Infrastrukturen und Middleware

Seminar "Smart Environments"

Assistent:
Oliver Kasten

René Müller
muellren@student.ethz.ch

20. Mai 2004

Zusammenfassung

Zu Smart Environments kann eine Vielzahl von Definitionen angegeben werden. In diesem Seminarbeitrag werden Smart Environments im Sinne von interaktiven Räumlichkeiten, so genannten Active Spaces, betrachtet. Als konkretes Anwendungsbeispiel wird ein mit moderner Computertechnik ausgerüsteter Konferenzraum dienen. Der Raum ist mit mehreren Wand-Bildschirmen und einer Vielzahl von Eingabegeräten ausgestattet. Die Hard- und Software-Infrastruktur des Raumes soll den Benutzern erlauben, ihre eigenen Geräte wie Notebook oder PDA einzubinden, so dass Kommunikation unter ihnen möglich wird. Die Geräte sollen sowohl vom Raum angebotene Dienste (wie z.B. Drucker, Terminplaner-Synchronisation unter den anwesenden Teilnehmer) in Anspruch nehmen, als auch selbst Dienste (z.B. Digitalkamera) für andere zur Verfügung stellen.

Zuerst werden die Anforderungen von möglichen Applikationen an die Infrastruktur untersucht. Anschliessend werden zwei konkrete Realisierungen vorgestellt und miteinander verglichen. Bei der ersten Realisierung handelt es sich um eine ereignisbasierte Kommunikations-Middleware, die auf einem erweiterten Linda-Tupelraum aufbaut. Die zweite ist als verteiltes Objekt-System implementiert und stellt den Anwendungen eine Reihe von Diensten zur Verfügung. Dazu gehören unter anderem auch kontextbezogene Dienste. Die Applikationen können vom Active Space Kontext-Informationen (z.B. Zahl der anwesenden Personen, deren Namen, oder der Grund des Zusammentreffens wie Konferenz oder Seminar) abfragen. Am Ende werden einige Probleme beleuchtet, die bei beiden Realisierungen vor dem praktischen Einsatz noch gelöst werden müssen.

Inhaltsverzeichnis

1 Einleitung

Im Zuge der Miniaturisierung der Mikrocomputer und der zunehmenden Integration von Prozessoren in Alltagsgegenständen wird die Vernetzung bis dahin isolierter Applikationen immer wichtiger. Wie bereits bei der Vernetzung von Workstations in den achtziger Jahren ist absehbar, dass die Zahl der Applikationen, die diese Vernetzung verwenden, ebenfalls zunehmen wird. Daher erscheint das Herausfaktorisieren oft verwendeter Applikationsfunktionen in eine Middleware auch bei Applikationen für smarte Umgebungen sinnvoll. Ähnliche grundlegende Paradigmen wie Remote-Procedure-Call oder verteilte Objektsysteme dürften auch hier zu erwarten sein.
In diesem Beitrag sollen Anforderungen, Eigenschaften und die Realisierung von Infrastukturen und Middleware für smarte Umgebungen diskutiert werden. Im folgenden Abschnitt wird eine allgemeine Definition für smarte Umgebungen angegeben, die anschliessend auf den hier betrachteten Bereich von interaktiven Räumen konkretisiert wird. Im zweiten Abschnitt werden die Anforderungen an Applikationen für smarte Umgebungen angegeben, sowie die hierfür notwendige Funktionalität der Infrastruktur und Middleware. Im dritten Abschnitt werden zwei konkrete Realisierungen betrachtet, die im letzten Teil miteinander verglichen werden. Probleme, die vor dem praktischen Einsatz dieser Prototypsysteme noch zu lösen sind, werden anschliessend diskutiert.

1.1 Smarte Umgebungen – Definition und Eigenschaften

Als smarte Umgebungen werden diejenigen physikalischen oder virtuellen Umgebungen umschrieben, die in einem noch näher zu bezeichnenden Sinn smart, also intelligent sind. Stefan Junstrand [1] charakterisiert smarte Umgebungen durch folgende sechs Punkte.

- Smarte Umgebungen sind interaktiv, d.h. der Benutzer kann das Verhalten der Umgebung beeinflussen.
- Smarte Umgebungen sind mit Informations- und Kommunikations-Technologien ausgestattet, dazu gehören Software-Komponenten, Prozessoren und Sensoren.
- Smarte Umgebungen ermöglichen die Kommunikation der Benutzer mit den Geräten, den Geräten untereinander und vor Benutzer mit dem Raum selbst.
- Sind ausgerüstet mit einer Vielfalt von Benutzerschnittstellen. Neben den gängigen Schnittstellen wie Maus, Tastatur und Bildschirm sollen auch neuartige Schnittstellen verwendet werden können, die über Sprache, Gesten oder durch Berühren sensitiver Oberflächen bedient werden.
- Die smarten Umgebungen passen sich den Bedürfnissen und Anforderungen der jeweiligen Benutzer an. Das smarte Gebäude stellt anhand der aufgezeichneten Bewegung einer Person fest, dass diese offenbar einen bestimmten Raum sucht. Der Raum bietet dann Hilfestellung an, indem auf der nächstgelegenen Anzeige ein Informations-Assistent oder ein Übersichtsplan angezeigt wird.
- Die smarten Umgebungen erlernen automatisch die Gewohnheiten und Vorlieben der Benutzer. Das smarte Wohnzimmer erkennt die Person und spielt deren bevorzugte Hintergrundmusik ab.

In diesem Beitrag soll eine smarte Umgebung im Sinn einer physikalischen Räumlichkeit verstanden werden. Manuel Román [7] führt in diesem Zusammenhang den Begriff *Active Space* ein. Wir sind der Meinung, dass diese Bezeichnung im Kontext dieses Beitrags äusserst gut geeignet ist. Er weist erstens auf den physikalischen Raum hin und zweitens streicht er den Aspekt der Interaktivität hervor. In seinem Paper [7] definiert Román Active Spaces wie folgt:

> Active Space: A physical space coordinated by a responsive context-based software infrastructure that enhances the ability of mobile users to interact and configure their physical and digital environments.

Zusätzlich zum oben Genannten, lassen sich für interaktive Räumen folgende Eigenschaften zuweisen:

- Interaktive Räume erfassen mit fest installierten Sensoren oder über die eingefügten Geräte Kontext-Informationen und verwalten diese. Die Applikationen können dadurch Kontext-Daten vom System abrufen.
- Interaktive Räume stellen den Endgeräten eine Soft- und Hardware-Infrastruktur bereit, über die Applikationen von verschiedenen Geräten miteinander kommunizieren können. So kann eine Terminplaner-Applikation, die auf einem PDA läuft, mit einer anderen Terminplaner-Applikation, die auf einem Notebook läuft, Verbindung aufnehmen und den Benutzern einen Vorschlag für den nächsten Sitzungstermin unterbreiten.
- In interaktiven Räumen soll eine Vielzahl von verschiedenen Geräten (wie z.B. Drucker, PDA, Notebook, Digitalkamera, ...) integriert werden. Einige dieser Geräte können fest im Raum installiert sein, während andere Geräte von den Benutzern mitgebracht und nur temporär in den Raum eingebunden werden. Eingebunden bedeutet hier, dass die von den jeweiligen Geräten angebotenen Dienste (z.B. Druckservice) anderen Geräten und Applikationen zur Verfügung gestellt wird.

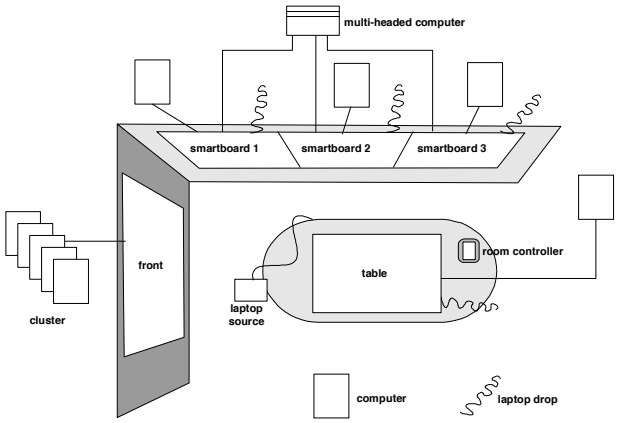

Abbildung 1: Installation Stanford iRoom (aus [5])

1.2 Anwendungs-Beispiel: interaktiver Konferenzraum

iRoom ist ein interaktiver Konferenzraum, der als Teil des iWork-Projekts[1] an der Stanford University entwickelt wurde. Neben seinem hauptsächlichen Einsatzweck als interaktiver Konferenzraum wird er auf dem Universitätscampus ebenfalls als interaktive Lehr- und Lernumgebung eingesetzt.
Die Ausstattung des iRooms ist in Abbildung 1 dargestellt. Die Darstellung wurde aus [5] entnommen. Drei grosse Touchdisplays (sog. Smartboards) sind an der Wand befestigt. Eine weitere Anzeige befindet sich auf der Tischoberfläche. In diesem Fall wird das Bild von einem weiteren Beamer erzeugt, der sich unter dem Tisch befindet. Das Bild wird hier auf die halbtransparente Tischplatte projiziert. Ebenfalls zur Rauminfrastruktur gehört ein Rechner-Cluster, welches die Anzeigen ansteuert und die Daten der Sensoren und Eingabegeräte verarbietet. Der Raum ist zusätzlich mit einem drahtlosen Netzwerk ausgerüstet. Dies ermöglicht das einfache Einbinden von zusätzlichen Geräten durch die Benutzer. Neben den konventionellen Eingabegeräten wie Bluetooth-Tastatur und -Maus sind im iRoom weitere proprietäre Eingabegeräte, wie ein drahtloser Zeigestift (iPen) oder ein programmierbarer Taster (iButton) vorhanden. Diese Eingabegeräte werden in Abschnitt drei genauer beschrieben.
Im Folgenden wird eine Präsentation als mögliches Anwendungsszenario für den iRoom vorgestellt. Der erste Benutzer betritt den Raum und aktiviert durch Drücken eines programmierbaren Tasters den Raum. Dabei werden Anzeigen eingeschaltet und der Raum auf eine im Voraus definiert Konfiguration eingestellt. Der Benutzer authentifiziert sich dann über den im Raum untergebrachten Fingerprint-Sensor oder indem er sich über seinen PDA oder Notebook anmeldet. Dadurch sind die persönlichen Dateien dieses Benutzers vom Raum aus zugreifbar. Jetzt bereitet er seine Präsentation vor und schaltet die gewünschten Wandbildschirme zu. Beispielsweise kann er die Präsentation auf dem Smartdisplay 2 anzeigen lassen, während das Inhaltsverzeichnis der Präsentation ständig auf dem Smartdisplay 1 für die Zuhörer sichtbar ist. Die Ausgabe der Projektplanungs-Applikation könnte er dann dem Smartdisplay 3 zuweisen.
Nachdem sich die Zuhörer mit ihren Notebooks ebenfalls angemeldet haben, erhalten sie die Folien direkt auf ihre Geräte. Sie können sich dann während dem Vortrag sogleich Bemerkungen zu einzelnen Folien notieren. Angenommen, ein Zuhörer hat zusätzliche Informationen zu einem Thema, die er dem Anwesenden zeigen möchte. Dann kann er die Ausgabe der Applikation von seinem Notebook auf eine beliebige Wandanzeige schalten. Am Ende der Präsentation können die Anwesenden den nächsten Sitzungstermin über ihre persönlichen Terminplaner absprechen. Der Raum übernimmt dann die Koordination der einzelnen Applikationen.

[1] iWork = interactive Workspaces

2 Anforderungen in smarten Umgebungen

In diesem Abschnitt werden die Anforderungen an die Applikationen für smarte Umgebungen diskutiert. Anschliessend werden daraus die Anforderungen an die Middleware und Infrastruktur abgeleitet.

2.1 Anforderungen an Applikationen

In [2] werden die unterschiedlichen Anforderungen an das Umgebungssystem (Runtime-System, Middleware) von mobilen Applikationen mit denjenigen konventionell verteilter Applikationen verglichen. Im Gegensatz zu konventionellen verteilten Systemen, sollte das System den Charakter der Verteilung deutlich nach aussen zeigen und nicht verbergen. Für die Unterstützung pervasiver Applikationen müssen folgende drei Anforderungen berücksichtigt werden:

Orts-Bewusstsein Da sich die Benutzer mit ihren Mobilgeräten in der physikalischen Welt bewegen, ändert sich der Ort, an dem die Applikation ausgeführt wird, sowie deren Kontext laufend. Das System muss daher diese Änderungen wahrnehmen und sie vor der Applikation nicht verbergen.

Dynamik Die Benutzer erwarten, dass Geräte und Applikationen dynamisch kombiniert, d.h. ad hoc konfiguriert werden können.

Information Sharing Wenn die Benutzer miteinander kollaborieren, müssen auch deren Geräte zusammenarbeiten. Die Geräte müssen also die in ihnen gespeicherten Informationen an andere weitergeben.

Die oben genannten Punkte können präzisiert und zudem noch weitere Kriterien angefügt werden. Die Applikationen müssen mehrere Ein- und Ausgabegeräte unterstützen. So sollte eine Anwendung Benutzereingaben

sowohl von herkömmlichen Eingabegeräten wie zum Beispiel Maus und Tastatur als auch von neuartigen Systemen wie Sprach- und Gestenerkennung verarbeiten können. Die Applikation soll Ausgaben auf verschiedene Geräte leiten können, unter Umständen sogar gleichzeitig. Eine Präsentationapplikation kann beispielsweise die Folien auf einem Display anzeigen, während das Inhaltsverzeichnis auf einem anderen dargestellt wird. Auf dem Notebook oder PDA des Vortragenden können Notizen zu den jeweiligen Folien dargestellt werden.
Die Applikation soll für den Datenaustausch mit anderen Applikationen vorbereitet sein. So könnte die Grant-Chart-Darstellung der Projektplanungs-Anwendung direkt in die Präsentationsapplikation eingebunden[2] werden.
Die Anwendung muss kontext-sensitiv sein, d.h. sie soll sich entsprechend dem aktuellen Kontext sinnvoll verhalten. Zum Kontext kann u.A. die Anzahl der Personen im Raum gehören. Eine Email-Applikation könnte die persönlichen Nachrichten des Benutzers zur besseren Lesbarkeit auf einer Wand-Anzeige darstellen, sofern sich der Benutzer alleine im Raum aufhält. Aus Gründen der Sicherheit ist es unangebracht, dass die Nachrichten auf einem ≪öffentlichen≫ Bildschirm – für alle sichtbar – angezeigt werden, sofern sich noch ein anderer Benutzer im Raum befindet. In diesem Fall ist es daher wünschenswert, wenn die Anwendung die persönlichen Nachrichten auf dem PDA oder dem Notebook anzeigt, sodass sie ausschliesslich für den jeweiligen Benutzer sichtbar sind.

2.2 Anforderungen an Middleware und Infrastruktur

Aus den Anforderungen, die an die Applikationen gestellt werden, lässt sich, wie bei der klassischen Middleware, Gemeinsames herausfaktorisieren und in der Middleware unterbringen. Im Folgenden wird zwischen funktionalen und nicht-funktionalen Aspekten unterschieden.
Um die Kommunikation unter den Applikationen zu ermöglichen, müssen entsprechende Kommunikationsmechanismen im System implementiert werden. Es ist daher zweckmässig, deren Implementierung in die Middleware zu verlagern. Applikationsentwickler müssen daher die Kommunikationsschnittstellen nicht stets von neuem implementieren. Vielmehr können sie die in die Middleware integrierte Funktionalität verwenden. Die Middleware spielt dann die Rolle eines Mediators zwischen Applikationen. Man kann die Middleware in dieser Hinsicht auch als ≪Kommunikationsschiene≫ verstehen, die sich mit mit der Mikrokernel-Architektur[3] vergleichen lässt.

[2]Ein entsprechender Ansatz ist heute bereits in OLE/DDE von Microsoft Windows zu finden.

[3]In dieser Betriebssystem-Architektur ist die Funktionalität des Kernels in mehrere Services gekapselt (z.B. Filesystem, Memory-Managment). Die einzelnen Services kommunizieren über den Mikrokernel untereinander. Dieser Kernel ist von minimaler Grösse. In der Mikrokernel-Architektur können beispielsweise mehrere Betriebssystem geleichzeitig emuliert werden, indem die jeweiligen Services an den Mikrokernel angeschlossen werden.

Neben den Kommunikationsdiensten muss die Middleware auch Kontextinformationen zum Raum verwalten. Als Kontext-Informationen können für einen smarten Raum die Menge der Benutzer, Geräte, Dienste und Applikationen aufgefasst werden, die sich momentan im Raum befinden oder sich zu einem früheren Zeitpunkt darin befunden haben. Die Rauminfrastruktur beinhaltet Sensoren, welche die dazu nötigen Daten erfassen. In Gaia [7] beispielsweise erfolgt die Erkennung der Benutzer bei der Authentifizierung über einen Fingerprint-Leser oder wenn sich der Benutzer über ein fest installiertes oder ein mitgebrachtes Gerät an Gaia anmeldet. Die Middleware verarbeitet die Sensordaten und erzeugt dabei höherwertige Kontext-Informationen. Beispiel: Aus den Daten vom Sensor der Türe und des daneben angebrachten Fingerprint-Lesers kann die Information ≪Person X betritt den Raum≫ und ≪Eine Person verlässt den Raum≫ extrahiert werden. Unter der Voraussetzung, dass sich jede Person, die den Raum betritt, über den Fingerprint-Leser authenzifizieren muss, kann die Middleware aus den beiden Informationen die Information ≪Es befinden sich zur Zeit n Personen im Raum≫ generieren. Eine Anwendung kann diese Zahl n von der Middleware abfragen, ohne sich selbst um die Auswertung der Sensoren zu kümmern.
Zusätzlich agiert die Middleware als sogenanntes *Meta-Betriebssystem* zwischen den Betriebssystemen der untergebrachten Geräte (wie z.B. Palm OS, Windows CE, Symbian, usw). In [7] beschreibt Manuel Román sein Gaia-System explizit als Meta-Betriebssystem. Auf den Endgeräten wird eine Adaptionsschicht eingefügt, welche die Funktionalität der Middleware den Applikationen auf dem jeweiligen Gerät zur Verfügung stellt.
Zu diesen funktionalen Aspekten lassen sich folgende nicht-funktionale Gesichtspunkte hinzufügen.

Kommunikation Die Middleware soll die Anbindungsmöglichkeit für eine Vielzahl verschiedener End-Geräte (wie Notebook, PDA, Smart-Phone, usw.) zur Verfügung stellen. Die eingesetzten Geräte verwenden unterschiedliche Bedienkonzepte (Eingabe- und Ausgabemöglichkeiten) und unterscheiden sich ebenfalls stark in ihrer Rechenleistung und Speicherausstattung (z.B. Notebook und Smart-Phone). Ebenfalls besitzen sie verschiedene Kommunikationsschnittstellen (IrDA, Bluetooth, WLAN). Die angeschlossenen Geräte sind bezüglich der verwendeten Technologien folglich äusserst heterogen.

Adaption Die Applikationen sollen für den Benutzer möglichst gleich oder ähnlich bedienbar sein, unabhängig vom verwendeten Gerät. Die Infrastruktur soll über die Middleware leistungsschwachen Geräten auch Rechenkapazität zur Verfügung stellen, oder die Informationen entsprechend den Fähigkeiten der Endgeräte aufbereiten (Bilder, die über die Middleware an eine Applikation auf einem PDA gesendet werden, können aus Effizienzgründen bereits bei der Auslieferung von der Middleware bezüglich Auflösung und Farbtiefe reduziert werden).

Robustheit Mobile Endgeräte können spontan in den Raum eingebunden und auch wieder entfernt werden. Für diese Fälle soll sichergestellt sein, dass das Gesamtsystem, d.h. die Middleware mit allen anderen Geräten und deren Applikationen, soweit möglich, weiter funktioniert. Der Ausfall eines Geräts oder Applikation darf nicht den Ausfall des Gesamtsystems zur Folge haben.

Vor-, Rückwärtskompatibilität Das API[4] über das die Applikationen Dienste der Middleware aufrufen, muss sorgfältig festgelegt werden. Es soll so definiert sein, dass auch zukünftige Applikationen und Geräte dieses verwenden können, d.h. das API soll auch für zukünftige Anwendungen und Dienste ausgelegt sein. Auf der anderen Seite sollen über das API auch ältere Geräte eingebunden werden, deren Verwendung in einer smarten Umgebung ursprünglich nicht vorgesehen wurde. In diesem Sinn muss beim API der Middleware sowohl *Vor-* als auch *Rückwärtskompatibilität* berücksichtigt werden.

3 Realisierungen

Im Hauptteil dieses Beitrags werden die beiden untersuchten Realisierungen vorgestellt. Im Folgenden wird das iROS-System, ein auf einem Event-Heap basiertes System beschrieben. Anschliessend wird Gaia vorgestellt. Bei diesem System wurde ein anderer Ansatz verfolgt. An Stelle eines Event-Heaps baut Gaia auf einem verteilten Objekt-System auf.

3.1 iROS – Interactive Room Operating System

Das Akronym iROS steht für *interactive Room Operating System*. Das System wurde als Teil des *iWork*[5] Projekts an der Stanford University entwickelt. iROS wird von den Entwicklern in den Papers [3] und [5] beschrieben. Die Aufgabe von iROS besteht darin, als Meta-Betriebssystem die angeschlossenen Geräte und Applikationen zu verbinden. Hierzu stellt die Middleware den Applikationen Koordinationsfunktionen zur Verfügung. iROS wird im iRoom, einem multimedialen und vernetzten Konferenz-Raum eingesetzt. Auf dem Campus sind mehrere iRooms installiert. Neben der ursprünglichen Funktion als Konferenz- und Seminar-Raum können sie von den Studierenden auch als interaktive Lernumgebung verwendet werden. Der Raum stellt die Infrastruktur bereit, wie Netzwerk-Anbindung, Rechner-Ressourcen, Ein- und Ausgabe-Geräte (Maus, Tastatur, proprietäre Zeigegeräte, Touchscreens usw). Abbildung 1 zeigt die Installation des iRooms. Die Installation des Raumes wurde in Abschnitt 1.2 vorgestellt. Der in iROS implementierte Koordinations-Mechanismus basiert auf dem Event-Heap-Modell, einer Erweiterung des Tupel-Raum-Modell.

[4] API: Application Programming Interface
[5] iWork: interactive Work spaces

Applikations-Koordination

Gemäss ihrem Paper [3] basiert iROS auf dem Konzept der Applikationskoordination, wie sie von Gelernter and Carriero in [6] vorgestellt wurde. In diesem Paper wird der Standpunkt vertreten, dass reale Sprach-Systeme immer aus zwei Subsprachen, der *Computational Language* und der *Coordination Language* bestehen. Die Coordination Language verbindet, d.h. koordiniert, mehrere Entitäten zu einem Ensemble. Die Entitäten können sowohl System-Prozesse, Tasks oder Threads als auch Benutzer sein.
Die Computational Language beinhaltet diejenigen Konstrukte, die für die Beschreibung einer Aktivität nötig sind. Im Prinzip bedeutet das, dass mit der Computational Language eine Turing-Maschine implementiert werden kann. Gelernter und Carriero argumentieren weiter, dass Computational Language und Coordination Language orthogonal zueinander stehen, d.h. dass die eine kann nicht durch die andere ausgedrückt werden kann. Reale Systeme besitzten stets beide Sprachen. Man kann sich unter der Computational Language alle gängigen Sprachkonstrukte der imperativen Programmierung (Sequenz, Selektion, Iteration) vorstellen, während Mechanismen für die Datenein- und ausgabe, sowie die Synchronisation von Threads zur Coordination Language gezählt werden können.
In [6] wird empfohlen diese Aufteilung explizit vorzunehmen, d.h. zwei Systeme können zwar eine unterschiedliche Computation Language haben, müssen aber die gleiche Coordination Language implementieren. Das ermöglicht die Koordination zweier Anwendungen, die für zwei verschiedene Systeme entwickelt wurden, falls die gleiche Coordination Language verwendet wird. Dadurch wird Koordination verschiedener Applikationen möglich, so dass die Heterogenität der einzelnen Sprach-Systemen kein Problem mehr darstellt. Im Übrigen lassen sich Applikationen leichter auf ein anderes Sprach-System portieren, insofern als dass nur der Teil in der Computational Language angepasst werden muss.
Gelernter und Carriero gehen noch weiter, indem sie vorschlagen, man soll eine allgemeine Coordination Language verwenden. Das sei sowohl ökonomisch als auch flexibel. Ihre Argumentation ist an sich trivial, denn sobald sich mehrere Entwickler auf ein einziges, allgemeines Sprachensystem einigen, wird die Effizienz erhöht. Flexibilität wird erreicht, indem mit der Verwendung einer allgemeinen Koordinationssprache jede Art der Applikations-Koordination realisiert werden kann. Diese Argumentation ist auch im Paper von Johnson und Fox [5] wiedergegeben.

Linda Tupel-Modell

Linda ist eine Koordinationssprache, die auf einem Tupelraum basiert. Die Entwickler von iROS sind bei der Entwicklung vom Konzept des Tupelraums ausgegangen. Der Tupelraum wie er in [4] von Gelernter und Bernstein vorgeschlagen wurde, besteht aus Datentupeln $(attr_1, attr_2, \ldots, attr_n)$ welche in einem globalen

Speicher abgelegt sind. Aktivitäten können mit der Anweisung `OUT(attr1, ..., attrN)` neue Tupel in den Raum einfügen. Mit `IN(attr1, ?attr2, ..., attrN)` werden Tupel konsumierend aus dem Tupelraum gelesen. Das passende Tupel wird dabei über die angegebenen aktuellen Parameter `attr1` und `attrN` ausgewählt. Der Parameter `?attr2` is ein formaler Parameter, der nach der IN-Operation entsprechend dem Attribut des gelesen Tupels gesetzt wird. Neben `IN` existiert noch eine weitere Operation `READ`. Bei dieser Operation wird das Tupel nach dem Lesen nicht aus dem Raum entfernt. Sowohl `IN` als auch `READ` blockieren die lesende Aktivität solange, bis ein passendes Tupel in den Raum einfügt wird.

Event-Heap-Modell

Johanson und Fox bescheiben in [5] einige Schwierigkeiten mit dem oben beschriebenen Tupelraum-Modell für smarte Umgebungen. Sie schlagen daher folgende Erweiterungen vor:

Self-describing Tuples Die Attribut-Felder der Event-Tupel besitzen nicht nur einen Typ und einen Wert sondern auch einen Namen (String). Dadurch soll den Applikationen ermöglicht werden, auch Tupel, deren Format sie noch nicht kennen, eine Bedeutung beimessen zu können. Eine Applikation kann dadurch den Event-Heap ≪abhören≫ und dann feststellen welche anderen Applikationen vorhanden sind.

Tuple Sequencing Wenn im Linda-Tupelraum mehrere Tupel vorhanden sind, die zu einem `READ` oder `IN` passen, wählt das System zufällig eines der passenen Exemplare aus. Dieses nicht deterministische Verhalten bereitet Probleme. Durch eine fortlaufende Nummerierung der Tupel wird sichergestellt, dass der Empfänger stets das älteste aus der Menge der passenden Tupel erhält. Eine Applikation, welche Event-Tupel mit Zustandsinformationen einer anderen Applikation empfängt, liest somit jedes Tupel in FIFO-Reihenfolge und genau einmal.

Expiration of Tuples Im Tupelraum können Applikationen als Produzent und/oder als Konsument auftreten. Es sind Situationen denkbar, wo zu einem Produzent kein Konsument existiert (zum Beispiel ein Uhrzeit-Dienst, der periodisch Zeit-Events in den Event-Heap einfügt, auch wenn keine Applikation vorhanden ist). Da in diesen Fällen keine Tupel konsumiert werden, würde die Anzahl der Tupel im Raum stetig zunehmen. Die Tupel müssen dennoch gespeichert werden, daher würde sich bei echten System eine Abnahme der Performanz zeigen. Johanson und Fox schlagen daher vor, für jedes Tupel ein *TimeToLive* Attribut zu verwenden. Nach Ablauf dieser Lebenszeit wird das Event-Tupel aus dem Event-Heap entfernt, resp. für den *Garbage Collector* freigegeben.

Query Registration Im Linda-Tupelraum müssen die Anwendungen den Raum mit `IN` oder `READ` pollen. Dadurch besteht die Möglichkeit, dass eine Applikation ein Event verpasst, der während einem Polling-Intervall eingefügt wird und sogleich wieder konsumierend gelesen oder vom Garbage Collector weggeräumt wird. Johanson und Fox schlagen hierzu einen Publish/Subscribe-Mechanismus vor. Die Anwendungen können sich beim System für bestimmte Tupel (durch Angabe des Muster der Tupel) registrieren. Dann wird die Applikation über einen Callback jedes Mal benachrichtigt, wenn ein entsprechendes Event-Tupel in den Event-Heap eingefügt wird.

Da die Semantik der Erweiterungen von derjenigen des Standard-Tupelraums abweicht, verwenden die Autoren eine andere Bezeichnung für Tupel und Tupelraum. Die Tupel bezeichnen sie als *Events* (oder Event-Tupel) und den Tupelraum als *Event-Heap*. Damit wollen sie der Tatsache Rechnung tragen, dass die Anwendungen in einer smarten Umgebung durch das Austauschen von Events mineinander kommunizieren.

Implementation

iROS wurde basierend auf einer Client/Server-Architektur implementiert. Der Event-Heap ist auf einem einzigen Server abgebildet. Der Event-Heap wird nicht persistent auf dem Server gehalten. Die Applikationen sind als Clients üblicherweise über eine permanente TCP-Verbindung an der Server angeschlossen. `IN`-, `OUT`- und `READ`-Operationen werden auf einem Wiring-Protokoll des Event-Heaps implementiert. Der Server wurde in Java realisiert. Für die Clients steht eine Java- und eine C-Library zur Verfügung, welche die Applikation über das Wiring-Protokoll an den Server anbindet. Für Python-Clients existiert ein Python-Java-Wrapper.
Daneben steht den Clients auch ein Web-Zugang zum Event zur Verfügung. Dies soll verschiedenen Low-End-Geräten (wie Smart-Phones und Palm-ähnliche Geräten) den Zugang zum Event-Heap ermöglichen. Im Übrigen erlaubt dieser Zugang den Benutzern den Event-Heap vom Dekstop-Rechner über einen Webbrowser zu benutzen.

Drahtlose UI-Komponenten – iStuff

iStuff ist eine Sammlung von drahtlosen, batteriebetriebenen Benutzer-Schnittstellen-Komponenten, die in den iRoom integriert werden können. Die Abbildung 2 zeigt eine Übersicht über die iStuff-Architektur. Ein- und Ausgabegeräte besitzen kleine Funk-Sender resp. -Empfänger (300 MHz). An die Rechner, die zur Rauminfrastruktur gehören, sind weitere Sender und Empfänger angeschlossen. Auf diesen Rechnern laufen Proxies, welche Events von den Eingabegeräten via Funk empfangen und die entsprechenden Event-Tupel in den Event-Heap einfügen. Ein Proxy für ein Ausgabegerät überwacht

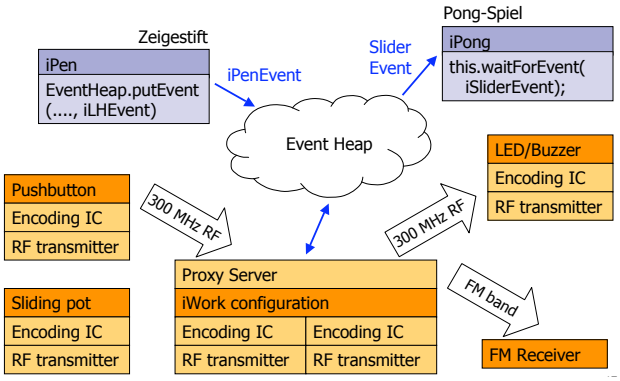

Abbildung 2: Drahtlose UI-Komponenten (frei nach [3])

den Event-Heap und sendet die zugehörigen Events über Funk an das Ausgabegerät.
In [3] stellen die iROS-Entwickler folgende Eingabegeräte vor: iButton (ein programmierbarer Taster), iPen (ein Zeigestift) und iSlider (Potentiometer). Als iStuff-Ausgabegeräte stehen zur Verfügung: iLED (LED-Anzeige), iBuzzer (Summer) sowie iSpeaker (Lautsprecher).
Die Applikationen verarbeiten Events von den Eingabegeräten durch Konsumieren der Event-Tupel, die von den Proxies in den Event-Heap gestellt werden. Die Steuerung der Ausgabegeräte erfolgt ebenfalls durch Events, die vom jeweiligen Proxy konsumiert werden. Für alle iStuff-Geräte existiert ein Web-Interface mit dem Ereignisse der Geräte mit den Event-Tuples assoziiert werden können.
Der Audio-Stream für den iSpeaker lässt sich nicht über Events realisieren. Die inheränt atomare Struktur der Events-Tupels eiget sich nicht für Streams, zumal im Event-Heap-System keine Garantien über die Latenz und Bandbreite der Event-Kommunikation gegeben sind. Daher stellt der iSpeaker eine Ausnahme unter den iStuff-Geräten dar. In diesem Fall wird lediglich ein Event-Tupel verwendet um den iSpeaker ein- und auszuschalten. Der eigentliche Audio-Stream wird ausserhalb des Event-Heaps über einen im Raum installierten UKW-Sender mit kurzer Reichweite an den Empfänger im iSpeaker übertragen. Der iSpeaker kann daher als tragbares UKW-Radio realisiert werden, das mit iStuff-Technologie für das Ein- und Ausschalten erweitert wurde.
Dieses Prinzip kann auch auf andere Geräte übertragen werden. Die bestehenden Geräte im Raum können mit iStuff nachgerüstet und somit über den Event-Heap ferngesteuert werden. Dadurch lassen sich Task-bezogene Fernbedienungen realisieren. Ein Task kann zum Beispiel das Starten einer Filmvorführung sein. Während heute eine Fernbedienung zum Dimmen der Beleuchtung, eine weitere zum Einschalten des Video-Projektors, eine dritte zur Steuerung des Sourround-Systems, und eine weitere zum Starten des DVD-Spielers erforderlich ist, könnte die Steuerung zukünftig vom iRoom-System übernommen werden. Hierfür müssen die End-Geräte durch eine iStuff-Erweiterung an den Event-Heap angebunden werden. Dann wird ein Task «Film starten» definiert, der über einen Event an einen iButton gekoppelt ist, so dass das Drücken dieses iButtons die Vorführung startet. Die heutigen Fernbedienungen sind weitgehend gerätebasiert. Durch die Entkopplung der Geräte von den Steuerbefehlen über den Event-Heap können solche Task-basierten Fernsteuerungen leicht realisiet werden.

Auswertung iROS/iRoom

In diesem Abschnitt soll iROS/iRoom kritisch untersucht werden. Auf Seiten der Realisierung fällt zunächst auf, dass der Event-Heap auf einem einzigen Rechner abgebildet wird. Das heisst, es findet keine Replikation statt. In den Papers [3] und [5] sind keine Angaben bezüglich Grösse und Anzahl der Tupel vorhanden. Ebenso fehlt die Angabe der Anzahl der Clients resp. Applikationen, die auf den Event-Heap zugreifen können. Es ist daher durchaus denkbar, dass mit zunehmender Anzahl von Applikationen und Geräten die Auslastung des einen zentralen Event-Heap-Servers zu gross wird, so dass eine Replikation des Event-Heaps auf mehrere Rechner erforderlich wird. In Paper [5] wird die Ein-Server-Lösung als Vorteil angepriesen. In diesem Fall, so die Autoren, sei nur ein «Single Point of Failure» vorhanden. Insbesondere sei neben dem Event-Heap-Server ein weiterer Server installiert, der ein Web-Interface bereitstellt, über das der Event-Heap-Server im Falle eines Absturzes innert kurzer Zeit neugestartet werden kann. Ebenso versuchen die Clients nach Ausfall selbständig wieder eine Verbindung zum Server zu erstellen.
Die Anbindung erfolgt über eine TCP-Verbindung. Da grundsätzlich Events ausgetauscht werden, ist nicht ersichtlich, weshalb hier für die Implementation eine verbindungsorientierte Kommunikation verwendet wird. Für die Wahl von TCP anstelle von UDP werden in den Papers [3] und [5] keine Gründe angegeben.
Angesichts der Tatsache, dass es sich im beschriebenen System um einen Proof-of-Concept-Prototyp handelt, sind die realisierungsspezifischen Punkte als nicht besonders schwerwiegend zu betrachten.
Auf der konzeptuellen Ebene mag das Prinzip von Events mit Verfallsdatum ungewohnt sein. Ein Ereignis hat als Punkt in der Raumzeit eine Bedeutung und ist kausal abhängig von vorherigen Ereignissen. In diesem Fall besitzt ein Ereignis jedoch eine Lebensdauer, d.h. nach einer gewissen Zeit verliert der Event an Bedeutung. Wird er vom Garbage Collector aufgeräumt, werden sämtliche Beweise entfernt, die bezeugen könnten, dass das Ereignis jemals stattgefunden hat.
Weiter fällt auf, dass zum Tupelraum-Modell durch *Query Registration* effektiv ein Publish/Subscribe-Mechanismus geschaffen wurde. Die Frage stellte sich, wozu noch ein Tupelraum wenn ein Publish/Subscribe-Mechanismus entsteht? In [5] weisen die Autoren auf einen Grund hin, weshalb Publish/Subscribe alleine nicht verwendet werden kann und infolgedessen ein Tupelraum-Modell verwendet wurde. Der Hauptunterschied bestehe darin, dass Meldungen in einem Publish/Subscribe-System nicht persistent sind. Dadurch ist es schwieriger,

neu gestarte Applikationen ins System zu integrieren, da hier für die Applikationen keine Möglichkeit besteht, die letzten Meldungen abzuholen. Publish/Subscribe auf vergangene Meldungen ist hier schwierig zu realisieren.
In den betrachteten Papers wird der Zusammenhang zwischen *Query Registration* und konsumierendem Lesen nicht erläutert. Das Aufrufen des Callbacks eines registrierten Clients ist an das Einfügen eines entsprechenden Event-Tupels gebunden. Was passiert, wenn einer von zwei benachrichtigten Clients mit einem konsumierenden Lesen das Event-Tupel aus dem Tupelraum entfernt noch bevor die Benachrichtigung den anderen erreicht? In diesem Fall erhält der zweite eine Benachrichtigung für ein Tupel, das nicht mehr im Tupelraum vorhanden ist. Die mögliche Lösung kann darin bestehen, dass bei der Benachrichtigung eine Kopie dieses Tupels mitgeschickt werden muss.
Im Weiteren geht nicht klar hervor, wie die sogenannten selbstbeschreibenden Tupel zu realisieren sind. In [3] und [5] wird nicht auf die Problematik hingewiesen, dass diese Attributnamen *automatisch* interpretiert werden müssen. Damit eine Anwendung aufgrund der Attributnamen einem Tupel eine Bedeutung beimessen kann, muss ein Begriff-System für Attributnamen (Ontologie) definiert werden. Schwieriger noch, das Begriff-System muss über Applikations-Grenzen hinaus gelten, da Applikationen in der Regel den Tupelraum überwachen und dabei Event-Tupel anderer Anwendungen inspizieren. Vermutlich ist eine Standardisierung des Begriff-Systems unausweichlich, wenn Anwendungen verschiedener Entwickler oder sogar unterschiedlicher Hersteller in der smarten Umgebung miteinander kommunizieren sollen. Es drängt sich die Frage auf, ob ein ausreichend flexibles Begriff-System überhaupt standardisiert werden kann. Ein Standard eines Begriff-Systems wird vermutlich nur für einen sehr spezifischen Kontext realisierbar sein, z.B. für einen smarten Konferenzraum.

3.2 Gaia

In diesem Abschnitt wird das Gaia-System beschrieben. Gaia wurde an der University Of Illinois in Urbana-Champaign entwickelt. Die Entwickler beschreiben das System in [7] als eine Middleware-Plattform für *Active Spaces*. Wie auch in iROS/iRoom kommt das System in einem interaktiven Konferenzraum zum Einsatz. Anders als iROS wird hier der Fokus nicht auf den Koordinationsmechanismus sondern auf eine Menge von sogenannten Kernel-Services gesetzt, welche das System charakterisieren. Gaia ist als verteiltes Objekt-System basierend auf CORBA realisiert worden. Die Kernel-Services bestehen aus dem *Event-Manager*, *Presence-Manager*, *Space-Repository*, *Context-Service* und dem *Context-Dateisystem*. Die Abbildung 3 zeigt die Architektur des Gaia-Systems.
Die Services sind auf den *Component Management Core* «aufgesteckt». Über diesen können Komponenten dynamisch geladen und entfernt werden. Zum Beispiel können Endgeräte neue Service-Komponenten über die Management-Schnittstellen hinzufügen. Wenn ein Gerät als Anbieter eines Dienstes in das System integriert werden soll, wird eine entsprechende Service-Komponente installiert. Die Anwendungen können dann den neuen Service über diese Komponente verwenden. Diese Architektur gleicht der JMX (Java-Management-Extension) mit den ManagedBeans als Komponenten. Der Component Management Core beinhaltet in der aktuellen Implementation auch den CORBA-ORB. Román schreibt in [7], dass neben CORBA grundsätzlich auch andere Technologien für verteilte Objekt-Systeme (RMI oder SOAP) verwendet werden können. Der Component Management Core und die Kernel-Services bilden den Gaia-Kernel. Darüber liegt ein Application Framework, welches den Active-Space-Anwendungen den Zugang zu den Diensten ermöglicht. In den folgenden Abschnitten werden die Kernel-Services im Einzelnen vorgestellt.

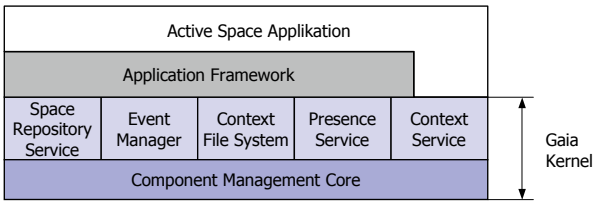

Abbildung 3: Architektur des Gaia-System (aus [7])

Event-Manager

Wie beim iROS-System scheint auch bei Gaia die Event-Kommunikation eine wichtige Rolle zu spielen. Das Gaia-Event-System stellt – wenn auch in kleinerem Umfang als iROS – einen Broadcast-Mechnismus für Zustandsänderungen der Applikationen und Geräte zur Verfügung. Das Event-System baut hier jedoch nicht auf einem Tupelraum auf, sondern basiert auf dem CORBA-Event-Service. Dadurch wird ein Kommunikations-Modell bestehend aus Event-Suppliers, -Consumers und Event-Channels bereit gestellt. Im Gegensatz zum Tupelraum werden die Events in diesem Publish/Subscribe-System nicht aufbewahrt. Das heisst, es ist hier nicht möglich, dass eine neu gestartete Applikation die letzten Events, die über das System versendet wurden, nachträglich abfragen kann.

Presence-Manager

Der Presence-Manager detektiert physikalische Entitäten (Personen, Geräte) sowie digitale Entitäten (Applikationen, Dienste), welche in den Active Space eingefügt oder entfernt werden. Sensoren, die im Raum integriert sind, liefern Informationen über physikalischen Entitäten. Mit dem Fingerprint-Leser oder dem Code-Schloss an der Türe erkennt der Presence-Service, wenn sich ein neuer Benutzer authentifiziert. Ein Bewegungsmelder kann die Anzahl der Personen im Raum bestimmen, resp. erkennen, wenn eine Person der Raum verlässt. Je nach Art

der Sensoren könnte auch die Position der Personen im Raum erkannt werden. Für digitale Entitäten wie Dienste oder Applikationen wird ein Beaconing-Mechanismus basierend auf Leases – ähnlich zu Jini-Leases – verwendet. Der Dienst meldet sich periodisch beim Presence-Manager (über ein Beacon). Vom Presence-Manager erhält der Dienst eine Lease zugeteilt. Nach einiger Zeit verliert die Lease ihre Gültigkeit. Dann erachtet der Presence-Manager den Dienst als nicht mehr registriert. Ein Dienst muss daher seine Lease periodisch erneuern. Die Leases tragen dem Umstand Rechnung, dass in smarten Umgebungen Geräte oft spontan entfernt werden, ohne vorher beim Registrationsdienst aus der Liste der registrierten Geräte ausgetragen zu werden. Damit keine nichtexistenten Dienste registriert bleiben, verliert jeder Registrationseintrag nach einiger Zeit seine Gültigkeit. Geräte, die spontan entfernt wurden, sind daher nach Ablauf der Lease im Presence-Manager nicht mehr sichtbar.

Space-Repository

Über das Space-Repository können die Applikationen Informationen über die Entitäten, die sich gerade im Raum befinden, abfragen. Sie können beispielsweise nach Diensten suchen, oder Name, Typ oder Besitzer eines Gerätes ausfindig machen. Die Auswahl der Dienste erfolgt durch Attribute. So kann eine Applikation ein Display für die Ausgabe suchen, welches eine Auflösung von 1024x768 Bildpunkten besitzt. In diesem Fall sucht die Applikationen nach einem Display-Service mit dem Attribut `Resolution=1024x768`. Das Space-Repository abonniert einen Event-Channel des Presence-Manager. Über diesen Event-Channel wird das Space-Repository vom Presence-Manager über ein- und austretende Entitäten informiert. Die Implementation des Space-Repository basiert auf *CORBA Trader*.

Context-Service

Wie in der Einleitung beschrieben, sind Applikationen für smarte Umgebungen ≪kontext-aware≫. Ein Email-Programm kann, falls sich keine andere Person im Raum befindet, die persönlichen Nachrichten auf der Wandanzeige darstellen statt auf dem Notebook-Display. Die Applikation muss sich entsprechend dem Kontext ≪Alleine im Raum≫ anders verhalten. Da der Kontext das Verhalten der Anwendung beeinflusst, muss diese die Kontext-Information selbst zusammenstellen oder diese bei der Middleware abfragen. Gaia stellt die Kontext-Information über den Context-Service den Anwendungen zu Verfügung.
Kontext-Informationen werden von mehreren sogenannten Context-Providern generiert, die im Context-Service registriert sind. Es wird zwischen high- und low-level Kontext unterschieden. Die low-level Kontext-Informationen werden von Sensoren generiert, die im Raum installiert sind. Das können zum Beispiel Temperatursensoren (Bewegungsmelder), Lautstärkesensoren oder ein Türsensor sein. Aus dem low-level Kontext wird durch Regelwerk, welches in Logik-Komponenten bereitgestellt wird, ein high-level Kontext abgeleitet. Die Logik-Komponenten können sowohl fest installiert, als auch dynamisch durch Geräte und Applikationen ausgetauscht werden. Das Kontext-Modell basiert auf einer Logik aus Quadrupel-Prädikaten. Der Kontext wird durch Klauseln definiert:

```
Context(<Context-Type>, <Subject>, <Relater>,
        <Object>).
```

Der Kontext besitzt einen Typ und stellt ein Subjekt mit einem Objekt in eine bestimmte Beziehung. Das Faktum, dass der Benutzer ≪René≫ den Raum ≪IFW C42≫ betritt, kann wie folgt modelliert werden:

```
Context(location, "Rene", entering, "IFW C42").
```

Aus den Klauseln können durch ein Regelwerk weitere Kontext-Daten abgeleitet werden. Folgende Regel besagt, wenn sich mehr als vier Personen einem Raum aufhalten und die Microsoft PowerPoint-Applikation im Raum gestartet ist, findet in besagten Raum eine Präsentation statt.

```
Context(numberOfPeople, Room, greater, 4) AND
   Context(application, "PowerPoint", is,
           running)
   => Context(socialActivity, Room, is,
              presentation)
```

`Room` wurde im obigen Beispiel als Platzhalter für einen beliebigen Raum verwendent, entsprechend der Prolog-Vorschrift.

Context-Dateisystem

Neben dem Context-Service stellt Gaia ebenfalls ein kontext-abhängiges Dateisystem zur Verfügung. Dessen Aufgabe besteht darin, den Applikationen und den Benutzern jene Dateien bereit zu stellen, die im aktuellen Kontext von Bedeutung sind. Wenn sich ein Benutzer in einem Konferenz-Raum für eine Sitzung anmeldet, wird nicht sein gesamtes Home-Verzeichnis angezeigt, sondern auschliesslich jene Dateien, die für die aktuelle Sitzung relevant sind. Die Middleware kann die Daten auch abhängig von den Fähigkeiten des Gerätes auf dem sie betrachtet werden, aufbereiten. So könnte die Middleware die Auflösung von Bilddateien bereits verkleinern, bevor sie auf einen PDA übertragen werden. Je nach Kontext, d.h. in diesem Fall Typ und Fähigkeit des Endgerätes, werden die Dateien von der Middleware angepasst.
Die Gaia-Entwickler haben sich ein weniger ambitiöses, aber nicht weniger interessantes Ziel gesetzt. Sie modellieren basierend auf den im letzten Abschnitt beschriebenen Kontext-Typen eine virtuelle Verzeichnis-Hierarchie. Dabei repräsentiert ein Verzeichnis einen Kontext. Die Dateien, welche sich in diesem Verzeichnis befinden, sind dann diesem Kontext zugeordnet.
Wie auch bei einem konventionellen Dateisystem wird ein Verzeichnis (hier Kontext) über einen Pfad erreicht.

Der Pfad beinhaltet hier die Kontext-Typen und deren Werte, welche zum Kontext führen. Folgender Pfad führt zum Seminar-Verzeichnis. Es beinhaltet alle Dateien, welche den Kontext mit Typ `location = IFW C42` und `situation = Seminar` haben.

```
/location:/IFW C42/situation:/Seminar
```

Kontext-Verzeichnisse können nicht auf Baum-Strukturen abgebildet werden, wie das bei konventionellen Dateisystemen der Fall ist, denn die Kontext-Typen können im Pfad vertauscht werden. Daher ist der Pfad

```
/situation:/Seminar/location:/IFW C42
```

mit dem obigen äquivalent.

Auswertung Gaia

Die bestehende Implementation des Component Management Cores basiert auf CORBA. Aus [7] ist zu entnehmen, dass neben CORBA auch andere Technologien verwendet werden können. Als Beispiele werden Java-RMI und SOAP aufgeführt. Wie in den letzten Abschnitten beschrieben, basiert die Implementation auf CORBA. Der Event-Service verwendet CORBA Events und der Presence-Service den CORBA Trader. Eine Portierung auf eine Technlogoie, die keinen mit CORBA-Events oder dem CORBA Trader vergleichbaren Mechanismus anbietet (z.B. Java-RMI), dürfte mit erheblichem Aufwand verbunden sein. In diesem Fall müssten grosse Teile des Event- und des Presence-Services neu implementiert werden. Daher ist die beschriebene ≪leichte Portierbarkeit≫ von Gaia auf andere Technologien zu relativieren.
Im Gaia-System stellt die Middleware Kontext-Informationen für die Anwendungen zur Verfügung. Das heisst, eine Applikation muss nicht selbst aus Sensorwerten (für physikalische Entitäten) und Repository-Informationen (für digitale Entitäten) Kontext-Informationen ableiten. Diese Aufgabe wird hier von der Middleware übernommen, und die Applikationen können bei der Middleware Kontext-Informationen abfragen. Die gewählte Architektur scheint auf den ersten Blick das Problem zu lösen, dass verschiedene Applikationen Kontext-Informationen unterschiedlich ableiten können. Für eine Applikation A müssten sich beispielsweise mindestens 10 Personen im Raum befinden, damit die Bedingung für den Kontext `meeting` erfüllt ist, während für B vier Personen ausreichen. Im Fall von Gaia ist daher einzig diejenige Interpretation relevant, die im Context-Service durchgeführt wird. Diese ist für alle Applikationen gleichermassen gültig; es gibt keine andere Interpretation. Damit ergeben sich jedoch folgende zwei Probleme. Zum einen braucht es eine Absprache zwischen den Middleware-Herstellern und den Applikations-Entwicklern. Die Entwickler von Applikationen müssen sicher gehen, dass jede Umgebung in der ihres Applikation laufen wird, einen Kontext vom Typ `meeting` bereit stellt. Zum anderen ist mit dem Kontext-Typ ebenfalls eine Semantik verbunden, d.h. eine Definition, was genau unter dem Kontext des Typs `meeting` verstanden wird. Welche Umstände sind erforderlich, damit ein `meeting` Kontext als solcher erkannt wird? Wenn nur ein Kontext-Typ `meeting` vorgesehen wird, kann dieser unterschiedlich definiert werden, z.B. durch eine unterschiedliche Anzahl von Personen im Raum für ein Meeting.
Abgesehen davon, dass sich die Hersteller von Middleware und die Entwickler von Anwendungen auf eine Menge von Kontext-Typen einigen müssen – was in einem engen Bereich, z.B. smarte Konferenz-Räume, prinzipiell möglich sein dürfte – drängt sich die Frage auf, ob grundsätzlich Regeln existieren, welche eine universelle Gültigkeit besitzen. Die Anzahl Personen, die für ein Meeting erforderlich sind, gehört offensichtlich nicht dazu. Ist dies nicht der Fall, dann existieren für ein und den selben Kontext mehrere Regeln. Die Frage ist nun, welche der Regeln in einem System verwendet werden, d.h. wer diese Regelndefiniert. Unter Umständen kann die Art und Weise, wie eine Regel definiert wird, einen grossen Einfluss auf die Anwendungen haben; bestimmt der Kontext doch das Verhalten der Applikation, was unter Umständen ein sicherheitstechnisches oder gar ein gesellschaftspolitisches Problem zur Folge haben kann. Doch dieser Aspekt soll hier nicht weiter untersucht werden.

4 Diskussion der Realisierungen

In Abschnitt 3.1 und 3.2 wurden iROS und Gaia als Realisierung einer Middleware für smarte Umgebungen vorgestellt. In diesem Abschnitt sollen die beiden Realisierungen einander gegenübergestellt und verglichen werden.
Eine vergleichende Betrachtung erscheint auf den ersten Blick schwierig, denn die Entwickler der Systeme haben beim Design unterscheidliche Ziele verfolgt. Bei iROS der Stanford University gilt das Hauptaugenmerk der Applikations-Koordination und dem daraus resultierende Kommunikations-Modell. Die iROS Middleware stellt den Applikationen einen Tupelraum zur Verfügung, worüber diese durch Austauschen von Events kommunizieren können. Das Novum bei iROS ist der vom Linda-Tupelraum abgeleitete Event-Heap.
Gaia verfolgt einen anderen Ansatz. Die Gaia Middleware besteht im Wesentlichen aus einer Sammlung von Kernel-Services. Die Middleware selbst basiert auf einem verteilten Objekt-Modell. Die Kernel-Services sind als CORBA Komponenten implementiert. Im Vergleich zu iROS wird hier bekannte Technologie verwendet.
Die iROS-Entwickler folgen einem low-level Ansatz, indem die Middleware lediglich die Koordinations-Infrastruktur bereit stellt, jedoch keine weiteren Dienste. Die Applikationen müssen den Tupelraum entweder durch Polling oder durch Verwendung des Publish/Subscribe-Mechanismus überwachen. Da die Attribute der Tupel Namen besitzen, ≪self-describing≫ sind, können oder sogar müssen die Applikationen *individuell* den vorhandenen Event-Tupel eine Bedeutung beimessen. Wie oben

beschrieben, ist das Verhalten von Anwendungen für smarte Umgebungen kontextabhängig. In iROS müssen die Applikationen selbst die erforderlichen Kontext-Informationen generieren. Die einzige Möglichkeit besteht darin die Tupel in geeigneter Weise zu interpretieren. Damit verbunden ist auch das Problem, dass verschiedene Anwendungen die Tupel unterschiedlich interpretieren können. Die Applikations-Entwickler müssen sich über eine gemeinsame Menge von Attribut-Namen einigen, ansonsten ist eine maschinelle Interpretation der Tupel-Attribute nicht möglich. Über die Menge der Attributnamen muss als sogenanntes Begriff-System Einigkeit unter den Applikationsentwicklern herrschen. Ansonsten ist eine maschinelle Interpretation nicht möglich. In einer Marktsituation von mehreren Software-Herstellern wird eine Einigkeit nur in Form einer Standardisierung möglich sein.

Im Gegensatz zum iROS-System beziehen die Applikationen in Gaia die Kontext-Informationen vom Context-Service. Hier ist es nicht erforderlich, dass die Anwendungen aufgrund eigener Regeln die Kontextdaten generieren. Der von Gaia festgelegte Kontext gilt für alle Applikationen. Unterschiede in der Auslegung des Regelwerkes gibt es hier nicht, da gundsätzlich nur eines existiert, nämlich das im Context-Service. Die Anwendungen handeln hier entsprechend dem zur Verfügung gestellten Kontext. Wie im letzten Abschnitt erläutert wurde, besteht hier ein Problem in der Universalität der Semantik der Kontext-Regeln. Damit die Applikationen in unterschiedlichen smarten Umgebungen betrieben werden können, ist eine einheitliche Definition des Regelwerkes erforderlich. Da im Fall von Gaia die Middleware die Kontext-Regeln auswertet, müssen sich hier hauptsächlich die Middleware-Hersteller oder letztlich die Administratoren von Middleware-Systemen auf ein einheitliches Regelwerk einigen. Ein Applikationsentwickler muss davon ausgehen können, dass beispielsweise für den Kontext `meeting` gleiche Regeln verwendet werden.

Wie bereits in iROS muss auch in Gaia ein Begriffs-System definiert werden. Während es bei iROS die Menge der Attribut-Namen umfasst, muss das Begriffs-System bei Gaia die Context-Typen beschreiben, welche die Anwendung von einer konformen Middleware abfragen kann. Eine Standardisierung dieses Begriff-Systems dürfte – wenn überhaupt – nur in einem engen Begriffsumfeld möglich sein.

Interessanterweise wird in keinem der betrachteten Paper [3], [5] und [7] auf diese Problematik hingewiesen. Aufgrund erwähnten Probleme haben diese Systeme die Marktreife noch nicht erreicht. Bevor kein einheitliches Begriff-System geschaffen wird, bleibt die Zahl der Applikationen klein. Die Anwendungen werden spezifisch für bestimmte Middleware-Systeme entwickelt. Sie werden daher nicht in einer anderen als der dafür vorgesehenen smarten Umgebung verwendet werden können.

Literatur

[1] Junestrand S.: *Private and Public Digital Spaces*. International Journal of Human Computer Interaction; Secial issue of Home use of IT. 2001.

[2] Grimm R.: *System Support for Pervasive Applications*. Dissertation, Univerity of Washington. 2002.

[3] Borchers J. et al.: *Stanford Interactive Workspaces: A Framework for Physical and Graphics User Interface Prototyping*. IEEE Wireless Communications, special issue on Smart Homes. 2002.

[4] Gelernter D., Bernstein A.: *Distributed Communication via Global Buffer*. Proceedings of ACM Symposium on Principles of Distributed Computing. August 1982, pp.10-18.

[5] Johnson B., Fox A.: *The Event Heap: A Coordination Infastructure for Interactive Workspaces*. Proc 4^{th} IEEE Workshop on Mobile Computing Systems and Applications. 2002.

[6] Gelernter D., Carriero N.: *Coordination Languages and their Significance*. Communications of the ACM, Vol. 35 No. 2, pp. 96-105, 1992.

[7] Román M. et al.: *Gaia: A Middleware Infrastructure to Enable Active Spaces.* IEEE pervasive computing, Vol. 1 No 4, pp. 74-83, 2002.