap als Kommunikationsbasis nutzten, soweit anzupassen, dass sie nun ihre Events über den ActiveMQ austauschen. Zunächst wurden exemplarisch drei Szenarien ausgewählt, die im folgenden Abschnitt im Hinblick auf ihre Funktion im Living Place sowie auf benötigte Veränderungen zum Einsatz in der neuen Architektur untersucht werden.

1. **Doorbell**
   Das Szenario der Doorbell beinhaltet in der derzeitigen Implementation eine Software-Türklingel, die ein reales Klingeln an der Haustür oder eine persönliche Nachricht am Computer simulieren kann. Sollte der Bewohner beschäftigt sein, da er beispielsweise gerade beim Fernsehen ist und es an der Tür klingelt, minimiert sich das aktuell aktive Programm und er erhält über eine Webcam das Bild des Besuchers. Wenn es sich nur um eine kurze Mitteilung handelt, kann der Benutzer per Klick wieder ins vorherige Programm wechseln und musste somit für den Erhalt dieser Nachricht weder seinen Platz verlassen noch eine zusätzliche Anwendung öffnen. Sollte sich jemand vor der Tür befinden, den der Bewohner in seinen Wohnbereich bitten möchte, kann er dieses dem System ebenfalls per Tastendruck mitteilen. In diesem Fall schaltet das aktuelle Programm automatisch in den Time-Shift-Modus, sodass der Bewohner zu einem späteren Zeitpunkt das Programm genau an der Stelle weiterführen kann, an der er unterbrochen wurde.
   Da es sich bei diesem System um eine reine Java Anwendung handelt, musste hier der Code, mit dem die Kommunikation mit dem Eventheap realisiert wurde, umgeschrieben werden. Aufgrund der bisherigen Tests mit dem ActiveMQ wurde der Code bedenkenlos ersetzt, sodass die Kommunikation zwischen den beteiligten Komponenten anhand von Topics durchgeführt wird.

[10] http://activemq.apache.org/jmeter-performance-tests.html

2. **XuukDisplay**
   Das elementare Gerät, das für dieses Szenario verwendet wird, ist die eyebox2[11] von Xuuk. Bei diesem Produkt handelt es sich um einen kamerabasierten Eyetracker, in dem mittels Infrarotlicht Pupillen im Kamerabild erkannt werden. Mithilfe dieses Gerätes soll auf dem Display nur dann ein Bild erscheinen, wenn jemand diesen fokussiert. Das bedeutet, dass sich der Monitor ausschalten soll, wenn er sicherstellen kann, dass niemand das aktuelle Bild auf dem Gerät wahrnimmt. Zur Implementierung dieser Anwendung werden die Daten des Eyetrackers ausgelesen und bei jeder Identifizierung von Augen ein Timer gestartet. Sollte der Timer abgelaufen sein, wird davon ausgegangen, dass niemand mehr auf das Display schaut und das aktuelle Bild wird minimiert.
   Damit diese Anwendung über den ActiveMQ kommuniziert, mussten die gleichen Veränderungen am Code vorgenommen werden, wie bereits beim Doorbell-Szenario, da es sich hier ebenfalls um eine ausschließlich auf Java-basierende Anwendung handelt.

3. **Location-based Screen**
   Innerhalb eines Wohnbereichs oder eines Bürokomplexes ist es denkbar, dass der Benutzer seinen aktuellen Standort verlässt, mit der Absicht kurzzeitig einer anderen Tätigkeit nachzugehen. Jedoch kann es vorkommen, dass er trotz seines Standortwechsels seine aktuellen Anwendungen wie z.B. eine Videokonferenz oder das Fernsehbild nicht verlieren möchte. Für diesen Fall wurde dieses dritte Szenario entwickelt. Hier wird über das Echtzeit-Ortungssystem der Firma Ubisense[12] die aktuelle Position der Person ermittelt und an alle im System verfügbaren Displays übermittelt. Da jedes Teilsystem seinen eigenen Standort kennt, interpretiert es die ankommenden Daten, um ggf. die Aufgabe des aktiven Bildschirms zu übernehmen. Dabei hat man sich bei der Realisierung dieses Szenarios bisher auf das Fernsehbild beschränkt, jedoch ist im nächsten Schritt geplant, beliebige Anwendungen in dieses System zu integrieren. Die Umstellung dieser Anwendung auf den ActiveMQ hat einige Vorteile mit sich gebracht. Zuvor musste mit einem Adapter gearbeitet werden, der die Daten des Lokationssystems an den Eventheap gesendet hat, da die als Dynamic Link Library (DLL) mitgelieferte Bibliothek keine Java-Bindings vorgibt und somit nicht direkt mit der Kommunikationsschnittstelle arbeiten konnte. Da der ActiveMQ u.a. eine Sprachunterstützung für C# bereitstellt, konnte im Zuge der Umstellung, der Adapter abgelöst werden und die Kommunikation direkt über das Topic realisiert werden.

[11] https://www.xuuk.com/Products.aspx
[12] http://www.ubisense.de/

2.4 Architektur des Living Place Hamburg

Nachdem wir uns für den ActiveMQ als Kommunikationsschnittstelle entschieden hatten, wurde in verschiedenen Gesprächen darüber diskutiert, wie eine Gesamtarchitektur des Living Place aussehen könnte. Hierbei konnten wir von unseren Erfahrungen aus dem vorherigen Semester (Otto (2010), Voskuhl (2010)) sowie von den Vorarbeiten vergangener Semester profitieren (u.a. Rahimi und Vogt (2010)).
Die Abbildung 2.3 beschreibt einen ersten Entwurf, wie die einzelnen Teilsysteme mit dem Blackboard kommunizieren und um welche Funktionalitäten der ActiveMQ erweitert werden müsste, damit er unseren Anforderungen genügt.

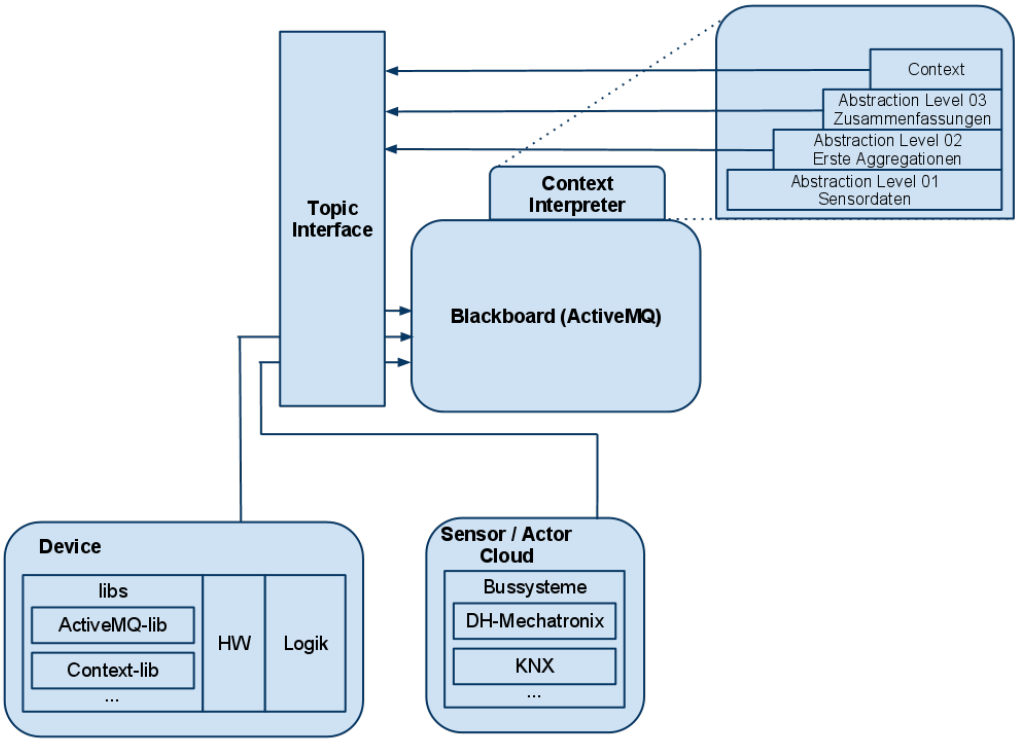

Abbildung 2.3: Kommunikation der Geräte mit dem Blackbaord

Im ersten Schritt wurde betrachtetet, wie die einzelnen Geräte innerhalb der Architektur auftreten. Hierbei wurde vorerst zwischen Devices und einzelnen Sensoren unterschieden. Während ein Device eine interne Logik besitzt und über verschiedene Libraries direkt über eine TCP/IP Verbindung auf das Blackboard zugreift, muss der Zugriff der Sensoren separat betrachtet werden. Mit der Vielfalt der Sensoren im Living Place geht eine Heterogenität der dazugehörigen Bussysteme einher. Dennoch muss sichergestellt werden, dass jede

Komponente ihre Informationen auf den AcitveMQ schreiben kann. Durch Absprachen mit anderen Projektteilnehmern konnten wir ermitteln, dass diese Problematik mit Hilfe von Adaptern unterstützter Sprachen gelöst wurde, welche die empfangenen Daten an das Blackboard weiterleiten.
In dieser Architektur soll die Kontexterstellung nicht den einzelnen Geräten überlassen werden, da diese Vorgehensweise eine hohe Rechenbelastung für die Komponenten bedeuten würde. Des Weiteren können die jeweiligen Teilsysteme nicht sicherstellen, dass sie jede vorhandene Kontextinformation in ihre Berechnung einbeziehen. Aus diesen Gründen wird in dieser Architektur ein „Context Interpreter“ eingesetzt. Diese Komponente versucht neu gelieferte Informationen mit bereits bestehenden in Verbindung zu setzen, mit dem Ziel den Kontext ein weiteres Stück zu konkretisieren. Somit gibt es in dieser Architektur eine Instanz, die zu jedem Zeitpunkt den aktuellen Zustand der Kontexterkennung kennt.
Damit den Geräten ein Gesamtkontext geliefert werden kann, muss der „Context Interpreter“ eine Sensor-Fusion der einzelnen Sensordaten vornehmen. Zur Realisierung dieser Aufgabe ist eine Aufteilung des Interpreters in mehrere „Abstraction Level“ geplant. Dabei repräsentiert jede Ebene den Fortschritt der Kontexterfassung. Ein geringes Level sollen die unverarbeiteten Sensordaten erhalten, da aus ihnen kein Kontext abgeleitet werden kann. Nach verschiedenen Aggregationen könnte sich aber herausstellen, dass sich jemand in der Küche befindet. Da diese Kontextinformation bereits Daten enthält, die eine bedeutende Rolle im Gesamtkontext spielen, wird diese Information auf einer höheren Ebene eingestuft.

Im weiteren Verlauf des Projekts haben wir uns intensiv mit der Kontexterfassung sowie dessen Repräsentation befasst. Die Abbildung 2.4 zeigt, wie zukünftig die Nachrichten des ActiveMQ in die Kontexterfassung einfließen werden.
Wenn eine Nachricht auf den ActiveMQ geschrieben wird, wird diese im ersten Schritt an den „Message Interpreter“ weitergeleitet. Dieser hat nun die Aufgabe, das erhaltene Event in eine Datenbank zu schreiben, damit alle Nachrichten in die von uns entwickelte Persistenzebene transportiert werden. Die in 2.3 erwähnte KahaDB dient dazu, bei einem Ausfall des ActiveMQ die bisher existierenden Queues und Topics sowie deren bisher nicht ausgelieferte Nachrichten wiederherzustellen. Die in der Abbildung beschriebene Datenbank soll dagegen die Aufgabe übernehmen, alle Nachrichten, die auf das Blackboard geschrieben werden, zu sichern. Diese Sicherung erfolgt mit dem Ziel, vergangene Events, wie beispielsweise die letzten Aufenthaltsorte der Person, erneut abfragen zu können. Zusätzlich könnten Lernverfahren eingesetzt werden, um aus den vorliegenden Daten Erfahrungen über die Verhaltensmuster einzelner Bewohner zu sammeln. Die in der Datenbank gespeicherten Datensätze können in einem weiteren Schritt dazu verwendet werden, die Ontologie mit Begriffsdefinitionen zu füllen und somit ein Modell für die Kontextdaten zu konstruieren.
Die „View-Abstraction-Layer“ dient zur Unterstützung des „Context Interpreters“, indem er verschiedene Datenbank-Views anbietet, mit denen der „Context Interpreter“ arbeiten kann. Beispielsweise ist es denkbar, dass zur Kontexterstellung die Termine des vorherigen Tages

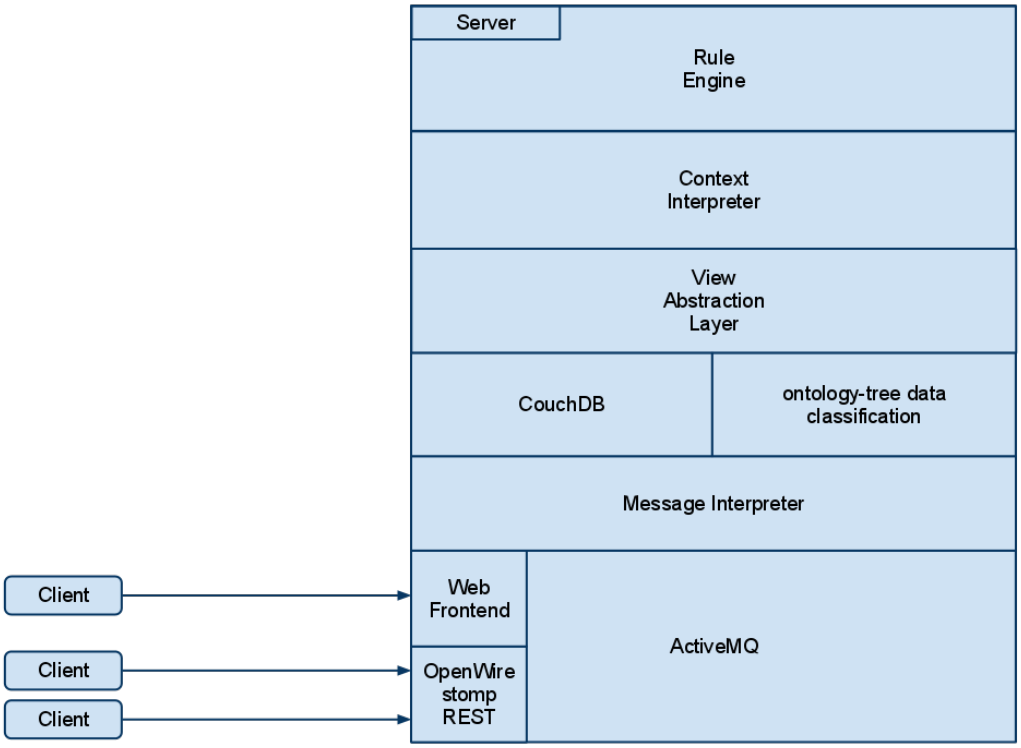

Abbildung 2.4: Repräsentation der Kontexterfassung in einer Schichtenarchitektur

abgefragt werden sollen. Damit keine komplexen Datenbankabfragen generiert werden müssen, bietet die „View-Abstraction-Layer“ eine Reihe vorbereiteter Views an, mit dem Ziel die Integration von neuen Informationen in den Gesamtkontext zu beschleunigen.
Die „Rule Engine“ wird vom „Context Interpreter“ verwendet, um anhand verschiedener Regeln auf Kontexte zu schließen. Diese Regeln können als logische Schlussfolgerungen verstanden werden, bei deren Erfüllung ein Kontext eingetreten ist.
Folgendes Szenario beschreibt, wie die „Rule Engine“ zur Kontextgenerierung beitragen könnte. Der Interpreter erhält die Information „Person X sitzt am Küchentisch“. Ein vorhergegangenes Event besagt, dass die Herdplatte ausgeschaltet wurde. Während der Interpreter die neue Kontextinformation analysiert, erfährt er mittels der „Rule Engine“, dass Person X gerade am Küchentisch isst.

3 Zusammenfassung

Zum Beginn von Projekt 1 haben wir unsere Hoffnungen in den EventHeap gesetzt und durch Besprechungen mit vorangegangenen Semestern, erste Zweifel an dieser Kommunikationsschnittstelle bekommen. Durch Tests mit dem Eventheap hat sich herausgestellt, dass dieser den Anforderungen im Living Place nicht gewachsen ist. Die unzureichende Performance und nicht vorhandene Persistenzschicht haben uns dazu veranlasst, neue Lösungen zu suchen.
Nach der ersten Recherche haben wir uns drei Systeme, RTI-DDS, RabbitMQ und ActiveMQ näher angesehen, um eine möglichst gute Auswahl zu treffen. Durch die hervorragende Dokumentation, Performance und Persistenz des ActiveMQ haben wir uns für dieses Projekt entschieden. Nachdem die Entscheidung besprochen und gefallen war, haben wir den ActiveMQ installiert und mit der Einarbeitung begonnen. Hierzu haben wir verschiedene Konfigurationsoptionen und die Architektur gesichtet und den ActiveMQ für erste Tests konfiguriert. Im nächsten Schritt wurden erste Applikationen mit Java entwickelt, um sich in die Zugriffsmechanismen und die API vom ActiveMQ einzuarbeiten. Nach eigenen Lasttests wurde deutlich, dass die Performance vom ActiveMQ deutlich besser ist als die des Eventheap.
Die bestehenden Anwendungen wurden zu Demonstrationszwecken des Forschungsinhaltes migriert und auf den ActiveMQ umgeschrieben. Diese Migration betraf zunächst die Projekte „Doorbell“, „XuukDisplay“ und „Location-based Screen“. Um die Features des ActiveMQ weiter zu prüfen, folgten auch andere Clientapplikationen in den Sprachen C# und Pyhton. Andere Studenten begannen den ActiveMQ erstmals unter realistischen Bedingungen in Betrieb zu nehmen.
Schnell wurde deutlich, dass ein einheitliches Datenformat für die Kommunikation benötigt wird. Wir haben uns für JSON als Textnachrichtenformat entschieden und ein Nachrichtenformat mit Minimalanforderungen entwickelt.

Nachdem andere Projektteams nun in der Lage waren, die zentrale Kommunikationsschnittstelle zu nutzen, haben wir begonnen die Architekturvisionen weiter zu verfeinern. Um die eventuell recht leistungsschwachen Komponenten im Living Place zu entlasten, haben wir unsere Überlegungen zu einer zentralen Architektur weiterverfolgt.
Die Blackboard-Architektur wurde weiter verfeinert und die Möglichkeit der Anbindung von Sensoren und Systemen in Betracht gezogen, die möglicherweise keine LAN Anbindung

haben. Auch die Kontexterkennung und -interpretation spielten eine entscheidende Rolle in den Überlegungen und führten zu der in 2.4 beschriebenen Vision.
Diese Vision wird im Ausblick weiter konkretisiert und es werden Vorschläge zur Integration dieser Architektur im Living Place Hamburg gemacht.

Ausblick

Der Ausblick für das Projekt 1 gibt eine Übersicht über geplante Schritte und zeigt Überlegungen für zukünftige Implementationen und Integrationen auf.

Die Architektur des Living Place besteht im Moment aus dem ActiveMQ als zentraler Kommunikationsschnittstelle und einzelnen Applikationen, die diese nutzen. Der ActiveMQ wird auch weiterhin ein zentraler Bestandteil der Architektur bleiben und muss daher ausfallsicher sein. Um diese Ausfallsicherheit zu gewährleisten, ist erforderlich, eine zweite Instanz des ActiveMQ mit dem aktuellen System zu verbinden oder das aktuelle System so zu konfigurieren, dass es nach einem Ausfall wieder startet und sich selbst in seinem ursprünglichen Zustand wiederherstellt. Der ActiveMQ bietet zum Skalieren von Queueanzahl[1] und -tiefe[2] jeweils eine Möglichkeit an. Sollten in Zukunft Probleme mit der Persistenz des ActiveMQ auftreten, kann der ActiveMQ umkonfiguriert werden, mit der Absicht eine andere Datenbank wie z.B. MySQL zu benutzen. Der Server, der den ActiveMQ betreibt, wird auch den Rest der Architekturüberlegung, wie zum Beispiel den „Message Interpreter“, „Context Interpreter“, die Datenbank etc. betreiben, siehe 2.3. Im Folgenden soll die Realisierung der Komponenten für die zukünftige Projektarbeit umrissen werden.

Konkrete Software für die Dienste, die auf dem ActiveMQ aufsetzen, ist nur teilweise vorhanden, denn das Einsatzgebiet sowie die Vielfalt der Informationen geben Anlass zur Entwicklung von Speziallösungen. Dennoch haben wir mit der Recherche von wiederverwendbaren Systemen begonnen.
Insbesondere für den Message Interpreter, siehe 2.4, wird es keine vorhandene Software geben. Die Software, die Nachrichten für die Einordnung in die Ontologie aufbereitet, soll eine konfigurierbare Abstraktionsschicht darstellen. Diese Abstraktionsschicht muss jeden Nachrichtentyp mit den von ActiveMQ gelieferten Metainformationen über Sender und Empfänger sowie Nachrichteninhalte vorverarbeiten. Die Ontologieeinordnung könnte eine Implementation von Protégé[3] mit einem Adapter zum Message Interpreter, der Datenbank und zur darüberliegenden Schicht, dem View-Abstraction-Layer, sein. Die Datenbank ist

[1] http://activemq.apache.org/scaling-queues.html
[2] http://activemq.apache.org/scaling-the-depth-of-a-queue.html
[3] http://protege.stanford.edu/

nach ersten Anwendungsüberlegungen als „Dokumentenbasierte-Datenbank“ geplant. Als Implementation einer solchen Datenbank haben wir MongoDB[4] und CouchDB[5] in Betracht gezogen. Nach weiteren Recherchen zum Thema „Dokumentenbasierte-Datenbanken“, haben wir uns für CouchDB entschieden[6] und werden diese im Folgeprojekt zum Einsatz bringen. Der „View-Abstraction-Layer“ stellt eine Abstraktionsschicht zum „Context Interpreter“ dar und bietet dieser Komponente eine abstraktere Sicht auf die Daten als die Datenbank ohne Abstraktionsmechanismus dies könnte. Die „Rule-Engine“ wird eine Reaktionsfähigkeit des Living Place implementieren. Dieses regelbasierte System soll eine einfache Möglichkeit zur Integration von Verhaltensmustern der Wohnung darstellen. Eine vielversprechende „Rule-Engine“ ist die JRuleEngine[7], welche basierend auf dem Java Specification Request (JSR) 94 implementiert wurde.

Mehr und mehr Clientapplikationen werden die Kommunikationsschnittstelle benutzen und sich im Living Place ansammeln. Um einen Überblick zu gewährleisten, werden wir im kommenden Projekt eine Monitoring Instanz für den gesamten Living Place installieren. Um alle Sensoren und Anwendungen im Überblick zu haben, bietet sich eine einheitliche Schnittstelle an. Das bevorzugte Protokoll zur Überwachung von PC-Systemen ist das Simple Network Management Protocol (SNMP), auf welches auf Nagios[8] aufbaut. Nagios stellt ein solches System dar und wir haben uns für die Benutzung dieser Software entschieden. Nagios bietet eine Sammlung von Modulen zur Überwachung von Netzwerken, Hosts und speziellen Diensten sowie eine Web-Schnittstelle zum Abfragen der gesammelten Daten. Nagios ist eine freie Software unter der GPL Lizenz und läuft unter zahlreichen unixähnlichen Betriebssystemen, wobei es auch Ansätze zur Integration in Windows gibt.

Des Weiteren soll der Umgang mit dem ActiveMQ in Kombination mit dem JSON-Nachrichtenformat weiter vereinfacht werden. Da alle beteiligten Kommunikationspartner im Living Place Umfeld den ActiveMQ nutzen werden, um zu kommunizieren, bietet sich eine gemeinsame Bibliothek für den vereinfachten Umgang an. Wir planen das nächste Semester mit der Entwicklung einer solchen Bibliothek zu beginnen.

[4] http://www.mongodb.org/
[5] http://couchdb.apache.org/
[6] http://www.korokithakis.net/node/116
[7] http://jruleengine.sourceforge.net/
[8] http://www.nagios.org/

Literaturverzeichnis

[Card u. a. 1983] CARD, Stuart K. ; NEWELL, Allen ; MORAN, Thomas P.: *The Psychology of Human-Computer Interaction*. Hillsdale, NJ, USA : L. Erlbaum Associates Inc., 1983. – ISBN 0898592437

[Eugster u. a. 2003] EUGSTER, Patrick T. ; FELBER, Pascal A. ; GUERRAOUI, Rachid ; KERMARREC, Anne-Marie: The many faces of publish/subscribe. In: *ACM Comput. Surv.* 35 (2003), Nr. 2, S. 114–131. – ISSN 0360-0300

[Johanson und Fox 2002] JOHANSON, Brad ; FOX, Armando: The Event Heap: A Coordination Infrastructure for Interactive Workspaces. In: *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*. Washington, DC, USA : IEEE Computer Society, 2002, S. 83. – ISBN 0-7695-1647-5

[Otto 2010] OTTO, Kjell: Clojure - concurrency revisited. (2010). – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master09-10-aw1/otto/bericht.pdf

[Rahimi und Vogt 2010] RAHIMI, Mohammadali ; VOGT, Matthias: Aufbau des Living Place Hamburg. (2010). – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master09-10-proj/rahimi-vogt.pdf

[Stegelmeier u. a. 2009] STEGELMEIER, Sven ; WENDT, Piotr ; LUCK, Kai von: iFlat - Eine dienstorientierte Architektur für intelligente Räume. (2009), S. 1 – 5. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/papers/aal2009.pdf

[The Apache Software Foundation 2010] THE APACHE SOFTWARE FOUNDATION: *Apache ActiveMQ Architecture*. 2010. – URL http://activemq.apache.org/code-overview.html

[TTM 2009] TTM, Total Transaction Management: AMonUserGuide. (2009). – URL http://www.ttmsolutions.com/Apache_Software/AMonUserGuide.pdf

[Voskuhl 2010] VOSKUHL, Sören: Bereitstellung einer Sensorwolke. (2010). – URL http://users.informatik.haw-hamburg.de/~ubicomp/projekte/master09-10-aw1/Voskuhl/bericht.pdf

[Weiser 1991] WEISER, Mark: *The Computer for the 21st Century*. 1991. – URL http://www.ubiq.com/hypertext/weiser/SciAmDraft3.html