Institut für Parallele und Verteilte Systeme

Abteilung Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Studienarbeit Nr. 1975

Entwicklung eines asynchronen und indirekten Kommunikationsmechanismus für ubiquitäre Systeme

Holger Katz

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Betreuer	Dipl.-Inf. M.Sc. (USA) Stephan Urbanski
Begonnen am:	01. November 2004
Beendet am:	03. Mai 2005
CR-Klassifikation:	C.2.4, D.2.11

Kurzfassung

Ubiquitäre Systeme zeichnen sich durch eine Vielzahl an kleinen und mobilen Geräten aus, die in Alltagsgegenstände integriert sind. Durch die hohe Mobilität weisen sie ein hohes Maß an Asynchronität auf und erfordern für die Kommunikation eine Entkopplung von Aufrufer und Dienst. Viele Verteilungsinfrastrukturen für ubiquitäre Systeme bieten allerdings nur eine synchrone und direkte Kommunikation an.

Diese Arbeit beschreibt die Entwicklung eines indirekten und asynchronen Kommunikationsmechanismus für ubiquitäre Systeme. Dabei werden die Anforderungen, die sich durch das Systemumfeld an einen solchen ergeben, aufgestellt und existierende Arbeiten untersucht. Es wird im Anschluss eine Architektur auf der Basis des Message Queuing Konzepts vorgeschlagen. Diese dient als Ausgangspunkt für die Realisierung eines konkreten Systems unter Verwendung der der Middleware BASE. Im Abschluss wird die Implementierung evaluiert und ein Ausblick auf zukünftige Erweiterungen gegeben.

Inhaltsverzeichnis

Inhaltsverzeichnis I
Abbildungsverzeichnis III
Tabellenverzeichnis IV

1 Einleitung 1
2 Grundlagen 2
2.1 Ubiquitäre Systeme 2
2.2 BASE 4
2.3 Kommunikationsformen 5
2.3.1 Shared Memory 7
2.3.1.1 Tuple Spaces 8
2.3.2 Message Queuing 9
3 Verwandte Arbeiten 11
3.1 Websphere MQ 11
3.2 Java Message Service 16
3.3 TSpaces 19
3.4 JavaSpaces 22
3.5 Lime – Linda in a Mobile Environment 24
3.7 Diskussion 27
4 Anforderungen 29
5 Konzeption 31
6 Implementierung 38
6.1 Integration in BASE 38
6.2 Nachrichtenstruktur 39
6.3 QueueManager 40
6.4 QueueLookupManager 42
6.5 Queues 42
6.5.1 QueueConnection 44
7 Evaluation 46

7.1 Implementierung 46
7.2 Speicherbedarf 47
7.2.1 Messung 48
7.2.2 Auswertung 49
7.3 Geschwindigkeit 50
7.3.1 Messung 50
7.3.2 Auswertung 53
8 Zusammenfassung und Ausblick 55
8.1 Zusammenfassung 55
8.2 Ausblick 57
Literaturverzeichnis 59

Abbildungsverzeichnis

Abbildung 1: BASE Architektur 4
Abbildung 2: Direkte symmetrische Adressierung 6
Abbildung 3: Mailbox 6
Abbildung 4: Shared Memory 8
Abbildung 5: Message Queuing 9
Abbildung 6: Websphere MQ Aufbau (Quelle: [15]) 12
Abbildung 7: Websphere MQ Nachrichtenformat (Quelle: [8]) 14
Abbildung 8: JMS Administration (Quelle: [24]) 17
Abbildung 9: JMS Queue (Quelle: [25]) 17
Abbildung 10: JMS Topic (Quelle: [25]) 18
Abbildung 11: TSapces Überblick (Quelle: [17]) 20
Abbildung 12: TSpaces Design (Quelle: [17]) 22
Abbildung 13: JavaSpaces Kommunikation (Quelle: [26]) 23
Abbildung 14: Lime Tuple-Spaces (Quelle: [22]) 25
Abbildung 15: Queuing-System Architektur 32
Abbildung 16: QueueConnection Kommunikation 36

Tabellenverzeichnis

Tabelle 1: TSpaces Query-Beispiele 21
Tabelle 2: QueueConnection Funktionen 45
Tabelle 3: Umfang und Komplexität..... 47
Tabelle 4: Speicherbedarf BASE 49
Tabelle 5: Speicherbedarf Queuing-System (inklusiv BASE) 49
Tabelle 6: Testumgebung 51
Tabelle 7: Geschwindigkeit BASE 51
Tabelle 8: Geschwindigkeit Queuing-System 53

1 Einleitung

Mit der fortschreitenden Entwicklung der Technik zeichnen sich verstärkt neue Einsatzmöglichkeiten für Rechnersysteme ab. Die eingesetzten Geräte wurden in den letzten Jahren immer kleiner und mobiler. Dabei sind die Systeme oft nicht mehr als Computer erkennbar, sondern werden in Alltagsgegenstände integriert und Menschen nutzen sie, ohne dass sie sich dessen bewusst sind. Dieser Trend führt zur Allgegenwärtigkeit von Rechnersystemen, die auch als Ubiquitous Computing bezeichnet wird. Er ist heute schon zum Beispiel an der Verbreitung von Mobiltelefonen oder Personal Digital Assistants zu erkennen (PDAs).

Ubiquitäre Systeme zeichnen sich also durch eine Vielzahl an mobilen und heterogenen Geräten aus, die spontan miteinander kommunizieren. An der Universität Stuttgart wird mit BASE eine Verteilungsinfrastruktur entwickelt, die speziell für den Einsatz in ubiquitären Systemen gedacht ist. Sie bietet eine Reihe von Mechanismen, die den Einsatz in spontan vernetzten Systemen ermöglichen und zeichnet sich durch hohe Flexibilität und Konfigurierbarkeit aus, womit sie auf verschiedensten Rechnern mit unterschiedlichen Ressourcen genutzt werden kann. BASE bietet derzeit Anwendungen die Möglichkeit zur synchronen und direkten Kommunikation.

Die Struktur von ubiquitären Systemen lässt allerdings vermuten, dass ein asynchroner und indirekter Kommunikationsmechanismus besser für die Kommunikation in diesen geeignet ist. Im Rahmen dieser Arbeit, soll deshalb ein solcher entwickelt und getestet werden. Dazu werden Inhalte bereits bestehender Projekte analysiert und darauf aufbauend eine Konzeption für die Realisierung des Anforderungsprofils auf der Basis von BASE definiert. Im Rahmen einer Evaluierung der Implementierung sollen Informationen über die Anwendbarkeit gewonnen werden.

2 Grundlagen

In diesem Kapitel werden die Grundlagen erläutert, die zum Verständnis der Arbeit notwendig sind. Dabei werden zuerst ubiquitäre Systeme vorgestellt, für deren Umgebung der Kommunikationsmechanismus entwickelt werden soll. Anschließend wird das Middleware-System BASE betrachtet, das als Grundlage dient. Im Abschluss werden verschiedene Kommunikationsformen näher betrachtet.

2.1 Ubiquitäre Systeme

Unter dem von Mark Weiser[32,33,34] geprägten Begriff Ubiquitous Computing wird die Allgegenwärtigkeit von Rechnersystemen verstanden, ohne dass sie für die Nutzer erkennbar sind.

> *„Ubiquitous computing is the method of enhancing computer use by making many computers available throughout the physical environment, but making them effectively invisible to the user.”[32]*

Ziel ist es, die Systeme so in den Alltag zu integrieren, dass nur noch die nötigsten Ein- und Ausgabegeräte sichtbar sind. Der Mensch ist einfach nur Nutzer ohne den Geräten größere Aufmerksamkeit schenken zu müssen. Die Computer werden Teil der physischen Umgebung. Dabei stehen ubiquitäre Systeme im Gegensatz zur virtuellen Realität, in der der Menschen in eine comptutergenerierte Welt gebracht werden bzw. die Welt im Computer abgebildet wird. Einen Schritt in Richtung Ubiquitous Computing gehen zum Beispiel Embedded Systems, die in Gebrauchsgegenständen wie Autos oder Waschmaschinen integriert sind, ohne dass es den jeweiligen Benutzern bewusst wird.

Die in ubiquitären Systemen zum Einsatz kommenden Geräte haben die unterschiedlichsten Leistungsklassen. Die Spanne reicht von ressourcenarmen Systemen, wie Sensoren, die nur ein Minimum an Speicher und Rechenleistung besitzen, bis hin zu leistungsstarken Rechnern, die für rechenintensive Aufgaben geeignet sind. Auch sind Systeme für unterschiedlichste Zwecke in einem

ubiquitären Umfeld vorhanden. So kommen hochspezialisierte Geräte für bestimmte Einsatzfelder genauso vor, wie universelle Rechner, die für fast alle Berechnungen genutzt werden können.

Ebenso vielfältig, wie die eingesetzten Geräte, sind die verwendeten Übertragungsmedien und Netzwerkprotokolle für die Kommunikation. Es werden zum Beispiel sowohl die herkömmliche Verkabelung eingesetzt als auch neuere Techniken wie die Funkübertragung per Bluetooth verwendet.

Eine weitere Eigenschaft von ubiquitären Systemen ist die hohe Dynamik. Mehrere Geräte können sich zu Gruppen gleichberechtigter Partner zusammenfinden (Peer-to-Peer) und innerhalb dieser miteinander kommunizieren. Ein System kann eine solche Gruppe spontan betreten oder wieder verlassen. Dabei ist der Abbruch der Kommunikation nicht nur auf den Ausfall eines Systems zurückzuführen, sondern kann auch deshalb geschehen, weil ein Gerät von seinem Benutzer aus dem Kommunikationsbereich der Gruppe herausgeführt wird und die Verbindung abbricht. Zum Beispiel finden sich die Geräte von Teilnehmern an einer Konferenz zu einer Gemeinschaft zusammen und tauschen Informationen aus. Verlässt ein Teilnehmer den Konferenzraum, so sind die Geräte, die er mit sich führt, nicht mehr Teil dieser Gruppe und können nicht mehr mit den anderen Teilnehmern kommunizieren.

Bei der Entwicklung einer Anwendung für ubiquitäre Systeme müssen folgende Punkte bedacht werden: Es ist darauf zu achten, dass die zum Einsatz kommenden Geräte nur über geringe Ressourcen verfügen können, dass unterschiedlichste Netzwerktechnologien eingesetzt werden und vor allem muss die Anwendung auf die hohe Dynamik in einem solchen System reagieren können. Auch ist auf einen möglichst geringen Energieverbrauch zu achten, zum Beispiel durch sparsamen Einsatz von verbrauchsintensiven Übertragungsmedien wie WLAN, da kleine Geräte, wie sie im ubiquitären Umfeld zum Einsatz kommen, meist nur über geringe Energievorräte verfügen.

2.2 BASE

BASE[1,28,35] ist eine Verteilungsinfrastruktur, die im Rahmen des Peer to Peer Pervasive Computing Projekts an der Universität Stuttgart entwickelt wurde. Das System wurde speziell für den Einsatz in ubiquitären Systemen konzipiert.

Clients können bei BASE, wie bei anderen Verteilungsinfrastrukturen auch, mittels Stellvertreterobjekten auf Dienste zugreifen. Das Kernstück von BASE, ein erweiterbarer mikrokernbasierter Object Request Broker (ORB), ist dabei für die Zustellung der Aufrufe verantwortlich. Er gibt sie entweder an einen lokalen Dienst oder einen entfernten ORB weiter, der mit dem aufgerufenen Dienst verbunden ist.

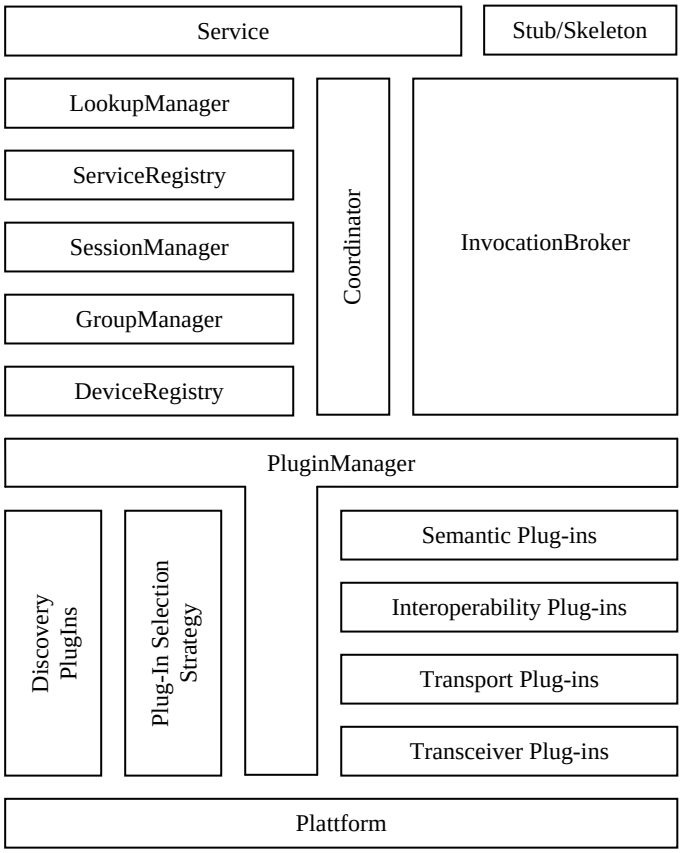

Abbildung 1: BASE Architektur

Der Kern des ORB besteht aus dem *InvocationBroker*, der für das eigentliche Zustellen von Aufrufen zuständig ist, und zwei Registrierungen, der *DeviceRegistry* und der *ServiceRegistry*, die ihn dabei unterstützen sollen. Die *DeviceRegistry* ist für

die Verwaltung von entfernten Geräten zuständig, während die *ServiceRegistry* lokale Objekte bzw. Services verwaltet.

Zur Kommunikation (Marshalling und Transport) verwendet BASE Plug-ins. Diese werden für jeden Kommunikationsversuch dynamisch ausgewählt. Dadurch können für das Senden und Empfangen desselben Aufrufs jeweils unterschiedliche Plug-ins gewählt werden. Durch den Plug-in-Mechanismus wird die Funktion des *InvocationBrokers* vom eigentlichen Versand der Aufrufe getrennt. Daraus ergibt sich der Vorteil, dass das System leicht an die unterschiedlichen Eigenschaften der Geräte bzw. der vorhandenen Kommunikationsmöglichkeiten angepasst werden und auf Änderungen bei der Verbindung reagieren kann.

Um das Auffinden von Geräten zu ermöglichen, benutzt Base einen Discovery-Mechanismus. Dieser ist allerdings nicht fest in Base integriert, wie zum Beispiel bei Jini[5,6,40], sondern wird über Discovery-Plug-ins hinzugefügt. Wird ein neues System entdeckt, werden Informationen über vorhandene Plug-ins ausgetauscht. Dies ermöglicht dem BASE-System die spätere Auswahl der Plug-ins für den Transport von Aufrufen.

Ein weiteres Merkmal von BASE sind die niedrigen Anforderungen an die Geräte. Damit wird auch der Einsatz auf stark ressourcenbeschränkten Rechnern ermöglicht. Durch dies und die verwendeten Plug-in-Mechanismen eignet sich das System besonders für den Einsatz in einem ubiquitären Umfeld.

2.3 Kommunikationsformen

Direkte vs. Indirekte Adressierung

Bei dem Versand von Nachrichten sind zwei Adressierungsarten zu unterscheiden. Die direkte Adressierung und die indirekte Adressierung.

Bei der direkten Adressierung wird der Empfänger vom Sender einer Nachricht explizit angegeben. Dabei unterscheidet man zwischen symmetrischer und asymmetrischer Adressierung. Bei der symmetrischen Adressierung sind Sender und

Empfänger gegenseitig bekannt. Bei der asymmetrischen Adressierung ist der Empfänger einer Nachricht nur dem Sender bekannt. Der Empfänger entnimmt die Adresse des Senders der Nachricht.

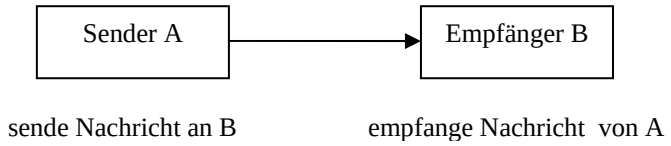

Abbildung 2: Direkte symmetrische Adressierung

Bei der indirekten Adressierung spricht der Sender den Empfänger nicht direkt an, sondern kommuniziert mit ihm über eine Empfangsstelle.

Indirekte Adressierung ist zum Beispiel über eine Mailbox möglich. Sie ist ein global bekanntes Objekt, welches über einen Puffer verfügt. Dort legen Prozesse Nachrichten ab, die von den Empfängern dort abgeholt werden. Eine Mailbox ermöglicht zudem eine n:m Kommunikation.

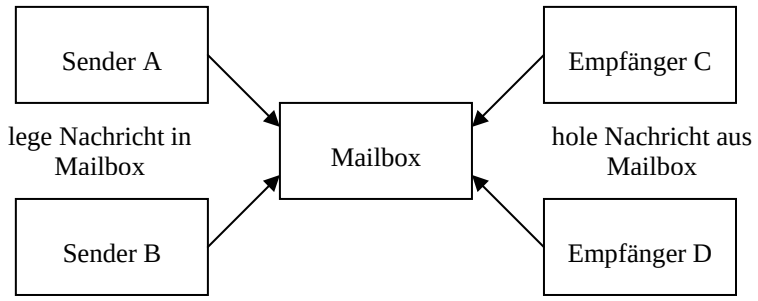

Abbildung 3: Mailbox

Eine weitere Möglichkeit der indirekten Adressierung ist die Kommunikation über Ports. Im Gegensatz zur Mailbox wird ein Prozess direkt an den Port gebunden, was nur eine n:1 Kommunikation zulässt. Es können zwar Nachrichten von mehreren Prozessen an den Port gesendet werden, aber nur der Besitzer des Ports kann Nachrichten von dort empfangen.

Synchrone vs. Asynchrone Kommunikation

Bei der synchronen (blockierenden) Kommunikation wartet der Sender einer Nachricht, bis diese vom Empfänger verarbeitet wurde und eine Antwort zurückkommt (RPC). Ein Vorteil der synchronen Kommunikation ist, dass kein Puffer verwendet werden muss um Nachrichten zwischenzuspeichern, da sie direkt in den Adressraum des Empfängers kopiert werden können.

Beim asynchronen Nachrichtenaustausch wartet der Sender einer Nachricht nicht bis diese beim Empfänger angekommen ist, sondern kehrt vom Sendeprozess direkt zurück. Dafür müssen die Nachrichten in einen Puffer geschrieben werden, aus dem sie zu einem späteren Zeitpunkt wieder abgeholt werden können, da nicht davon ausgegangen werden kann, dass ein Prozess zum Sendezeitpunkt bereit ist, eine Nachricht entgegenzunehmen. Vorteilhaft bei der asynchronen Kommunikation ist die zeitliche Entkopplung von Sende- und Empfangvorgang. Der Sender wird nicht blockiert bis der Empfänger bereit ist die Nachricht entgegenzunehmen und kann so seine Arbeit fortsetzen.

In den nächsten Kapiteln werden Methoden vorgestellt, die eine indirekte asynchrone Kommunikation für Anwendungen ermöglichen.

2.3.1 Shared Memory

Beim Konzept des Shared Memory greifen mehrere Anwendungen auf einen gemeinsamen Speicher zu. Für die beteiligten Prozesse liegt dieser im eigenen Adressraum und kann so mit normalen Speicherzugriffsoperationen manipuliert werden. Um den gleichzeitigen Zugriff der Anwendungen auf eine Seite des gemeinsamen Speichers zu verhindern, müssen Locking-Strategien eingesetzt werden.

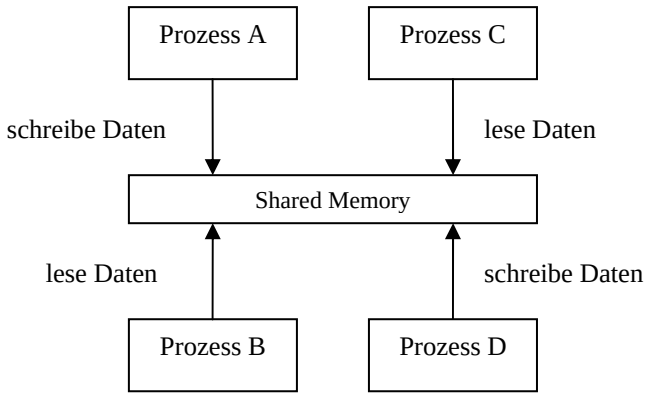

Abbildung 4: Shared Memory

Um diese Methode der Interprozesskommunikation (IPC) auch in Rechnernetzen einsetzen zu können, nutzt man das Konzept des Distributed Shared Memory (DSM). In einem DSM-System wird ein globaler Adressraum simuliert, auf den Anwendungen zugreifen können. Dadurch ist es für die Prozesse möglich mit den üblichen Speicherzugriffsoperationen auf entfernten Speichern zu arbeiten. Änderungen im Speicher werden über das Netzwerk an die beteiligten Rechner propagiert, um ihn konsistent zu halten. Ein DSM-System wird beispielsweise mit Plurix[42,43] angeboten.

2.3.1.1 Tuple Spaces

Das TupleSpace-Konzept[41,47] wurde von David Gelernter als Teil von Linda entwickelt. Ein Tuple Space ist ein assoziativer geteilter Speicher. Anwendungen tauschen hier Daten in Form von Tupeln über einen Tupelraum aus. Die einzelnen Tupel liegen dabei ungeordnet im Tupelraum, müssen dabei aber nicht einzigartig sein. Ein Tupel ist ein Vector aus Werten eines festgelegten Typs. Die Tupel werden von Prozessen erzeugt und in den Tupelraum geschrieben. Beim Lesen werden die Tuple mittels Pattern-Matching ausgesucht und das erste passende Tupel wird zurückgegeben. Die Auswahl des Tupels ist abhängig von der jeweiligen Implementierung.

Für das Pattern-Matching werden Templates eingesetzt. Ein Template ist ebenfalls ein Vector aus Werten, bei dem allerdings einige Felder aus typisierten Platzhaltern

bestehen können, so genannten formalen Feldern. Ein formales Feld in einem Template entspricht einem Tupelfeld, wenn beide den gleichen Typ besitzen. Bei nicht formalen Feldern müssen beide außerdem den gleichen Wert aufweisen. Ein Template entspricht also einem Tupel, wenn beide die gleiche Länge haben und die Templatefelder den dazugehörigen Tupelfeldern entsprechen. Zum Beispiel passt das Template <integer, “hello world“, 345> zu dem Tupel <12, “hello world“, 345>.

2.3.2 Message Queuing

Beim Message Queuing kommunizieren Anwendungen miteinander, indem sie Nachrichten über Queues austauschen.

Eine Nachricht besteht dabei im Normalfall aus zwei Teilen: dem Nachrichtenkopf und den Nutzdaten der Anwendungen. Der Nachrichtenkopf enthält zum Beispiel eine eindeutige Nachrichten-ID und Parameter wie Prioritäten oder Eigenschaften, die von den Anwendungen gesetzt werden können.

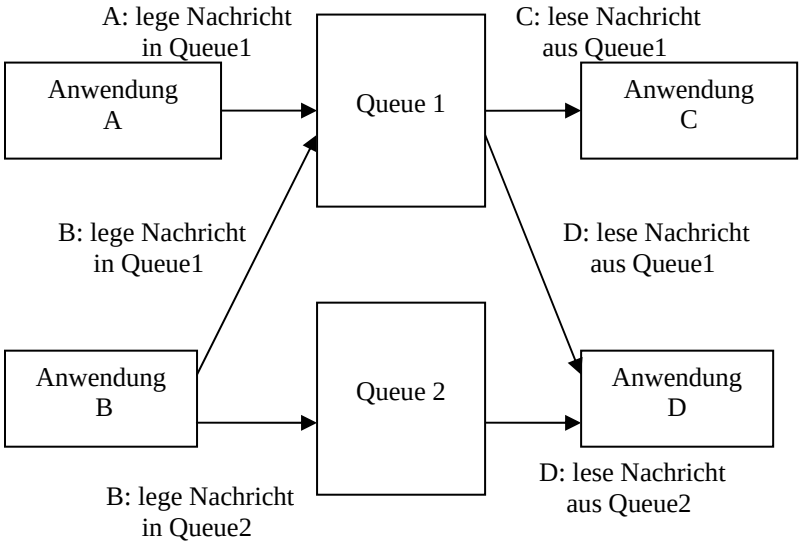

Abbildung 5: Message Queuing

Queues besitzen einen global eindeutigen Namen über den sie adressiert werden können. Sender stellen Nachrichten in die Queue. Diese werden dort gehalten bis

der Empfänger sie abholt. Mehrere Sender bzw. Empfänger können dieselbe Queue nutzen um Daten auszutauschen. Die Nachrichten werden im Normalfall in der Reihenfolge, in der sie eingeliefert wurden, wieder ausgegeben (First-in-First-out, FIFO). Viele Message-Queuing-Systeme unterstützen allerdings Prioritäten und den Zugriff auf bestimmte Nachrichten innerhalb einer Queue. Auch bieten diese Systeme an, die Nachrichten persistent zu speichern, so dass diese auch nach dem Ausfall eines Systems weiter vorhanden sind.

In diesem Kapitel wurde zuerst auf ubiquitäre Systeme eingegangen. Dabei wurde festgestellt, dass Entwicklungen für dieses Umfeld auf die Heterogenität, Dynamik und mögliche Ressourcenknappheit der Geräte eingehen müssen. Im Anschluss wurde BASE vorgestellt. Diese Middleware soll als Ausgangspunkt für den zu entwickelnden Kommunikationsmechanismus dienen. Abschließend wurden die gängigsten Kommunikationsformen und Adressierungsarten in Verteilten Systemen vorgestellt. Dabei wurde festgehalten, dass es im Wesentlichen zwei Konzepte für die asynchrone und indirekte Kommunikation gibt. Zum einen ein Shared Memory-Ansatz zum Beispiel unter Verwendung von Tupelräumen zum anderen das Prinzip des Message Queuing.

3 Verwandte Arbeiten

In diesem Kapitel sollen verschiedene Projekte vorgestellt werden, die sich mit indirekter asynchroner Kommunikation beschäftigen. Anschließend findet eine Bewertung der Systeme über ihre Einsatzmöglichkeiten in ubiquitären Systemen statt.

3.1 Websphere MQ

Websphere MQ (früher MQSeries)[9,10,11,12,13,14,15] ist ein Message-Queuing-System, das von der Firma IBM entwickelt wurde. Es unterstützt mehrere Plattformen und garantiert das einmalige Senden wichtiger Nachrichten. Außerdem stellt es ein umfangreiches Application Programming Interface (API) zur Verfügung, das einen einfachen Zugriff auf die Messaging- Dienste erlaubt.

QueueManager

Das Kernstück von Websphere MQ ist der Queue-Manager. Er ist für die Verwaltung von Queues und Messages zuständig. Dafür stellt er Funktionen bereit, mit denen Administratoren Queues anlegen, Eigenschaften von existierenden Queues verändern oder diese löschen können. Außerdem wird von ihm das Message-Queuing-Interface (MQI) angeboten, mit dem Anwendungen Nachrichten in Queues legen bzw. aus ihnen abholen können. Programmen ist es mit dem MQI auch möglich eine Nachricht an eine Queue auf einem entfernten System bzw. QueueManager zu senden. Dabei übergibt die Anwendung dem lokalen QueueManager die Nachricht und überlässt es ihm, die Zielmaschine zu suchen und die Nachricht zu übertragen. Ist die Nachricht zu lang, bietet der QueueManager auch die Möglichkeit diese zu segmentieren. Ein QueueManager erlaubt auch eine sichere Nachrichtenübermittlung durch Verschlüsselung, Zugriffskontrolle und sichere Kanäle. Damit ein Queuemanager identifiziert werden kann, besitzt er einen netzwerkweit eindeutigen Namen. QueueManager können auch so genannte Cluster bilden. Hierbei werden mehrere QueueManager zusammengeschlossen um eine Lastverteilung und einfachere Kommunikation zu ermöglichen.

Anwendungen müssen allerdings nicht auf einem System mit eigenem QueueManager ausgeführt werden. Es wird auch die Möglichkeit angeboten sich als Websphere MQ-Client mit einem QueueManager auf einer anderen Maschine zu verbinden. Dieser Server kann dann wie ein lokaler QueueManager genutzt werden.

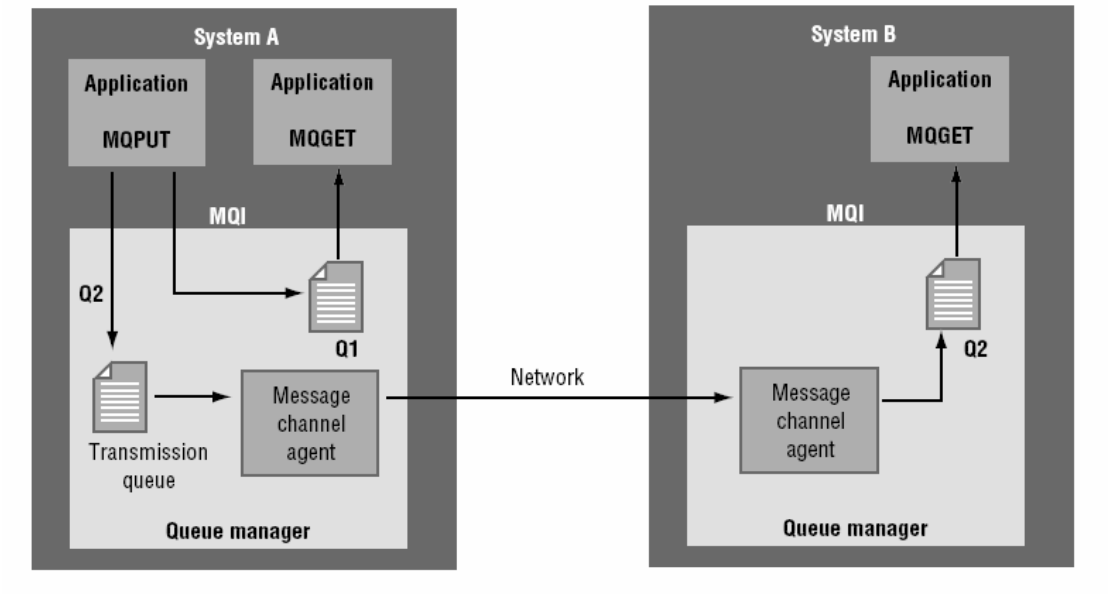

Abbildung 6: Websphere MQ Aufbau (Quelle: [15])

Queues

Um Nachrichten von Programmen zu speichern, nutzt Websphere MQ Queues. Jede von ihnen gehört dabei zu genau einem Queue-Manager. Eine Queue verfügt eine Vielzahl von Attributen, die Informationen über sie bereitstellen. Das Wichtigste davon ist der Queue-Name, der innerhalb eines Queue-Managers eindeutig ist. Weitere Eigenschaften sind zum Beispiel der Queue-Typ oder eine Beschreibung der Queue in Textform. Über die Kombination von Queue-Manager-Name und Queue-Name kann eine Queue eindeutig adressiert werden. Queues werden in lokale und entfernte Queues unterteilt. Ob eine Queue lokal oder entfernt ist, hängt von der Perspektive der Anwendung ab. Außerdem gibt es noch so genannte Alias-Queues. Diese sind keine richtigen Queues, sondern nur Stellvertreterobjekte für die eigentliche Queue. Dadurch können verschiedene Anwendungen auf dieselbe Queue zugreifen obwohl sie verschiedene Namen benutzen.

Lokale Queues sind aus der Sicht einer Anwendung diejenigen, die von dem QueueManager verwaltet werden, mit dem sie gerade verbunden ist. Eine entfernte

Queue gehört dagegen zu einem QueueManager, zu dem eine Verbindung nur über den lokalen QueueManager besteht. Sie stellt keine eigentliche Queue dar, sie ist nur ein Objekt, das die Informationen enthält, die für den lokalen Queue-Manager nötig sind, um Nachrichten an die entfernte Queue weiterzuleiten. Entfernte Queues werden mit so genannten Transmission Queues verknüpft. Nachrichten können nicht von entfernten Queues gelesen werden.

Transmission-Queues sind spezielle lokale Queues. Sie werden als Zwischenschritt genutzt, wenn eine Nachricht an eine entfernt Queue gesendet werden soll. In ihnen werden die Nachrichten für die entfernte Queue abgelegt, bis eine Verbindung zu dem zuständigen Queue-Manager hergestellt ist. Die Nachrichten werden aus ihr gelöscht sobald sie weitergeleitet wurden. Für entfernte Queue-Manager können auch mehrere Transmission-Queues angelegt werden, die zum Beispiel für verschiedene Dienste zuständig sind.

Channels

Zur Kommunikation in Websphere MQ werden so genannte Channels eingesetzt. Es wird zwischen zwei Typen unterschieden: MQI- und Message-Channels. Erstere ist bidirektional und verbindet MQI-Clients mit einem Queue-Manager. Ein MQI-Client besitzt dabei keinen eigenen QueueManager, sondern es wird mit einem Server via RPCs kommuniziert.

Ein Message-Channel ist eine unidirektionale Verbindung. Er verbindet zwei QueueManager über Message Channel Agents (MCAs) um Nachrichten weiterzuleiten. Ein MCA ist ein Programm, das Nachrichten aus einer Transmission-Queue liest und diese über eine Kommunikationsverbindung an die Zielqueue überträgt. Für eine bidirektionale Kommunikation müssen zwei Message-Channels verwendet werden.

Nachrichten

In Websphere MQ gibt es vier verschiedene Arten von Nachrichten: Datagram, Request, Reply und Report. Ein Datagram ist eine Nachricht, bei der keine Antwort von der Anwendung erwartet wird für die sie bestimmt ist. Ein Request wird genutzt, wenn eine Antwort von dem Programm erwartet wird an das sie gerichtet ist. Die

Antwort auf ein Request ist ein Reply. Reports werden verwendet, um Nachrichten über ein Ereignis zu informieren, zum Beispiel wenn während der Zustellung einer Nachricht ein Fehler auftritt. Ein Report kann vom QueueManager, einem MCA oder einer Anwendung erzeugt werden. Außerdem können zusätzliche Nachrichtentypen definiert werden.

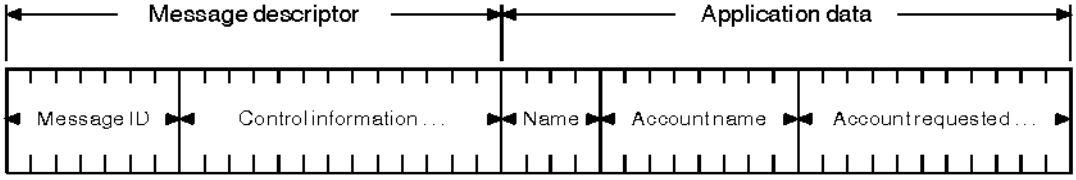

Abbildung 7: Websphere MQ Nachrichtenformat (Quelle: [8])

Eine Nachricht besteht aus zwei Teilen: dem Nachrichten-Descriptor und den Daten der Anwendung. Der Descriptor enthält die Eigenschaften einer Nachricht, die der QueueManager und der Empfänger benötigt, um die Nachricht zu verarbeiten. Er enthält unter anderem eine Nachrichten-ID, die eine eindeutige Identifikation erlaubt. Zusätzlich zu ihr gibt es eine Correlation-ID. Sie kann frei benutzt und von einer Anwendung vergeben werden, um beispielsweise eine Nachricht einer anderen zu zuordnen.

Websphere MQ unterscheidet zwischen persistenten und nicht persistenten Nachrichten. Eine als persistent gekennzeichnete Nachricht wird in einer Log-Datei protokolliert und kann so nach einem Systemausfall wieder hergestellt werden andernfalls geht sie verloren. Wird eine Nachricht einem QueueManager übergeben, so garantiert Websphere MQ eine exactly-once Zustellung dieser, d.h. die Nachricht wird genau einmal in der Zielqueue abgelegt, falls diese existiert. Wird die Nachricht als persistent gekennzeichnet, so wird auch eine exactly-once Zustellung auch im Fehlerfall, also zum Beispiel beim Absturz eines Geräts garantiert.

Bei Request-Nachrichten kann im Nachrichtenkopf von den Anwendungen auch noch angegeben werden an welchen QueueManager bzw. welche Queue die Antwort gesendet werden soll.

Den Nachrichten können auch verschiedene Prioritäten zugewiesen werden, um die Reihenfolge ihrer Auslieferung zu beeinflussen. Je höher die Priorität einer Nachricht

desto weiter vorne wird sie in einer Queue eingereiht. Wird keine Priorität vergeben, wird der Defaultwert der für die Queue gesetzt ist verwendet.

Für eine Nachricht kann auch noch ein Ablaufdatum vergeben werden. Wird die Nachricht nicht innerhalb des gewählten Zeitraumes abgeholt, wird Sie vom QueueManager aus der Queue gelöscht.

Sollte es einem entfernten QueueManager nicht möglich sein, eine Nachricht in eine Queue zu stellen, so werden verschiedene Wege angeboten, wie dies behandelt werden kann. Der QueueManager kann die Nachricht innerhalb eines Reports zurück an den Sender schicken. Die Nachricht kann in einer lokalen *Dead-Letter*-Queue abgelegt werden, in der sie verbleibt bis sie abgeholt wird. Bevor eine Nachricht in eine solche Queue gestellt wird, versucht der QueueManager allerdings diese noch einmal in die Queue zu stellen. Der Wert, wie oft dies wiederholt werden soll, kann im Nachrichtenkopf spezifiziert werden.

Websphere MQ everyplace (MQe)

Mit Websphere MQ everyplace[37,38,39] hat IBM ein Produkt geschaffen, das Message-Queuing auch für kleine und mobile Geräte anbietet. Bei diesem System verfügt jedes Gerät über einen eigenen Queuemanager, der im Wesentlichen dieselben Funktionen wie der von Websphere MQ hat. Es wird allerdings zum Beispiel keine Clusterbildung oder Segmentierung von Nachrichten erlaubt. Über den QueueManager wird auch auf sämtliche lokalen und entfernten Queues zugegriffen. Für die Übertragung von Nachrichten durch den QueueManager an einen entfernten QueueManager bzw. an eine entfernte Queue werden zwei Möglichkeiten angeboten, synchron und asynchron. Im ersten Fall wird mit RPCs auf den entfernten QueueManager zugegriffen und die sendende Anwendung wird solange blockiert, bis die Nachricht in die Zielqueue gelegt wurde. Ist die Möglichkeit gegeben, lokale Queues zu erzeugen, gestattet der QueueManager auch die asynchrone Übertragung von Nachrichten an die Zielqueue. Dabei wird genauso verfahren wie bei Transmission-Queues bei Websphere MQ. Um auf MQ-Server zugreifen zu können, wird von MQe noch eine Schnittstelle für die Kommunikation mit diesen angeboten.

Die Nachrichten in MQe bestehen nicht aus Nachrichtenkopf und –körper wie in MQ, sondern setzen sich aus einer Nachrichten-Id, einem Namen, einem Datentyp und den Anwendungsdaten zusammen. Die Persistenz von Nachrichten wird bei MQe auf Queueebene realisiert. Eine Queue kann also so konfiguriert werden, dass sie Nachrichten persistent speichert. Ansonsten werden im Wesentlichen die gleichen Queue-Typen angeboten, wie bei Websphere MQ.

3.2 Java Message Service

Der Java Message Service[23,24,25,26] (JMS) wurde von Sun entwickelt, um Java-Entwicklern eine einheitliche Schnittstelle zu bieten, mit der sie auf Enterprise-Messaging-Systeme zugreifen können. JMS ist eine Sammlung von Schnittstellen und schreibt keine konkrete Implementierung vor. Sie wird dem Hersteller der jeweiligen Message-oriented Middleware (MOM) überlassen. JMS abstrahiert also von den herstellerspezifischen APIs der zugrunde liegenden Systeme. Implementierungen für JMS gibt es zum Beispiel für Websphere MQ oder Weblogic.

Die Architektur des JMS besteht aus folgenden Komponenten:

- Dem JMS-Provider. Dies ist das Messaging-System, das die JMS Schnittstelle zusätzlich zu den Administrations- und Kontrollkomponenten, die für ein vollwertiges Messaging-System notwendig sind, implementiert.
- Den JMS-Clients. Die JMS-Clients sind Java-Anwendungen, die Nachrichten senden oder empfangen. Sie sind also die eigentlichen Nutzer des JMS.
- Den ausgetauschten Informationen in Form von Message-Objekten.
- Und den Administered Objects. Diese sind vorkonfigurierte JMS Objekte, die von einem Administrator angelegt werden und Informationen über den Provider enthalten, die von den Clients genutzt werden können.

Im JMS gibt es zwei Arten von Administered Objects: Connection Factories und Destinations. Eine Connection Factory wird vom Client verwendet, um eine Verbindung mit dem Provider herzustellen. Destinations nutzt der Client, um das Ziel oder die Quelle einer Nachricht zu spezifizieren, die er senden bzw. empfangen will. Die Destinations repräsentieren also die Warteschlangen des Providers.

Die Administered Objects können von Anwendungen über das Java Naming und Directory Interface (JNDI) lokalisiert und eingebunden werden. Dabei kann eine JMS-Implementation einen eigenen Serviceprovider anbieten oder auf einen vorhandenen zurückgreifen, zum Beispiel einen LDAP-Server.

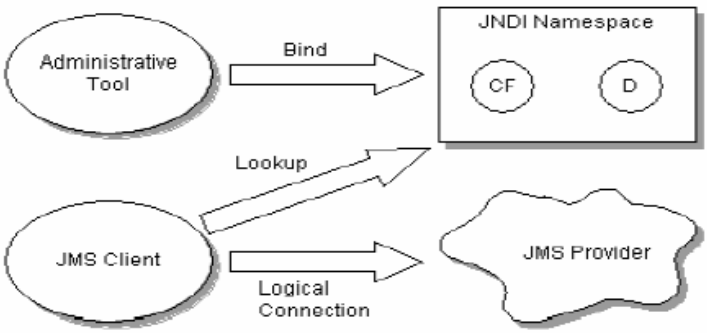

Abbildung 8: JMS Administration (Quelle: [24])

Der JMS unterstützt zwei Kommunikationsmöglichkeiten, Queues und Topics. Queues dienen dazu, Punkt-zu-Punkt Verbindungen zwischen zwei Partnern herzustellen. Die Nachrichten werden dabei an eine bestimmte Queue adressiert und verbleiben in ihr, bis sie abgeholt werden oder verfallen. Bei dieser Kommunikationsform besitzt jede Nachricht nur einen Empfänger.

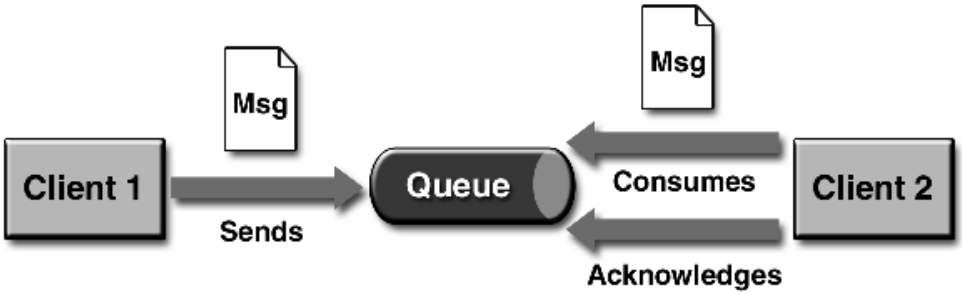

Abbildung 9: JMS Queue (Quelle: [25])

Topics funktionieren nach dem Publish-Subscribe Prinzip. Ein Sender verschickt eine Nachricht an ein Topic. Alle Empfänger, die sich für dieses registriert haben, erhalten die Nachricht. Ein Empfänger erhält die Nachrichten nur, wenn er eingeschrieben ist. Nachrichten, die vorher oder später eintreffen, sind nicht zugänglich. Der JMS bietet allerdings die Möglichkeit so genannte Durable Subscriptions zu erstellen. Ein Client

kann diese Nachrichten auch empfangen, wenn sie publiziert wurden solange er nicht in das Topic eingeschrieben war, da sie dauerhaft im Topic verbleiben.

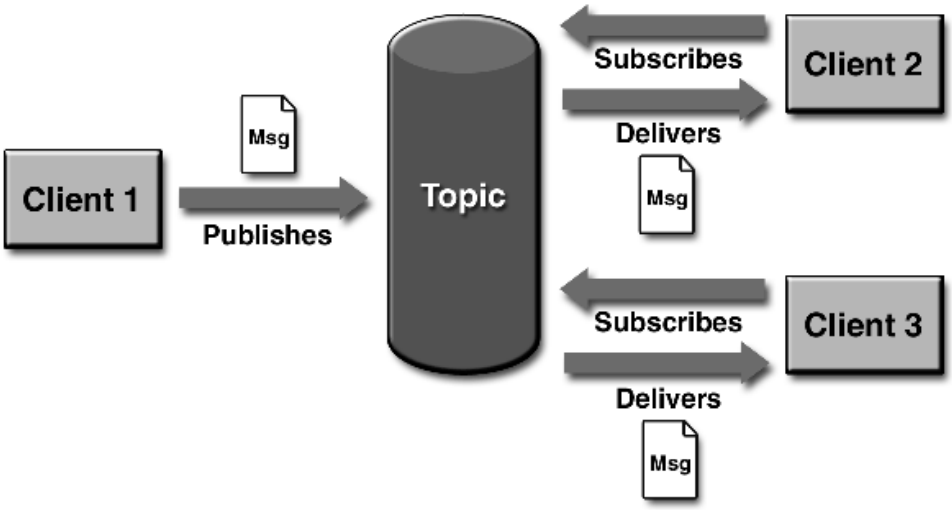

Abbildung 10: JMS Topic (Quelle: [25])

Um die Zuverlässigkeit der Nachrichtenübermittlung zu gewährleisten, stellt das JMS verschiedene Methoden bereit. Zum einen muss der Empfang einer Nachricht bestätigt werden. Bis zum Eintreffen einer Empfangsbestätigung gilt eine Nachricht als nicht ausgeliefert. Es werden drei Vorgehensweisen unterschieden: *AUTO_ACKNOWLEDGE*, d.h. die Nachrichten werden automatisch nach der erfolgreichen Auslieferung bestätigt. Beim *CLIENT_ACKNOWLEDGE* bestätigt der Empfänger explizit den Erhalt von Nachrichten. Die Bestätigung erfolgt in diesem Fall jeweils für eine ganze Sitzung, zum Beispiel wenn der Empfänger zehn Nachrichten während einer Sitzung empfangen hat und nach der fünften eine Bestätigung sendet, gelten auch die nachfolgenden als bestätigt. Die letzte Vorgehensweise ist der *DUPS_OK_ACKNOWLEDGE*. Hier werden die Nachrichten von der Sitzung nur ab und zu automatisch bestätigt, was dazu führen kann, dass Nachrichten mehrfach übermittelt werden.

Es gibt außerdem die Möglichkeit, Nachrichten als *PERSISTENT* zu kennzeichnen. In diesem Fall stellt das Massaging-System sicher, dass die Nachrichten auch beim Ausfall eines Providers nicht verloren gehen.

Den Mitteilungen lassen sich auch Prioritäten zuordnen. Je höher die Prioritätsstufe ist, desto früher wird eine Nachricht ausgeliefert. Wird einer Nachricht ein Verfallsdatum zugewiesen, wird diese nach Ablauf der festgesetzten Zeit aus der Queue entfernt und geht verloren.

Eine Nachricht besteht in JMS aus drei Teilen, dem *Header*, *Properties* und dem *Body*. Jede Nachricht verwendet dabei dieselben Headerfelder, die im *Message*-Interface definiert werden. Sie enthalten Informationen, die für Clients und Provider notwendig sind, um eine Nachricht zu identifizieren und weiterzuleiten. Im Wesentlichen entsprechen die Headerfelder denen von Websphere MQ-Nachrichten. Die *Properties* bieten die Möglichkeit, den *Header* um weitere Felder zu erweitern. Sie sind ein Mittel, mit dem Nachrichten an die Bedürfnisse von Providern und Anwendungen angepasst werden können. Für den Body stehen fünf Formen zur Verfügung, die unterschiedliche Daten aufnehmen können und vom *Message*-Interface abgeleitet werden: *ByteMessage*, *MapMessage*, *ObjectMessage*, *StreamMessage* und *TextMessage*.

3.3 TSpaces

TSpaces[16,17,18] ist ein Projekt des IBM Almaden Research Center. Es implementiert ein erweitertes TupleSpace-Konzept in Java mit Datenbankfunktionalität. Durch die Verwendung von Java ist das System nahezu plattformunabhängig.

TSpaces unterscheidet zwischen Clients und Servern. Letztere halten einen oder mehrere Tuple Spaces, über die die Clients miteinander kommunizieren können. Zu jedem Tupelraum, den ein Client nutzen möchte, muss er eine separate Verbindung aufbauen. Da den einzelnen Aufrufen eines Clients eindeutige IDs zugeordnet werden ist es möglich, dass die Clients auf derselben Maschine die gleiche Verbindung zu einem TSpace-Server nutzen.

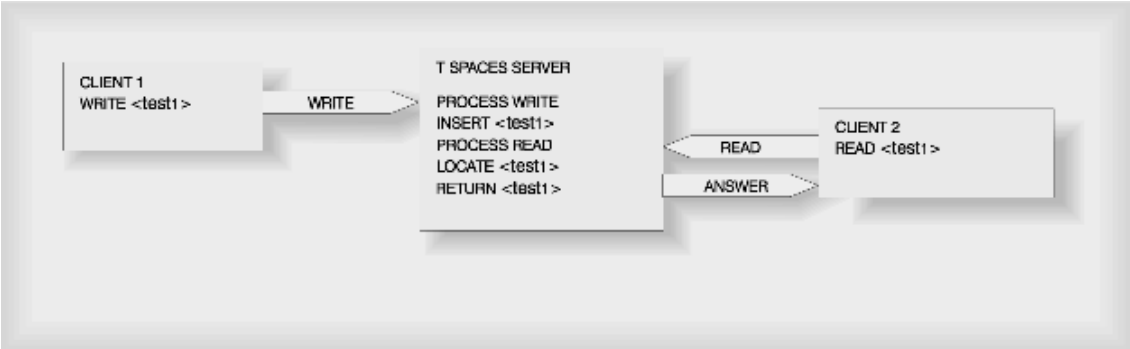

Abbildung 11: TSapces Überblick (Quelle: [17])

Daten werden in TSpaces in Form von Tupeln ausgetauscht. Diese bestehen aus einem oder mehreren Feldern (*Fields*). Ein Feld besteht aus einem Typ (z.B. *java.lang.String*) und einem Wert. Optional können auch Feldnamen vergeben werden. Außer den Tupeln gibt es noch Templates, die für Vergleiche bei Leseoperationen benutzt werden. Ein oder mehrere Felder in ihnen können aus Platzhaltern (Formale Felder) bestehen, die keinen Wert besitzen, sondern nur einen Typ. Tupel bzw. Templates können auch noch über die Klasse *SubclassableTuple* erweitert werden.

Dem Client steht eine Schnittstelle zur Verfügung, die Operationen anbietet, mit denen er auf einen Tuple Space zugreifen kann. Die grundlegenden Operationen sind *write*, *take* und *read*. *Write* schreibt ein Tupel in den Tupelraum. *Read* und *take* lesen ein Tupel aus dem Tupelraum. Ihnen wird als Argument ein Template übergeben, das mit den Tupeln verglichen wird. *Take* entfernt das erste passende Tupel, wogegen *read* den Tupelraum unverändert lässt und nur eine Kopie liefert. Es existieren auch blockierende Varianten von *read* und *take*, d.h. der Client wird solange gestoppt bis ein passendes Tupel in den Tupelraum geschrieben wird. Es gibt auch noch Operationen mit denen alle passenden Tupel aus dem Tupelraum gelesen werden können.

Für das Vergleichen von Tupel und Template nutzt TSpaces einen erweiterten Linda-Algorithmus (siehe Kapitel 2.3.1.1). Die Erweiterungen sind: object compatibility, named fields und query semantics. Bei einem Vergleich mit object compatibility muss zusätzlich zu den normalen Vergleichen der Typ des Tupels eine Unterklasse des Typs des Templates sein. Die zweite Erweiterung sind die named fields. Bei Tupeln

wird ein Index über die Felder erzeugt, denen ein Name zugewiesen wurde. Dies gibt Clients die Möglichkeit, Anfragen nach Tupeln zu starten, indem sie nur einen Namen und Wert angeben, ohne die restlichen Felder des Tupels zu betrachten. Zum Beispiel gibt eine Anfrage nach („foo“, 8) diejenigen Tupel zurück, die ein Feld mit dem Namen „foo“, dessen Wert 8 ist, besitzen. Außerdem gibt es noch Queries. Es werden vier Typen von Queries angeboten: *Match*, *Index*, *And* und *Or*.

Match Queries werden für object compatibility oder strukturelle Anfragen genutzt. Bei *Index* Queries werden Anfragen über die Indizes der Tupel durchgeführt. Mit And und Or können die beiden ersten Anfragetypen kombiniert werden.

Queries	

Match	template = new Tuple("A", 10);
	ErgebnisMenge = tupelraum.scan(new MatchQuery(template));
	template = new Tuple(new IndexQuery("Wert", "index"));
	ErgebnisMenge = ts.scan(queryTuple);

Index	template = new Tuple(new AndQuery(new IndexQuery("Wert", “index”), new IndexQuery("Wert2",”index2”))); ErgebnisMenge = ts.scan(queryTuple);
And	template = new Tuple(new OrQuery(new IndexQuery("Wert", “index”), new IndexQuery("Wert2",”index2”))); ErgebnisMenge = ts.scan(queryTuple);
Or	

Tabelle 1: TSpaces Query-Beispiele

Ruft ein Client eine Operation auf einem Tupelraum auf, so werden die Daten, die zur Ausführung der Operation nötig sind, via RPCs an den Server übertragen. Auf diesem wird mittels der *Tuplespace-Lookup-Machinery* der angesprochene Tupelraum lokalisiert. Ein Tupelraum besteht bei TSpaces im Wesentlichen aus zwei Komponenten: Factories und einer Datenbank. Die Factories, die in Schichten aufeinander aufbauen, sind für die Verwaltung der Operationen zuständig, die auf den Tupelräumen ausgeführt werden können. Jede Factory enthält eine Reihe von Handlern, die die jeweiligen Operationen implementieren. Wird einer Factory also ein Funktionsname, ein Tupel und eine ClientID übergeben so liefert sie eine konkrete Implementierung der Funktion. Bietet eine Schicht der Factories eine Funktion nicht

an, so kann sie auf die darunter liegende zugreifen. Die Handler können mit den Tupelräumen über das Datenbank API von TSpaces kommunizieren, das als TSDB (Tublespace Database) Interface bezeichnet wird.

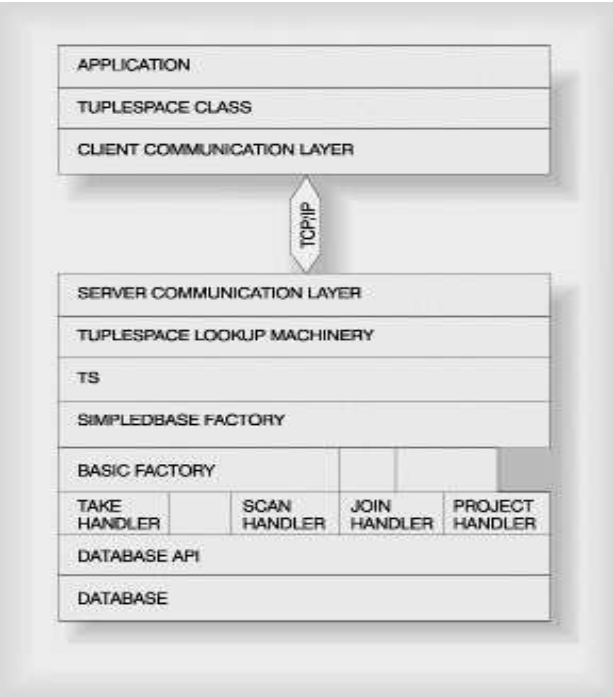

Abbildung 12: TSpaces Design (Quelle: [17])

Der Server gestattet es, verschiedene Datenbankimplementierungen für die einzelnen Tupelräume zu verwenden. Er sorgt auch dafür, dass durchgeführte Transaktionen dauerhaft bleiben. Um dies zu gewährleisten wird ein Redo-Log verwendet, in dem alle Operationen protokolliert werden. Zusätzlich wird in periodischen Abständen eine Sicherung der Tupelräume durchgeführt.

3.4 JavaSpaces

Sun hat mit JavaSpaces[26,40], die Linda-Idee aufgegriffen und um objektorientierte Konzepte erweitert. JavaSpaces sind ein Bestandteil von Jini und in Java implementiert.

Die Tupel werden in JavaSpaces Entry genannt. Ein Objekt muss das Interface *net.jini.core.entry.Entry* implementieren, um in den Tupelraum geschrieben werden

zu können. Die Attribute des Objekts sind die Felder, auf denen Vergleiche durchgeführt werden können, um ein Objekt aus dem Tupelraum zu lesen. Der Tupelraum wird in JavaSpaces schlicht *Space* genannt und von einem zentralen Server verwaltet. Der Client verwendet, um auf einen *Space* zugreifen zu können, ein Proxy-Objekt. Die wesentlichen Operationen auf einem Tupelraum sind *write*, *read* und *take*. Zusätzlich zu diesen Grundfunktionen gibt es noch Transaktionen und einen Event-Mechanismus.

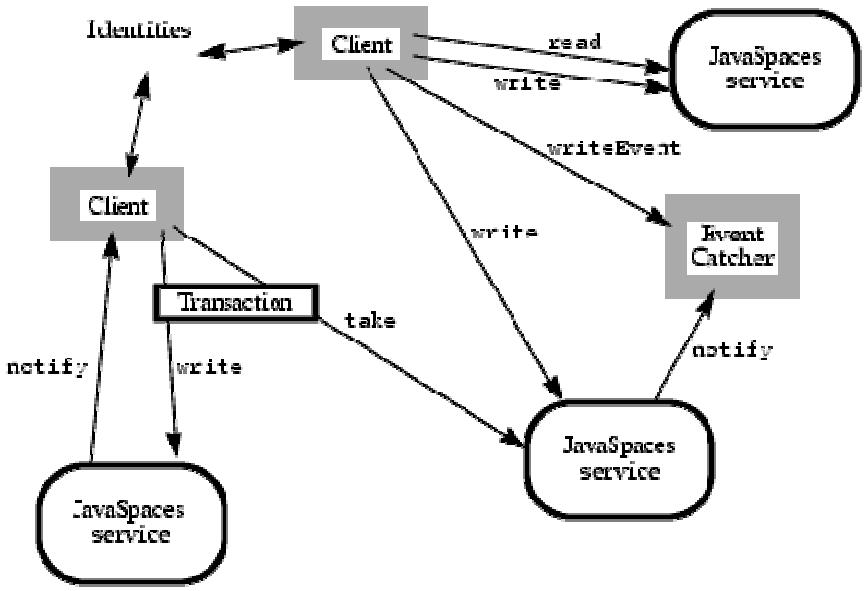

Abbildung 13: JavaSpaces Kommunikation (Quelle: [26])

Beim Schreiben eines Tupels, mit *write* wird ein Lease definiert, das die Gültigkeit des Eintrags angibt. Nach Ablauf, des in dem Lease angegebenen Zeitraumes, wird der Tupel aus dem Space gelöscht. Sollte ein Client ein Lease für ein Tupel anfordern, das zu lange für den Server ist, wird es reduziert und der verkürzte Wert an den Client zurückgegeben. *Read* und *take* liefern einen Eintrag aus dem Space zurück. Ist kein passendes Tupel vorhanden, so blockieren diese beiden Methoden den Client für eine Zeitspanne, die als Parameter übergeben werden kann. Alternativ zu diesen blockierenden Methoden existieren auch nicht blockierende *readIfExists* und *takeIfExists*. Um nach einem Tupel zu suchen, nutzt JavaSpaces ebenfalls Templates. Es existiert allerdings keine spezielle Templateklasse, sondern es werden *Entry*-Objekte genutzt. Auch hier können wie bei TSpaces Platzhalter für die einzelnen Felder vergeben werden. Der Space-Server durchsucht den Tupelraum nach dem übergebenen Template und gibt das erste passende Objekt zurück (siehe

Kapitel 2.3.1.1). Das Tupel kann allerdings auch eine Unterklasse des Templateobjekts sein. Der Unterschied zwischen *read* und *take* ist, wie in Tuple-Space-Systemen üblich, dass letzteres den Eintrag aus dem Tupelraum entfernt.

Als ein weiteres Argument kann den Grundfunktionen eine Transaktion übergeben werden. Schreiboperationen im Tupelraum werden dann erst nach Abschluss dieser sichtbar. Wird ein Tupel während einer Transaktion gelesen, so können keine anderen Transaktionen darauf zugreifen bis diese beendet ist. Wird den Methoden anstatt einer Transaktion ein Nullwert übergeben, so verhalten sie sich so, als ob jede Operation in einer einzelnen Transaktion ausgeführt worden wäre.

Eine weitere Ergänzung zum Linda-Konzept ist ein Event-Mechanismus. Mit diesem können Clients ihr Interesse an neu abgelegten Tupeln, die einem Template entsprechen, dem Server bekannt geben. Trifft ein passender Eintrag zu dem Template ein, so wird der vom Client mitgegebene EventHandler aufgerufen und der Client kann auf das Ereignis reagieren.

Da ein Client öfters dasselbe Template oder Tupel verwendet, um auf einen Tupelraum zuzugreifen, bietet JavaSpaces die Funktion *snapshot* an. Diese soll den Overhead reduzieren der entsteht, wenn ein Objekt an den Space gesendet wird. Wird sie von einem Client aufgerufen, wird ihm ein Stellvertreterobjekt übergeben, welches das übergebene Tupel im Tupelraum repräsentiert. Ein Objekt, das mittels snapshot übertragen wurde, kann nur auf dem Space verwendet werden, auf dem es erzeugt wurde. Wird es in einem anderen Tupelraum verwendet, so tritt ein Fehler auf.

3.5 Lime – Linda in a Mobile Environment

Lime[19,20,21,22] ist eine in Java realisierte Umsetzung des in Linda vorgestellten Tuple Space-Konzepts für mobile Umgebungen. Im Gegensatz zu Linda geht Lime nicht von einem statisch angelegten und global zugreifbaren Tupelraum aus, sondern es werden mehrere kleine Tupelräume dynamisch vereinigt. Lime führt deshalb den Begriff der „transiently shared tuple spaces“ ein.

Die einzelnen Anwendungen werden in Lime „Mobile Agents“ genannt. Diese liegen dabei auf „Mobile Hosts“, die als Container dienen. Die einzelnen Agenten verfügen in Lime über mindestens einen Tupelraum (Interface Tuple Space/ITS). Sie können allerdings auch mehrere zu verschiedenen Zwecken verwalten. Migriert ein Agent von einem Gerät zu einem anderen, so werden auch seine ITS transferiert und alle Tupel die darin enthalten sind. Die einzelnen Tupelräume der Agenten auf einem Host werden zu einem Host-Level Tuple Space zusammengefasst. Falls eine Netzwerkverbindung zwischen den verschiedenen mobilen Geräten besteht, sind die Agenten über einen Federated Tuple Space verbunden. Dieser setzt sich aus den einzelnen Host-Level Tuple Spaces zusammen, also aus den ITS der Agenten. Diese Tupelräume werden immer rekonfiguriert, sobald ein Agent hinzukommt (Engagement) oder nicht mehr erreichbar ist (Disengagement). Jeder Host verfügt zusätzlich noch über einen so genannten Lime System TupleSpace. Dieser enthält Informationen über die zurzeit erreichbaren Rechner. Tritt ein Host einer Gruppe von Geräten bzw. einem Federated Tuple Space bei, so sendet er während des Engagement-Prozesses Tupel mit Informationen über sich an die anderen Mitglieder bzw. deren LimeSystem TupleSpace. Verlässt ein Gerät die Gruppe, so werden diese wieder entfernt.

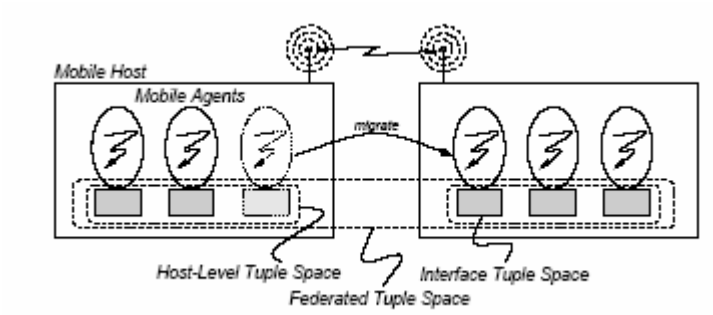

Abbildung 14: Lime Tuple-Spaces (Quelle: [22])

Lime bietet für den Zugriff auf den Tupelraum die in Linda üblichen Funktionen an: *out*, *rd*, und *in*. *Out* schreibt einen Tupel in den lokalen ITS des Agenten, der die Operation ausführt. *Rd* und *in* lesen ein Tupel aus dem Tupelraum, das einem bestimmten Template entspricht (siehe Kapitel 2.3.1.1), wobei letzteres es auch noch aus dem Tupelraum entfernt. Diese beiden Funktionen greifen allerdings nicht nur auf den Tupelraum des Agenten zu, sondern durchsuchen den gesamten Federated

Tuple Space, indem sie die Angaben im LimeSystem Tupelraum nutzen. Da Agenten auch mehrere ITS besitzen können, muss bei den Operationen immer der Name angegeben werden, auf welchen zugegriffen werden soll. Zwei Agenten haben dabei Zugriff auf denselben Tupelraum, wenn die Namen übereinstimmen. Angenommen A besitzt die ITSs X,Y und B X,Z, so teilen sich A und B den Tupelraum X.

Da ein Agent, der den Federated Tuple Space verlässt, auch immer die Tupel in seinen ITS mit sich nimmt, auf die dann nicht mehr zugegriffen werden kann, ist in Lime die Möglichkeit implementiert worden, auf den Ort eines Tupels Bezug zu nehmen. Der Ort eines Tupels ist dabei immer sein Tupelraum, welcher an einen Agenten gebunden ist. Der jeweilige ITS kann hier durch seinen Namen und die Identifikation des Agenten eindeutig bestimmt werden. Die einzelnen Grundoperationen wurden dabei um Lokationsparameter erweitert. Wird ein Tupel mit Angabe eines Ortes geschrieben, so wird das Tupel zuerst in den lokalen ITS des Agenten gelegt. Besteht eine Verbindung zum Ziel, so wird es übertragen. Sollte der angegebene Tupelraum nicht erreichbar sein, so wird das Tupel durch Lime als deplaziert (misplaced) markiert und verweilt im lokalen ITS bis es entfernt wird oder gesendet werden kann. Kommt ein Agent zu einem Federated Tupel Space hinzu, werden während des Engagement-Prozesses die markierten Tupel, für die der Zielort gefunden werden konnte, ausgetauscht. Bei *rd* bzw. *in* können jeweils zwei Lokationsparameter angegeben werden. Der erste gibt den momentanen Aufenthalt eines Tupels an, der zweite den Zielort. Einer der beiden Orte kann dabei unbenannt bleiben. Zum Beispiel greift die Funktion *in(A,-)(p)* auf alle Tupel zu, deren momentaner Ort A ist und die dem Template p entsprechen.

Da Lime für mobile Systeme entwickelt wurde, in denen schnell, aufgrund der hohen Dynamik, auf Vorkommnisse wie die Ankunft eines Agenten reagiert werden muss, wird ein Mechanismus zur Ereignisbehandlung angeboten. Ereignisse werden in Lime in der Form von Tupeln repräsentiert. Einem Agenten steht nun die Möglichkeit zur Verfügung, eine so genannte *reaction* bei einem Tupelraum zu registrieren. *Reactions* bestehen im Wesentlichen aus zwei Komponenten: einem Template und einem Stück Programmcode. Wird zu dem Template ein passendes Tupel gefunden, so wird der Code ausgeführt. Sind mehrere reactions zu einem Ereignis eingetragen, werden sie nacheinander ausgeführt. Eine Ereignisbehandlung findet nach jeder

Operation in dem Tupelraum statt. Erst nach ihrem Ablauf werden die normalen Operationen fortgesetzt. Eine *reaction* kann auch mit Ortskontext aufgestellt werden. Dabei wird zwischen starken und schwachen *reactions* unterschieden. Bei einer starken wird eine Ausführung des Programmcodes sofort nach Entdecken des Tupels garantiert. Sie darf allerdings nicht über den Ortsbezug eines Hosts hinaus angewendet werden, da Lime über kein Transaktionssystem verfügt. Bei einer schwachen *reaction* wird auf jedem derzeit erreichbaren ITS eine starke *reaction* registriert, die bei Zutreffen des übergebenen Templates eine Kopie des Tupels an den Tupelraum des Initiators sendet. Dort wird dann der Programmcode asynchron ausgeführt.

3.7 Diskussion

In diesem Abschnitt sollen die in diesem Kapitel vorgestellten Systeme im Hinblick auf die Anforderungen, die ubiquitäre Systeme an einen indirekten und asynchronen Kommunikationsmechanismus stellen, betrachtet werden.

Websphere MQ bietet mit der Umsetzung des Message-Queuing-Konzepts eine gute Möglichkeit, Daten asynchron und indirekt auszutauschen. Allerdings sind die Anforderungen an die beteiligten Systeme, die die QueueManager und damit Queues beherbergen, zu groß für den Einsatz im ubiquitous Computing. Es wird zum Beispiel davon ausgegangen, dass die Geräte Nachrichten auf einen stabilen Speicher schreiben können und die Server relativ statisch sind. Auch kann es nicht an die Erfordernisse der einzelnen Geräte angepasst werden. Mit Websphere MQ everyplace wurde von IBM deswegen ein Messaging-System eingeführt, das auch auf kleineren Geräten zum Einsatz kommen kann. Es stellt nur minimale Anforderungen an die Ressourcen, indem z.B. nur die notwendigsten Bibliotheken auf die Geräte transferiert werden. Auch ist Websphere MQ everyplace durch die Möglichkeit, die Queues an ein System anzupassen, gut für den Einsatz in einem ubiquitären Umfeld geeignet. Allerdings bietet es, genau wie Websphere MQ, Anwendungen keine Möglichkeit Queues zu suchen. Es geht davon aus, dass die verfügbaren Queues bei der Programmierung bekannt sind und irgendwann eine Verbindung zu ihnen aufgebaut werden kann. In einem ubiquitären System möchten

Anwendungen allerdings häufig die im Moment vorhandenen Queues nutzen und über sie kommunizieren.

Der JMS liefert mit seinen Schnittstellen einen guten Ausgangspunkt für die Implementierung eines Messaging-Systems für auf Java basierende Anwendungen. Die meisten Systeme, die auf dem JMS beruhen bzw. es implementieren, gehen allerdings von einem statischen Server aus, auf dem die Queues bzw. Topics residieren. JMS sieht außerdem auch keinen Mechanismus vor, der die Erkennung bzw. das Suchen von Queues erlauben würde.

Das System von TSpaces kann durch die Trennung von Zugriffsoperationen und Tupelräumen gut an die Bedürfnisse der einzelnen Geräte angepasst werden. Außerdem können verschiedene Implementierungen für die Datenhaltung in einem Tupelraum verwendet werden, die nicht unbedingt einen stabilen Speicher voraussetzen. Allerdings beruht TSpaces auf dem Konzept eines global zugreifbaren Tupelraums, der in der Umgebung verankert ist. Ebenfalls geht es von einer ständigen Verbindung zu diesem aus. Dadurch eignet es sich weniger für ubiquitäre Systeme, in denen sich aufgrund der Dynamik, die Verbindungen häufig ändern und bestimmte Server nicht immer zu erreichen sind.

Bei JavaSpaces wird auch von statischen Tupelräumen ausgegangen, bei denen zudem verlangt wird, dass sie die Tupel persistent speichern können. Durch die Einbettung in Jini verfügt das System allerdings über einen Mechanismus, mit dem die Spaces aufgefunden werden können. Anwendungen wird hiermit die Möglichkeit gegeben, sich an die momentane Umgebung anzupassen.

Lime bietet mit der Implementierung von „transiently shared tuple spaces“ einen guten Ansatz zur indirekten und asynchronen Kommunikation in mobilen Systemen. Problematisch ist dabei, dass bei jeder Änderung des Netzwerkes der Tupelraum neu berechnet werden muss. Je dynamischer die Umgebung ist und je mehr Komponenten beteiligt sind, desto öfter muss eine Reorganisation stattfinden. Als weiteres setzt es bei den Geräten genügend Ressourcen voraus, damit die einzelnen Agenten ihren eigenen ITS unterhalten können.

4 Anforderungen

Im Folgenden sollen die Anforderungen, die an den asynchronen und indirekten Kommunikationsmechanismus für ubiquitäre Systeme gestellt werden, näher betrachtet werden.

Systeme, die für ein ubiquitäres Umfeld entwickelt werden, müssen in der Lage sein, mobile, d.h. spontan vernetzte und heterogene Geräte zu unterstützen. Als Grundlage für den Kommunikationsmechanismus soll die in Kapitel 2.2 vorgestellte Middleware BASE dienen, die speziell für ubiquitäre Systeme entwickelt wurde. Aus diesen Rahmenbedingungen lassen sich verschiedene Anforderungen an den Kommunikationsmechanismus ableiten.

Asynchrone und Indirekte Kommunikation

Die Anforderung die sich schon aus der Aufgabenstellung ergibt ist, dass das zu entwickelnde System Anwendungen erlaubt asynchron und indirekt zu kommunizieren. Es muss Anwendungen also möglich sein, Nachrichten an einen Dienst zu senden, ohne dass sie auf deren Verarbeitung warten müssen und so mit ihrer eigenen Arbeit fortfahren können. Außerdem sollen ein oder mehrere Empfänger der Daten angesprochen werden können, ohne diese direkt zu benennen.

Minimalität

In ubiquitären Systemen kommen gerade auch Geräte zum Einsatz, die nur über geringe Ressourcen verfügen. Es ist deshalb erforderlich, dass der zu entwickelnde Kommunikationsmechanismus im Kern nur minimale Anforderungen an die Hard- bzw. Software stellt. Beispielsweise würde ein zu hoher Bedarf an Rechenleistung oder Speicher ganze Geräteklassen, wie Sensoren, von der Nutzung des Systems ausschließen.

Anpassbarkeit

Ubiquitäre Systeme zeichnen sich durch eine Vielfalt unterschiedlichster Geräte aus, die über verschiedenste Leistungsklassen verfügen. Es kommen sowohl

leistungsstarke als auch ressourcenarme Rechner zum Einsatz. Den Anwendungen muss daher die Möglichkeit gegeben werden, den Kommunikationsmechanismus an die zur Verfügung stehenden Ressourcen, bzw. die jeweiligen Bedürfnisse anzupassen.

Einfache Schnittstellen

Um auch die Entwicklung von einfachen Anwendungen für ressourcenarme Geräte zu unterstützen, sollte der Kommunikationsmechanismus möglichst wenige und einfache Schnittstellen bereitstellen, die von Entwicklern ohne größeren Aufwand genutzt werden können, da breitere Schnittstellen, gerade bei kleinen Anwendungen, oft zu einer geringeren Effizienz führen können.

Unabhängigkeit von Zentralen Systemen

In einem ubiquitären System kann nicht von einer Netzwerkinfrastruktur ausgegangen werden, die einen Zugriff auf zentral verfügbare Dienste gewährleistet, da sich hier auch mehrere Geräte spontan zu Gruppen zusammenfinden können, die nur eine Verbindung untereinander besitzen. Das System muss daher unabhängig von zentralen Diensten sein. Aus diesem Grund muss ein Discovery-Mechanismus bereitgestellt werden, der das Auffinden von Kommunikationsmöglichkeiten ermöglicht.

5 Konzeption

Das Ziel dieser Arbeit besteht darin, einen asynchronen und indirekten Kommunikationsmechanismus für ubiquitäre Systeme zu entwickeln, der den in Kapitel 4 gestellten Anforderungen genügt. In diesem Kapitel sollen nun die für die Lösung eingesetzten Konzepte veranschaulicht werden.

Für die indirekte und asynchrone Kommunikation ist es nötig, die zu sendenden Daten zwischen Sender und Empfänger zwischenzuspeichern. Außerdem muss ein Sender einen Dienst ansprechen können, ohne dass er diesen speziell adressiert. In verteilten Systemen stehen im Wesentlichen zwei Methoden zur Verfügung. Zum einen ein Shared-Memory Ansatz, zum Beispiel in der Form von Tuple Spaces oder das Message Queuing Prinzip (siehe Kapitel 2.3).

Bei der Umsetzung des Mechanismus wurde das Konzept des Message Queuing verwendet. Das Prinzip des Shared Memory wurde nicht gewählt, weil hier entweder von zentral zugänglichen und ständig verfügbaren Servern ausgegangen wird oder bei einer Verteilung der Speicherbereiche, wie zum Beispiel in Lime, bei jeder Änderung der Netzwerkkonstellation eine Rekonfigurierung des zusammengesetzten Speichers stattfinden muss. Da ubiquitäre Systeme im Allgemeinen eine hohe Dynamik aufweisen und viele Systeme hierbei beteiligt sind, kann es sein, dass durch das Erfordernis des Rekonfigurierens die gesamte Kommunikation blockiert wird.

Das entwickelte Queuing-System besteht hauptsächlich aus fünf Komponenten: **Nachrichten**, **Queuemanager**, **Queues**, **Queue-Connection** und **Queue-Lookupmanager**. Anwendungen nutzen dabei den **Queuemanager**, um das System zu konfigurieren. Um auf eine **Queue** zu zugreifen, werden **Queue-Connections** verwendet, über die **Nachrichten** geschrieben bzw. gelesen werden können. Der **QueueLookupmanager** bietet die Möglichkeit, **Queues** im Netzwerk zu suchen. Die **Queues** dienen als Datenpuffer zwischen verschiedenen Anwendungen. In diesen werden Nachrichten gespeichert, bis sie von einem Dienst verarbeitet werden können.

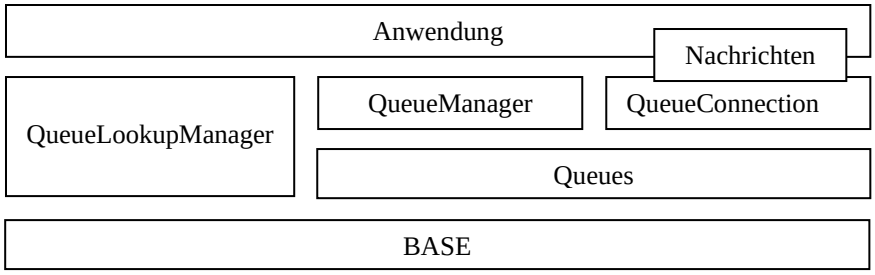

Abbildung 15: Queuing-System Architektur

Nachrichten

Die Nachrichten werden von Anwendungen zum Datenaustausch genutzt. Sie bestehen, wie bei Websphere MQ oder JMS, aus zwei Teilen: einem Nachrichtenkopf und den Anwendungsdaten. Der Nachrichtenkopf enthält Felder, die das Queuing-System oder Anwendungen benötigen, um die Nachrichten zu verarbeiten. Er besteht aus:

- Nachrichten-ID: Sie kennzeichnet eine Nachricht global eindeutig und wird vom System vergeben.
- Reply-ID: Mit ihr können Anwendungen auf eine frühere Nachricht Bezug nehmen, zum Beispiel, wenn wie bei Request-Nachrichten in Websphere MQ eine Antwort verlangt wurde.
- Reply-Queue: In diesem Feld kann definiert werden, an welche Queue eine Antwort gesendet werden soll.
- Objektklasse: Die Objektklasse gibt an, von welchem Typ die enthaltenen Anwendungsdaten sind.
- Ablaufzeit: Hiermit kann eine Zeitspanne für die Nachricht vergeben werden, für die sie Gültigkeit hat. Nach ihrem Ablauf wird die Nachricht vom System nicht mehr ausgeliefert.
- Zeitstempel: Dieser wird vom System gesetzt, wenn die Nachricht in eine Queue gestellt wird.
- Priorität: Durch das Setzen der Priorität kann eine Anwendung Einfluss auf die Reihenfolge nehmen, in der die Nachricht weitergegeben werden soll.

- Eigenschaften: Das Feld Eigenschaften ist eine Liste aus Schlüssel-Wert-Paaren, die von Anwendungen frei vergeben werden können, um die Nachrichten an ihre Bedürfnisse anzupassen. Beispielsweise können hier Angaben über den Verwendungszweck der Daten gemacht werden.

Queuemanager

Der Queuemanager ist die zentrale Komponente des Kommunikationsmechanismus. Er ist auf jedem System genau einmal vorhanden und ist für die Verwaltung der Queues zuständig. Dafür stellt der Manager Funktionen zum Erzeugen und Löschen von diesen bereit. Anwendungen können mit ihm allerdings keine Queues auf entfernten Geräten verwalten. Um eine Queue zu erstellen, muss dem Queuemanager eine Beschreibung dieser übergeben werden. Der Queuemanager sucht anhand der Beschreibung den passenden Queue-Typ aus und erstellt die Queue. Um die Ansprüche an die Ressourcen von Geräten möglichst gering zu halten und um möglichst flexibel und anpassbar zu bleiben, sind die Queue-Typen nicht fest in den Queuemanager integriert, sondern können je nach Bedarf eingebunden werden. Auf diese Weise kann der Kommunikationsmechanismus auch problemlos erweitert werden. Außerdem stellt der QueueManager mit den Queue-Connections eine Schnittstelle bereit, mit der die Anwendungen sowohl auf lokale als auch auf Queues auf entfernten Geräten einfach zugreifen können.

Queues

Die Queues werden dazu verwendet, um von Programmen gesendete Nachrichten zu speichern, bis die Daten von den Zielanwendungen abgeholt werden können.

Jede Queue verfügt über eine Beschreibung, die Informationen über sie enthält und Anwendungen Auskunft über ihre Eigenschaften geben kann. Zusätzlich zum Queue-Typ wird sie dem Queuemanager beim Erzeugen der Queue übergeben, die dann mit den Angaben in der Beschreibung konfiguriert wird. Die Beschreibung enthält folgende Felder:

- Queue-ID: Diese wird beim Erzeugen der Queue vom Queuemanager erstellt und identifiziert sie global eindeutig.

- Queuename: Der Name einer Queue entspricht normalerweise der Queue-ID. Er kann allerdings auch frei vergeben werden.
- Beschreibung: Die Beschreibung ist ein Textfeld, das von der erstellenden Anwendung dazu genutzt werden kann, um den Zweck dieser Queue zu definieren. Zum Beispiel kann hier angegeben werden, welche Dienste lesend auf die Queue zugreifen.
- Ablaufzeit: In diesem Feld kann festgelegt werden, ob die Queue die Ablaufzeiten bei Nachrichten beachtet. Ist es nicht gesetzt, werden die Nachrichten, trotz Angabe einer Lebensspanne, auch nach Ablauf dieser an die Anwendungen weitergegeben.
- Prioritäten: Diese Eigenschaft gibt an, ob die Queue bei Nachrichten die Priorität beachten soll. Wenn die Prioritäten nicht benutzt werden sollen, erfolgt eine Auslieferung nach der Reihenfolge, in der die Nachrichten eintreffen.
- Persistenz: Sie gibt an, ob die Daten in der Queue auf stabilen Speicher geschrieben werden. Unterstützt die Queue dies, werden die Nachrichten in ihr nach einem Absturz wieder hergestellt ansonsten gehen sie verloren.
- Persistente-Beschreibung: Queues, die über eine persistente Beschreibung verfügen, werden nach dem Ausfall eines Gerätes beim Neustart wieder mit dieser hergestellt. Dies gewährleistet, dass Queues auch nach einem Ausfall des Queuing-Systems, wieder unter gleicher Beschreibung bzw. ID verfügbar sind.
- Eigenschaften: Dieses Feld ist wie bei den Nachrichten eine Liste aus Schlüssel-Wert-Paaren, die von Anwendungen frei vergeben werden können, um die Eigenschaften der Queue näher zu spezifizieren. Wenn die Queue beispielsweise für einen Druckdienst zuständig ist, kann hier festgelegt werden welche Papierformate zur Verfügung stehen. Die Eigenschaften dienen auch dazu spätere Erweiterungen des Queuing-Systems zu ermöglichen.
- Maximale-Nachrichtenanzahl: Hier wird die Anzahl der Nachrichten festgelegt, die maximal in eine Queue gestellt werden dürfen. Wird diese erreicht, können keine weiteren Nachrichten in die Queue geschrieben werden bis wieder einige entfernt wurden. Damit lassen sich die Queues auch an die zur Verfügung stehenden Ressourcen der Geräte anpassen.

- Nachrichtenanzahl: Sie gibt die Anzahl der derzeit in der Queue liegenden Nachrichten an.

Die Nachrichten in einer Queue werden normalerweise in der Reihenfolge wieder ausgeliefert, in der sie in der Queue abgelegt wurden (FIFO), außer ihnen wurde eine Priorität zugeordnet und die Queue wurde so konfiguriert, dass sie dies unterstützt. In diesem Fall werden immer Nachrichten mit der höchsten Priorität zuerst weitergegeben. Innerhalb der Prioritätsstufen wird das FIFO-Prinzip angewendet. Außerdem wird aber auch die Möglichkeit angeboten, die jeweilige Queue nach einer bestimmten Nachricht zu durchsuchen, indem ihr ein Nachrichtenkopf übergeben wird. Darin müssen nicht alle Felder mit Werten besetzt sein, sondern können durch Platzhalter ersetzt werden. Der Suchkopf wird mit den Nachrichtenköpfen der in der Queue enthaltenen Nachrichten verglichen. Es wird immer die erste Nachricht in der Queue zurückgegeben, die dem Suchkopf entspricht. Sind Felder mit einem Platzhalter versehen, werden diese beim Vergleich nicht berücksichtigt. Die Anderen müssen sich in den Werten entsprechen. Beim Vergleich der Liste der Eigenschaften im Nachrichtenkopf wird eine Übereinstimmung erzielt, wenn alle Schlüssel-Wert-Paare aus dem Suchkopf in den Eigenschaften der Nachricht enthalten sind.

Außer den Funktionen für das Einreihen einer Nachricht oder das Abholen der Ersten stehen noch Operationen bereit, mit denen mehrere Nachrichten gleichzeitig in die Queue gelegt bzw. ausgelesen werden können. Beim Auslesen können hier entweder alle Nachrichten aus der Queue entfernt werden oder alle, die einem bestimmten Nachrichtenkopf entsprechen. Programme können auch eine Liste der Nachrichtenköpfe von Nachrichten anfordern, die sich derzeit in der Queue befinden.

Wegen der Heterogenität der Geräte, in ubiquitären Systemen und der Anforderung der Anpassbarkeit, wurden im Queuing-System zwei Queue-Typen bereitgestellt. Der erste Queue-Typ speichert die Nachrichten nur im flüchtigen Speicher, während der zweite die Möglichkeit bietet, sie auf stabilen Speicher zu schreiben. Unabhängig von dem Typ können, wie bereits erwähnt, die Beschreibungen der Queues auf stabilen Speicher geschrieben werden. Nur die Queues, die mit einer persistenten Beschreibung erzeugt wurden, werden bei einem Neustart wieder hergestellt. Um

eine persistente Speicherung zu erlauben, muss dem Queuemanager beim Start ein gültiger Pfad zu einem Verzeichnis übergeben werden, in dem die Dateien für die Wiederherstellung, sowohl der Queue-Beschreibungen bzw. der Queues als auch der Nachrichten für den persistenten Queue-Typ angelegt werden. Wird kein gültiger Pfad angegeben, können nur Queues erzeugt werden, die den flüchtigen Speicher nutzen. Um die künftige Flexibilität des Systems zu gewährleisten besteht die Möglichkeit weitere Queue-Typen hinzuzufügen.

Queue-Connection

Die Queue-Connection bietet eine einheitliche Schnittstelle, mit der Anwendungen lesend bzw. schreibend auf Queues zugreifen können. Bei der Kommunikation mit einer lokalen Queue (Abbildung 16 QueueConnection1) werden die Nachrichten direkt an diese weitergegeben. Residiert die Queue auf einem entfernten Gerät werden RPCs von BASE zur Übertragung der Daten genutzt (Abbildung 16 QueueConnection2). Dabei ist es für die Programme transparent, ob die Queue lokal im System oder auf entfernten Geräten liegt, da dieselbe Schnittstelle für beide Varianten verwendet wird. Für jede Queue, auf die eine Anwendung zugreifen möchte, muss jeweils eine neue Queue-Connection verwendet werden. Diese kann mittels des Queuemanagers erzeugt werden. Dafür muss ihm eine Queue-Beschreibung bzw. eine Queue-ID übergeben werden. Anhand dieser kann der Queuemanager feststellen, welches Queue-Connection Objekt an die Anwendung zurückgegeben werden muss. Entweder das für eine lokale oder eine entfernte Queue.

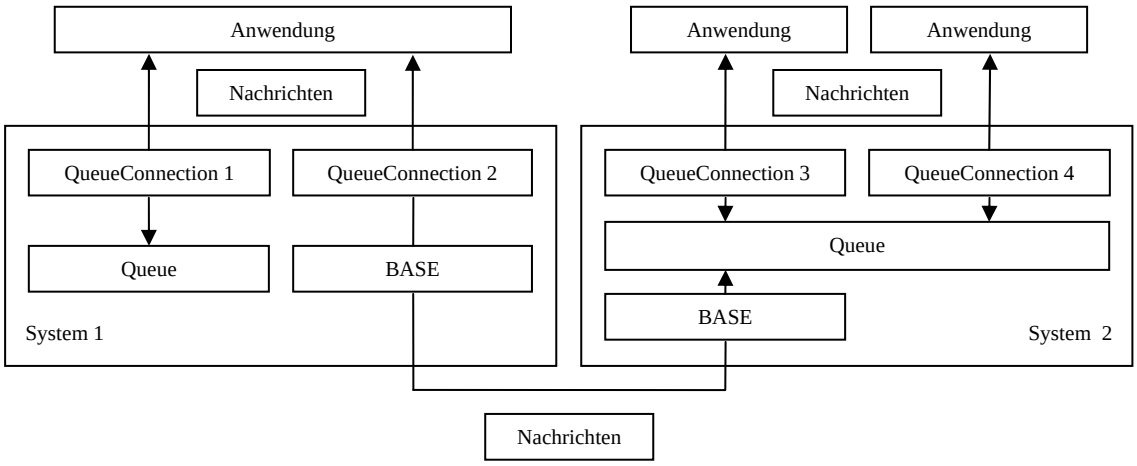

Abbildung 16: QueueConnection Kommunikation

Die Funktionen, die Queue-Connections anbieten, sind im Wesentlichen dieselben wie die der Queues. Die Operationen erlauben, wie schon im Abschnitt Queues erwähnt, das Lesen bzw. Schreiben von oder den Zugriff auf bestimmte Nachrichten.

Queue-Lookupmanager

Der Queue-Lookupmanager bietet einen Dienst an, mit dem Queues im Netzwerk gesucht werden können. Dabei kann festgelegt werden, ob nur lokal, entfernt oder überall gesucht werden soll. Mittels Queue-Beschreibungen können Kriterien festgelegt werden, anhand derer die zurückgegebenen Queues gefiltert werden. Dabei können, wie bei der Suche nach Nachrichten in einer Queue, den einzelnen Feldern in der Beschreibung, Platzhalter zugewiesen werden. Bei der Auswahl müssen sich hier, wenn Werte gesetzt wurden, die Felder in der Queue-Beschreibung mit denen in der Suchbeschreibung decken. Die Felder, die einen Platzhalter aufweisen, werden nicht berücksichtigt. Die Liste der Eigenschaften im Suchkopf(1) entspricht der Eigenschaftenliste in den Queue-Beschreibungen(2) der erreichbaren Queues, wenn mindestens alle Schlüssel-Wert-Paare von (1) in (2) enthalten sind.

6 Implementierung

In diesem Kapitel soll ein Überblick über die Implementierung des in Kapitel 5 vorgestellten Systems gegeben werden. Zuerst soll dabei auf die Integration des Kommunikationsmechanismus in BASE eingegangen werden. Danach sollen die wesentlichen Teile der Implementierung des Queuing-Systems betrachtet werden.

6.1 Integration in BASE

Das in Kapitel 2.2 vorgestellte BASE-System dient als Grundlage für den zu erstellenden Kommunikationsmechanismus. Für die Integration in die vorliegende Infrastruktur gibt es verschiedene Möglichkeiten. Zum einen kann der Mechanismus direkt in das bestehende System eingebettet werden, indem die relevanten Teile identifiziert und erweitert werden. Zum anderen kann die Erweiterung ohne Veränderung von BASE erfolgen.

Die direkte Integration ermöglicht eine bessere Leistung. Sie führt jedoch auch dazu, dass der Umfang des Kernsystems größer wird und sich dadurch die Anforderungen an die Ressourcen der Geräte erhöhen, auch wenn die Anwendungen auf diesen keine indirekte und asynchrone Kommunikation einsetzen möchten. Außerdem leidet die Wartbarkeit der Quellen bei einer solchen Vorgehensweise, da diese durch die Erweiterungen im Normalfall unübersichtlicher werden.

Bei der Implementierung wurde die zweite Vorgehensweise gewählt, bei der das Kernsystem unverändert bleibt. Der Kommunikationsmechanismus setzt also auf einem vollständigen BASE-System auf und nutzt dieses nur für den Zugriff auf entfernte Geräte mittels Stubs bzw. Skeletons. Dadurch ergibt sich der Vorteil, dass zukünftige Änderungen im BASE-Kern keine Auswirkungen auf das Queuing-System haben. Außerdem kann es auf diese Weise leichter erweitert werden und Anwendungen können das Gesamtsystem besser an ihre Bedürfnisse oder die zur Verfügung stehenden Ressourcen anpassen.

Um Plattformunabhängigkeit zu erreichen und um heterogene Geräte zu unterstützen, wurde bei der Implementierung des Queuing-System wie bei BASE die Java 2 Micro Edition verwendet.

6.2 Nachrichtenstruktur

Die Nachrichten(*Message*) bestehen, wie schon erwähnt, aus den Daten die eine Anwendung versenden möchte und einem Nachrichtenkopf(*MessageDescription*). Die Anwendungsdaten können dabei aus einem beliebigen Java-Objekt bestehen. Damit eine Nachricht in entfernte Queues gestellt oder auf stabilen Speicher geschrieben werden kann, muss das Anwendungsdaten-Objekt allerdings von BASE bzw. dem Queuing-System serialisiert werden können. Dies kann beispielsweise dadurch ermöglicht werden, indem seine Klasse die *Serializable*-Schnittstelle implementiert.

Der Nachrichtenkopf dient zur Beschreibung der Nachricht und bietet Informationen für das Queuing-System und Anwendungen, wie diese behandelt werden soll.

Die in ihm enthaltene Nachrichten-ID(*MessageID*) ist ein *String*. Dieser setzt sich aus drei Teilen zusammen: Einem *Long*-Wert, der bei jeder neuen Nachricht erhöht wird, einer Epoche und der *String*-Repräsentation der BASE-System-ID des Gerätes auf dem die Anwendung liegt, die die Nachricht sendet. Die Epoche ist dafür nötig, damit die Nachrichten nach einem Neustart des Gerätes, von älteren, aus demselben System, unterscheidbar bleiben. Durch diesen Aufbau erhält jede Nachricht eine global eindeutige ID.

Für die Priorität(*priority*) einer Nachricht können im Kopf Werte von 0 (niedrigste) bis 9 (höchste) vergeben werden. Wird von einer Anwendung einer Nachricht keine Priorität zugewiesen, so wird ein Default-Wert von 4 genutzt. Je höher die Priorität, desto weiter vorne wird sie in der Queue eingereiht.

Der Zeitstempel(*timestamp*) im Nachrichtenkopf wird von der Queue gesetzt, an die die Nachricht gesendet wurde. Die Ablaufzeit(*ttl*) einer Nachricht, die von der sendenden Anwendung festgelegt wird, gibt die Zeit in Millisekunden an, für die sie in

der Queue gehalten werden soll, bevor sie gelöscht und nicht mehr ausgeliefert wird. Es gibt auch die Möglichkeit das Feld auf einen Wert zu setzen, der eine unendliche *ttl* repräsentiert.

Das Eigenschaftenfeld(*properties*) im Nachrichtenkopf wurde als ein *Hashtable* implementiert, der jeweils Schlüssel-Wert-Paare, die jeweils aus einem String bestehen, aufnimmt. Dieses Feld ist momentan nur für Anwendungen gedacht. Es kann aber auch dazu verwendet werden, bei Erweiterung des Queuing-Systems, weitere Verarbeitungshinweise für die Queues zu liefern.

Die Felder Reply-Queue(*replyQueue*) bzw. Reply-ID(*replyToID*) können von Anwendungen zur Spezifizierung eingesetzt werden, an welche Queue eine Antwortnachricht gesendet werden soll (*replyQueue*) bzw. auf welche Nachrichten geantwortet wird (*replyToID*). Das Reply-Queue-Feld ist dabei ein QueueID-Objekt (siehe Kapitel 6.5), die Reply-ID ein *String*.

Das letzte Feld im Nachrichtenkopf ist die Objektklasse(*objectClass*). Mit diesem *String* kann eine Anwendung angeben, von welcher Klasse das gesendete Objekt ist. Beispielsweise „*java.lang.String*“. Damit können Anwendungen erkennen, ob eine Nachricht Daten enthält, die für sie relevant sind.

Sowohl die *Message*- als auch die *MessageDescription*-Klasse implementieren jeweils die BASE *Serializable*-Schnittstelle, aus den schon bei den Nachrichtendaten erläuterten Gründen.

6.3 QueueManager

Der *QueueManager* wurde als Singleton implementiert, es gibt also pro Gerät nur eine Instanz von ihm. Beim ersten Aufruf muss ihm ein Pfad übergeben werden, unter dem Daten gespeichert werden können. Dieses Verzeichnis wird während der Initialisierung auch nach persistenten Queue-Beschreibungen durchsucht. Findet der *QueueManager* solche, stellt er die Queues mit derselben Beschreibung wieder her. Als Verzeichnis kann auch ein *null*-Wert übergeben werden. In diesem Fall greift das

System nicht auf den stabilen Speicher zu. Ein weiteres Argument, das angegeben werden muss, ist ein *IDGenerator*. Dieser ist für die Erzeugung von *MessageIDs* und *QueueIDs* zuständig. Der implementierte IDGenerator erlaubt die Auswahl, ob bei der Generierung der IDs die Systemzeit eingesetzt wird oder ob diese nicht herangezogen werden soll. Mit der Möglichkeit, verschiedene IDGeneratoren einzusetzen, kann das Queuing-System an die Eigenschaften der Geräte bzw. die Anforderungen der Anwendungen besser angepasst werden.

Da die verschiedenen Queue-Arten nicht fest in den *QueueManager* integriert sind, muss der Typ, mit dem eine Queue angelegt werden soll, zuerst mittels der *registerQueueType* Methode registriert werden. Eine Queue-Klasse muss dabei das *Queue*-Interface implementieren. Ist dies geschehen, kann durch die *createQueue* Methode eine neue Queue erzeugt werden. Für das Erstellen einer Queue muss dem *QueueManager* ein *QueueDescription*-Objekt (Queue-Beschreibung) übergeben werden, das die Eigenschaften der Queue festlegt. In weiteren Argumenten muss der Typ angegeben werden und ob die Queue-Beschreibung persistent gespeichert werden soll. Wird ein Queue-Objekt angelegt, so vergibt der *QueueManager* dafür zuerst eine Queue-ID. Anschließend ruft er die *startQueueMethode* der Queue auf, die für die nötigen Initialisierungen innerhalb dieser sorgt, zum Beispiel das Anlegen der Dateien in denen Nachrichten abgelegt werden können. Danach erzeugt der *QueueManager* ein neues Skeleton(siehe Kapitel 6.5.1) für die Queue, dass er beim BASE-System registriert, um auch entfernte Zugriffe darauf zu ermöglichen. Als letzten Schritt informiert er den *Queue-Lookupmanager* über die neue Queue.

Für das Löschen einer Queue wird die Methode *deleteQueue* bereitgestellt. Diese ruft die Methode *stopQueue* des Queue-Objekts auf. Anschließend entfernt sie das zugehörige Skeleton aus BASE und gibt dem Queue-Lookupmanager bekannt, dass die Queue gelöscht wurde. Bei Queues mit einer persistenten Beschreibung wird diese aus der Datei entfernt, in der die *MessageDescriptions* enthalten sind.

6.4 QueueLookupManager

Genau wie der *QueueManager,* wurde der *QueueLookupManager* als Singleton erstellt. Weiterhin wurde er, um die Suche auch bei entfernten Geräten zu ermöglichen, als ein so genannter „well-known-Service“ in BASE registriert.

Für Anwendungen stellt der *QueueLookupManager* drei Bereiche bereit, in denen nach Queues gesucht werden kann: lokal (*LOOKUP_LOCAL_ONLY*), entfernt (*LOOKUP_REMOTE_ONLY*) und in beiden zusammen (*LOOKUP_BOTH*). Den Programmen ist es dabei möglich, nach allen Queues in einem bestimmten Bereich zu suchen oder zur Filterung nach bestimmten Eigenschaften einer Queue ein *QueueDescription*-Objekt als Parameter mit anzugeben. Wird in einem oder mehreren Feldern von dem *QueueDescription*-Objekt ein *null*-Wert gesetzt, so werden diese beim Vergleich mit den gefundenen Queues nicht betrachtet.

Bei einer lokalen Suche werden nur die Queues, die auf dem lokalen System residieren, durchsucht und die Beschreibungen derjenigen zurückgegeben, die den angegebenen Suchparametern entsprechen.

Um bei entferntem Suchen die Geräte im Netzwerk aufzufinden, nutzt der *QueueLookupManager* die *DeviceRegistry* von BASE. Wird eine entfernte Suche initialisiert, werden an alle darin enthaltenen Geräte die Suchparameter mittels RPC-Aufrufen übermittelt. Die entfernten Geräte führen anschließend eine lokale Suche durch und filtern gegebenenfalls in ihrem System liegenden Queues nach der angegeben *Queue-Description*. Die Ergebnisse werden, nachdem dies abgeschlossen ist, beim aufrufenden *QueueLookupManager* zusammengeführt und an die Anwendung weitergereicht. Bei LOOKUP_BOTH werden nacheinander eine lokale und eine entfernte Suche durchgeführt.

6.5 Queues

Für den Kommunikationsmechanismus wurden, wegen der Heterogenität des Systemumfelds, zwei Arten von Queues implementiert. Ein Typ hält die Nachrichten

nur im flüchtigen Speicher(*SimpleQueue*), der andere schreibt sie zusätzlich noch in den stabilen Speicher(*SimplePersistentQueue*). Beide Arten bieten ansonsten die gleiche Funktionalität.

Die einzelnen Queues können von Anwendungen mittels der Queue-Beschreibung (*QueueDescription*) konfiguriert werden. Die in ihr enthaltene *QueueID* setzt sich, genau wie eine *MessageID,* aus drei Teilen zusammen: Einem Zähler und einer Epoche. Diese beiden Bestandteile garantieren, dass die Queue innerhalb eines Systems eindeutig identifiziert werden kann. Die Epoche gewährleistet dabei, dass nach einem Neustart keine doppelten IDs vergeben werden. Zusätzlich wird noch die BASE-*SystemID* hinzugefügt, um eine netzwerkweite Zuordnung zu ermöglichen.

In der *QueueDescription* kann außerdem festgelegt werden, ob die Ablaufzeiten von Nachrichten innerhalb der Queue beachtet werden sollen(*usingTimeouts*). Wird eine Nachricht in eine Queue geschrieben, so wird ihr *timestamp*-Feld auf die aktuelle Systemzeit gesetzt. Vor dem Ausliefern von Nachrichten wird, wenn *usingTimeouts* bei der Queue gesetzt ist, überprüft ob der *timestamp* der Nachricht plus ihre *ttl* kleiner als die aktuelle Systemzeit ist. Trifft dies zu, wird die Nachricht aus der Queue entfernt und stattdessen die nächste passende Nachricht weitergegeben. Außerdem wird auch noch in periodischen Abständen überprüft, ob Nachrichten in der Queue enthalten sind deren Ablaufzeit überschritten ist.

Zur Unterstützung der Prioritäten von Nachrichten verwenden die Queue-Typen mehrere *Vectoren*. Innerhalb der Prioritätsstufen werden die Nachrichten nach ihrem Eintreffen in der Queue abgelegt bzw. wieder ausgeliefert. Die Queues bieten allerdings auch die Möglichkeit, Nachrichten wahlweise auszulesen, indem ein *MessageDescription*-Objekt als Parameter angegeben wird. Dieses wird mit den Nachrichtenköpfen der gespeicherten Nachrichten verglichen wird. Dabei wird das in Kapitel 5 vorgestellte Verfahren verwendet.

Tritt während der Verarbeitung in der Queue ein Fehler auf, so wird eine *QueueException* geworfen. Diese wird an die Anwendung, die den Aufruf initiiert hat, weitergegeben. Fehler können beispielsweise auftreten, wenn versucht wird, in eine

Queue Nachrichten abzulegen, deren maximale Nachrichtenanzahl überschritten ist oder wenn aus leeren Queues eine Nachricht ausgelesen werden soll.

Für die persistente Speicherung von Nachrichten, die der *SimplePersistentQueue*-Typ anbietet, werden aus Performancegründen zwei Dateien verwendet, die für jedes einzelne *Queue*-Objekt angelegt werden. In die erste Datei werden alle Nachrichten, die in der Queue abgelegt wurden, geschrieben. Die zweite Datei enthält alle Nachrichten, die aus der Queue gelesen wurden. Bei einem Neustart, werden diese beiden verglichen und die Nachrichten in der ersten Datei abzüglich derer in der zweiten wieder hergestellt. Um den Speicherbedarf möglichst gering zu halten, erfolgt ein Abgleich der Dateien zusätzlich noch in periodischen Abständen.

6.5.1 QueueConnection

Um auf eine Queue zugreifen zu können, müssen Anwendungen vom *QueueManager* eine *QueueConnection* anfordern. Dafür muss ihm ein *QueueDescription*- bzw. *QueueID*-Objekt übergeben werden. Anhand der darin enthaltenen BASE-SystemID liefert der Queuemanager die Implementierung entweder für den lokalen (*LocalQueueConnection*) oder entfernten (*RemoteQueueConnection*) Zugriff auf die Queue. Durch diese Vorgehensweise ist es für die Anwendungen transparent, auf welchem System die Queue liegt.

Die *QueueConnection*-Schnittstelle bietet Funktionen an, mit denen lesend bzw. schreibend auf eine Queue zugegriffen werden kann. Vor dem Senden einer Nachricht vergibt sie zusätzlich noch die IDs für diese. Tritt während eines Aufrufs ein Fehler auf, so werden die geworfenen *Exceptions* durch die jeweilige QueueConnection an die Anwendung weitergereicht.

Funktion	Beschreibung
putMessage(Message)	Stellt ein Message-Objekt in die Queue.
putAllMessages(Vector)	Stellt mehrere Message-Objekte in die Queue.
getMessage()	Gibt die erste Nachricht in der Queue zurück.
getMessage(MessageDescription)	Gibt die erste Nachricht zurück, deren Nachrichtenkopf der MessageDescription entspricht.
getAllMessages()	Gibt alle Nachrichten in der Queue zurück.
getAllMessages(MessageDescription)	Gibt alle Nachrichten in der Queue zurück, deren Nachrichtenkopf der MessageDescription entsprechen
getMessageDescriptions()	Liefert eine Liste der MessageDescriptions, der in der Queue enthaltenen Nachrichten
isEmpty()	Hiermit kann abgefragt werden, ob die Queue leer ist.
isFull()	Diese Funktion dient zur Abfrage, ob die maximale Nachrichtenanzahl in einer Queue erreicht ist.
getDescription()	Gibt die Beschreibung der Queue zurück.

Tabelle 2: QueueConnection Funktionen

Eine *LocalQueueConnection* leitet die Funktionsaufrufe von Programmen direkt an das Queue-Objekt weiter. Die *RemoteQueueConnection* teilt sich in zwei Teile auf, einen *Proxy* und ein *Skeleton*. Das *Skeleton* wird während der Erzeugung einer Queue an diese gebunden. Der *Proxy* nutzt für die Übertragung der Aufrufe von Anwendungen RPCs über BASE.

7 Evaluation

Nach der Vorstellung der Implementierung des Queuing-Systems soll in diesem Kapitel geklärt werden, welche Kosten ein asynchroner und indirekter Kommunikationsmechanismus verursacht. Dazu werden zuerst einige Kennzahlen erhoben, die es erlauben, den Entwicklungsaufwand abzuschätzen. Im Weiteren wird eine Abschätzung des benötigten Speicherbedarfs durchgeführt, der vor allem in ubiquitären Systemen eine wichtige Rolle spielt. Zuletzt erfolgt eine Zeitmessung für die Interaktion mit dem Queuing-System. Diese Werte sollen jeweils mit einem unveränderten BASE verglichen werden.

7.1 Implementierung

Die in diesem Kapitel durchgeführten Messungen sollen den Entwicklungsaufwand der Implementierung widerspiegeln. Als Vorlage für die Auswahl von Kennzahlen dienten die Messungen, die für BASE in [28] und für PCom in [29] durchgeführt wurden. Es wurden hierbei fünf Metriken ausgewählt:

- *Anzahl der Codezeilen*, ohne Leerzeilen und Kommentare
- *Anzahl der Klassen*
- *Anzahl der Interfaces*
- *Zyklomatische Komplexität* nach McCabe (Durchschnitt pro Methode)
- *Verschachtelungstiefe* (Durchschnitt pro Methode)

Die beiden letzteren Metriken dienen zur Darstellung der Komplexität der Implementierung, während die ersten drei den Umfang dieser veranschaulichen sollen. Alle Metriken wurden jeweils für ein vollständiges BASE- und Queuing-System erhoben.

Die Messungen wurden mit dem Eclipse Metrics Plugin (Version 1.3.5) durchgeführt. Dieses kann unter http://metrics.sourceforge.net kostenlos heruntergeladen werden.

Um das Plugin zu verwenden, ist zusätzlich eine Eclipse-Entwicklungsumgebung nötig, die unter http://www.eclipse.org bezogen werden kann.

Die Ergebnisse der Messung werden in folgender Tabelle dargestellt:

	Zahl der Codezeilen	Zahl der Klassen	Zahl der Interfaces	Zyklomatische Komplexität	Verschachtelungs-tiefe
BASE	3953	48	15	3,051	2,049
Queuing-System	1549	16	5	3,192	1,874

Tabelle 3: Umfang und Komplexität

Betrachtet man den Umfang des Queuing-Systems, fällt auf, dass dieses zu einer beträchtlichen Erweiterung des BASE-Systems führt. Aus diesem Gesichtspunkt scheint die Wahl das Queuing-System nicht in BASE einzubetten, sondern auf einem unveränderten Kernsystem aufzusetzen, richtig gewesen zu sein.

Bei der Komplexitätsmessung bleibt festzuhalten, dass die Werte des Queuing-Systems sehr stark durch Funktionen, wie den Vergleichen der *Message*- oder *QueueDescriptions* in den Queues bzw. dem *QueueLookup*-Manager erhöht werden. Da diese Funktionen relativ einfach gehalten sind, ergibt sich daraus kein Nachteil für die Wartung des Quellcodes.

7.2 Speicherbedarf

Der Speicherbedarf spielt in ubiquitären Systemen eine nicht unbedeutende Rolle, da Geräte zum Einsatz kommen können, die nur über geringe Ressourcen verfügen. Deshalb soll in diesem Abschnitt ermittelt werden, wie viel Speicher das Implementierte Queuing-System benötigt.

7.2.1 Messung

Die hier für die Abschätzung des Speicherbedarfs durchgeführten Messungen gliedern sich in zwei Bereiche. Zum einen wurde die statische Größe des Bytecodes auf dem Datenträger ermittelt, zum anderen der während der Ausführung des Programms benötigten Speicher abgeschätzt. Da das Queuing-System eine Erweiterung von BASE ist, wurden alle Werte für dieses, inklusive eines vollständigen BASE-Systems, ermittelt.

Um den statischen Platzbedarf des Bytecodes zu bestimmen, wurden jeweils Java-Archive für BASE und das Queuing-System erstellt. Bei ihrer Generierung wurde die Standardkomprimierung verwendet.

Für die Erfassung des Speicherbedarfs während der Laufzeit wurde der für die Messung von PCom und BASE in [29] eingesetzte Programmcode verwendet:

```
runtime.gc();                            // Hinweis für Garbage Collector
long total = runtime.totalMemory();      // reservierter Speicher
long free = runtime.freeMemory();        // freier Speicher
long used = total - free;                // genutzer Speicher
System.out.println("Total Memory: " + total);
System.out.println("Free Memory : " + free);
System.out.println("Used Memory : " + used);
```

Der Code wurde in einem zusätzlichen Thread ausgeführt, der ca. jede Sekunde eine neue Messung durchführte.

Da während der Laufzeit der Speicherverbrauch eines Systems abhängig von dem jeweiligen Zustand ist, wurden die Messungen in verschiedenen Situationen durchgeführt, die einen Aufschluss darüber geben sollen, wie sich das System im Einsatz verhält:

- Queuing-System ohne Queue
- Queuing-System mit einer *SimpleQueue*
- Queuing-System mit einer SimplePersistentQueue
- BASE ohne Dienst
- BASE mit Dienst

Der bei BASE verwendete Dienst enthielt dabei keine Funktionen und die jeweiligen Queues enthielten keine Nachrichten. In jedem Fall wurde ein funktionsfähiges BASE Kernsystem, mit Semantic-, Interoperability-, Transport-, Transceiver-, und Discovery-PlugIn, verwendet.

Für die Ermittlung der Ergebnisse wurde eine IBM J9 2.2.0 Java Virtual Machine (JCL Personal Profile 1.0) eingesetzt, die auf einem Rechner unter Windows XP Professional mit Service Pack 2 lief. Die ermittelten Werte für Base werden in der nachstehenden Tabelle zusammengefasst:

Java-Archiv (JAR)	ohne Dienst	mit Dienst
109 KB	120 KB	123 KB

Tabelle 4: Speicherbedarf BASE

Für das Queuing-System ergaben sich folgende Ergebnisse:

Java-Archiv (JAR)	ohne Queue	SimpleQueue	SimplePersistent Queue
147 KB	122 KB	124 KB	126 KB

Tabelle 5: Speicherbedarf Queuing-System (inklusiv BASE)

7.2.2 Auswertung

Vergleicht man die in den Tabellen aufgeführten Ergebnisse für die statische Größe des Bytecodes, lässt sich feststellen, dass das Queuing-System im Vergleich zu einem unveränderten BASE einen um ca. 35% höheren Speicherbedarf hat. Dies spiegelt die schon in der Implementierung erfassten Werte wieder.

Betrachtet man daneben den Speicherverbrauch zur Laufzeit ohne Dienst bzw. ohne Queue, scheint der Unterschied von ca. 2 KB sehr gering. Bei genauerer Betrachtung muss hier allerdings festgestellt werden, dass die Klassen für die Queues hier nicht geladen werden. Der zusätzlich verwendete Speicher des

Queuing-Systems im Vergleich zu BASE wird größtenteils vom *QueueManager* bzw. den von ihm benötigten Klassen, zum Beispiel dem *IDGenerator*, genutzt. Aber auch mit geladenen Queues scheint der Zusatzaufwand noch vertretbar.

Der unterschiedliche Speicherbedarf, der sich bei den beiden Queue-Typen ergibt, ist darauf zurückzuführen, dass die *SimplePersistentQueue* noch zusätzliche Klassen für den Zugriff auf den stabilen Speicher benötigt.

Bei den gemessenen Werten für die Queues muss allerdings beachtet werden, dass diese keine Nachrichten enthielten. Der tatsächliche Speicherbedarf des Systems steigt im Verhältnis zu den in den Queues abgelegten *Message*-Objekten und der Größe der in ihnen enthaltenen Anwendungsdaten.

7.3 Geschwindigkeit

In diesem Abschnitt soll eine Betrachtung der Geschwindigkeit des Queuing-Systems stattfinden. Da das Queuing-System auf BASE aufsetzt, stellt sich hier vor allem die Frage, welche zusätzlichen Kosten für das Lesen und Schreiben von Nachrichten aus bzw. in Queues entstehen.

7.3.1 Messung

Wie für die Messung des Speichers, wurde auch für die Geschwindigkeitsmessung eine IBM J9 2.2.0 Java Virtual Machine (JCL Personal Profile 1.0) verwendet. Hierbei wurde allerdings für die Messungen der Just-In-Time-Compiler abgeschaltet. Außerdem wurden zwei Rechner eingesetzt, die in Tabelle 6 beschrieben werden. Beide Rechner waren über ein 100MBit Ethernet verbunden, das durch einen Switch vom restlichen Netzwerk getrennt war.

	Rechner 1	Rechner 2
Prozessor	Pentium M 1,70GHz	Athlon XP 2000+
Speicher	512 MByte	1024 MByte
Betriebssystem	Windows XP Service Pack 2	Windows XP Service Pack 1

Tabelle 6: Testumgebung

Für die Messung von BASE wurden zwei Dienste verwendet. Der eine bot eine Schnittstelle an, auf die der andere als Nutzer zugriff. Der Dienstanbieter stellte dabei eine Methode bereit, die als Argument ein beliebiges Objekt entgegennahm und keinen Wert zurücklieferte. Diese kehrte nach ihrem Aufruf direkt zurück, d.h. das übergebene Objekt wurde nicht verarbeitet.

Um die Übertragungszeit von BASE zu erfassen, wurden jeweils 1000 lokale und 100 entfernte Aufrufe durchgeführt. Einmal wurde dabei der Methode des Dienstes ein *null*-Objekt, bei einem zweiten Durchlauf ein String aus 1000 Zeichen übergeben. Jede dieser Messungen wurde 20 Mal wiederholt. Für die Zeitmessung wurde folgender Programmcode verwendet:

```
Object = null;                                        //Objekt initialisieren
Long start = System.currentTimeMillis();              //Startzeit in ms
for (int i = 0; i < 1000; i++)
       service.put(n);
long stop = System.currentTimeMillis();               //Ende des Durchlaufs
System.out.println("Time elapsed: "+ (stop-start)); //Dauer bestimmen
```

Für die lokalen Aufrufe wurde ausschließlich der erste Rechner aus Tabelle 5 verwendet. Bei entfernten Aufrufen lief der Dienstnutzer auf dem zweiten der Dienstanbieter auf dem ersten PC. Die ermittelten Ergebnisse der Geschwindigkeit von BASE werden in folgender Tabelle aufgelistet:

	1000 lokale Aufrufe	100 entfernte Aufrufe
Null-Objekt	29 (30/40) ms	709 (625/922) ms
1000 Zeichen String	31 (30/40) ms	766 (687/984) ms

Tabelle 7: Geschwindigkeit BASE

Zur Erfassung der Geschwindigkeit des Queueing-Systems wurden mehrere Dienste implementiert, die auf eine SimpleQueue und eine SimplePersistentQueue lesend bzw. schreibend zugriffen. Die Queues wurden jeweils mit der Unterstützung von Prioritäten und der Beachtung der Lebenszeiten in Nachrichten erstellt.

Gemessen wurde jeweils das Versenden bzw. Empfangen von 1000 lokalen und 100 entfernten Nachrichten. Für die entfernte Kommunikation wurden folgende Operationen auf den Queues getestet: *putMessage(Message), putAllMessages(Vector)*, *getMessage(), getAllMesages()*. Lokal wurde zusätzlich noch *getMessage(MessageDescription) und getAllMesages(MessageDescription)* ausgeführt. Dabei wurden als Anwendungsdaten jeweils ein 1000 Zeichen langer String übergeben. Bei den Funktionen *putMessage()* und *getMessage()* wurde zusätzlich noch ein Durchlauf mit einem *null*-Objekt als Daten durchgeführt.

Für die Zeitmessung wurde ebenfalls der bei BASE verwendete Code genutzt. Genau wie dort wurden alle lokalen Messungen auf dem ersten Rechner ausgeführt. Bei der Messung der entfernten Aufrufe liefen die Anwendungen auf dem zweiten während die Queues auf dem ersten PC ausgeführt wurden. Insgesamt wurde auch hier jede Messung 20 Mal wiederholt, dabei wurden die Queues für jeden Durchlauf neu erstellt. Tabelle 8 zeigt die Werte dieser Messung.

	SimpleQueue (lokal/1000 Nachrichten)	SimplePersistentQueue (lokal/1000 Nachrichten)	SimpleQueue (entfernt/100 Nachrichten)	SimplePersistentQueue (entfernt /100 Nachrichten)
putMessage() (null-Objekt)	20 (10/30) ms	2546 (2483/3483) ms	713 (687/750) ms	848 (703/922) ms
getMessage() (null-Objekt)	15 (10/20) ms	2495 (2414/2533) ms	710 (687/750) ms	900 (672/1036) ms
putMessage() (1000 Zeichen)	20 (20/21) ms	3365 (3345/3395) ms	807 (750/985) ms	892 (672/1235) ms
getMessage() (1000 Zeichen)	18 (10/20) ms	3378 (3345/3425) ms	982 (750/1125) ms	1016 (625/1328) ms
putAllMessages() (1000 Zeichen)	20 (20/20) ms	3374 (3334/3455) ms	185 (171/203) ms	335 (219/438) ms
getAllMessage() (1000 Zeichen)	ca. 0-10 ms	ca. 0-10 ms	202 (140/234) ms	200 (109/360) ms
getMessage(MD) (1000 Zeichen)	821 (811/841) ms	4158 (4135/4246) ms	-	-
getAllMessages(MD) (1000 Zeichen)	209 (200/211) ms	3791 (3775/3846) ms	-	-

Tabelle 8: Geschwindigkeit Queuing-System

7.3.2 Auswertung

Betrachtet man die Ergebnisse, die für lokale Aufrufe von BASE ermittelt wurden und vergleicht diese mit denen auf einer *SimpleQueue,* kann festgestellt werden, dass diese nahezu gleich sind. Dies kann darauf zurückgeführt werden, dass das Queuing-System für den lokalen Zugriff auf eine Queue BASE nicht verwendet, sondern die Nachrichten direkt an diese weitergibt. Auch kann festgestellt werden, dass die Zeiten für die verschiedenen Operationen sich kaum unterscheiden. Nur die Zeiten für das Auslesen aller Nachrichten aus der Queue und das Suchen nach bestimmten Nachrichten fallen aus der Rolle. Im ersten Fall liegt die kürzere Dauer daran, dass die Nachrichten nicht einzeln übermittelt werden müssen sondern einfach alle auf einmal übergeben werden können. Dabei scheint es überraschend, dass nicht auch beim Senden mehrerer Nachrichten auf einmal, mittels *putAllMessages,* eine kürzere Zeitspanne gemessen wurde. Hier muss bedacht werden, dass die Nachrichten innerhalb der Queue einzeln verarbeitet werden müssen und zuvor jeder einzelnen eine ID zugewiesen wird. Beim zweiten Fall liegen die Gründe für die erhöhte Übertragungszeit auf der Hand, die Queue muss zuerst

nach einer passenden Nachricht durchsucht werden. Die unterschiedlichen Zeiten bei den Suchoperationen erklären sich aus den schon beim Auslesen aller Nachrichten genannten Gründen.

Die stark erhöhten Messungen bei der *SimplePersistentQueue* entstehen durch den Zugriff auf stabilen Speicher. Im Wesentlichen zeigt diese sonst das gleiche Verhalten. Nur die Übertragungszeiten für eine Nachricht mit *null*-Objekt sind im Verhältnis kürzer. Dies liegt vor allem daran, dass dieses schneller serialisiert werden kann.

Vergleicht man nun die entfernten Aufrufe von BASE mit den Übertragungszeiten der Nachrichten bei *putMessage* bzw. *getMessage* auf einer *SimpleQueue*, sind diese leicht erhöht. Dies kann darauf zurückgeführt werden, dass zusätzlich zu den Anwendungsdaten, die bei beiden Übermittlungen identisch sind, beim Queuing-System noch der Nachrichtenkopf übertragen werden muss. Beim Empfangen aller Nachrichten aus einer Queue ist die gemessene Zeit ebenfalls wesentlich kürzer als bei der Einzelübertragung der Nachrichten, da nicht alle Nachrichten einzeln serialisiert werden. Im Gegensatz zum Zugriff auf lokale Queues ist im entfernten Fall auch die Operation *putAllMessages()* schneller als das Versenden von einzelnen Nachrichten. Dies liegt wiederum an der Serialisierung von Daten für die Übertragung. Die SimplePersistentQueue zeigt auch hier im Grundsatz das gleiche Verhalten wie die *SimpleQueue*.

Nutzt man den Just-In-Time-Compiler verbessern sich die Werte deutlich.

8 Zusammenfassung und Ausblick

In diesem Kapitel soll ein abschließender Überblick über die Arbeit gegeben werden. Den Schluss bildet ein Ausblick mit möglichen Erweiterungen für das entwickelte System.

8.1 Zusammenfassung

Ziel dieser Arbeit war es, einen asynchronen und indirekten Kommunikationsmechanismus für ubiquitäre Systeme zu entwickeln. Dafür wurden zuerst existierende Systeme untersucht und deren Eignung für den Einsatz in einem ubiquitären Umfeld diskutiert. Dabei stellte sich heraus, dass diese derzeit nur bedingt dafür geeignet sind. Anschließend wurden aus den besonderen Eigenschaften ubiquitärer Rechnersysteme Anforderungen für das zu entwickelnde System abgeleitet.

Als Grundlage für die Konzeption des Kommunikationsmechanismus diente das Konzept des Message Queuing. Es wurde ein funktionsfähiges und erweiterbares Queuing-System erarbeitet, das eine indirekte und asynchrone Kommunikation unter Verwendung von Queues und Nachrichten ermöglicht. Die Implementierung des Systems baut auf der Verteilungsinfrastruktur BASE auf und machte sich die Plattformunabhängigkeit von Java zu Nutze. Bei der Entwicklung wurde vor allem auf geringen Speicherbedarf, hohe Geschwindigkeit, leichte Erweiterbarkeit und Wartbarkeit geachtet.

Bei der anschließenden Evaluation des implementierten Queuing-Systems wurde festgestellt, dass das System eine vergleichsweise hohe Geschwindigkeit für die Interaktion mit einfachen Queues aufwies. Allerdings sind die Ergebnisse für die Operationen auf Queues mit persistenter Nachrichtenspeicherung nicht vollständig zufrieden stellend. Beim Speicherbedarf wurde festgestellt, dass der Aufwand stark von der Anzahl und Größe der in den Queues abgelegten Nachrichtenobjekten

abhängt. Das Kernsystem weist im Vergleich zu Base nur einen geringen Mehrbedarf an Speicher auf.

Im Folgenden soll noch einmal zusammengefasst werden, wodurch die gestellten Anforderungen an das Queuing-System erfüllt werden:

Eine **indirekte und asynchrone Kommunikation** wurde durch den Einsatz von Queues ermöglicht, die als Kommunikationspuffer zwischen Anwendungen dienen. In ihnen können Nachrichten zwischengespeichert werden, bis sie vom Empfänger entgegengenommen werden können. Durch die Queues muss ein Dienst nicht mehr direkt angesprochen werden, sondern es ist beispielsweise möglich, eine Dienstklasse anzusprechen, deren Instanzen auf eine Queue zugreifen. Sendern muss also die konkrete Adresse eines Dienstes, den sie nutzen möchten nicht mehr bekannt sein.

Die Forderung nach **Minimalität** kam in der Nutzung der Java 2 Micro Edition und von BASE zum Ausdruck, die speziell für ressourcenarme Geräte ausgelegt sind. Des Weiteren trug die Entwicklung eines erweiterbaren Queuemanagers dazu bei, dass die Anforderungen des Kernsystems an die Ressourcen der Geräte verringert wurden.

Die **Anpassbarkeit** wird ebenfalls durch den Queuemanager erreicht, da in diesen, je nach Bedarf, verschiedene Queue-Typen integriert werden können. Auch die Queues sind mittels der Queuebeschreibungen an die jeweiligen Anforderungen und Eigenschaften der Anwendungen bzw. Geräte anpassbar.

Mit den Queue-Connections wird eine einheitliche und **einfache Schnittstelle** für den Zugriff sowohl auf lokale als auch entfernte Queues bereitgestellt. Außerdem bietet der Queuemanager relativ einfache Funktionen für die Verwaltung von Queues an.

Die **Unabhängigkeit von zentralen Diensten** wurde durch den Einsatz des QueueLookupmanagers erreicht, da der LookupManager das Suchen nach Queues auf allen Geräten ermöglicht, die dem BASE-System bekannt sind.

8.2 Ausblick

Das entwickelte Queuing-System stellt eine funktionsfähige Erweiterung für BASE dar. Im Rahmen der Bearbeitung wurden noch verschiedene Ansatzpunkte für Verbesserungs- und Erweiterungsmöglichkeiten identifiziert.

Ein wichtiger Punkt bei den Verbesserungsmöglichkeiten ist die Behebung der Folgen eines Verbindungsausfalls zwischen Anwendung und Queue. Wird die Verbindung während der Datenübertragung unterbrochen, kann dies zu einem Verlust von Nachrichten führen. Eine Möglichkeit dies zu beheben, wäre beispielsweise die Bereitstellung eines Mechanismus in BASE, der auch Dienste darüber Informiert, dass das Senden einer Antwort auf einen Aufruf fehlgeschlagen ist.

Eine weitere Verbesserungsmöglichkeit wäre die Einführung eines besser geeigneten Verfahrens zur persistenten Speicherung von Nachrichten, um die Geschwindigkeit bei dieser zu erhöhen. Dabei müsste untersucht werden, welche Methoden sich für den Einsatz in ubiquitären Systemen eignen. Zum Beispiel könnten Queue-Typen eingeführt werden, die auf eine Datenbank zugreifen. Es könnten auch Möglichkeiten angeboten werden, Nachrichten zu komprimieren um eine schnellere und platzschonende Speicherung zu ermöglichen. Dies hätte auch den Vorteil, dass weniger Daten über das Netzwerk ausgetauscht werden müssen.

Es bietet sich auch an, ein Zugriffskontrollsystem für die einzelnen Queues zu realisieren, um zu gewährleisten, dass nur bestimmte Anwendungen Zugriff auf bestimmte Nachrichten bzw. Daten bekommen. Dies ist besonders im Hinblick auf die Dynamik in einem ubiquitären Umfeld wichtig, da sich hier verschiedene Geräte von verschiedenen Nutzern spontan verbinden und nicht davon ausgegangen werden kann, dass nur mit einer Gruppe feststehender und zuverlässiger Systeme kommuniziert wird. Durch den Einsatz von Verschlüsselungsverfahren kann auch eine Verbesserung der Sicherheit der Anwendungsdaten erreicht werden. Dazu kann

zum Beispiel in einer *QueueConnection* die Möglichkeit integriert werden die Daten vor dem Senden zu verschlüsseln.

Generell ist zu sagen, dass die Einführung weiterer Queue-Typen für das System notwendig ist, um es noch besser an die verschiedenen Eigenschaften und zur Verfügung stehenden Ressourcen der Geräte in ubiquitären Systemen anzupassen.

Literaturverzeichnis

[1] **3PC**, http://www.3pc.info, Universität Stuttgart

[2] **CORBA**, http://www.corba.org, Open Management Group

[3] **Java 2 Plattform Micro Edition Connected Limited Device Configuration (CDLC)**, http://java.sun.com/products/cldc/index.jsp, Sun Microsystems

[4] **Java 2 Platform Micro Edition Overview**, http://java.sun.com/j2me/j2me-ds.pdf, Sun Microsystems

[5] **Jini**, http://wwws.sun.com/software/jini/, Sun Microsystems

[6] **Jini Networking Technology**, http://www.sun.com/software/jini/whitepapers/jini-execoverview.pdf, Sun Microsystems

[7] **Metrics plugin for Eclipse**, http://metrics.sourceforge.net/, Frank Sauer

[8] **Websphere MQ Application Programming Guide**, http://publibfp.boulder.ibm.com/epubs/pdf/csqzal09.pdf, IBM, 2003

[9] **Websphere MQ Application Programming Reference**, http://publibfp.boulder.ibm.com/epubs/pdf/csqzak09.pdf, IBM, 2003

[10] **Websphere MQ Clients**, http://publibfp.boulder.ibm.com/epubs/pdf/csqzaf07.pdf, IBM, 2002

[11] **Websphere MQ Intercommunication**, http://publibfp.boulder.ibm.com/epubs/pdf/csqzae09.pdf, IBM, 2003

[12] **Websphere MQ Messages**, http://publibfp.boulder.ibm.com/epubs/pdf/amqzao04.pdf, IBM, 2002

[13] **Websphere MQ Messages**,
http://publibfp.boulder.ibm.com/epubs/pdf/amqzao04.pdf, IBM, 2002

[14] **Websphere MQ**, http://www-306.ibm.com/software/integration/wmq/, IBM

[15] James Hart, **Connecting your applications without complex programming.**,
http://publibfp.boulder.ibm.com/epubs/pdf/p5239000.pdf , IBM,September 2003

[16] **TSapces**, http://www.almaden.ibm.com/cs/TSpaces/, IBM

[17] Peter. Wyckoff, **TSpaces**, IBM Systems Journal, August 1998

[18] **IBM TSpaces Version 3 Client Programmer's Guide**,
http://www.almaden.ibm.com/cs/TSpaces/Version3/ClientProgrGuide.html, IBM

[19] **Lime: Linda in a Mobile Environment**, http://lime.sourceforge.net/

[20] Gian Pietro Picco, Amy L. Murphy, Gruia-Catalin Roman, **Lime: Linda Meets Mobility**, In Proceedings of the 21stInternational Conference on Software Engineering (ICSE'99), Los Angeles (USA), May 1999

[21] Amy L. Murphy, Gian Pietro Picco, Gruia-Catalin Roman, **Lime: A Middleware for Physical and Logical Mobility**, In Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS-21), Phoenix, AZ, USA, April 16-19 2001, pp. 524-233.

[22] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman, **Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents**,
http://www.elet.polimi.it/upload/picco/papers/formalLime.pdf

[23] **Java Message Service**, http://java.sun.com/products/jms/, Sun Microsystems

[24] **Java Message Service Specification** , http://java.sun.com/products/jms/docs.html, Sun Microsystems, April 2002

[25] **Java Message Service Tutorial**, http://java.sun.com/products/jms/tutorial/1_3_1-fcs/doc/copyright.html, , Sun Microsystems, 2002

[26] **JavaSpaces Service Specification**, http://www.sun.com/software/jini/specs/js2_0.pdf, Sun Microsystems, 2003

[27] David Garlan, João Pedro Sousa, **Aura: An Architectural Framework for User Mobility in Ubiquitous Computing Environments**, Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture, August 2002

[28] Holger Gubbels, **Basisdienste für Anwendungen in ubiquitären Rechnersystemen**,Universität Stuttgart, Dezember 2002

[29] Marcus Handte, **Entwicklung eines Komponentensystems für Anwendungen in ubiquitären Rechnersystemen**, Universität Stuttgart, 2003

[30] Manuel Román, Christopher Hess, Renato Cerqueira, Roy Campbell, Klara Nahrstedt, **Gaia: A Middleware Infrastructure to Enable Active Spaces**, UIUC, Juni 2002

[31] Manuel Román, Roy Campbell, Christopher Hess, Roy Campbell, **Gaia: An OO Middleware Infrastructure for Ubiquitous Computing Environments**, ECOOP-OOOSWS, Magala, Spanien, 2002

[32] Mark Weiser, **Some Computer Science Issues in Ubiquitous Computing**, Communications of the ACM, 36(7), 74-84, July 1993

[33] Mark Weiser, **The Computer for the 21st Century**, Scientific American, 265(3), 94-104, September 1991

[34] Mark Weiser, **Ubiquitous Computing**, IEEE Computer Hot Topics, 71-72, Oktober 1993

[35] Christian Becker, Gregor Schiele, Holger Gubbels, **BASE – A Micro-broker-based Middleware For Pervasive Computing**, Proceedings of the 1st IEEE International Conference on Pervasive Computing and Communication, 443-451, März 2003

[36] **Java 2 Platform Standard Edition (Version 1.4.2) API Specification**, http://java.sun.com/j2se/1.4.2/docs/api/index.html, Sun Microsystems, 2003

[37] **Websphere MQ everyplace**, http://www-306.ibm.com/software/integration/wmqe/, IBM

[38] **MQe Getting Started** ftp://ftp.software.ibm.com/software/websphere/mqeveryplace/pmqeint01.pdf, IBM, Juli 2004

[39] **MQe Application Programming**, ftp://ftp.software.ibm.com/software/websphere/mqeveryplace/pmqeapg01.pdf, IBM, Juli 2004

[40] W. Keith Edwards, **Core Jini**, Prentice Hall PTR, 2001

[41] Tuple Spaces, http://c2.com/cgi/wiki?TupleSpace

[42] **Plurix**, http://www.plurix.de/, Universität Ulm

[43] M. Schoettner, S. Traub , P. Schulthess, A transactional DSM Operating System in Java, Universität Ulm, 1998

[44] Robert Grimm, **System Support for Pervasive Applications**, ACM Transactions on Computer Systems (TOCS), Volume 22, Issue 4, pages 421-486, November 2004

[45] Brad Johanson, Armando Fox, **Extending Tuplespaces for Coordination in Interaktive Workspaces**, Journal of Systems and Software, Volume 69, Issue 3, pages 243-266, 2004

[46] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, Carl H. Hauser, **Managing update conflicts in Bayou, a weakly connected replicated storage system**, In Proceedings of the Fifteenth ACM Symposium on Operating systems principles, pages 172--188, Copper Mountain Resort, Colorado, December 1995

[47] **Linda**, http://lindaspaces.com/about/index.html, Scientific Computing Associates Inc.

[48] Andrew Tanenbaum, Marten van Steen, Verteilte Systeme. Grundlagen und Paradigmen, Prentice Hall, 2002

Erklärung

Ich versichere, dass ich diese Arbeit selbständig verfasst und nur die angegebenen Hilfsmittel verwendet habe.

_____

(Holger Katz)