Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Biologie und Informatik

Diplomarbeit

Entwicklung von Komponenten für eine Verteilungsplattform zur Unterstützung kollaborativer Anwendungen

Christian Fertig

3. November 2003

Gutachter:
Prof. Dr. Oswald Drobnik
Architektur und Betrieb Verteilter Systeme / Telematik

(7) It is always something.
(7a) (corollary). Good, Fast, Cheap: Pick any two (you can’t have all three).

R. Callon, RFC 1925, The Twelve Networking Truths [Cal96]

„Entwicklung von Komponenten für eine Verteilungsplattform
zur Unterstützung kollaborativer Anwendungen“
gesetzt mit pdfTEX/pdfLATEX 2$_\varepsilon$ unter Linux zuletzt am 3. November 2003

Danksagung

Ich möchte mich hiermit bei allen Personen bedanken, die mich bei der Anfertigung dieser Arbeit unterstützt haben. Mein besonderer Dank gilt Michael Lauer, Michael Matthes und Prof. Dr. Oswald Drobnik für ihre ausgezeichnete Betreuung und ihre wertvollen Anregungen.

Christian Fertig

Hiermit bestätige ich, daß ich die vorliegende Arbeit selbständig verfaßt habe und keine anderen Quellen oder Hilfsmittel als die in dieser Arbeit angegebenen verwendet habe.

Frankfurt am Main, den 3. November 2003

Christian Fertig

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Ziele der Arbeit . . . 2
- 1.2 Aufbau der Arbeit . . . 3

2 Entwicklung von Anwendungen in mobilen Ad-hoc Netzen **4**
- 2.1 Eigenschaften von Ad-hoc Netzwerken . . . 4
- 2.2 Verteilungsplattformen in Ad-hoc Netzwerken . . . 5
  - 2.2.1 Middleware . . . 5
  - 2.2.2 Asynchrone Nachrichten . . . 6
  - 2.2.3 Awareness . . . 6
  - 2.2.4 Reflection Techniques . . . 7
- 2.3 Kollaborationsumgebungen . . . 8
  - 2.3.1 E-Learning in Ad-hoc Netzen (E.L.A.N) . . . 8
  - 2.3.2 Aufbau des E.L.A.N-Projekts . . . 9
- 2.4 Experimentierplattformen . . . 10
  - 2.4.1 Wireless LAN . . . 10
  - 2.4.2 Zielplattformen . . . 11
  - 2.4.3 Die Klassenbibliotheken Qt und Qt / Embedded . . . 11
  - 2.4.4 Simulierung der Oberflächen über VNC und QVFB . . . 13
  - 2.4.5 Die Programmiersprache Python . . . 13

3 Konzeption der Verteilungsplattform **15**
- 3.1 Kommunikationsschichten . . . 15
- 3.2 Verteilungsinhalte . . . 16
  - 3.2.1 Verteilung von Awareness-Information . . . 16
  - 3.2.2 Verteilung von Dienstinformationen . . . 18
  - 3.2.3 Verteilung von Diensten . . . 19
  - 3.2.4 Verteilung von Nachrichten . . . 20
- 3.3 Middleware mit integriertem Awareness-System (midas) . . . 20
  - 3.3.1 Aufbau der Middleware . . . 20
  - 3.3.2 Kommunikationskomponenten . . . 21
  - 3.3.3 Verwaltungskomponenten . . . 22
- 3.4 Routing-Schicht . . . 23
  - 3.4.1 Aufgaben des Routings . . . 23
  - 3.4.2 Konzeption der Routingschicht . . . 23

3.4.3 LanRouter 24

4 Umsetzung und Implementierung von Middleware und LanRouter 26
4.1 Das Handling von WiLiDips 26
4.1.1 Verwaltung der WiLiDips 27
4.1.2 Identifikation der WiLiDips 28
4.2 Implementierung der Middleware 28
4.2.1 ECom 28
4.2.2 XML-RPC-Schnittstelle 29
4.2.3 extIn- und extOut-Kommunikation 30
4.2.4 Protokollmaschine 33
4.2.5 WilidipManager 34
4.2.6 ServiceDescriptoren 37
4.2.7 ServiceManager 37
4.2.8 Kontrollfluß 37
4.3 Implementierung des LanRouters 39
4.3.1 Programmablauf 39
4.3.2 Nachrichtenverarbeitung 41
4.3.3 Routing 41
4.3.4 XML-RPC-Schnittstelle 42
4.3.5 Optionale GUI 43
4.3.6 Handshake Middleware und Router 44

5 Umsetzung und Implementierung von Evaluationskomponenten 46
5.1 Das Komponentenmodell 46
5.2 Die Awareness-Komponente: awarenessView 48
5.2.1 Aufbau von awarenessView 48
5.2.2 Steuerung der awarenessView Komponente 49
5.3 Die Identifikationskomponente: identd 49
5.4 Die Debuggingkomponente middlewareActivity 50
5.5 Die Konfigurationskomponente elanConfig 52
5.6 Nutzung der unterschiedlichen Dienste 52
5.6.1 Nutzung spezifischer Dienste: Filesharing 52
5.6.2 Nutzung unspezifiziert-expliziter Dienste: messageService 53
5.6.3 Nutzung unspezifiziert-impliziter Dienste: converter/textviewer 55

6 Zusammenfassung und Ausblick 58

Anhang 62
A.1 lego-Nachrichten 62
A.2 Middleware UDP Pakete – MUPs 69
A.3 Wichtige Funktionen 71
A.4 XML-RPC-Schnittstellen 75

Abbildungsverzeichnis **79**

Literaturverzeichnis **81**

1 Einleitung

Immer mehr Hersteller bringen kleine mobile Endgeräte (PDAs[1]) auf den Markt, die via WLAN[2] (IEEE 802.11), Bluetooth oder UMTS[3] drahtlos Kontakt mit der Außenwelt aufnehmen können. Dabei werden neben zunehmend höheren Übertragungsraten auch große Fortschritte in der Miniaturisierung erzielt. Viele PDAs besitzen bereits eingebaute Funknetzwerk-Karten, bei vielen läßt sich eine solche Karte einfach zusätzlich in das Gerät einstecken.

Durch die steigende Rechenleistung der Geräte und die gleichzeitig steigenden Übertragungsraten der Funkverbindungen entstehen diverse neue Anwendungsmöglichkeiten, die allerdings noch wenig ausgenutzt werden. Es überwiegen die Standardaufgaben der PDAs, wie die Termin- und die Adressenverwaltung. Ein eventuell vorhandener Zugang zu Netzwerken wird meist ebenfalls nur für Standardanwendungen verwendet, wie z.B. die Synchronisation von Daten mit einem Server, der Zugang zum World Wide Web oder zum Versenden von e-Mails.

Der herkömmliche Zugang zu Netzwerken ist meist ortsgebunden. Im sogenannten Infrastruktur-Modus ist man auf eine Basisstation (Access-Point) angewiesen. Bewegt man sich in Bereichen, in denen kein solcher Access-Point zur Verfügung steht, kann eine Netzwerkverbindung nicht hergestellt werden. Die Mobilität des Benutzers ist auf die Funkreichweite seines Geräts zu dieser Basisstation beschränkt. Die Bezeichnung „mobil“ trifft also nur auf das Endgerät, nicht auf die Basisstation zu. Genutzt werden üblicherweise die gleichen Dienste wie in einem kabelgebundenem Netz, nämlich Server des Intra-, bzw. Internets.

Die eigenständige Vernetzung mobiler Geräte untereinander, der sogenannte *Ad-hoc-Modus*[4], eröffnet neuartige Anwendungsgebiete. Ursprünglich dafür gedacht, zwei Stationen schnell und einfach über Funk miteinander zu verbinden, können durch eine Erweiterung beliebig viele Stationen zu einem spontanen und mobilen Netzwerk zusammengefaßt werden (mobiles Ad-hoc Netzwerk, MANET). Eine zentral verwaltete Infrastruktur entfällt dabei vollständig. Die Aufgaben des Routings muß jede einzelne der beteiligten Stationen übernehmen. Auf diese Weise erhöht sich die Funkreichweite aller beteiligten Stationen, da nicht ein zentraler Access-Point, sondern jeder beliebige Teilnehmer in Reichweite den Zugriff auf das Netzwerk ermöglicht. Es muß dabei allerdings davon ausgegangen werden, daß sich die Stationen in einem solchen Netz ständig bewegen, das Netz verlassen oder neu hinzukommen können. Somit verändert sich andauernd die Netzwerktopologie und ein kontinuierliches und selbstständiges Neuorganisieren des Netzes wird notwendig. Man spricht daher von *selbstorganisierenden* mobilen Netzen.

Der Informationsfluß in Ad-hoc Netzen ist üblicherweise ein anderer als in Festnetzen. Es entsteht eine eher interessensbezogene Kommunikation, weniger eine, die auf dem Wissen von bekannten Endpunkten im Netzwerk basiert. In Ad-hoc Netzwerken können Endpunkte und

[1] Personal Digital Assistants
[2] Wireless Local Area Network
[3] Universal Mobile Telecommunications System
[4] lateinisch „ad hoc“: sofort, aus dem Augenblick entstanden

Adressen aufgrund der sich permanent ändernden Nutzerzusammensetzung naturgemäß nicht bekannt sein. Stattdessen gibt jeder Nutzer Informationen über seine Interessen und seine Angebote im Netzwerk bekannt. Findet sich ein zu diesem Interessensprofil passendes Angebot, so kommt eine Verbindung zustande.

Die Flexibilität der PDAs, ihre Mobilität und ihre ständig steigende Leistung lassen die Entwicklung von mobilen Ad-hoc-Anwendungen sinnvoll erscheinen. MANET Netzwerke sind in ihrer Auslegung flexibel und schnell, genau so, wie es die modernen PDAs sind. Es eröffnet sich durch durch beides eine große Zahl neuer und innovativer Anwendungsmöglichkeiten. Eine davon, das so genannte E-Learning, soll im Folgenden näher betrachtet werden. E-Learning ist ein Beispiel für eine Anwendung in Ad-hoc Netzen, die besonders geeignet erscheint, die Beweglichkeit ihrer Anwender in vielerlei Hinsicht zu unterstützen. Durch die Eigenschaft, geeignete Kollaborationspartner und Informationen schnell und gezielt im Netzwerk finden zu können, wird ein spontaner Wissensaustausch über die bisher bestehende Grenzen hinaus möglich.

Es werden im Rahmen dieser Diplomarbeit Mechanismen behandelt, die dem Anwender die Nutzung von kollaborativen Anwendungen wie dem E-Learning ermöglichen und ihn bei dessen Nutzung unterstützen sollen.

1.1 Ziele der Arbeit

Im Rahmen des von der Deutschen Forschungsgesellschaft (DFG) eingerichteten Schwerpunktprogramms „Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Systeme“ wurde an der Professur für Architektur und Betrieb verteilter Systeme / Telematik ein Teilprojekt bearbeitet. Es wird eine „adaptive Kollaborationsumgebung für E-Learning in mobilen Ad-hoc Netzen (ELAN)“ entwickelt, die eine Softwareplattform für die Anforderungen des E-Learnings bilden soll.

Die Plattform soll die Kontaktaufnahme der Geräte untereinander ermöglichen und der Verteilung von Informationen in einem spontanen und mobilen Netzwerk dienen. Dazu war es zunächst erforderlich, die benötigten Komponenten einer solchen Plattform im Rahmen des ELAN-Projekts zu bestimmen.

Zu diesen Komponenten gehört insbesondere eine sogenannte Middleware, die geeignete Schnittstellen zur Eingliederung in die parallel zu entwickelnden Routing- und Framework-Schichten zur Verfügung stellen sollte. Eine solche Middleware wurde in der vorliegenden Arbeit entwickelt. Ihre Umsetzung erfolgte in der Programmiersprache Python, da durch den zentralen Charakter der Komponente „Middleware“ eine schnelle prototypische Implementierung erforderlich war. Sie sollte auf PDAs mit dem Betriebssystem Linux arbeiten (z.B. Compaq iPAQ, Sharp Zaurus SL-5500), aber auch auf Notebooks und Workstations lauffähig sein. Die verwendeten Geräte sollten mit WLAN Funkkarten nach IEEE 802.11(b) arbeiten.

Zur Verteilung von Dienstprofilen (Wilidips) wurde eine Verteilungsstrategie erarbeitet und diese in die Middleware implementiert. Dabei mußte auf die dynamische Netzstruktur der Ad-hoc Netzwerke besondere Rücksicht genommen werden, da ihnen eine zentrale Speicher- bzw. Serverinstanz fehlt. Die Middleware sollte sich dabei reflektiv und adaptiv an die Veränderungen der Umgebung durch das Hinzukommen und Wegfallen von Stationen anpassen können.

Eine weitere Anforderung an die Middleware war die Bereitstellung von Context-Awareness-

Informationen an das Framework. Context-Awareness bezeichnet hierbei Informationen über den Ausführungskontext, d.h. die Umgebung eines Gerätes. Darunter fallen z.B. Informationen über den Batteriezustand mobiler Geräte oder über die Qualität einer Funkverbindung.

Desweiteren wurde eine Schnittstelle zur Framework-Framework-Kommunikation implementiert. Als Testcase wurde ein simpler Anwendungsfall, ein Chat, verwendet (messageService).

Neben der Middleware als zentraler Komponente waren noch einige weitere Komponenten zu erarbeiten.

Zur Darstellung möglicher Kollaborationspartner, bzw. möglicher verfügbarer Dienste, ist eine Darstellungskomponente entwickelt worden, die sog. Benutzer- und Diensteansicht (awarenessView). Sie soll u.a. die Eigenschaft haben, die Dynamik des Netzes, sowie die Kontext- und Lokationsabhängigkeit vor den oberen Schichten zu verbergen.

Es folgten einige Beispieldienste, die zum Testen der Umgebung dienen sollten. So z.B. ein Dienst, der auf auf der lokalen Station Hilfstools suchen und Dateitypen umwandeln kann (converter), ein Identifikationsdienst, der Informationen über Kollaborationspartner wie z.B. ein Pictogramm abfragt (identd) und ein Dienst, der zum Debugging bestimmt ist (middlewareActivity).

1.2 Aufbau der Arbeit

Die Arbeit ist in sechs Kapitel wie folgt gegliedert:

Nach einer Einführung in Kapitel 1, die die Motivation und die Ziele der Arbeit beschreibt, behandelt Kapitel 2 zunächst die notwendigen Grundlagen, die zur Entwicklung von Anwendungen in mobilen Ad-hoc-Netzen notwendig sind. Dazu gehören neben den Grundlagen von Middleware-Architekturen die Grundlagen über die benötigte Hard- und Software.

In Kapitel 3 wird der Kommunkikationsfluß innerhalb und zwischen E.L.A.N-Systemen, sowie die zu verteiltenden Inhalte beschrieben. Anschließend wird eine Middleware und deren benötigen Schnittstellen konzeptionell vorgestellt, sowie Aufgaben und Anforderungen an das Routing erörtert und eine Routingschicht für ein Festnetz konzipiert.

Kapitel 4 beschreibt die konkrete Umsetzung der in Kapitel 3 entworfenen Komponenten und deren Integration in das E.L.A.N-Framework. Insbesondere werden auf die Umsetzung und Funktionen der Middleware und des LanRouters eingegangen.

Zur Demonstration der Funktionalität wurden eine Reihe von grafischen Evaluationskomponenten für das Framework entworfen. Die Beschreibung der Funktionsweise dieser Komponenten, sowie eine Beschreibung der Nutzung der vorhandenen Dienste erfolgt in Kapitel 5 der Arbeit.

Eine Zusammenfassung der wesentlichen Punkte der Arbeit, sowie Vorschläge zur Erweiterung des E.L.A.N-Projekts erfolgen in Kapitel 6. Abschließend wird ein Ausblick gegeben.

2 Entwicklung von Anwendungen in mobilen Ad-hoc Netzen

2.1 Eigenschaften von Ad-hoc Netzwerken

Der Begriff Ad-hoc Netzwerk steht für ein mobiles, selbstorganisierendes Netz. Typischerweise kommunizieren die Stationen eines mobilen Netzes (auch Knoten genannt) über Funkverbindungen. Die normalerweise kugelförmigen Empfangsbereiche der Funknetzwerkkarten sind oftmals durch Wände oder andere Hindernisse gestört. Sie bekommen durch die örtlichen Gegebenheiten bedingte Deformationen. Diese Funkzellen werden daher oft als „Funkwolken“ bezeichnet. Sind zwei Stationen in Empfangsreichweite zueinander, spricht man davon, daß sie sich „sehen“. Mehrere Stationen in einem Ad-hoc Netzwerk bilden eine Ad-hoc Wolke.

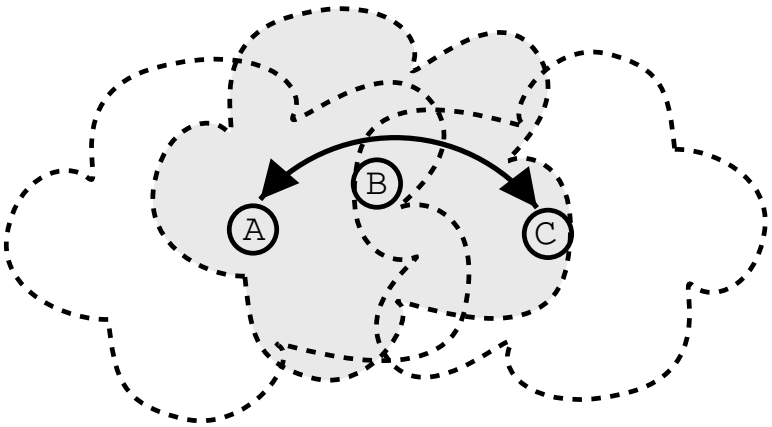

Abbildung 2.1: Eine Ad-hoc Wolke mit drei Stationen

Im Gegensatz zu herkömmlichen Netzen existieren im Ad-hoc Netz keine Spezialknoten, die Serverdienste wie Routing, Paketweiterleitung oder Namensauflösung (DNS, DHCP) übernehmen. Weiterhin gibt es oftmals keine Benutzerauthentifizierung oder ähnliche Sicherheitsmechanismen. Ad-hoc Netze sind infrastrukturlos. Jeder Knoten wirkt zugleich als Router und als Anwendungsknoten. Er leitet Datenpakete auch für seine Nachbarn weiter. Abbildung 2.1 zeigt drei Stationen in einer Ad-hoc Wolke. Station **A** und **C** „sehen“ jeweils nur die Station **B**, sie haben keinen direkten Kontakt miteinander. Damit sie trotzdem kommunizieren können, muß Station **B** die Pakete von **A** und **C** jeweils weiterleiten.

Ad-hoc Netze haben einige besondere Eigenschaften, die sie von den bisher üblichen Netz-

werken unterscheiden. Ein wesentlicher Unterschied ist die dynamische Netzwerktopologie. Da die Netzwerkknoten beweglich sind, ändert sich die Zusammensetzung des Netzes ständig. Das gesamte Netz kann sich in kleinere Teilnetze partitionieren oder zwei disjunkte, d.h. völlig getrennte Netzwerkwolken, können sich vereinigen. Das Netz muß sich darauf einstellen können. Ad-hoc Netzwerke mit diesen Eigenschaften werden daher als *mobile Ad-hoc Netzwerke* (MANETs) bezeichnet.

Ein wesentlicher Nachteil der Funknetzverbindungen sind die gegenüber den festen Netzwerken noch deutlich geringeren Bandbreiten. Zusätzlich wird ein Teil der Bandbreite für Topologieinformationen und Routingpakete anderer Stationen benötigt, wodurch sich die Bandbreite weiter verringert. Den mobilen Geräten selbst steht meist wenig Speicherplatz, verhältnismäßig geringe CPU-Leistung und begrenzte Energie zur Verfügung. Diese Faktoren wirken sich limitierend auf ihre Verwendung in mobilen Netzwerken aus.

Die geringe Sicherheit der Funkverbindungen erhöhen grundsätzlich die Gefahr von Angriffen und vom Abhören sensibler Informationen. Die Sicherheit ist gering, da ein eventueller Angreifer bereits dann physikalischen Zugang zum Netzwerk hat, wenn er sich in dessen Funkreichweite befindet. Die im Infrastruktur-Modus vorgesehene Verschlüsselung über WEP[1] ist schon vor einiger Zeit umgangen worden und gilt seither als nicht mehr sicher. Eine Umsetzung dieser oder anderer Verschüsselungsmethoden stehen für den Ad-hoc Modus z.Zt. noch nicht zur Verfügung. Der Datenverkehr im Ad-hoc Netz muß infolgedessen unverschlüsselt ablaufen.

2.2 Verteilungsplattformen in Ad-hoc Netzwerken

2.2.1 Middleware

Beim Entwickeln verteilter Systeme möchten Anwendungsprogrammierer nicht direkt mit den Eigenschaften der Verteilung wie Ressourcenteilung, Netzwerkkommunikation, Skalabilität usw. konfrontiert werden. Dies würde zu ständig neuen und proprietären Lösungsansätzen führen. Man versucht deshalb, die Komplexität der Verteilung durch Einfügen einer weiteren Abstraktionsschicht zwischen den Anwendungen und dem Netzwerk zu verringern. Diese neue Abstraktionsschicht, die sogenannte Middleware, übernimmt die Kommunikation mit dem Netzwerk und stellt der Anwendungsschicht eine vereinfachte Schnittstelle zur Verfügung.

Gängige Middleware-Architekturen, wie Corba[2], DCOM[3] und RPC[4] wurden unter dem Gesichtspunkt einer *Black Box* entwickelt, die sowohl dem Anwendungsprogrammierer als auch dem Anwender vermittelt, es mit einem einzigen System zu tun zu haben [EAB+99]. Externe Prozeduren können in solchen Systemen ausgeführt werden, als wären sie lokal verfügbar. Es ist nicht direkt ersichtlich, ob diese Prozeduren lokal ablaufen oder nicht. In objektorientierten Middleware-Architekturen können ganze Instanzen von Objekten behandelt werden, als wären sie auf dem aktuellen System vorhanden. Dies erlaubt eine orts- und plattformunabhängige Kommunikation zwischen den Applikationen.

[1] Wired Equivalent Privacy
[2] Common Object Request Broker Architecture, `http://www.corba.org/`
[3] Distributed Component Object Model
[4] Remote Procedure Call

Diese Plattformen sind primär für kabelgebundene Netze entwickelt worden und setzen daher oftmals eine hohe Bandbreite und die ständige Verfügbarkeit der Kommunikationspartner voraus. Weiterhin basieren viele der objektorientierten Systeme – wie beispielsweise Corba – auf stehenden Punkt-zu-Punkt Verbindungen.

Das Verbergen der Netzwerkeigenschaften ist im dynamischen Netz sehr viel schwieriger als sonst üblich. Die Middleware muß hier Entscheidungen anhand der aktuellen Anwendung treffen. Oftmals ist es effizienter, solche Entscheidungen den Anwendungen selber zu überlassen. Dies kann nur durch ein gewisses Maß an Transparenz, also das Durchreichen von Informationen von der Middleware an die Anwendungsschicht, erreicht werden [CEM01b].

2.2.2 Asynchrone Nachrichten

Die bereits oben angesprochenen Punkt-zu-Punkt-Verbindungen via TCP/IP sind für Ad-hoc Netze wegen der ständigen Topologieveränderungen nicht geeignet, denn TCP / IP arbeitet verbindungsorientiert. Es benötigt zwingend eine Hin- und eine Rückroute. Diese müssen für jede Verbindung aufgebaut und gehalten werden. Nach eventuellen Verbindungsabrüchen müssen sie neu initialisiert werden. TCP / IP-Verbindungen benötigen aufgrund des 3-Wege-Handshakes dafür relativ lange. Das verbindungslose UDP-Protokoll hingegen vermeidet diese Nachteile und bietet sich deshalb für Ad-hoc-Netzwerke an.

Die Kommunikation muß daher über ein einfaches, asynchrones und unidirektionales Nachrichtensystem nach dem „fire and forget“-Prinzip, also dem Versenden von Nachrichten ohne auf weitere Rückmeldung zu warten, erfolgen. Dafür bieten sich UDP-Broadcasts oder Unicasts an. Eine weitere Begründung für die Wahl von UDP-Broadcasts ist das eventuelle Fehlen einer Rückroute. Die im vorliegenden Fall konzipierte Routingschicht baut eine Rückroute nur dann auf, wenn sie explizit angefordert wurde.

Ddas Fehlen der Rückmeldung einer Nachricht ist gleichzeitig Hauptnachteil asynchroner Nachrichtensysteme. Der Empänger einer Nachricht muß manuell eine Antwort-Nachricht zurücksenden. Ob eine Nachricht angekommen ist oder nicht, läßt sich also nicht ohne weiteres herausfinden. Ein auf asynchroner Kommunikation aufgebautes Framework muß in der Lage sein, einen eventuellen Nachrichtenverlust kompensieren zu können. Notfalls muß dies durch Neuverschicken der Nachricht geschehen.

2.2.3 Awareness

Der Begriff *Awareness* wird im Zusammenhang mit User / Service-Awareness, Context-Awareness und Location-Awareness verwendet. Im Allgemeinen versteht man darunter das „sich-bewußt-sein“ einer Anwendung über ihre Umgebung.

Eine awareness-basierte Middleware muß Informationen über den aktuellen Zustand und die Veränderung in einem Ad-hoc Netzwerk sammeln. Dazu müssen geeignete Profile verteilt werden, die die vorhandenen oder gesuchten Dienste im Netz repräsentieren. Die Verteilung soll möglichst effizient erfolgen, um die Bandbreite des Ad-hoc Netzes nicht übermäßig zu beanspruchen. Ein Profil soll über das gesamte Netzwerk verteilt werden. Durch dessen dynamische Veränderungen wird es notwendig werden, die Verteilung regelmäßig zu wiederholen. Verteilungsalgorithmen müssen die Veränderungen im Netz in ihre Strategien mit einbeziehen. Das

passive Sammeln von Daten sollte dabei bevorzugt werden, um eine erhöhte Netzflutung zu vermeiden [LMD03]. Man unterscheidet verschiedene Arten von Awareness.

Unter *Context-Awareness* versteht man Anwendungen, die sich ihrerer Ausführungsumgebung „bewußt“ sind. Damit sind Informationen gemeint, die Einfluß auf eine Anwendung haben können. Man unterscheidet dabei zwischen der Context-Awareness bezüglich des eigenen Geräts (*Device-Awareness*) und bezüglich der örtlichen Umgebung (*Environment-Awareness* oder *Location-Awareness*).

Die *Device-Awareness* bezieht sich physisch auf das aktuelle Gerät. Darunter fallen zum Beispiel Monitor-Größe, verfügbarer Arbeitsspeicher, verfügbare Rechenleistung, der Zustand der Batterieversorgung und die Qualität der Funkverbindung. Werte von Umgebungssensoren, die die Ausführung eines Programms beeinflussen können, fallen unter die Environment-Awareness. Ein Beispiel hierfür wäre die Übertragung von Werten eines Temperaturfühlers zur Vermeidung von Überhitzung in extremen Umgebungen.

Anwendungen, die Informationen über ihren genauen geographischen Aufenthaltsort verwenden, bezeichnet man als *location-aware*. Diese Informationen können z.B. über ein satellitenbasiertes Ortungssystem wie *GPS*[5] gewonnen werden. Aber auch ausgewählte Bezugspunkte eines geschlossenen Systems, die als solche erkannt werden und deren Position bekannt ist, fallen unter die Location-Awareness.

Ein Drucker muß spontan im Netz gefunden werden können. Dazu muß dabei den örtlichen Gegebenheiten Rechnung getragen werden. Denn üblicherweise möchte man den nächstliegenden Drucker verwenden, und nicht einen, der z.B. in einem weit entfernten Gebäude steht.

2.2.4 Reflection Techniques

Dadurch, daß sich ein System seines Ausführungskontextes bewußt ist, kann es Veränderungen dieses Kontextes erkennen und berücksichtigen.

Ein System, das in der Lage ist, sich als Antwort auf Veränderungen der Umgebung dynamisch zur Laufzeit und ohne Benutzereinwirkung selbst zu rekonfigurieren, wird *reflection-based* genannt [CBC98, KCBC02].

Reflection-Techniken einzusetzen erscheint im Kontext der Anforderungen für Anwendungen in Ad-hoc Netzen sehr geeignet, da sich dort die Umgebung andauernd ändert und eine Anpassung erfolgen muß.

Gesteuert durch Reflection können Anwendungsteile nachträglich geladen und entladen werden. Auch wenn PDAs immer leistungsstärker werden, kommen sie nicht in die selbe Leistungsklasse wie Desktop-Rechner und Workstations. Insbesondere die Begrenzung des Arbeitsspeichers auf die heutezutage üblichen 32 bis 64 MBytes legen es nahe, Komponenten nur dann zu laden, wenn sie auch gebraucht werden (*loading on demand*). Dies wird durch Framework-Architekturen, denen leichtgewichtige Plugin-Architekturen integriert sind, erleichtert.

Ebenso ist die Fähigkeit, Teile eines Programms nachzuladen, nicht auf die lokale Station beschränkt. Auch aktualisierte Versionen von Plugins können automatisch über das Netz abgeglichen werden. Gemeint ist hier nicht nur mobiler Code im Sinne von mobilen Agenten (die

[5] Global Positioning System

selbst aktiv zur Laufzeit von einer Station auf die nächste migrieren können), sondern ganze Programmteile, die zur Ausführung benötigt werden.

Middleware-Architekturen können den Wegfall eines Dienstes durch Suchen eines Ersatz-Dienstes kompensieren. Dies geschieht für den Anwendungs-Entwickler, aber auch für den Endanwender völlig transparent. Verringert sich die Übertragungsqualität oder geht die Batterieladung eines Teilnehmersgerätes im Netzwerk ihrem Ende zu, besteht die Möglichkeit, aktiv Dienste auf andere Geräte migrieren zu können, die diese dann übernehmen. Dienstspezifisches Routing ist möglich. Es versucht, die Routen zu wichtigen Diensten zu optimieren, während für untergeordnete Dienste ein weniger optimaler Weg in Kauf genommen werden kann. Ebenso ist ein Routing, das auf die Erhöhung der Qualität der Verbindungswege verstärkten Wert legt, vorstellbar.

2.3 Kollaborationsumgebungen

Die Lerngruppe, die sich spontan trifft und Wissen austauscht, ist ein typisches Beispiel für ein kollaboratives Netz. Informationen, die dem einem Mitglied fehlen, können von einem anderen Mitglied bezogen werden. Dieses versorgt sich seinerseits mit fehlenden Informationen an anderer Stelle in der Gruppe. Auf diese Weise wird ein Wissensaustausch ermöglicht, der das vorhandene Wissen zum Nutzen aller Beteiligter verteilt. Zunächst ist es jedoch nötig, eine Gruppe möglicher Kollaborationspartner zu finden, die die benötigten Informationen auch bieten kann. Zu diesem Zweck erstellt jeder Benutzer ein Profil seiner Interessenlage, die er im Netzwerk verbreitet. Finden sich Partner mit ähnlich gelagerten Interessen, so gelangt der Suchende mit hoher Wahrscheinlichkeit an die gewünschten Informationen.

Ein weiteres Beispiel einer kollaborativen Verbindung ist eine Zimmervermittlung, in der sich Zimmersuchende und Vermieter treffen. Die Basis einer erfolgreichen Vermittlung sind detaillierte Angaben über die zur Verfügung stehenden, und die weniger spezifischen Angaben über die Zahl der benötigten Zimmer in den Profilen der Anbieter und Mieter.

2.3.1 E-Learning in Ad-hoc Netzen (E.L.A.N)

E-Learning in Ad-hoc Netzen (E.L.A.N) ist ein Teilprojekt des Schwerpunktprojekts SPP1140. Im Vordergrund des Projekts steht die Unterstützung von E-Learning Szenarien unter besonderer Berücksichtigung der Eigenschaften von Ad-hoc Netzen. Ziel ist die Entwicklung einer adaptiven Infrastruktur, die sich den dynamischen Veränderungen von Ad-hoc Netzen anpassen kann und stets die höchstmögliche Lernunterstützung bereitstellt. Dies soll insbesondere interessen- und wissensbezogen geschehen. Zu diesem Zweck werden Awareness-Profile im Netz verteilt, die sogenannte *Wissens-, Lerninteressens- und Dienstprofile* (WiLiDips). Stationen können ihr eigenes Profil mit dem anderer Stationen vergleichen und entscheiden, inwiefern eine andere Station „interessant“ ist und einen möglichen Kollaborationspartner darstellt.

2.3.2 Aufbau des E.L.A.N-Projekts

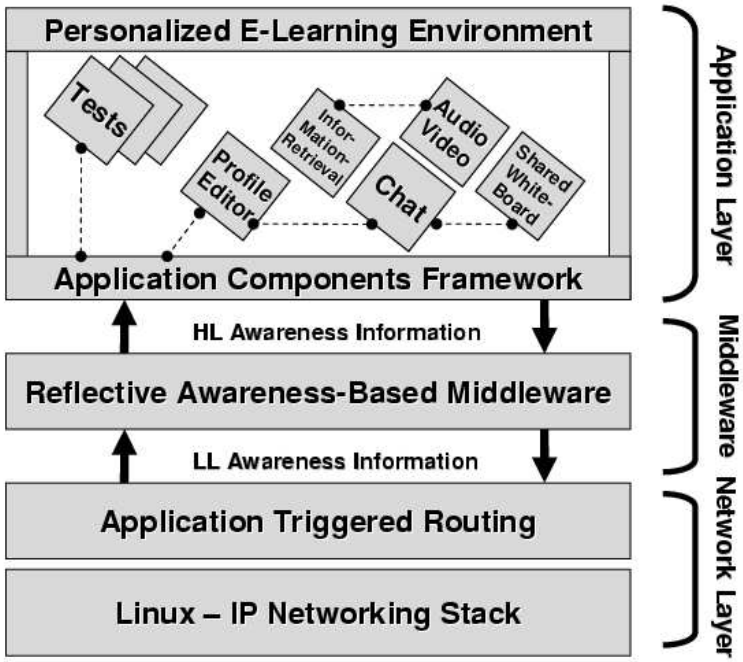

Abbildung 2.2: Die E.L.A.N Architektur

Abbildung 2.2 stellt die verschiedenen Schichten des E.L.A.N-Projekts stellt dar.

Die oberste Schicht ist der *Application Layer*, die Anwendungsschicht der E.L.A.N-Umgebung. Basierend auf einer komponentengestützten Plugin-Struktur laufen diverse Dienste und Anwendungen innerhalb eines grafischen Frameworks (*Application Components Framework*). Lernunterstützende Komponenten, wie ein Editor zum Bearbeiten der Lerninteressen, sind hier angesiedelt. Das Framework soll ein *Personalized E-Learning Environment* bilden, d.h. eine Umgebung, die sich dem Lernverhalten von Benutzern anpassen kann. Dazu soll das Erlernen eines Themas durch einen Benutzer so gefördert werden, wie dieser es bevorzugt, indem er verschiedene Lernmethoden wählen kann.

Die *Middleware* übernimmt die zentralen Aufgaben der Verwaltung von Awareness-Informationen. Dazu verteilt sie das Awareness-Profil des Benutzers im Netzwerk und sammelt Awareness-Profile von anderen Benutzern. Über einen Vergleich der Profile ermittelt sie interessante Kollaborationspartner und interessante Dienste. Sie gibt die gesammelten Informationen an die Anwendungsschicht als High-Level Awareness weiter. Außerdem kann sie von dieser konfiguriert und zum Versenden von Nachrichten an entfernte Stationen verwendet werden. Auf Veränderungen im Netzwerk soll sie dynamisch reagieren können, diese jedoch möglichst vor dem Framework verbergen (*Reflective Awareness-Based Middleware*).

Auf der Netzwerkschicht (*Network Layer*) arbeitet die Routingschicht auf der von Linux zur Verfügung gestellten TCP / IP-Netzwerkinfrastruktur. Die Routingschicht greift in die Routingta-

bellen des Kernels ein, um Netzwerk-Routen zu entfernten Stationen aufbauen zu können. Dabei kann sie teilweise durch andere Anwendungen gesteuert werden um z.B. awarnessbasiertes oder dienstspezifisches Routing, also *Application Triggered Routing* zu ermöglichen. Weiterhin hat sie die Aufgabe Informationen über ihre Umgebung in Form von Stations- und Nachbarschaftslisten an die höheren Schichten des Systems weiterzugeben (im Bild 2.2 mit LL für Low-Level Awareness bezeichnet).

E.L.A.N wurde parallel in mehreren Arbeiten entworfen. Für die Routingschicht entstand das Teilprojekt *arena*[6], das einen Routingalgorithmus in der Arbeit „Entwicklung von anwendungsorientierten, adaptiven Routing-Mechanismen für optimale Selbstorganisation in mobilen Ad-Hoc-Netzwerken“ (Felix Apitzsch) [Api03] implementiert. Die Implementierung der WiLiDips wurde in der Arbeit von Thorsten Wodarz „Entwicklung von editierbaren, korrelierbaren Interessensprofilen und deren Integration in ein Rahmenwerk für Kollaborative Anwendungen“ durchgeführt [Wod03]. Im Zuge dieser Arbeit wurde auch der Vergleichsalgorithmus zum Auffinden von Kollaborationspartnern sowie die WiLiDip-Editor-Komponente entwickelt. Das Framework lego und die Kommunikationsschnittstelle ECom wurden von Michael Lauer erarbeitet [LMD03, Lau04]. Weiterhin wurde SiMon, ein Simulator für Ad-hoc Netzwerke mit Anwendungsschnittstellen, in der Arbeit „Modellierung und Visualisierung von Mobilität und Gruppenbildung in Ad-hoc-Netzen“ von Nataliya Amirova entwickelt [Ami04].

2.4 Experimentierplattformen

2.4.1 Wireless LAN

Das als Wireless LAN (WLAN) oder Airport bezeichnete drahtlose Übertragungsverfahren basiert auf den IEEE 802.11-Standards für drahtlose Netze, die vom amerikanischen Institute of Electrical and Electronics Engineers (IEEE) festgelegt wurden.

Der heute am weitesten verbreitete Standard ist IEEE 802.11b, der Übertragungsraten von maximal 11 Mbps erreichen kann und damit, zumindest theoretisch, schneller als kabelgebundenes 10-MBit-Ethernet ist. Transferraten von 11 MBit/s (neuerdings auch 54 MBit/s im Standard 802.11g) sind jedoch lediglich Brutto-Datenraten. Die Rate für Nutzdaten liegt dagegen bei rund 7 MBit/s, die jedoch, je nach Verbindungsqualität, sich bis 1 Mbit verringern kann.

Der Standard 802.11b verwendet Frequenzen zwischen 2.4 und 2.485 Gigahertz, unterteilt in 11 Kanäle. Die verwendeten Frequenzen sind Teil des ISM-Bandes und können von jedermann benutzt werden, ohne um eine Lizenz ansuchen zu müssen.

Der Standard basiert auf zwei Betriebsarten. Es lässt sich wahlweise im Ad-hoc- oder im Infrastrukturmodus arbeiten.

Im Ad-hoc-Modus kommunizieren die Stationen direkt miteinander, im Infrastrukturmodus hingegen vermittelt eine Basisstation, ein Access Point, zwischen den Clients. Er dient zum einen als Bridge zum drahtgebundenen Netz, vermittelt also Pakete zwischen den Netzen hin und her, zum anderen arbeitet er als Repeater. D.h., er empfängt Pakete und leitet sie weiter.

Nachteile der WLAN-Technologie sind durch Frequenzüberschneidungen mit anderen Funktechniken, so z.B. Bluetooth. gegeben. Auch besteht keine garantierte Kompatibilität zwischen

[6] Application triggered Route Establishment / Network Awareness System

Geräten zweier verschiedener Anbieter, trotz des Industrielabels „Wi-Fi“ (Wireless Fidelity). Die relativ geringe Sicherheit von WLAN ist durch die unzureichenden Sicherheitsmechanismen bedingt und macht es für den kommerziellen Einsatz z.Zt., ohne zusätzliche Schutzmaßnahmen, ungeeignet.

Für das vorliegende Projekt wurden Lucent Orinoco Karten für die Laptops, Symbol Spectrum 24 Compact Flash Karten für die PDAs verwendet. Alle Karten arbeiteten nach dem IEEE 802.11b Standard.

2.4.2 Zielplattformen

E.L.A.N wurde insbesondere für den Einsatz auf mobilen Geräten (PDAs, Laptops, Webpads) entworfen. Da es insbesondere für das Betriebssystem Linux konzipiert ist, wurden die zu Testzwecken benutzen Compaq Ipaq PDAs mit diesem Betriebssystem versehen. Dazu mußten geeignete Distributionen auf den Flashspeichern der Geräte installiert werden. Ein PDA, auf dem die Tests hauptsächlich durchgeführt wurden, ist bereits ab Werk mit Linux ausgeliefert, der Sharp Zaurus SL-5500, .

Der SL-5500G (Abb. 2.3) hat einen 206 MHz Intel StrongARM Prozessor mit 64 MByte SDRAM. Sein TFT TouchScreen hat eine Auflösung von 240 x 320 Pixel bei 65536 Farben. Der PDA besitzt zwei Karteneinschübe. Einer für SD / Multimedia und einer für Compact Flash Karten. Es kann ein Micro Hard Drive und sogar eine Videokamera angeschlossen werden.

Der Zaurus eignet sich hervorragend für die Entwicklung von mobilen Anwendungen. Durch seine zwei Karteneinschübe kann gleichzeitig eine Funknetzwerkkarte und eine Speichererweiterung verwendet werden, auf der die nötigen Entwicklungstools installiert sind. Besonders hervorzuheben ist eine hinter der Frontblende verborgene Mini-Tastatur, durch die die Entwicklung von Anwendungen unterstützt wird, da bei Eingaben auf die teilweise umständlichen Eingabehilfen verzichtet werden kann.

Durch die Möglichkeit, sich mittels ssh oder telnet in den PDA einzuloggen und dort direkt Anwendungen starten zu können, wird die Entwicklung nachhaltig unterstützt. Ebenso können über das Netz Laufwerke angebunden werden. Somit ist das Ausführen von Code, der auf einem Desktop-Rechner entwickelt wurde, über das Netz möglich. Andernfalls müßte nach jedem Testschritt das Programm auf den Ziel-PDA durch Flashen oder durch Kopieren auf Einsteckkarten von neuem Übertragen werden.

2.4.3 Die Klassenbibliotheken Qt und Qt / Embedded

Qt ist eine der bekanntesten C++ Klassenbibliotheken zur Entwicklung plattformunabhängiger graphischer Benutzeroberflächen. Sie enthält mehr als hundert Klassen, darunter sowohl GUI-spezifische, als auch allgemeine Hilfsklassen, wie Strings und Hashtabellen (Dictionaries). Entwickelt wurde sie von der norwegischen Firma Troll Tech[7]. Qt ist sehr schnell, kompakt und gut dokumentiert. Sie basiert unter X-Windows direkt auf der Xlib, kommt also ohne Motif- oder Xt-Zwischenschicht aus. Für den nichtkommerziellen Einsatz ist sie frei erhältlich. Zur Entwicklung grafischer Oberflächen enthält sie mehrere unterstützende Tools, so z.B. den Qt

[7] http://www.troll.no/intro.html

Designer, mit dessen Hilfe konfortabel grafische Benutzerschnittstellen (GUI[8]) erstellt werden können. Mit dem Qt Linguist können diese Dialoge halbautomatisiert in verschiedene Sprachen übersetzt werden.

In den meisten GUI-Toolkits haben Widgets[9] einen Callback für jede Aktion, die sie auslösen können. Ein solcher Callback ist ein Zeiger auf eine Funktion, was leicht zu Programmierfehlern führen kann. In Qt wurden diese Funktionspointer durch sogenannte Signale und Slots ersetzt, die die Bindung zur Laufzeit flexibel und dynamisch durchführen.

Qt / Embedded ist das Pendant zu Qt für mobile Endgeräte. Qt wurde nicht nur für die verschiedenen Prozessortypen der Geräte portiert, sondern ist den limitierten Ressourcen der Geräte angepaßt. So stand bei weitgehender Kompatibilität zu Qt ein Optimum an Leistung bei einem Minimum an verwendeten Ressourcen im Vordergrund. Um Speicher zu Sparen ist Qt / Embedded nicht von X11 und der Xlib abhängig, sondern setzt direkt auf dem seit langem in Linux vorhandenen Framebuffer-Device auf. Es hat direkten Zugriff auf den Bildspeicher.

Eine auf Qt basierende Anwendung läßt sich im optimalen Fall durch Neuübersetzen auf die Zielplattform portieren. Dann müssen nur die Bibliotheken ausgetauscht und das Anwendungsfenster auf die verkleinerte Bildschirmauflösung umgestellt werden. Gegebenenfalls müssen noch die grafischen Bedieneinheiten neu erstellt oder umgearbeitet werden.

Qt / Embedded ist lediglich ein Toolkit für Applikationen. Für den Einsatz auf PDAs hat Trolltech deshalb Qtopia entwickelt, das auf dem Qt Palmtop Environment (QPE) basiert. Dieses ist, wie Qt / Embedded, ebenfalls unter der GPL Lizenz verfügbar. Es bietet eine Anwendungsumgebung ähnlich der grafischen Oberfläche KDE[10] unter X11. Um mehrere Anwendungen gleichzeit verwenden zu können, werden Mechanismen wie eine Start- und eine Taskleiste zur Verfügung gestellt. PDAs haben in der Regel keine Tastatur, deshalb sind in Qtopia unterschiedliche Eingabemethoden vorgesehen. Neben einer virtuellen Tastatur gibt es ein Pickboard, das eine Eingabe ähnlich der eines Mobiltelefons mit Schrifterkennung ermöglicht. Weiterhin sind in Qtopia PIM-Anwendungen[11], wie ein Adressbuch, ein Kalender, sowie eigene Synchronisationsfunktionen enthalten.

Abbildung 2.3: E.L.A.N auf dem Sharp Zaurus SL-5500

Eine Abspaltung von Qtopia bildet das OPIE-Projekt[12], das die in Qtopia enthaltenen Appli-

[8] Graphical User Interfaces
[9] Der Name Widget ist zusammengesetzt aus window (Fenster) und gadget (technisches Spielzeug).
[10] K Desktop Environment (`http://www.kde.org/`)
[11] Personal Information Manager
[12] Open Integrated Palmtop Environment (`http://opie.handhelds.org/`)

kationen und Bibliotheken unabhängig von Trolltech verbessert hat. Anwendungen, die unter OPIE entwickelt wurden, laufen auch unter Qtopia und umgekehrt.

2.4.4 Simulierung der Oberflächen über VNC und QVFB

Der QVFB ist ein virtueller Framebuffer für die X11-Oberfläche. Über diesen lassen sich Anwendungen, die für Qt / E entwickelt wurden, innerhalb eines Fensters auf der Desktop-Oberfläche starten. Dabei kann der virtuelle Framebuffer so konfiguriert werden, daß die Fenstergröße der des verwendeten PDAs entspricht. Die während der Entwicklung verwendete Umgebung bestand aus dem QVFB und einer binär für die Intel ix86 übersetzte OPIE-Version, die innerhalb des Framebuffers gestartet wurde. Innerhalb dieser wurde anschließen das E.L.A.N System evaluiert.

Eine ähnliche Methode zum Testen der Oberfläche wurde über VNC[13] zur Verfügung gestellt. Dabei wird der Bildschirminhalts eines Rechners auf einen anderen Rechner umgeleitet. Auf dem Gerät, dessen Bildschirm betrachtet werden soll, läuft ein VNC-Viewer. Auf jeder Plattform, auf der ein VNC-Client existiert, ist nun ein Darstellen des Bildschirms möglich. Tastatur- und Mauseingaben werden zum VNC-Server weitergeleitet und ermöglichen das Verwenden der dort laufenden Programme.

2.4.5 Die Programmiersprache Python

E.L.A.N wurde in der Programmiersprache Python realisiert. Python ist eine interpretierte Hochsprache (Skriptsprache) mit ausgebildeten Basisdatentypen wie Mengen und assoziativen Feldern (Dictionaries). Sie ist objektorientiert, allerdings ohne strenge Typisierung. Somit ist es leicht möglich, größere Prototypen zu entwerfen ohne auf die Einschränkungen anderer Hochsprachen Rücksicht nehmen zu müssen. Durch die Eigenschaft, Programmblöcke durch Einrückungen unterscheiden zu können, wird der Programmtext übersichtlich und der Wartungsaufwand reduziert.

Durch die Erweiterbarkeit mit Modulen wird die Programmiersprache zu einem Allzweckwerkzeug. Eine reichhaltige Modulbibliotek ist in der Distribution enthalten und stellt von einfachen Komprimierwerkzeugen bis zu Webservern eine Fülle von Anwendungen zur Verfügung.

Es ist außerdem möglich, binäre Module einzubinden. Diese werden durch den Wrapper SIP[14] angebunden.

Ein weiterer Vorteil der Sprache ist ihr Interpreter-Charakter. Es ist möglich, das Programm zur Laufzeit anzuhalten und Werte durch sogenannte Introspektion auszulesen und zu verändern. Während E.L.A.N läuft, existiert eine optionale Interpreter-Komponente, die es ermöglicht, direkt im laufenden Programm Daten zu verändern und Nachrichten zu verschicken.

Erkauft werden die genannten Vorteile leider durch eine langsamere Ausführungsgeschwindigkeit. Es gibt keine Möglichkeit, Python-Programme zu kompilieren, um die Geschwindigkeit zu erhöhen. Allerdings haben andere Hochsprachen das Problem, daß sie binäre Programme auf verteilten Systemen kapseln müssen. Aus diesem Grund müssen für entfernte Aufrufe spezielle

[13] Virtual Network Computing, `http://www.realvnc.com/`

[14] A sip of swig (Simplified Wrapper Interface Generator), `http://www.riverbankcomputing.co.uk/sip/`

Interfaces gebildet werden, die über virtuelle Funktionstabellen die echten Adressen herausfinden müssen. Mit Python dagegen ist es möglich, einfach den Programmcode zu einer entfernten Maschine zu übertragen und dort dem Interpreter zu übergeben. Auf diese Weise läßt sich z.B. die Migration von Diensten realisieren.

Python ist für mehrere Plattformen völlig frei verfügbar, es enthält keinen Code aus dem GNU-Projekt und kann somit in kommerziellen Anwendungen problemlos eingesetzt werden.

3 Konzeption der Verteilungsplattform

Dieses Kapitel beschreibt den konzeptionellen Aufbau der Komponenten der Verteilungsplattform. Es wird zuerst auf die benötigten Kommunikationsschichten und -wege eingegangen. Anschließend werden die Verteilungsinhalte, die zwischen den Schichten ausgetauscht werden, dargelegt.

Analog zu der in Kapitel 2.3 vorgestellten Architektur des E.L.A.N-Systems wird im Anschluß die Konzeption einer Middleware für verteilte Anwendungen in Ad-hoc-Netzen dargelegt. Es mußte eine Kapselung der dort vorhandenen Routingschicht realisiert werden. Dies geschieht in Form des LanRouters, dessen Aufbau das Kapitel abschließt.

3.1 Kommunikationsschichten

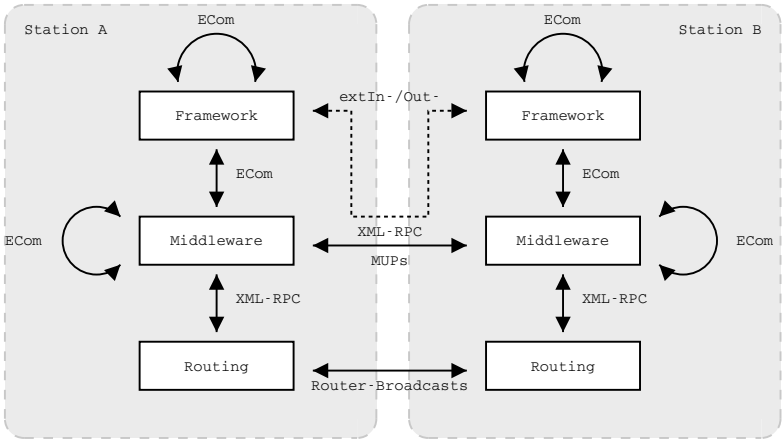

Abbildung 3.1: Der E.L.A.N-Kommunikationsfluß zwischen den verschiedenen Komponentenschichten

Abbildung 3.1 zeigt den zu verarbeitenden Informationsfluß zwischen den verschiedenen Schichten innerhalb einer Station und zwischen zwei E.L.A.N-Instanzen innerhalb eines Netzes.

Innerhalb einer Station wird zwischen der Routingschicht und der Middleware über XML-RPC[1] kommuniziert. Auf Middleware- und Framework-Ebene wird die ECom-Schnittstelle von lego verwendet, die auf dem Versand asynchroner Nachrichten ausgelegt ist.

[1] eXtensible Markup Language-Remote Procedure Call

Auf der untersten Schicht kommunizieren die Router aller Stationen über UDP-Broadcasts. Sowohl die von arena als auch die vom LanRouter versendeten UDP-Datagramme haben eine konstante Größe von wenigen Bytes. Durch die Maximalgröße von 65535 Bytes eines Datagramms bleibt innerhalb eines versendeten Pakets genügend Platz für die sogenannten *Cargo-Daten* („Nutzlast-Daten“, cargo-data) verwendet. Um die Latenzzeit des gesamten Systems zu verringern werden die Cargo-Daten zum Transport eines *Global Unique Identifiers* (GuID) verwendet, der dem System das genaue Identifizieren einer Station erlaubt.

Die Middleware verteilt die Awareness-Informationen in Form von WiLiDips. Sie verwendet dazu sogenannte *Middleware UDP-Pakete* (MUPs), die sie über UDP-Broadcasts im Netz verteilt. Die MUPs können ihren Inhalt komprimieren und stellen somit sicher, daß beim Transport eines WiLiDips sowohl Bandbreite gespart, als auch die maximale Größe eines UDP-Datagramms nicht überschritten wird. Für die direkte Kommunikation mit einer entfernten Station verwendet die Middleware XML-RPC-Aufrufe. Da diese TCP / IP-basiert sind, muß sie vor dem Aufruf die Routingschicht veranlassen, eine Hin- und eine Rückroute zu dieser Station aufzubauen.

Die in der Anwendungsschicht laufenden Komponenten benötigen eine Möglichkeit, mit anderen Komponenten kommunizieren zu können, um auf diese Weise auf Dienste anderer Stationen zuzugreifen. Eine solche Kommunikationsschnittstelle stellt das Framework als sogenannte extIn- / extOut-Kommunikation zur Verfügung. Die einzelnen Nachrichten werden über die Middleware verschickt, die den entfernten Aufruf dann über XML-RPC ausführt.

Neben der eigentlichen Verteilungsplattform, der Middleware, mußten die Protokolle zur Kommunikation zwischen den einzelnen Schichten sowohl horizontal als auch vertikal berücksichtigt und zum Teil erst entwickelt werden.

3.2 Verteilungsinhalte

3.2.1 Verteilung von Awareness-Information

Die im E.L.A.N-System verteilten Awareness-Informationen beziehen sich auf das Wissen über „interessante“ Kollaborationspartner und nutzbare Dienste. Diese Informationen werden in WiLiDips gespeichert, den Wissens-, Lerninteressens- und Dienstprofilen.

Das Ergebnis des Vergleichs zweier WiLiDips, nämlich des eigenen und des eines anderen Mitglieds des Ad-hoc-Netzes, ist der Schlüssel zum Finden von Kollaborationspartnern. Der Vergleich wird über einen vorgegeben Korrelationsalgorythmus durchgeführt [Wod03] und gibt einen normaliserten Wert zurück.

Wissens-, Lerninteressens- und Dienstprofile (WiLiDips)

Ein Benutzer des Systems kann über den Profil-Editor `wilied`, der in Abb. 3.2 dargestellt ist, sein persönliches WiLiDip editieren.

Dazu gibt er in Prozentwerten an, in welchem Wissenbereich er über Kenntnisse verfügt und für welche Wissensbereiche er sich interessiert. In der Abbildung 3.2 ist zu sehen, wie ein Benutzer ein Lerninteresse von 40% im Bereich Geologie einstellt.

Die Wissensbereiche sind hierarchisch in Kategorien strukturiert. Es können ganze Teilbäume oder nur Unterpunkte markiert werden. So kann ein Benutzer beschreiben, daß er z.B. umfassendes Wissen innerhalb der Kategorie Biologie im Bereich der Humangenetik besitzt, jedoch im Unterbereich Immungenetik noch starkes Interesse an Fortbildung besteht.

Die Kategorien sind zur Zeit noch statisch vorgegeben. Im vorliegenden Fall sind es Teilgebiete des universitären Alltags. Es ist jedoch jederzeit möglich die Wissensgebiete um eigene Konzepte nachträglich zu erweitern.

Über den Profil-Editor ist es weiterhin möglich, bestimmte Dienste an das WiLiDip zu binden.

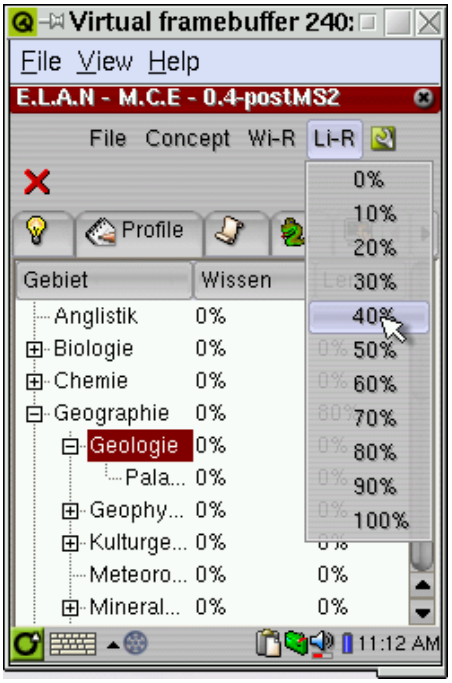

Abbildung 3.2: WiLiDip-Editor-Komponente (wilied)

Adressierung über GuIDs

Die GuID bezeichnet einen globalen und eindeutigen Identifier für einen E.L.A.N-Teilnehmer bzw. dessen WiLiDip. Zu jedem Benutzer gibt es genau ein WiLiDip, das verteilt werden muß. Dieses kann jedoch vom Benutzer jederzeit verändert werden und somit verschiedene Versionen im Netz vorhanden sein. Da nur das aktuelle WiLiDip eines Benutzers interessant sein kann, muß eine Versionskontrolle erfolgen. Ältere WiLiDips müssen von neueren ersetzt werden. Dazu enthält die GuID eine Sequenznummer, realisiert als Zeitstempel. Zusätzlicher Bestandteil der GuID ist ein sogenannter Spitzname, ein „Nick“, der dazu da ist, die GuID lesbarer zu gestalten. Es ermöglicht dem Benutzer zu sehen, ob ein Teilnehmer bekannt ist.

Middleware UDP-Pakete: MUPs

Um Awareness-Informationen zu versenden mußte eine Transportbeschreibung entworfen werden, die Middleware UDP-Pakete (MUPs). Bei ihrer Konzeption standen mehrere Punkte im Vordergrund: Sie sollten neben den Nutzdaten eine eindeutige Nachrichten-Identifikation (Message-ID) enthalten, um zuordnen zu können, ob das Paket bereits weitergeleitet oder verarbeitet wurde. Pakete, die als Antwort auf ein anderes Paket versendet werden, sollen diese ursprüngliche Message-ID (Referenced Message-ID) ebenfalls beinhalten.

Für die Verarbeitung des Pakets ist eine Identifikation des Pakettyps in Form einer Funktionsbeschreibung (Function-ID) hilfreich, um ohne genauere Analyse des Paketinhalts dieses den jeweils zuständigen Programmteilen übergeben zu können. Der Zugriff auf den Transportinhalt und das Erzeugen eines MUPs sollte über eine einfach zu verwendende Schnittstelle ermöglicht werden. Durch Komprimierung des Transportinhalts sollte die Größe des Paketes reduziert werden, um in ein einziges UDP-Datagramm zu passen. Schließlich sollten die Schnittstelle zu den MUPs erweiterbar sein, damit in zukünftigen Versionen der Middleware weitere Pakettypen hinzugefügt werden können.

Da die Pakete asynchron verschickt werden, gibt es für jeden Pakettyp einen Antwort-Pakettyp. Für den aktuellen Betrieb der Middleware reichen 2 Pakettypen und das jeweilige Antwortpaket aus.

Ein *Ping* ist ein Paket, das zu Testzwecken gebroadcastet wird. Jede Station die es empfängt, versendet ein *Pong* als Antwort. Weder Ping noch Pong transportieren Nutzdaten. Über die Analyse der Paketheader ist jedoch zu sehen, welche Stationen im Netz vorhanden sind und wie weit diese Stationen entfernt sind. Für zukünftige Testzwecke ist daran gedacht das Pong-Paket um Infformationen über den zurückgelegten Weg zu erweitern. *WilidipRequest* und *WilidipTransport* verhalten sich prinzipiell genauso wie Ping und Pong. Das WilidipTransport-Paket beinhaltet jedoch noch das WiLiDip der entfernten Station. Weiterhin wird das WilidipTransport-Paket zur normalen Verteilung verwendet. Eine Unterscheidung zwischen der Antwort auf einen Request und dem selbstständigen propagieren erschien nicht sinnvoll, da der Paketinhalt in beiden Fällen gleich ist.

Anfordern oder Propagieren

Während der Konzeption stellte sich die Frage, wie die WiLiDips verteilt werden sollten. Dies könnte auf verschiedene Arten erfolgen. Entweder wird, wenn eine Station zum Netz hinzukommt das WiLiDip explizit angefordert oder es wird gewartet, bis die Station es von sich aus durch das Netz flutet.

Ein optimaler Verteilungsalgorithmus würde ein Profil nur an diejenigen Stationen weiterleiten, die auch daran interessiert sind. Dies würde jedoch bedingen, daß die WiLiDips verändert werden dürften. Nach einer Veränderung eines WiLiDips würde der Vergleich ungültig werden.

Voraussetzung ist also, daß jede Station das aktuelle WiLiDip jeder anderen Station einmal gesehen haben muß, was ein komplettes Fluten des Netzes nötig machte.

Die Entscheidung fiel auf eine Kompromißlösung: das eigene WiLiDip wird nach dem Start und nach jeder Veränderung aktiv gebroadcastet. Alle anderen Stationen empfangen dieses passiv. Stationen, die später zum Netz hinzukommen, können fehlende WiLiDips anfordern (welche das sind erfahren sie über den Router). Die Beantwortung der Anfrage wird nicht notwendigerweise von der Station durchgeführt, von der das WiLiDip ursprünglich stammte. Jede Station in Reichweite, die das WiLiDip bereits hat, darf antworten. Auf diese Weise soll ein erneutes Fluten des Netzes vermieden werden.

Damit nicht mehrere Stationen auf eine Anfrage gleichzeitig reagieren, wird die Beantwortung um eine zufällige Zeitspanne verzögert. Dadurch, daß eine Station geantwortet hat erkennen mögliche weitere Beantworter, anhand der Referenced Message-ID, daß sie dieses Paket nicht noch einmal senden müssen.

3.2.2 Verteilung von Dienstinformationen

Um Komponenten das Nutzen von entfernten Diensten zu ermöglichen, ist es notwendig, daß Informationen über vorhandene Dienste verteilt werden. E.L.A.N unterscheidet dabei zwischen verschiedenen Arten von Diensten, die auch verschieden behandelt werden.

Dienste sind Programme, die auf Stationen im Netz ablaufen und die anderen Teilnehmern des Netzes Ressourcen zur Verfügung stellen, die diese nutzen können. Unter *Dienstgüte* (Quality

of Service) versteht man die Beeinflussung des Dienstes durch äußere Faktoren. So hat z.B. ein Dienst, der wenige Stationen entfernt abläuft eine höhere Dienstgüte als ein weiter entfernter Dienst, da die Nutzungsgeschwindigkeit durch die Entfernung beeinflußt wird. Ebenso fließt die Rechenleistung einer Station, die den Dienst anbietet, in die Dienstgüte ein.

Dienste innerhalb von E.L.A.N werden bezüglich ihres Inhalts in verschiedene Typen, nämlich in *unspezifizierte* und *spezifizierte* Dienste eingeteilt. Unspezifizierte Dienste sind darüber hinaus unterteilt in *implizit-unspezifizierte* und *explizit-unspezifizierte* Dienste.

Unspezifizierte Dienste sind Dienste, die mehrmals vorhanden sein können, wie z.B. ein allgemeiner Druckdienst oder ein Konverter. Ihr besonderes Merkmal ist die automatische Verteilung ihrer Information im Netz, die nur davon abhängt, ob ein Dienst läuft oder nicht.

Sollten mehrere *Unspezifiziert-Explizite* Dienste im Netz vorhanden sein, bieten diese zwar jeweils die gleiche Funktionalität, sind aber auf ihre spezielle Maschine beschränkt. Darunter fallen u.a. Nachrichtendienste. Diese haben die Aufgabe, eine eingehende Nachricht eines entfernten Teilnehmers dem Benutzer anzuzeigen und ihm die Möglichkeit zu geben, auf diese Nachricht zu antworten. Ein prototypischer Nachrichtendienst wird mit dem `messageService` in 5.6.2 vorgestellt.

Unspezifiziert-Implizite Dienste sind austauschbare Dienste: Ein Konverter, der z.B. PostScript in reinen Text umwandelt, kann auf mehreren Maschinen laufen. Wichtig ist allein die Existenz eines Konverters in Reichweite. Die Station, die einen interessanten Dienst anbietet, muß in diesem Fall kein möglicher Kollaborationspartner sein. Die Middleware speichert zu einem unspezifizert-impliziten Dienst diejenigen Stationen, die ihn zur Verfügung stellen. Soll ein Dienst in Anspruch genommen werden, wählt sie eine der möglichen Stationen aus. Welche das ist, wird durch den ServiceManager bestimmt. Dieser entscheidet z.B. anhand der Dienstgüte der verfügbaren Dienste, welcher Dienst verwendet werden soll. Ebenso kann er versuchen, die Last gleichmäßig über die dienstanbietenden Stationen zu verteilen, um nicht die Ressourcen einer Station übermäßig zu belasten.

Die Benutzung unspezifiziert-impliziter Dienste erfolgt ohne weiteres Eingreifen des Benutzers. Sie wird allein durch die Middleware gesteuert.

Ein Beispiel einer prototypischen Nutzung eines unspezifiziert-impliziten Dienstes wird mit der `converter-` und der `TextViewer`-Komponente in Kapitel 5.6.3 vorgestellt.

Spezifizierte Dienste sind Dienste, die von einem bestimmten Mitglied des Netzes entweder explizit angeboten werden oder deren Inhalt von Anbieter zu Anbieter variiert. Letzte können nicht als redundant betrachtet werden. Das System muß Kontakt zu einer bestimmten Maschine aufnehmen, um den Dienst nutzen zu können. Spezifizierte Dienste sind weiterhin dadurch definiert, daß sie nicht automatisch verteilt werden, sondern explizit vom Benutzer an das Dienstprofil angehängt werden.

3.2.3 Verteilung von Diensten

Die Dienste werden nicht separat verteilt, sondern an ein WiLiDip angehängt. Nach dem Empfang eines WiLiDips werden sie daraus extrahiert.

Es wird anhand des Diensttyps entschieden, *wer* den Dienst an das WiLiDip anhängt. Spezifizerte Dienste werden, wie schon erwähnt, von den Benutzern an das ganze oder auch nur

an Unterpunkte des Wissensprofils gebunden. Dies ermöglicht die Zuordnung von Diensten zu Wissensbereichen.

Unspezifizierte Dienste werden von der Middleware angehängt - je nach dem, ob der Dienst auf der aktuellen Maschine läuft oder nicht. Dabei erkennt die Middleware, ob ein Dienst nach bereits erfolgtem Verteilen des Wilidips gestartet oder beendet wurde und erzwingt eine Neuverteilung.

3.2.4 Verteilung von Nachrichten

Applikationen aus dem Framework müssen in der Lage sein, untereinander Nachrichten auszutauschen. Es muß eine asynchrone Nachrichtenübermittlung implementiert werden, die dies den Framework-Komponenten ermöglicht.

Eine Komponente, die einer entfernten Komponente Nachrichten schicken möchte, könnte dies selbstständig z.B. durch Aufbauen einer Socket-Verbindung oder durch einen XML-RPC-Aufruf durchführen. Dieses Vorgehen hätte jedoch den Nachteil, daß im Ad-hoc-Netzwerk Hin- und Rückroute zu der entfernten Station existiert müßte. Weiterhin würde für jede Komponente ein Kommunikationsprotokoll sowie ein Empfänger notwendig werden, sowie das Wissen un den verwendeten Empfangs-Port.

Um diese Probleme zu umgehen, wird die Kommunikation gekapselt und der Middleware überlassen. Die sogenannte extIn- / extOut-Kommunikation läuft über die XML-RPC-Schnittstelle der Middleware.

3.3 Middleware mit integriertem Awareness-System (midas)

Die Middleware – einer der Schwerpunkte der vorliegenden Diplomarbeit – verteilt die Awareness-Informationen, also die WiLiDips, eingebettet in MUPs, an alle Rechner im Ad-hoc Netz. Ebenso empfängt sie fremde WiLiDips. Diese werden über einen Korrelationsalgorithmus verglichen und anhand der Ergebnisse eine Liste von Stationen bzw. WiLiDips erstellt und verwaltet, die im Sinne des Anwenders „interessant“ sind.

Durch die Verknüpfung der Dienstbeschreibungen mit den WiLiDips wird auch eine Liste von interessanten Diensten geführt. Diese umfaßt auch alle unspezifiziert-impliziten Dienste im Netz.

Weiterhin beobachtet die Middleware die Ausführungsumgebung im Sinne der Context-Awareness, um auf Veränderungen spontan reagieren zu können. Dies betrifft verfügbare Batterie-Kapazitäten und die Qualität der Funkverbindung.

Die gesammelten Informationen werden der Anwendungsschicht zur Verfügung gestellt, insbesondere um dem Benutzer eine Übersicht über die Angebote im Netz zu geben.

Sie übernimmt auch die Kommunikation der eigenen Anwendungsschicht mit Anwendungsschichten entfernter Komponenten.

3.3.1 Aufbau der Middleware

Die Middleware ist in mehrere Einzelkomponenten unterteilt, um alle beschriebenen Anforderungen zu erfüllen. Abbildung 3.3 zeigt die Darstellung ihres Aufbaus. Sie besteht aus mehre-

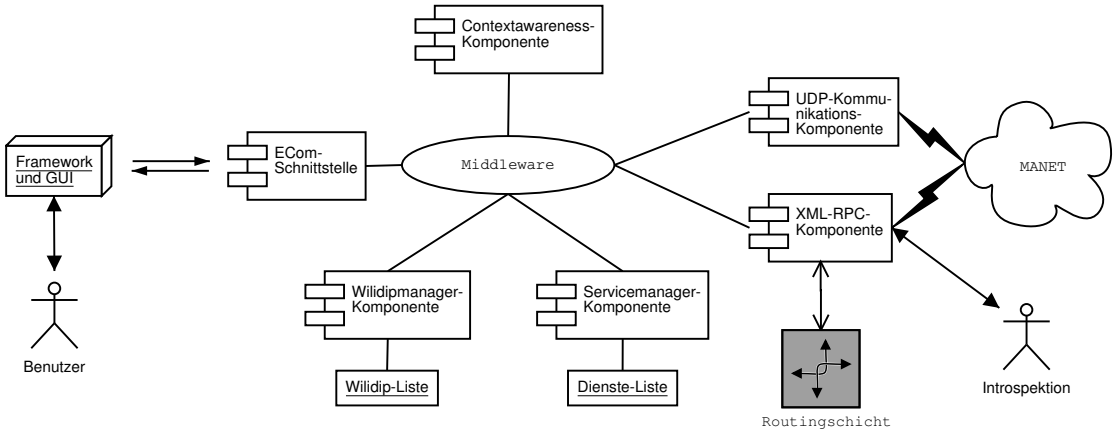

Abbildung 3.3: Aufbau von midas

ren Kommunikationskomponenten, die die in Kapitel 3.1 vorgestellten Kommunikationswege zwischen den verschiedenen Schichten umsetzen. Weiterhin beinhaltet sie die für den Betrieb innerhalb eines Ad-hoc Netzes benötigte Awareness-Komponenten.

3.3.2 Kommunikationskomponenten

Durch die Anforderungen an die Kommunikation in alle Richtungen besitzt die Middleware drei Kommunikationskomponenten. Die UDP-Kommunikations-Komponente, die sogenannte *Protokollmaschine* übernimmt die Verteilung von WiLiDips innerhalb des Netzes. Externe Kommunikation wird durch die XML-RPC-Schnittstelle, dem *XMLRPCCommunicator* durchgeführt. Dieser übernimmt auch die Kommunikation mit der Routingschicht. Zwischen Framework und Middleware wird die ECom-Schnittstelle zum Versand asynchroner Nachrichten verwendet.

Protokollmaschine

Die Protokollmaschine übernimmt die grundsätzliche Verteilung von WiLiDips im Netz. Dazu broadcastet sie das eigene WiLiDip jeweils nach dem Systemstart und nach einer erfolgten Veränderung an alle Stationen in Empfangsreichweite.

Weiterhin wartet ein UDP-Server auf eingehende MUPs. Er beantwortet selbstständig Anfragepakete, indem er das zugehörige Antwortpaket erstellt und verändert. Enthält das eingegangene Paket ein WiLiDip, so wird es dem Wilidipmanager zur Weiterverarbeitung übergeben.

Die Protokollmaschine ist auch für das Routen von Paketen zuständig. Damit ein Netz über einen Broadcast geflutet werden kann, müssen die Pakete von allen Stationen weitergeleitet werden. Anhand der Message-ID des Pakets ermittelt, ob es bereits weitergeleitet wurde. Ein mehrfaches Versenden wird somit vermieden.

ECom-Schnittstelle

Da Teile der Middleware und des Frameworks nebenläufig in verschieden Threads ablaufen, mußte ein Kommunikationsmodell entworfen werden, das auf dem Versand asynchronener Nachrichten basiert. Um dieses in allen Teilen des Systems einheitlich zu halten wurde die ECom-Schnittstelle entworfen.

Die ECom-Schnittstelle ist allen Komponenten von E.L.A.N zugänglich. Sie dient dem Framework dazu, mit seinen Komponenten zu kommunizieren. Ebenso wird sie von verschiedenen Teilen der Middleware verwendet, um untereinander Nachrichten auszutauschen.

XMLRPCCommunicator

Der XMLRPCCommunicator ist hauptsächlich für die Kommunikation zwischen der Routingschicht und der Middleware-Schicht verantwortlich. Für die Kommunikation mit der Routingschicht wurde XML-RPC gewählt, da diese über das lokale Loopback-Device erfolgt. Somit ist sie unabhängig von den Verbindungsproblemen, die innerhalb eines Ad-hoc Netzes auftreten können. XML-RPC ist ein offener Standard, der einfach zu implementieren ist und der für verschiedene Programmiersprachen zur Verfügung steht. Während der Konzeptionsphase war nicht festgelegt, ob die Routingschicht arena ebenfalls in Python implementiert werden würde und somit hat sich ein sprachübergreifendes Protokoll angeboten.

Als Erweiterung wird XML-RPC auch verwendet, um mit entfernten Stationen kommunizieren zu können, trotz des Wissens, daß eine TCP / IP-basierende Kommunikation im Ad-hoc-Netzwerk nicht optimal ist. Die direkte XML-RPC-Kommunikation zwischen einzelnen E.L.A.N-Stationen soll zukünftig über ein neues, UDP-basiertes XML-RPC-Protokoll durchgeführt werden.

Weiterhin wird eine wichtige Introspektions-Methode zur Verfügung gestellt. Von E.L.A.N unabhängige Programme, die nicht auf der lokalen Maschine laufen müssen, können durch simple XML-RPC-Aufrufe den internen Zustand einer E.L.A.N Instanz analysieren und verändern. Diese Schnittstelle wurde zu Evaluationszwecken und zur Anbindung von Simulationsprogrammen eingefügt.

3.3.3 Verwaltungskomponenten

Die Verwaltung der WiLiDips und der Dienste verlaufen getrennt. Daher gibt es einschließlich der Context-Awareness, auch hier drei Verwaltungskomponenten.

Context-Awareness Informationen werden innerhalb des Frameworks, der Middleware-Komponente und der Routingkomponente ausgetauscht. Dazu beobachtet die Middleware neben ihren weiteren Tätigkeiten den Batteriezustand des Geräts über das Powermanagement ACPI bzw. APM und die Qualitiät der Funkverbindung über die Linux-Wireless-Extensions.

Die Verwaltung der User-Awareness-Informationen übernimmt die *Wilidipmanager*-Komponente. Sie speichert Stationen über ihre GuID bzw. IP-Adresse und ordnet dieser die entsprechenden WiLiDips und zusätzlichen Informationen über die Stationen zu. Fehlende Informationen werden erkannt und entsprechend über die Kommunikationskomponenten angefordert.

Die Verwaltung der unspezifiziert-impliziten Dienste wird von der *Servicemanager*-Komponente übernommen. Diese speichert zu einer Dienstbeschreibung diejenigen Stationen, die den

Dienst anbieten. Wird ein Dienst von der Middleware angefordert, kann sie entscheiden, welche Station für die Aufgabe am geeignesten ist.

Die expliziten Dienste sind an WiLiDips gebunden und benötigen keine weitere Verwaltungseinheit. Ist ein WiLiDip interessant sind auch die zugehörigen Dienste interessant. Es muß also nur bei einer Veränderung der Interesseslage dem Framework mitgeteilt werden, welche Dienste dadurch neuerdings interessant oder uninteressant werden.

3.4 Routing-Schicht

3.4.1 Aufgaben des Routings

Die Routingschicht übernimmt das Setzten und Entfernen von Routen zu Stationen im Ad-hoc-Netzwerk. Dazu ermittelt sie zunächst, welche Stationen überhaupt vorhanden sind. Diese Informationenen sind der Middleware zugänglich zu machen. Ebenso ist ihr eine Schnittstelle zur Verfügung zu stellen, die anwendungsbasiertes Routing ermöglicht.

Um die Netzwerkbelastung zu optimieren, sollen den sowieso verwendeten Router-Paketen jeweils noch die Cargo-Daten in Form der GuID mitgegeben werden. Dieses wird von der Middleware übergeben.

3.4.2 Konzeption der Routingschicht

Es existieren grundsätzlich zwei Möglichkeiten, eine Routingschicht zu implementieren. Die eine ist *Peer-To-Peer Routing* direkt zwischen den Anwendungen, die andere ist *kernelnahes Routing*, das direkt auf die Routing-Tabellen im Betriebssystemkern zugreift.

Peer-To-Peer Routing wird in gängigen Filesharing-Anwendungen, wie Gnutella und giFT[2] eingesetzt [Gnu01, giF01]. Ihr Vorteil ist, daß die Anwendungen überall, ohne Einrichtungsaufwand für Routing und ohne spezielle Rechte zum Setzen der Routen funktionieren. Allerdings funktionert das Weiterleiten von Paketen dann nur innerhalb der Anwendung, die dieses Routing einsetzt und nur mit dem Protokoll, das die Anwendung zum Versand ihrer Pakete verwendete. TCP / IP oder UDP basierende Protokolle können nur dann verwendet werden, wenn die Anwendung in der Lage ist, diese innerhalb ihres Protokollstroms zu tunneln.

Bei kernelnahem Routing werden direkt die Routing-Tabellen im Kern verändert. Der dazu benötigte Zugriff benötigt allerdings administrative Rechte. Ein direkter Eingriff in diese Routing-Tabellen betrifft den gesamten IP-Stack der Maschine. Somit wird das Weiterverwenden sämtlicher IP basierter Festnetz-Protokolle (wie z.B. XML-RPC) und Anwendungen ermöglicht.

Gewählt wurde schließlich das kernelnahe Routing über ein eigenständiges Programm, das mit administrativen Rechten läuft. Dadurch, daß dieses Programm nicht in das restliche System integriert ist, ist es möglich E.L.A.N mit den eingeschränkten Rechten eines normalen Benutzers zu betreiben.

Weiterhin kann es sinnvoll sein, daß Stationen existieren, die nur das Routing beherrschen aber keine Lernumgebung laufen lassen. Auf diese Weise kann die Reichweite der Ad-hoc Wolke durch Teilnehmer, die nur die Routing-Anwendung laufen lassen, vergrößert werden.

[2] eine der existierenden Bedeutungen der Abkürzung ist das rekursive Akronym: giFT is not FastTrack

Das Projekt arena aus der Arbeit „Entwicklung von anwendungsorientierten, adaptiven Routing-Mechanismen für optimale Selbstorganisation in mobilen Ad-Hoc-Netzwerken“ [Api03] befaßt sich mit der Implementierung einer solchen Routingschicht für das E.L.A.N Projekt. Basierend auf dem TORA-Algorithmus[3] [PC97] wurde ein applikationsgesteuerter Routing-Algorithmus entworfen. Die Schwerpunkte des Projekts liegen auf geringem Kommunikationsoverhead und Minimierung von Anpassungen nach Topologieänderungen des Netzwerkes.

Die Entwicklung von arena und der Verteilungsplattform fand zeitgleich statt. Daher stand die Ad-hoc-Routingschicht der Middleware zu Projektstart noch nicht zur Verfügung. Da sie grundlegend für das Funktionieren des Gesamtsystems ist, mußte ein prototypisches Programm konzipiert werden, das sich nach Außen wie die eigentliche Routingschicht verhält. Das Programm ist in Form des LanRouters realisiert worden.

3.4.3 LanRouter

Der LanRouter dient als Kapselung der arena-Funktionalität in einem Festnetz und zu Evaluierungszwecken der Schnittstelle zwischen beiden Schichten.

Der LanRouter trägt die gleiche XML-RPC-Schnittstelle zur Middleware wie sie arena implementiert. Ebenso verteilt er die Cargo-Daten und übermittelt der Middleware die Liste der aktuell im Netz verfügbaren Stationen. Damit die Stationslisten verwaltet werden können sendet jeder Router in regelmäßigen Intervallen einen Broadcast aus, der andere Stationen über das Vorhandensein der eigenen Station informiert. Dieser Broadcast enthält auch die Cargo-Daten.

Im Unterschied zu arena wird nicht auf die Routing-Tabellen des Kerns zugegriffen. Daher funktioniert E.L.A.N bei Verwendung des LanRouters nur in Festnetzen. Dies war für die Entwicklung, die im Festnetz stattfand, kein Nachteil. Die PDAs, die zur Evaluierung des Verhaltens der Middleware und der Benutzerschnittstelle verwendet wurden, waren über WLAN im Infrastruktur-Modus an das Festnetz angebunden. Das Hinzukommen oder Wegfallen aus dem Ad-hoc Netz wurde durch Starten und Beenden der Anwendung simuliert. Durch die Verwendung des Festnetzes stand eine größere Zahl möglicher Teilnehmer zur Verfügung, als mobile Geräte vorhanden gewesen wären.

Komponenten des Lanrouters

Der LanRouter ist wie die Middleware in einzelne Komponenten unterteilt, wie Abbildung 3.4 zeigt.

Über den *UDP-Broadcaster* werden in regelmäßigen Intervallen LanRouter-Pakete ins Netz gebroadcasted. Über den *UDP-Listener* werden solche Pakete empfangen. Die Broadcasts enthalten eine Kennung, die sie als LanRouter-Pakete identifizieren. Weiterhin enthalten sie eine Message-ID, analog zu den in 3.2.1 vorgestellten MUPs. Eingegangene Pakete werden weitergeleitet. Auf diese Weise können mehrere Subnets in größeren Netzwerken verbunden werden. Direkte Nachbarn werden durch die Anzahl der zurückgelegten Hops, die in den Paketen enthalten ist, festgestellt.

Empfange Pakete werden ausgewertet und anhand ihres Inhalts eine Liste von Knoten im Netz verwaltet. Aktualisierungen dieser Liste werden der Middleware über XML-RPC mitgeteilt.

[3] Temporally Ordered Routing Algorithm (Corson/Park, 1997)

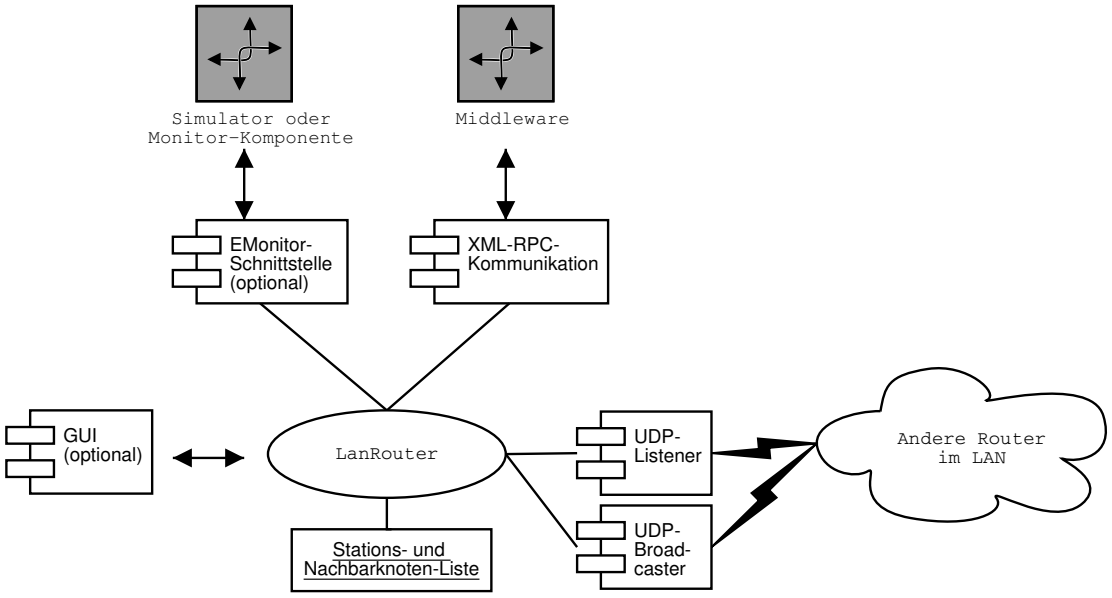

Abbildung 3.4: Aufbau des LanRouters

Analog zum XMLRPCCommunicator der Middleware verfügt der LanRouter daher über eine XML-RPC-Kommunikationskomponente, die einen Server und einen Client implementiert.

Weiterhin enthält der LanRouter ein optionales grafisches Benutzerinterface (GUI). Dieses diente während der Entwicklung zur übersichtlichen Darstellung gefundener Stationen und des internen Zustands. Im Normalfall wird der Router jedoch als Daemon im Hintergrund gestartet.

Um in nachfolgenden Arbeiten dennoch auf den internen Zustand des LanRouters zugreifen zu können wurde weiterhin eine *EMonitor-Schnittstelle* integriert. Mit ihrer Hilfe sollen die Router und Stationen an einen Simulator angebunden werden.

4 Umsetzung und Implementierung von Middleware und LanRouter

Im folgenden Kapitel werden die in Kapitel 3 vorgestellten Konzepte für eine Middleware und eines LanRouters implementiert. Dazu werden geeignete Schnittstellen zur Verwendung von WiLiDips innerhalb der Middleware und des Frameworks vorgestellt, sowie die verschiedenen Nachrichtensysteme zwischen Framework, Middleware, Routing und den jeweils entfernten Komponenten implementiert. Das Kapitel schließt mit der Beschreibung der Umsetzung des LanRouters ab.

4.1 Das Handling von WiLiDips

WiLiDips werden im XML-Format[1] gespeichert. In Listing 4.1 werden Ausschnitte der XML-Darstellung eines WiLiDip-Profils gezeigt. Man sieht, daß bei dem betreffenden WiLiDip 30% Wissen über Informatik vorhanden ist und 20% Lerninteresse daran bekundet werden. Weiterhin ist ein Chat-Service an das Profil für die Bereiche Mathematik und Biologie angehängt.

```
<wilidip>
  <concept name="Informatik" id="130" wi="30" li="20">
    <subConceptOfId="0"/>
  </concept>
  ...
  <service id="1">
    <name>Chat</name>
    <sconcept name="Mathematik" id="148" wi="100" subid="0"/>
    <sconcept name="Biologie" id="100" wi="30" subid="0"/>
  </service>
  ...
</wilidip>
```

Listing 4.1: Teile der XML-Darstellung eines WiLiDips

Den Komponenten aus E.L.A.N wird mit der Klasse `Wilidip` eine Abstraktionsschicht zur Verfügung gestellt, damit diese nicht direkt mit der XML-Darstellung arbeiten müssen. Stattdessen können sie direkt auf Methoden von `wilidip`-Instanzen operieren.

Der Inhalt einer Instanz eines WiLiDips kann über die Methode `saveAsXML` als XML-Datei abgespeichert werden. Das Laden eines WiLiDips erfolgt analog über die Methode `loadFromXML`. Um ein WiLiDip im Netz verteilen zu können, wird es über die Methode `pickle` in

[1] eXtensible Markup Language

eine Pickle-Repräsentierung umgewandelt. Als „pickle“ bezeichnet man unter Python die Serialisierung, also die Umwandlung von Datenstrukturen in Byte-Ströme. Über die `compareTo`-Methode kann die für den Vergleich zweier Profile notwendige Korrelationsmethode aufgerufen werden. Sie gibt den Prozentwert der Übereinstimmung der Profile zurück. Die Middleware überprüft anhand eines Schwellenwerts, ob das verglichene WiLiDip interessant ist. Zusätzlich beinhaltet die Klasse eine Reihe von dienstspezifischen Methoden, wie z.B. `addService`, um einen Dienst hinzuzufügen, `removeService`, um einen Dienst zu löschen und `listServices`, um alle angehängten Dienste anzeigen zu lassen.

4.1.1 Verwaltung der WiLiDips

Die Middleware verwaltet die WiLiDip-Instanz für das gesamte Framework. Dazu lädt und speichert sie das WiLiDip auf einem nicht flüchtigen Speicher (Compact Flash oder Festplatte), en einem in der Konfiguration vorgegeben Pfad.

Damit Komponenten des Frameworks auf das WiLiDip zugreifen können, muß die Middleware diesen eine Schnittstelle zur Verfügung stellen. Diese wird über lego-Nachrichten umgesetzt.

Die Nachricht `updateWilidip` dient dazu, andere Teilnehmer des Kommunikationssystems über eine Aktualisierung des WiLiDips zu informieren. Dabei wird das aktuelle WiLiDip als Argument versendet. Die jeweiligen Empfänger können das WiLiDip direkt verwenden. Ist die Middleware Empfänger der Nachricht, speichert sie das mitgelieferte WiLiDip zusätzlich ab. Außerdem aktualisiert sie die GuID und stößt eine Neuverteilung des Profils an. Üblicherweise empfängt sie die Nachricht, nachdem im Profileditor das WiLiDip verändert wurde. Wenn der Nachricht anstelle des WiLiDips das Argument `None` übergeben wurde, liest die Middleware das WiLiDip aus dem vorkonfigurierten Profilpfad ein. Auf diese Weise kann das Framework das initiale Laden des WiLiDips steuern. Ebenso kann das Framework durch Versenden der Nachricht `requestWilidip` von der Middleware das eigene WiLiDip anfordern.

Nach dem Start des Profil-Editors `wilied` sendet die Middleware ebenfalls das aktuelle WiLiDip über die `updateWilidip`-Nachricht an das Framework, damit der Editor dieses darstellen kann. Dazu empängt sie die `componentReady`-Nachricht, die das Frameworks immer versendet, wenn eine Komponente geladen wird. Der Name der Komponente wird dabei als Argument übergeben und von der Middleware ausgewertet.

Sobald eine Dienst-Komponente innerhalb des Frameworks gestartet wird, versendet dieses die Nachricht `addService` mit dem ServiceDescriptor des Dienstes. Analog versendet das Framework `removeService`, sobald ein Dienst beendet wurde. Da die Middleware die unspezifizierten Dienste verwaltet, muß sie einen solchen an das WiLiDip anfügen, sobald er gestartet wird und entfernen, sobald er beendet wurde.

Nach jeder Veränderung, die die Middleware an einem WiLiDip vornimmt, speichert sie es ab und inkrementiert die GuID. Nach jeder Veränderung der GuID wird das WiLiDip neu verteilt.

Zusätzlich zu den beschriebenen Nachrichten sendet die Middleware über die Nachricht `fullWilidipList` in regelmäßigen Intervallen (im Abstand weniger Sekunden) die vollständige Liste von Stationen bzw. WiLiDips, die die Middleware verwaltet. Dies beinhaltet also auch die Informationen über uninteressante Teilnehmer des Netzes. Dies dient zur Steuerung der `middlewareActivity`-Komponente und wird zu Debugging-Zwecken verwendet. Durch die geraume Größe, die eine solche Nachricht annehmen kann, wird diese nur versendet, wenn

eine middlewareActivity läuft. Dies wird wie beim Profileditor über die `componentReady`-Nachricht festgestellt. Die Beendigung der middlewareActivity-Komponente wird analog über die dann versandte Nachricht `componentShutdown` erkannt.

4.1.2 Identifikation der WiLiDips

Zur Identifikation eines WiLiDips wird ein Schlüssel benötigt, da in diesem keinerlei spezifische Informationen über seinen Ursprung und seinen Besitzer vorliegen. Der Schlüssel kann in einem Dictionary verwendet werden, um einerseits an das WiLiDip selbst heranzukommen und um andererseits weitere Informationen, die dem WiLiDip zugeordnet sind, abzufragen. Als Schlüssel eignen sich die IP-Adresse der Station, deren MAC-Adresse oder eine selbstgenerierte ID.

Die IP-Adresse einer Station ist eindeutig, wenn man davon ausgeht, daß auf einer Station nur eine E.L.A.N-Instanz läuft. Es kann jedoch Probleme geben, wenn eine Station mehrere Netzwerkkarten hat.

Die MAC-Adresse ist eine eindeutige Identifikation einer Netzwerkkarte, die von den Herstellern vergeben wird. Sie wäre für unsere Zwecke isomorph auf eine IP-Adresse abbildbar. Daher gelten die gleichen Vor- und Nachteile wie bei dieser, d.h. ein Rechner könnte mehrere MAC-Adressen haben, wenn er mehrere Netzwerkkarten besitzt.

Auf Routing-Ebene wurde entschieden, die IP-Adresse als Schlüssel zu verwenden, da dies die Implementierung vereinfacht. Ein logischer Aufruf auf den Router ist eine Funktion wie „route nach IP-Adresse x.x.x.x". Auf Middleware- bzw. auf Komponenten-Ebene hat sich gezeigt, daß die Verwendung der IP-Adresse aus den oben genannten Gründen ein Problem darstellt. Daher wird zur Überprüfung der Eindeutigkeit die sogenannte GuID verwendet, eine selbstgenerierte, im E.L.A.N System eindeutige ID.

In der Middleware werden in einem Dictionary WiLiDips und deren Eigenschaften in Form von Instanzen der Klasse `WilidipInfo` gespeichert. Dies umfaßt das eigentliche WiLiDip und dessen GuID, den Status in der Wilidipmanager-Statusmaschine, den Vergleichswert, den Zeitpunkt zu dem das Wilidip zuletzt empfangen wurde und eine Liste von IP-Adressen, die die zugehörigen Absenderadressen kennzeichnen. Als Schlüssel zu den Elementen des Dictionaries wird die IP-Adresse genommen, die vom Router übergeben wird.

4.2 Implementierung der Middleware

4.2.1 ECom

Ein Problem der Programmierung mit Qt besteht darin, daß die Qt-Bibliothek nicht threadübergreifend verwendet werden kann. Nur ein einziger Thread darf GUI-Funktionen ausführen. Weitere Threads müssen dem Gui-Thread ihre Informationen über Warteschlangen, in Python durch das Queue-Modul implementiert, zur Verfügung stellen.

Um dieses Problem zu umgehen wurde *ECom* entwickelt [Lau04], eine der Kommunikationsinfrastrukturen von E.L.A.N. ECom besteht aus mehreren Kommunikations-Kapselungen und ermöglicht eine asynchrone Kommunikation der verschiedenen Komponenten über sogenannte lego-Nachrichten. Eine Auflistung aller im System versendeten lego-Nachrichten findet sich in A.1.

Dabei wird zwischen Monothread-Kommunikation und Multithread-Kommunikation unterschieden. Die Monothread-Kommunikation erfolgt im Gui-Thread. Hier laufen mehrere Komponenten innerhalb eines Threads ab. Um z.B. mit den Threads der Middleware zu kommunizieren, muß eine threadübergreifende Kommunikation erfolgen.

Monothread-Kommunikation

Die Monothread-Kommunikation erfolgt über die Klassen `ECommunicationChannel` und `ECommunicationComponent`. Die Klasse ECommunicationChannel abstrahiert einen gemeinsamen Kommunikationskanal zum Senden und Empfangen von Nachrichten. Möchte eine Komponente am Nachrichtensystem teilhaben, registriert sie sich am Kanal und empfängt sofort alle Nachrichten, für die sie Empfänger bereitstellt. Ein Empfänger bezeichnet hierbei eine Methode, deren Präfix `lego_` ist. Sendet eine Komponente eine Nachricht, z.B. die Nachricht `self.post('test')`, so wird in allen partizipierenden Komponenten die Methode `lego_test()` aufgerufen, wenn sie in den jeweiligen Komponenten definiert wurde. Eine von ECommunicationComponent abgeleitete Klasse registriert und deregistriert sich selbstständig am Kommunikationskanal.

Multithread-Kommunikation

Für threadübergreifende Kommunikation wird eine threadsicher Warteschlange benötigt. Eine solche realisiert der `EQueuedCommunicationChannel` und die dazugehörige Abstraktionsklasse `EQueuedCommunicationComponent`. Wie in der Monothread-Kommunikation können Nachrichten mit `post` gesendet und über Empfänger abgearbeitet werden. Ein wichtiger Unterschied ist, daß aktiv in der Warteschlange nach Nachrichten gesucht werden muß. Dazu wird die Methode `handleMessages()` regelmäßig aufgerufen.

Kommunikationsbrücke

Die Klasse `ECrossOver` implementiert eine Kommunikationsbrücke zwischen der Threadübergreifenden Kommunikationswarteschlange und der Monothread-Kommunikation. Sie sorgt dafür, daß alle Nachrichten des einen Systems im jeweils anderen empfangen werden können.

4.2.2 XML-RPC-Schnittstelle

Die XML-RPC-Schnittstelle der Middleware besteht, aus mehreren Komponenten. Ein Server, der `ElanXMLRPCServer`, wartet am vordefinierten TCP / IP-Port 5557 auf eingehende XML-RPC-Aufrufe. Diese übergibt er zur weiteren Bearbeitung an einen Handler. Der Handler ist aus Gründen der Übersicht in den `ELANXMLRPCDispatcher` und den `ElanXMLRPCHandler`, der die zur Verfügung stehenden XML-RPC-Methoden beinhaltet, unterteilt. Da der `ElanXMLRPCServer` in einem eigenen Thread läuft, ist er als `EQueuedCommunicationComponent` an das ECom-Nachrichtensystem angeschlossen. Eingehende XML-RPC-Aufrufe werden von ihm über lego-Nachrichten an das restliche System zur Verarbeitung weitergleitet. Der `ElanXMLRPCServer` ist von der Klasse `EXMLRPCServer` abgeleitet, die im Gegensatz zur Klasse `SimpleXMLRPCServer` eine „saubere“ Methode zum Herunterfahrens des Servers

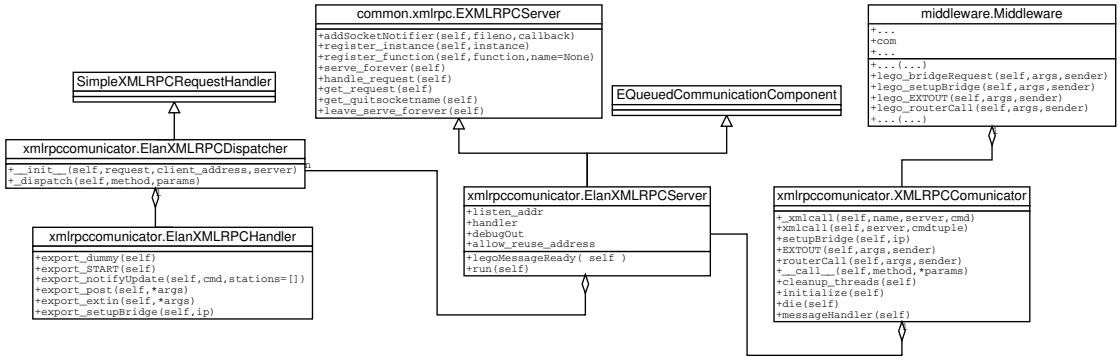

Abbildung 4.1: UML-Ansicht des XMLRPCCommunicators

enthält. Diese ist notwendig, da beim normalen Beenden eines Serverprozesses der verwendete Port blockieren kann, bis das Betriebssystem bemerkt, daß an diesem kein Server mehr auf eingehende Verbindungen wartet.

Diese einzelnen Komponenten werden von vom `XMLRPCCommunicator` gestartet, der die zentrale Steuerung der XML-RPC-Kommunikation übernimmt. Er ist daher auch für ausgehende Aufrufe zuständig. Für diese startet der `XMLRPCCommunicator` jeweils über die Methode `xmlcall` einen eigenen Funktionsthread, der nach Abarbeitung des Aufrufs wieder beendet wird. Dies soll das restliche System vor kurzzeitig blockierenden Verbindungen oder länger dauernden XML-RPC-Aufrufen, bewahren. XML-RPC-Verbindungen, die aus Gründen von Netzwerkfehlern länger blockieren, werden über einen Timeout abgebrochen. In diesem Fall wird keine Nachricht an das Framework gesendet, der externe Aufruf geht also verloren. Damit soll ein interner Kommunikationsoverhead vermieden werden.

Die von der XML-RPC-Schnittstelle zur Verfügung gestellten Methoden werden im Anhang A.4.2 vollständig besprochen.

4.2.3 extIn- und extOut-Kommunikation

Die extIn- und extOut-Kommunikation stellt eine Kommunikationsbrücke für die Kommunikation von E.L.A.N-Komponenten untereinander dar.

Sie wird durch die Kombination der XML-RPC-Schnittstelle und der internen ECom-Schnittstelle hergestellt. Sie befindet sich nicht auf Framework-Ebene, sondern wird auf der Ebene der Middleware-Kommunikation durchgereicht.

Als Funktionsbeispiel dient die Awareness-Komponente awarenessView, die von der Identifikations-Komponente identd, eines entfernten Rechners Informationen über einen Benutzer haben möchte. Abb. 4.2 zeigt die dabei entstehenden Kommunikationsverläufe.

Die awarenessView-Komponente sendet dabei folgende lego-Nachricht an die Middleware:

```
self.post('EXTOUT','awarenessview','identd',ip,'requestIdent',())
```

Diese Nachricht besagt, daß die Komponente `awarenessview` der Komponente `identd` auf der Maschine mit IP-Adresse `ip` die Nachricht `requestIdent` ohne weitere Argumente

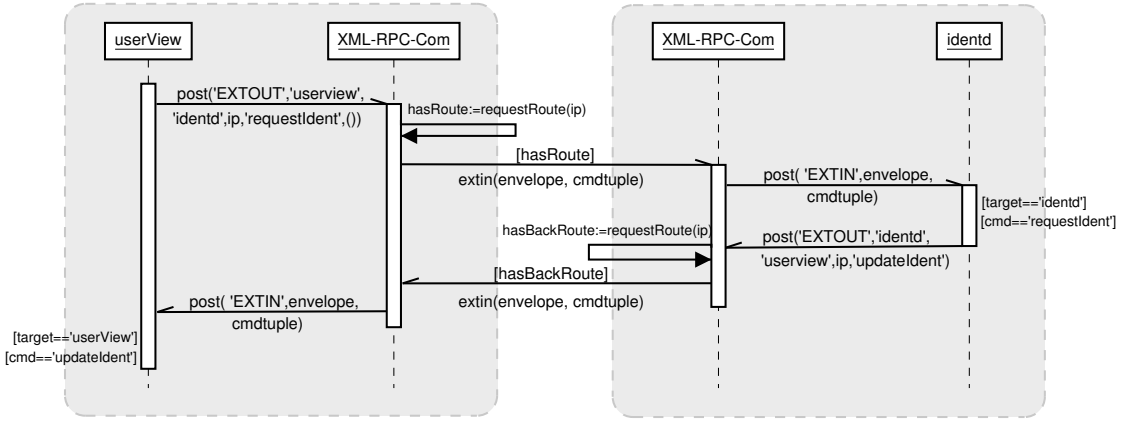

Abbildung 4.2: extIn- / extOut-Kommunikation

schicken möchte.

In der Middleware wird diese Nachricht empfangen und über den XMLRPCCommunicator eine Verbindung initialisiert. Dazu verpackt dieser die Nachricht mit einigen Zusatzinformationen in ein logisches Paket bestehend aus den IP-Adressen von Absender und Empfänger, den Komponentennamen von Absender und Empänger und dem eigentlichen Befehl mit seinen Argumenten.

Anschließend wird eine Hin- und Rück-Route zur entfernten Station initialisiert und der entfernte Aufruf `extin` durchgeführt:

```
def EXTOUT(self, args, sender):
    sourceService,targetService,targetIP,targetcmd,targetparms=args
    envelope = ((sourceService,sourceIP), (targetService,targetIP))
    cmdtuple = (targetcmd, targetparms,)
    cmd = (envelope, cmdtuple)

    self.mw.lego_requestRouteTo((targetIP,), self)
    server = "http://%s:%s" % (targetIP, port)
    server.extin(cmd)
```

Auf Empfängerseite wird die Nachricht über eine lego-Nachricht weitergeleitet:

```
def export_extin( self, *args):
    cmd = args[0]
    envelope, cmdtuple = cmd[0], cmd[1]
    self.server.post( 'EXTIN', envelope, cmdtuple)
```

Die identd-Komponente ist Empfänger von lego-EXTIN-Nachrichten. Sie prüft, ob sie Empfänger der Nachricht ist und ob das Kommando ein gültiges ist. Falls ja, kann sie Empfänger und Sender-Parameter vertauschen und mit dem gleichen Mechanismus antworten. In diesem Fall sendet sie eine `updateIdent`-Nachricht mit den Informationen über einen Benutzer und mit seinem Benutzerbild zurück.

```
def lego_EXTIN(self, params, sender):
```

```
envelope, cmdtuple = params[0], params[1]
sourceService, sourceIP = envelope[0]
targetService, targetIP = envelope[1]
cmd, args = cmdtuple

if not targetService==’identd’:
    return

if cmd == ’requestIdent’:
    sourceService, targetService = targetService, sourceService
    sourceIP, targetIP = targetIP, sourceIP
    self.post( ’EXTOUT’, sourceService, targetService, targetIP,
                ’updateIdent’, (self.ident, self.pixmap, ) )
```

MUPs

Die in 3.2.1 vorgestellten *Middleware UDP Pakete* (MUPs) sie sind als Klassenhierarchie implementiert. Die Vererbungshierarchie ihrer Klassen wird in Abbildung 4.3 dargestellt. Die MUPs bestehen aus der Basisklasse `Message` für ausgehende Pakete und der erweiterten Basisklasse `RMessage` (Received Message), die der Repräsentation von eingehenden Paketen dient.

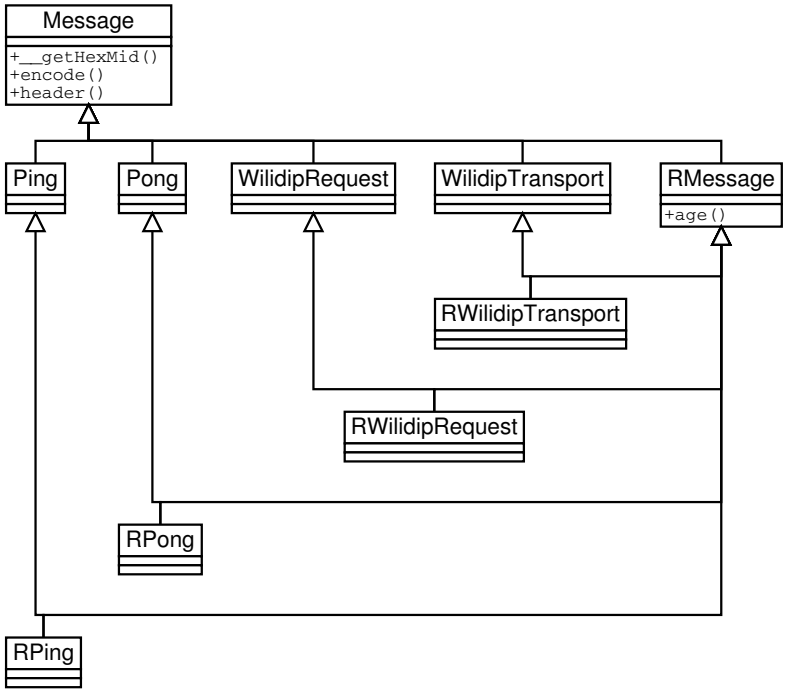

Abbildung 4.3: Klassenstruktur der MUPs (*Middleware UDP Packets*)

Wie gezeigt ist, werden die Pakettypen `Ping`, `Pong`, `WilidipRequest` und `WilidipTransport` von `Message` abgeleitet. Die Pakete `RPing`, `RPong`, `RWilidipRequest` und

`RWilidipTransport` erben zusätzlich noch die Eigenschaften von `RMessage`. Eine genaue Beschreibung ihrer Inhalte ist in Anhang A.2 beschrieben.

Die Antwortdatentypen unterscheiden sich von den normalen Paketen dadurch, daß ihr Konstruktor ein empfangenes binäres MUP als Argument haben kann. Dieses kann er automatisch in die ursprüngliche Datenstruktur entpacken, so daß ein einfacher Zugriff auf seine Bestandteile möglich ist. Somit wird eine Abstraktion zu den binären Paketen zur Verfügung gestellt. Weiterhin enthalten sie die Methode `age`, die TTL dekrementiert und HOPS inkrementiert. Über die Methode `encode` können die Nutzdaten in binärer, komprimierter Form abgefragt werden, um sie anschließend per UDP versenden zu können.

4.2.4 Protokollmaschine

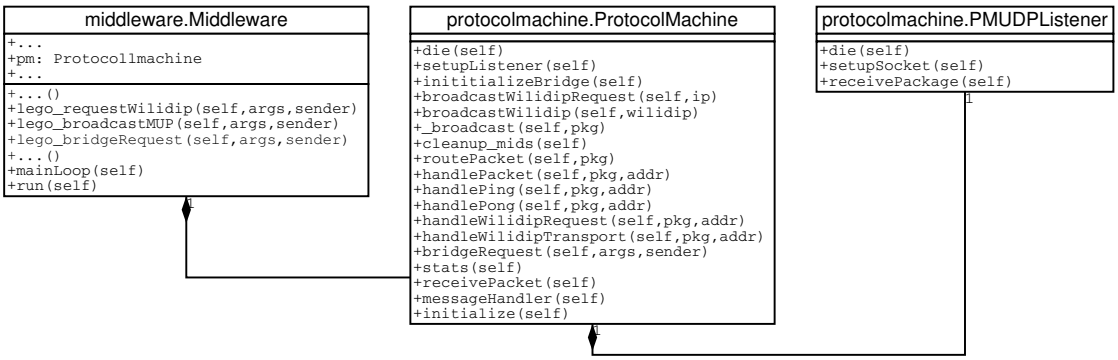

Abbildung 4.4: UML-Ansicht der Protokollmaschine

Die Protokollmaschine besteht aus zwei Klassen. Die Klasse `ProtocolMachine` implementiert die Hauptklasse der Protokollmaschine. Diese enthält die Methoden zum Versenden von Paketen (MUPs) über Broadcasts und die Methoden zum Verarbeiten von eingehenden Paketen. Zusätzlich verwaltet sie eine Instanz der Klasse `PMUDPListener`, die den Server implementiert, also die Komponente, die eingehende Pakete empfängt. Eine UML-Darstellung dieser Klassen ist in Abb. 4.4 zu erstellt. Die Protokollmaschine empfängt selbst keine lego-Nachrichten. Diese werden von der Hauptklasse der Middleware empfangen, die sie an entsprechende Methoden der Protokollmaschine durchreicht.

Der Empfang eines Pakets erfolgt innerhalb des PMUDPListeners durch seine `receivePackage`-Methode. Diese prüft, ob ein empfangenes Paket ein gültiges MUP ist. Wenn ja, wird es der Methode `receivePacket` der Protokollmaschine übergeben. Diese prüft, ob das MUP zu alt ist, also den Vorgabewert von 7 Hops überschritten hat. Wenn dies der Fall ist, wird das Paket verworfen, wenn nicht wird der Inhalt über die Methode `handlePacket` analysiert.

Ist das Paket ein Ping, so wird ein Pong-Paket erstellt und versendet. Das Ping wird ebenfalls weitergeroutet. Im Falle eines WilidipRequests wird untersucht, ob das gesuchte WiLiDip von zurückgesendet werden kann. Wenn nicht, wird das Paket weitergeleitet, sonst wird es verworfen und statt dessen eine WilidipTransport zurückgesendet. Ein Pong-Paket wird dem Framework über die lego-Nachricht `receivedPong` mitgeteilt. Ein durch einen WilidipTransport

empfangenes WiLiDip wird dem Wilidipmanager zur weiteren Bearbeitung übergeben.

Verteilung der MUPs

Die Protokollmaschine implementiert einen Flutalgorithmus. Ein empfangenes Paket wird über alle Interfaces, einschließlich des Interfaces, über den das Paket empfangen wurde, einmalig weitergebroadcastet. Weiterhin wird das Paket an alle zusätzlich angegeben IP-Adressen per Unicast gesendet. Die Einmaligkeit des Sendens wird durch eine eindeutige Message-ID gewährleistet. Sie broadcastet das Paket auch über die Schnittstelle, über die es ursprünglich ankam. Im Funknetz wirkt eine Station so wie ein Repeater. Üblicherweise wird ein PDA nur über eine Funknetzwerkkarte verfügen. Da er sich im Mittelpunkt seiner eigenen Funkwolke befindet, kann er durch erneutes Broadcasten die Reichweite des ursprünglichen Senders im besten Fall verdoppeln. Es bleibt keine andere Möglichkeit, als das Pakete über die empfangende WLAN-Karte erneut zu versenden, auch wenn die Station, die das Paket ursprünglich versandte das Paket wieder empfangen sollte.

Der Empfangsport ist bei allen Stationen vorgegeben (*Well-known-port*-Schema) und wird aus der Konfiguration ausgelesen. Für die Protocolmaschine ist der Vorgabewert Port 5555.

4.2.5 WilidipManager

Der Wilidipmanager verwaltet ein Dictionary von Stationen namens `stations`. Schlüssel des Dictionaries sind die IP-Adressen von Stationen, die Werte sind Instanzen der Klasse Wilidipinfo.

Instanzen der Klasse Wilidipinfo dienen als Container für alle Informationen, die über eine Station bzw. über ein WiLiDip benötigt werden. Insbesondere gehören hierzu das WiLiDip selbst, die GuID der Station und der Status in der Statusmaschine des Wilidipmanagers. Darüber hinaus bietet sie die benötigten Zugriffsmethoden auf diese Information (siehe UML-Darstellung in Abb. 4.5).

Abbildung 4.5: Die Klasse Wilidipinfo

Die Klasse `WiLiDipmanger` enthält alle Methoden, die zur Aktualisierung der Liste notwendig sind. Die Middleware enthält eine Instanz dieser Klasse. An den `WiLiDipmanger` direkt gekoppelt ist der später beschriebene `ServiceManager` (vgl. 4.2.7). In Abb. 4.6 ist die UML-Darstellung der Klassen mit ihren Methoden und Attributen zu sehen.

Aktualisieren der Liste

Die Liste der Stationen wird beim Aufruf der `handleListUpdate`-Methode aktualisiert. Diese wird von der zentralen Middleware-Instanz aufgerufen, wenn diese vom XMLRPCCommunicator über die lego-Nachricht `MW_routerNotifiedUpdate` mitgeteilt bekommt, daß der Router neue Informationen gesendet hat. Wenn die Protokollmaschine ein neues WiLiDip empfängt, teilt sie dieses dem Wilidipmanager über die Methode `handleWiLiDipUpdate` mit. Kam es noch nicht in der Liste vor, so wird ebenfalls die Liste aktualisiert.

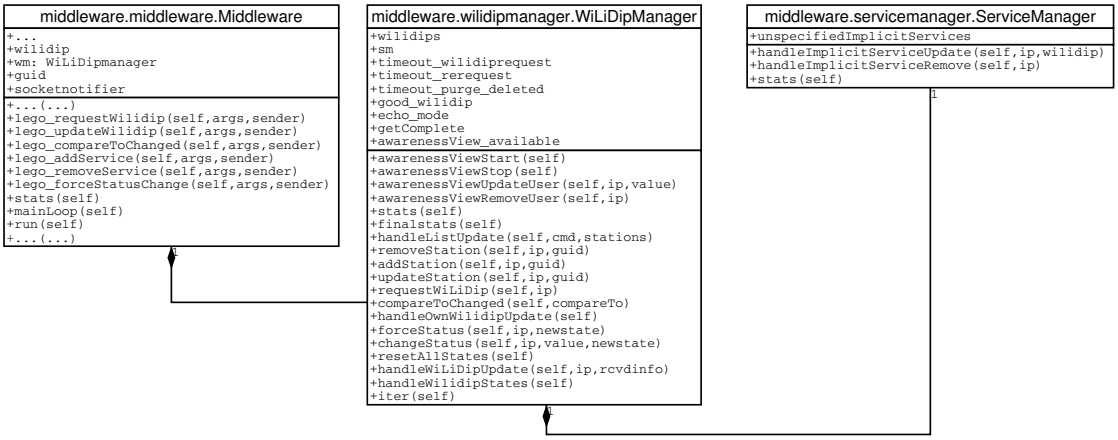

Abbildung 4.6: UML-Ansicht des WiLiDipmanagers und des ServiceManagers

`handleListUpdate` bekommt als Argumente ein Kommando und eine Liste von Stationen. Diese Liste entspricht dem Übergabeformat vom Router zur Middleware, enthält also neben der Liste noch eines der Kommandos `add`, `remove`, `update` oder `full` (vgl. A.4.2). Die GuID liegt als String vor und muß in eine GuID-Instanz gewandelt werden. Die Liste wird zur Weiterverarbeitung in ein Dictionary umgewandelt.

Anschließend wird das interne Stations-Dictionary entsprechend des übergeben Kommandos aktualisiert. Für neue Stationen wird eine leere Wilidipinfo-Instanz angelegt, die nur die übergebene GuID enthält.

Der Wilidipmanager empfängt WiLiDips über seine `handleWiLiDipUpdate`-Methode, die von der Protokollmaschine aufgerufen wird, sobald ein neues WiLiDip empfangen wurde.

Die `handleWiLiDipUpdate`-Methode hat einige Verwaltungsaufgaben zu erledigen, bis sie ein WiLiDip in der Stationsliste aktualisiert. Dazu gehört Insbesondere das Vergleichen des Alters der GuID zu einem bestehenden Eintrag. Anhand diesem kann erkannt werden, ob ein WiLiDip jünger oder älter ist, als ein vorhandenes. Ein empfangenes älteres WiLiDips wird nicht aktualisert, sondern verworfen. Da der Zeitstempel immer nur auf der Ursprungsmaschine aktualisiert und kein Vergleich zur aktuellen Uhrzeit, sondern nur zu bestehenen Zeitstempeln des gleichen Ursprungs durchgeführt wird, können keine Probleme mit unterschiedlich laufenden Uhren entstehen.

Erst wenn alle Prüfungen positiv verlaufen sind, wird das WiLiDip in der Liste aktualisiert. Dies entspricht dem Übergang von „Empfangen“ auf „ok“ oder „error“ in Abb. 4.7.

Die Statusmaschine

In jedem Takt der Middleware durchläuft der Wilidipmanager seine `handleWilidipStates`-Methode. Diese entspricht einem Zustandsautomaten, der im wesentlichen die in Abbildung 4.7 dargestellten Zustände durchläuft.

Nach dem Empfang ist der Status eines WiLiDips „ok“ oder „error“, je nachdem ob die Übertragung in Ordnung war. Es wird nun durch Aufruf der Korrelationfunktion `compareTo` über-

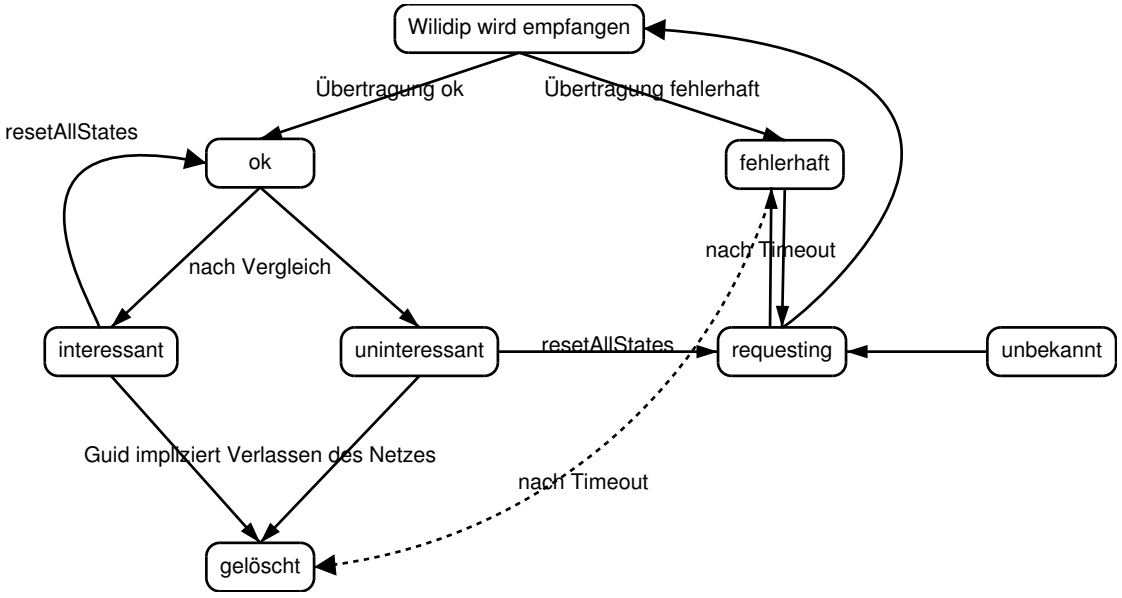

Abbildung 4.7: Statusmaschine des Wilidipmanagers

prüft, ob ein WiLiDip bezüglich des eigenen WiLiDips interessant ist. Der entsprechende Status wird dann gesetzt. Wenn ein WiLiDip nicht interessant ist, so wird es gelöscht, um Speicher zu sparen. Die Middleware verwaltet im Folgenden dann nur noch die Existenz eines Netzteilnehmers, der nicht weiter interessant ist.

Von allen Zuständen existiert ein Übergang nach „unbekannt“, wenn eine neue GuID impliziert, daß das WiLiDip veraltet ist. Der Übergang in den Zustand „requesting“ kann jederzeit durch die Middleware forciert werden, wenn diese eine Aktualisierung des WiLiDips wünscht. Da ein WiLiDip jederzeit empfangen werden kann, ist ein direkter Statusübergang zu den gültigen Empfangsstadien „ok“ oder „error“ ebenfalls jederzeit möglich.

Fällt ein Vergleich positiv aus, ist ein WiLiDip also interessant, so wird der awarenessView-Komponente die lego-Nachricht `awareness_updateUser` geschickt, um Ihr diese Information mitzuteilen. Ebenso werden Ihr die expliziten Dienste des möglichen Kollaborationspartners über die lego-Nachricht `awareness_updateService` mitgeteilt.

Ändert sich der Status eines interessanten WiLiDips im weiteren Betrieb auf „gelöscht“ oder „uninteressant“, werden der awarenessView-Komponente entsprechend die Nachrichten `awareness_removeUser` und `awareness_removeExplicitServices` gesendet.

Ändert sich das eigene WiLiDip oder der Schwellenwert für die Vergleiche, so werden die Ergebnisse vorheriger Vergleiche ungültig. Es ist nur durch Neuvergleich festellbar, ob ein WiLiDip interessant war oder nicht. Daher wird in einem solchen Fall der Status von interessanten WiLiDips auf „ok“ und von uninteressanten, und daher gelöschten WiLiDips auf „requesting“ zurückgesetzt.

4.2.6 ServiceDescriptoren

Die Klasse `Service` enthält die Beschreibung eines Dienstes. Zur Zeit ist dies neben dem Dienstnamen der Typ (vgl. 3.2.2) und eine MetaInfo, die von jedem Dienst benutzt werden kann, um genauere Einzelheiten zu verwalten. So speichert der converter hier z.B. seine möglichen Quell- und Ziel-Typen und diejenigen Programme, die zum Umwandeln notwendig sind. Außerdem kann der Wilidip-Editor hier hinterlegen, zu welchen Konzepten ein Dienst zugeordnet ist.

Die Klasse ist mit entsprechenden Zugriffs- (`setName`, `setType`, etc.) und Abfrage-Methoden (`getMetaInfo`, `isUnspecified`, `isExplicit` etc.) ausgestattet, die hier nicht näher erläutert werden sollen.

4.2.7 ServiceManager

Der ServiceManager ist eine Abstraktionsklasse für die Verwaltung unspezifiziert-impliziter Dienste. Er speichert solche Dienste in einem als Dictionary realisierten Dienstverzeichnis. Zu einem Dienstnamen als Schlüssel wird ein weiteres Dicitionary als Wert gespeichert, das die Zuordnung von IP-Adresse und Service-Deskriptoren enhält.

Der ServiceManager läuft als Instanz des Wilidipmanagers und wird über diesen aufgerufen. Das geschieht stets dann, wenn ein WiLiDip empfangen wurde oder wenn eine Station sicher das Netz verlassen hat. Dies entspricht in Abbildung 4.7 den Zuständen „Ok“ und „Gelöscht“.

Über die Methode `handleImplicitServiceUpdate` wird ihm von diesem ein neues oder aktualisiertes WiLiDip einer Station übergeben. Aus dem WiLiDip werden alle angehängten unspezifiziert-impliziten Dienste extrahiert und mit dem Dienst-Verzeichnis abgeglichen. Die Veränderungen werden der E.L.A.N-awarenessView-Komponente über die lego-Nachricht `awareness_updateService` mitgeteilt.

Analog lassen sich bei Wegfall einer Station deren implizite Dienste über die Methode `handleImplicitServiceRemove` löschen. Dazu wird im Verzeichnis unter allen Dienstnamen nachgesehen, ob die Station einen oder mehrere Dienste zur Verfügung gestellt hatte. Diese Einträge werden daraufhin entsprechend entfernt. Sollten unter dem Dienstnamen keine Stationen vorhanden sein, die den Dienst anbieten, wird der Dienstname aus dem Verzeichnis gelöscht. Sollte eine Veränderung des Dictionaries stattgefunden haben, so wird dies der awarenessView-Komponente über die Nachrich `awareness_removeImplicitService` mitgeteilt.

4.2.8 Kontrollfluß

Die Middleware muß während des Betriebs regelmäßige Wartungsarbeiten, wie das Durchlaufen der Statusmaschine des WiLiDipmanagers und das Bereinigen der Message-ID-Listen der Protokollmaschine durchführen. Gleichzeitig muß sie eingehende ECom-Nachrichten, eingehende MUP-Pakete in der Protokollmaschine und eingehende XML-RPC-Aufrufe im XMLRPC-Communicator bearbeiten. Zur Bearbeitung der ECom-Nachrichten muß sie als `EQueuedCommunicationComponent` in regelmäßigen Abständen ihre Kommunikations-Queue leeren (vgl. ECom in 4.2.1).

Im ursprünglichen Entwurf wurde dies durch mehrere nebenläufig ablaufende Threads gelöst, da dies als die konzeptionell logischste Lösung erschien. Dadurch entstand jedoch ein starker Overhead, weil in den einzelnen Hauptroutinen der Threads jeweils *Busy-Waiting*[2] durchgeführt wurde. Um das Problem zu lösen wurde der EMultipleSocketNotifier entworfen.

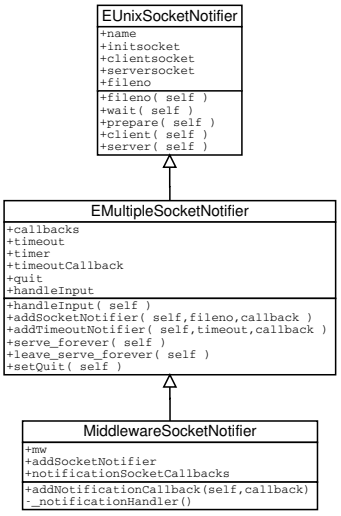

Abbildung 4.8: Die Socket-Notifier-Hierarchie

Die in Abb. 4.8 gezeigte Klasse `EMultipleSocketNotifier` wird vom Framework zur Verfügung gestellt. Sie kapselt den POSIX[3] `select`-Aufruf. `select` bekommt eine Menge von Dateideskriptoren übergeben und optional einen Timeout. Ein Aufruf von `select` blockiert nun so lange, bis sich an einem beliebigen Dateideskriptor eine Änderung ergibt oder der Timeout abgelaufen ist. Dabei wird das Blockieren nicht durch busy-waiting, sondern durch einen Aufruf im Betriebssystem-Kern gesteuert. Der Rückgabewert des `select`-Aufrufs enthält den Deskriptor, in dem die Änderung enthalten ist und der somit ausgelesen werden kann. In der vererbten Klasse `MiddlewareSocketNotifier` wird der `EMultipleSocketNotifier` um die Möglichkeit erweitert, Callback-Methoden aufzurufen, wenn Nachrichten an der ECom-Schnittstelle vorliegen. Die Middleware, als `EQueuedCommunicationComponent`, enthält die vererbte Methode `notificationSocket`, die den Dateideskriptor der ECom-Schnittstelle zurückgibt. Dieser wird innerhalb des select-Aufrufs ausgewertet, um bei vorliegenden ECom-Nachrichten die Callback-Methoden auszuführen.

Abbildung 4.9 zeigt den Kontrollfluß innerhalb der Middleware. Nur die Middleware selbst bezüglich des restlichen Framework und ihre echten Netzwerk-Server, der PMUDPListener und der XMLRPCListener sind als einzelne Threads implementiert. Ihre Server-Sockets werden der `MiddlewareSocketNotifier` Instanz namens socketnotifier übergeben. Weiterhin wurde als Callback-Methode für eingehende ECom-Nachrichten die Methode `handleMessages` eingerichtet. Über einen Timeout wird die Hauptschleife der Middleware (`mainLoop`) aufgerufen. Diese ruft für alle Komponenten der Middleware jeweils die Methode `iter` auf, in der diese ihre durchzuführenden Aufgaben erledigen können. Durch diesen Aufruf ist eine Art Hauptschleife realisiert, die in regelmäßigen Intervallen durchlaufen wird. Es entsteht auf diese Weise eine Taktung der Middleware, gesteuert durch das Intervall, das dem socketnotifier als Timeout übergeben wird.

[2] Als Busy-Waiting bezeichnet man das dauerhafte, CPU-belastende Durchlaufen einer Schleife, in der die meiste Zeit nichts getan wird

[3] Portable Operating System Interface

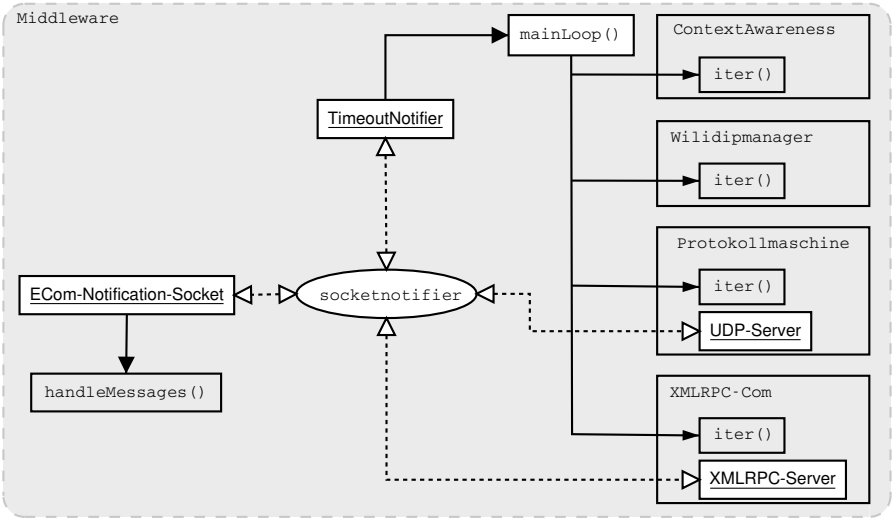

Abbildung 4.9: Kontrollfluß in der Middleware

4.3 Implementierung des LanRouters

Abbildung 4.10 zeigt anhand einer UML-Darstellung die Umsetzung des in Kapitel 3.4.3 vorgestellten Konzepts des Routers.

Der LanRouter ist aus mehreren Komponenten aufgebaut. Innerhalb seiner Hauptklasse LanRouter verxwaltet er die Liste der Stationen im Netz in einem Dictionary, das IP-Adressen als Schlüssel und Instanzen von StationValues als Werte verwendet.

Seine Hauptschleife läuft als eigenständiger Thread. Zusätzlich zur Hauptschleife laufen drei weitere Threads gleichzeitig. Einer davon ist der Server-Prozess LanRouterListener, der eingehende Broadcastpakete empfängt. Der ElanXMLRPCListener ist ebenfalls als eigenständiger Thread implementiert. Er empfängt XML-RPC-Aufrufe der Middleware. Um regelmäßig Broadcasts aussenden zu können, ist auch der Broadcaster als Thread ausgelegt.

4.3.1 Programmablauf

Je nach Kommandozeilenoption kann der LanRouter als Hintergrundprozess (Daemon), als Konsolenprogramm oder als grafisches Programm gestartet werden. Die Ausgabe seiner Statusmeldungen und der Informationen über seinen internen Zustand werden unterdrückt, wenn der LanRouter als Hintergrundprozess gestartet wird. Ansonsten werden sie entweder auf der Konsole oder in der Log-Ansicht des LanRouter-Fensters dargestellt.

In jedem Fall wird die eine Instanz der Klasse `LanRouter` erstellt, die die Hauptroutine des Routers beinhaltet. Der Programmablauf innerhalb der Hauptroutine ist in Abbildung 4.11 dargestellt. Nach dem Start werden die Listener-Threads, also der LanRouterXMLRPCServer und der LanRouterUDPListener gestartet. Ebenso wird der Broadcaster-Thread gestarstet.

Die versendeten Pakete des Broadcasters bestehen aus strukturiertem Text. Ein Paket hat fol-

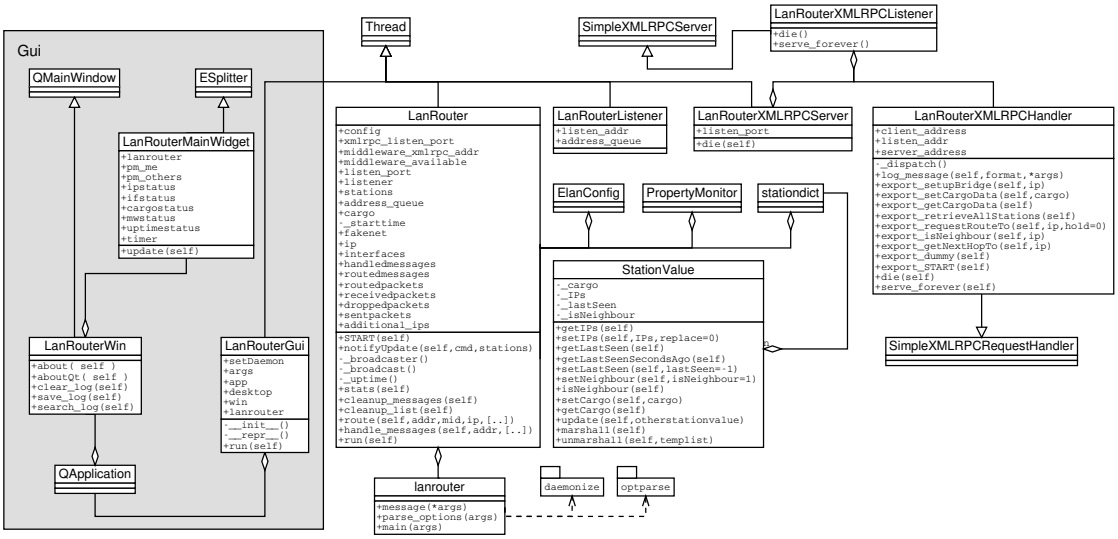

Abbildung 4.10: LanRouter-UML-Ansicht

gende Form:

```
msg = ( BROADCAST_PREFIX+'\r\n'
  + str(mid)+'\r\n'
  + str(ip)+'\r\n'
  + str(ttl)+'\r\n'
  + str(Hops)+'\r\n'
  + str(cargo) + '\r\n' )
```

Dabei ist das Broadcast-Präfix ein festgelegter vordefinierter String für LanRouter-Broadcasts (`'Elan Router Broadcast 0.4'`). Neben diesem enthält ein Paket eine Message-ID, die eine eindeutige Zuordnung des Pakets ermöglicht. Ebenso enthält das Paket die IP-Adresse des LanRouters, sowie die Zähler TTL und Hops. Über die TTL wird eine maximale Lebensdauer eines Pakets festgelegt. Hops bezeichnet die Anzahl der Stationen, die ein Paket weitergeleitet haben.

Um eingehende Pakete zwischen dem LanRouterListener-Thread und der Hauptschleife transportieren zu können, wird eine threadsichere Paket-Warteschlange, die durch das Python-Queue-Modul zur Verfügung gestellt wird, verwendet.

Nach dem Empfang eines Pakets wird über das Broadcast-Präfix validiert, ob das Paket ein gültiges LanRouter-Paket ist. Ein gültiges Paket wird über einen regulären Ausdruck in seine ursprünglichen Bestandteile zerlegt und in die Adress-Warteschlange eingereiht.

Ein Abruf eines Elements aus dieser Warteschlange wird durch das Queue-Modul blockiert, solange die Warteschlange leer ist. Sobald ein LanRouter-Paket empfangen wurde, kehrt der Aufruf zurück. Da der LanRouter auch seine eigenen Pakete empfängt, wird die Hauptschleife dennoch regelmäßig in den Intervallen durchlaufen, in denen der Router selber Pakete versendet (BROADCAST_SLEEP).

Ist ein eingehendes Paket nicht zu alt, ist also seine TTL noch größer als Null, wird das Paket verarbeitet, ansonsten wird es verworfen. Dazu wird es sowohl geroutet, als auch für die Stationslisten ausgewertet.

4.3.2 Nachrichtenverarbeitung

Die Verarbeitung von Nachrichten erfolgt in der Methode `handle_messages`. Die Nachricht wird in dieser Methode für die Stationsliste ausgewertet.

Wenn der Paket-Absender noch nicht in dieser Liste vorhanden ist, wird er jetzt hinzugefügt. Anschließend wird die Middleware über die neue Station informiert.

Der Hops-Wert wird analysiert um zu prüfen, ob eine Station als direkter Nachbar eingetragen werden soll. Ein Paket, das von einer benachbarten Station versendet wurde, kann nur einen Hop zurückgelegt haben. Ist die Zahl der Hops größer, dann kam es nicht von einem Nachbarn. Ebenso wird im StationValue der lastSeen-Wert, also der Zeitpunkt, an dem eine Station zuletzt gesehen wurde, mit einem aktuellen Zeitstempel aktualisiert.

Um ein mehrfaches Analysieren der Nachricht zu vermeiden wird über die Message-ID geprüft, ob ein Paket bereits bearbeitet wurde. Dazu wird dieses als Schlüssel mit dem lastSeen-Wert des Pakets in ein Dictionary eingetragen.

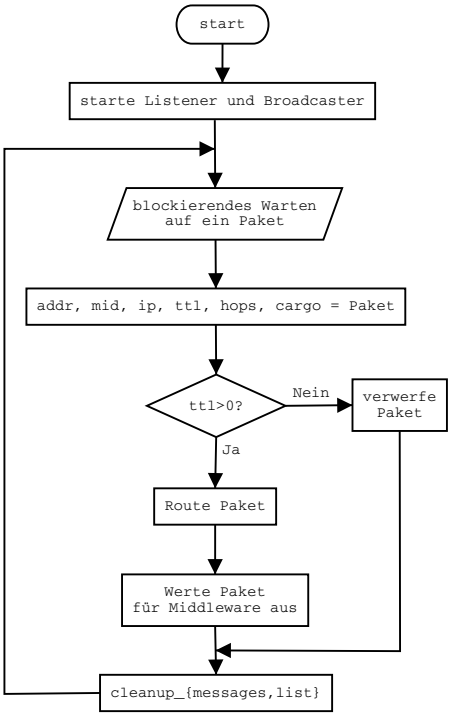

Abbildung 4.11: Programmfluß der Hauptroutine

4.3.3 Routing

Indem er seine Pakete als Broadcast versendet, flutet der LanRouter das Netzwerk. Andere LanRouter im Netz empfangen diese Pakete. Empfangene Pakete werden von einem LanRouter weitergeleitet, wenn er über mehr als eine Netzwerkschnittstelle verfügt. Dazu broadcastet er ein Paket erneut, nachdem er den Wert der TTL dekrementiert und die Anzahl der Hops inkrementiert hat. Dies erfolgt über alle verfügbaren Netzwerkschnittstellen, mit Ausnahme derjenigen, über die das Paket ursprünglich empfangen wurde.

Um ein mehrfaches Versenden von Paketen zu vermeiden, wird analog zur Nachrichtenverarbeitung, die Message-ID und ein Zeitstempel in einem Dictionary gespeichert. Vor dem Weiterleiten wird überprüft, ob das Paket schon enthalten ist. Sollte dies der Fall sein, wird ein Paket nicht nochmals versendet.

Das Routing funktionert über Grenzen von Subnets nur, wenn der Rechner, der die beiden Subnets verbindet, einen LanRouter laufen läßt. Ein fehlender weiterleitender Rechner genügt, um eine Unterbrechung des Weiterleitens auszulösen. Da das Routing nur ein Weiterleiten von Paketen innerhalb der LanRouter-Anwendung darstellt, kann nicht über die Grenzen eines LANs hinaus geroutet werden. Um solche Entfernungen überwinden zu können, wurde die Brückenfunktionalität implementiert. Zwei LanRouter aus verschiedenen Netzen senden sich die Pakete als Unicasts direkt und ermöglichen somit eine Verbindung der beiden Netzwerke.

Um eine Brücke zu einem entfernten LanRouter zu intitalisieren, muß die IP-Adresse vom Benutzer in die Konfigurationsdatei eingetragen werden. Der Router liest diese nach dem Programmstart aus und speichert die dort gefunden IP-Adressen in einer Liste (`additional_ips`). Wenn ein Paket weitergeleitet wird, sendet der Router nun, zusätzlich zu seinem normalen Broadcast, an alle in der Liste abgelegten IP-Adressen einen Unicast. Damit er auch Pakete in Rückrichtung von diesen Stationen bekommt, muß die Brückenfunktion an den jeweilgen Stationen ebenfalls aktiviert werden. Dies konfiguriert der Router nach dem Einlesen der Stationen automatisch. Er sendet dem Router an der jeweiligen entfernten Station den XML-RPC-Befehl `setupBridge`, der als Argument die eigene IP-Adresse beinhaltet. Auf diesen Befehl hin fügt der entfernte Router die übergebene IP-Adresse seiner Liste der zusätzlichen IP-Adressen hinzu.

Aufräumarbeiten

Um die Dictionaries, die bearbeitete bzw. geroutete Pakete kennzeichnen, nicht über alle Grenzen wachsen zu lassen, werden diese in regelmäßigen Abständen bereinigt. Dazu werden alle Message-IDs, deren Zeitstempel ein gewisses Alter erreicht haben, gelöscht. Das maximale Alter wird im Falle der bearbeiteten und Pakete durch die Konstanten `TIMEOUT_MESSAGES` und im Falle der weitergeleiteten Pakete durch die Konstante `TIMEOUT_LASTSEEN` bestimmt.

Ebenso wird die Stationsliste bereinigt. Da ein Router Stationen durch deren Broadcasts erkennt, kann er das Entfernen einer entfernten Station nur dadurch ermitteln, daß eine gewisse Zeitlang keine Broadcasts mehr von dieser Station empfangen wurden. Diese Zeitspanne wird ebenfalls durch die Konstante `TIMEOUT_LASTSEEN` bestimmt. Ist die Zeitspanne überschritten, werden diese Stationen aus der Stationsliste gelöscht.

4.3.4 XML-RPC-Schnittstelle

Die XML-RPC-Schnittstelle des Routers besteht aus drei Komponenten. Durch die Klasse `LanRouterXMLRPCServer` wird ein XML-RPC-Server implementiert. Dieser wird von Python durch das Modul `SimpleXMLRPCServer` zur Verfügung gestellt. Durch dieses wird der `LanRouterXMLRPCListener` als eigenständiger Thread gestartet und wartet am vordefinierten TCP / IP-Port 5558 auf Aufrufe. Eingehende Aufrufe werden jeweils einem Handler, dem `LanRouterXMLRPCRequestHandler`, übergeben, um sie abzuarbeiten. Dadurch kann der Server weiter Aufrufe annehmen, während ein vorangegangener Aufruf noch abgearbeitet wird.

Durch den Handler sind die möglichen XML-RPC-Aufrufe definiert. So ist es z.B. über den Aufruf von `setCargoData` möglich, dem Router die GuID zu übergeben. Dieser Aufruf wird von der Middleware durchgeführt, wenn die GuID aktualisiert wird. Über die Methode `retrieveAllStations` kann sie den Router anweisen, ihr alle Stationen im Netz zu senden. Entfernte LanRouter können die Methode `setupBridge` aufrufen, um die schon erwähnte Brückenfunktionalität zu initialisieren. Die vollständig zur Verfügung gestellte XML-RPC-Schnittstelle wird im Anhang A.4.1 besprochen.

Empfangene Daten verändert der XML-RPC-Server direkt in der Instanz des LanRouters, ohne z.B. eine threadsichere Warteschlange zu verwenden. Da er der einzige Thread ist, der auf die Cargo-Daten schreibend zugreift, ist dieser Zugriff threadsicher. Es entstehen durch das

regelmäßige Neuversenden der Cargo-Daten ebenfalls keine Probleme durch Nebenläufigkeit beim Lesen der Daten.

4.3.5 Optionale GUI

Knotenansicht

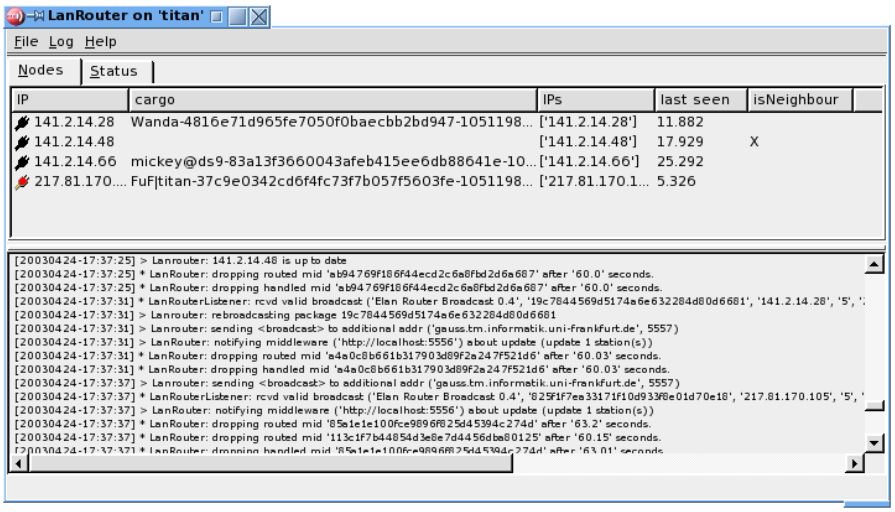

Abbildung 4.12: LanRouter GUI: Knotenansicht

Der LanRouter verfügt über eine optionale GUI, die primär zu Evaluierungszwecken der PyQt-Schnittstelle, also der Schnittstelle zwischen Python und der Qt-Bibliothek entstand. Dabei sollte die GUI sowohl auf mobilen Geräten, als auch auf Desktop-Rechnern einzusetzen sein. Sie besteht aus einer Knoten- und einer Statusansicht.

In der In der Knotenansicht des LanRouters werden alle erkannten Stationen des Netzwerks angezeigt. In Abbildung 4.12 sind dies vier Stationen. Die eigene Station ist durch andere Farbe gekennzeichnet. Die Station mit der IP-Adresse `141.2.14.48` ist als direkter Nachbar gekennzeichnet. Sie transportierte also keine Cargo-Daten, was darauf schließen läßt, daß auf dieser Station keine E.L.A.N-Anwendung läuft. Sie ist ein direkter Nachbar der eigenen Station, weil beide über die Brückenfunktion verbunden sind und daher Pakete über einen Hop weitergeleitet werden. Hop ist hier als die Entfernung zwischen zwei LanRoutern zu verstehen, auf IP-Ebene legen die versendeten Pakete mehrere Hops zwischen den beiden Stationen zurück.

Der untere Bereich der Ansicht enthält ein Log-Fenster, das alle Debugging-Ausgaben des Routers enthält.

Statusansicht

Abbildung 4.13 zeigt die Statusansicht des Lanrouters. Sie stellt Informationen über die Laufzeitumgebung des Routers dar. IP zeigt die aktuell verwendete IP-Adresse an. Interfaces ist eine

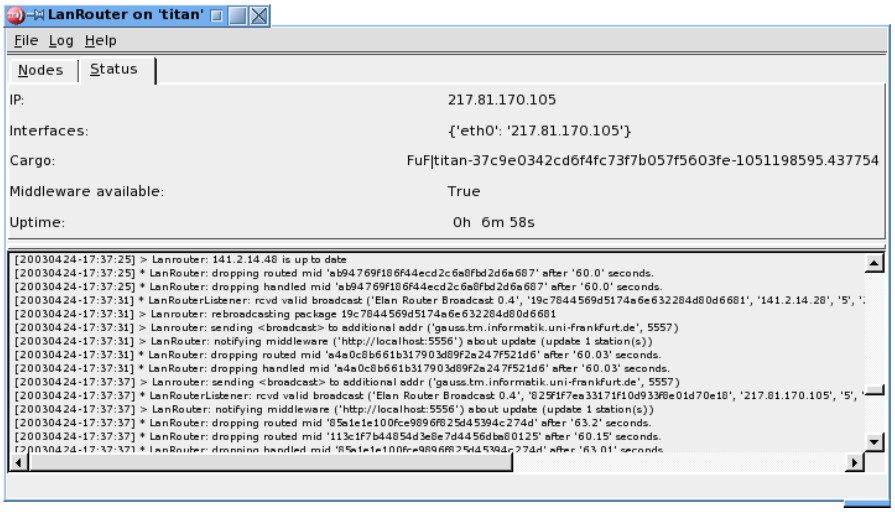

Abbildung 4.13: LanRouter GUI: Statusansicht

Darstellung der verfügbaren Netzwerk-Schnittstellen, bestehend aus dem Namen der Schnittstelle und der zugeordneten IP-Adresse. Dieser Wert ist interessant, wenn mehrere Netzwerk-Schnittstellen vorhanden sind. Der Wert von Cargo zeigt den aktuellen Inhalt der Cargo-Daten an. Dies ist die Low-Level-Awarenessinformation in Form der GuID der Station. Uptime schließlich zeigt an, wie lange der Router bereits läuft.

4.3.6 Handshake Middleware und Router

Die Routing-Schicht läuft unabhängig von der Middleware-Schicht. Die Middleware ist jedoch auf sie angewiesen, da sie von ihr die Stationslisten als Awareness-Information erhält. Das Senden dieser Listen ist von Seiten des Routers aus nur dann notwendig, wenn eine Middleware-Schicht läuft. Um unnötiges Versenden zu vermeiden, ist ein Handshake zwischen beiden Schichten nötig. Dieses ermöglicht es insbesondere, daß beide Schichten unabhängig voneinander neu gestartet werden können. Die jeweils andere Schicht wird durch den Handshake zurückgesetzt.

Abbildung 4.14 zeigt ein UML-Sequenzdiagramm des Handshakes. Ist eine Schicht aktiv, wird dies durch einen vertikalen, weißen Balken angezeigt.

Die Middleware versucht kontinuierlich durch Senden des XML-RPC-Aufrufs START auf den Router zuzugreifen. Dies ist nötig, da ein Router für die Funktion der Middleware unbedingt notwendig ist. Solange der Aufruf scheitert, verbleibt sie im Zustand „kein Router vorhanden“. Gelingt der Zugriff, wechselt sie in den Zustand „Router vorhanden“.

Scheitert ein beliebiger XML-RPC-Aufruf im Zustand „Router vorhanden“ wechselt die Middleware wieder in den „kein Router“-Zustand und damit wieder in die Warteposition zurück.

Der Router dagegen wartet so lange im Zustand „keine Middleware vorhanden’", bis er von der Middleware einen START-Aufruf von dieser erhält. Er wechselt dann in den Zustand „Middleware vorhanden“. In diesem Zustand sendet der Router der Middleware die Listen der Stationen per XML-RPC, bis ein Fehler auftritt. Dies ist z.B. der Fall, wenn eine Middleware beendet

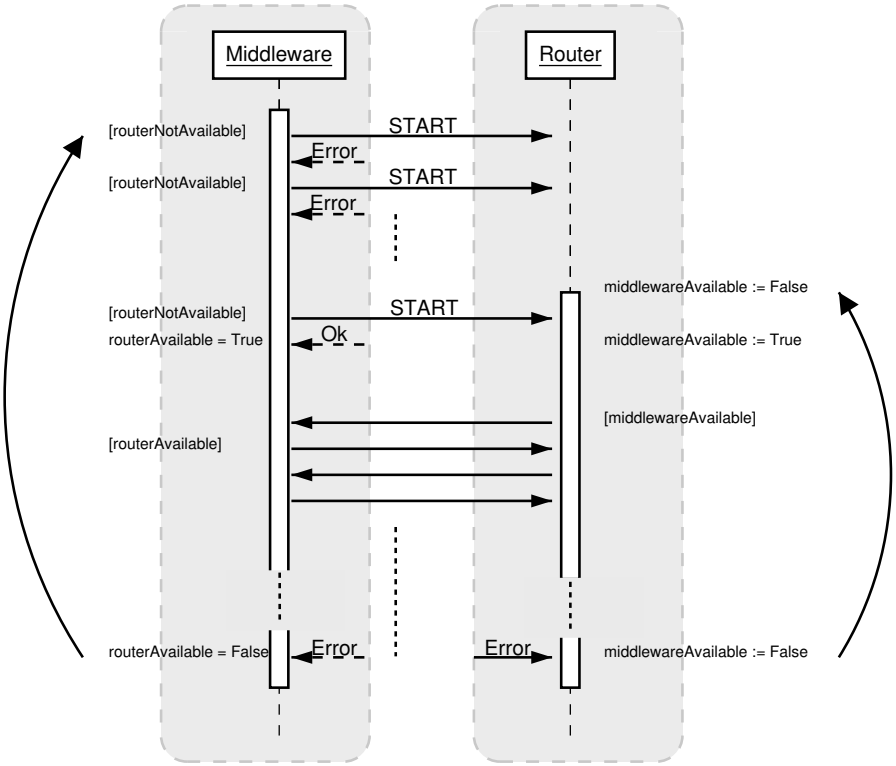

Abbildung 4.14: Middleware-Routing-Handshake

wurde. Durch einen solchen Fehler wechselt er wieder in den Zustand „keine Middleware vorhanden“.

Durch Aufruf von START im Zustand „Middleware vorhanden“ signalisiert dem Router, daß die Middleware neu gestartet wurde.

Wird auf der anderen Seite ein Router neu gestartet, erkennt die Middleware dies entweder indirekt dadurch, daß sie keine Stationsinformationen mehr erhält oder dadurch, daß ein XML-RPC-Aufruf an den Router fehlschlägt.

5 Umsetzung und Implementierung von Evaluationskomponenten

Um die vorgestellte Middleware und ihre Funktionalität im Rahmen des E.L.A.N-Projekts zu testen, werden in diesem Kapitel eine Reihe von E.L.A.N-Komponenten zur Demonstration der Funktionalität vorgestellt.

Es wird zuerst vorgestellt, wie eine E.L.A.N-Komponente allgemein aufgebaut ist. Anschließend werden einige wichtige Komponenten des E.L.A.N-Systems vorgestellt. Abschließend folgen drei Szenarien zur Nutzung der vorgestellten verschiedenen Diensttypen und die Beschreibung der dafür implementierten Dienstkomponenten.

5.1 Das Komponentenmodell

Das lego-Framework verfügt über eine flexible Architektur zur Darstellung von Plugins, innerhalb des Hauptfensters der E.L.A.N-Anwendung. Diese Plugins werden in Laschen im Hauptfenster ausgeführt. Der Benutzer kann im Hauptfenster einzelne Komponenten durch Anklicken der jeweiligen Lasche auswählen.

Abb. 5.1 zeigt das E.L.A.N-Hauptfenster. In diesem ist die Komponentenansicht aktiviert, die selbst eine E.L.A.N-Komponente ist. Die Komponentenansicht dient zur Anzeige der im System laufenden Plugins. Die Darstellung der Informationen über die Komponenten erfolgt in tabellarischer Form. Dargestellt werden das jeweilige Komponentensymbol, der Namen der Komponente, ihre Version und der Names Autors. Die Spalte „Dienst“ zeigt an, ob eine Komponente ein Dienst ist oder nicht. Im abgebildeten Fall von `FileSharing`-Dienst, `converter`, `identd` und `messageService` handelt es sich jeweils um einen Dienst, was mit „X“ aufgezeigt wird.

Die Spalte „Benutzt“ dient zur Anzeige, ob eine diensterbringende Komponente von einer entfernten Station verwendet wird. Die Spalten GUI zeigt zusätzlich an, ob es sich um eine grafische Komponente handelt. Nur grafische Komponenten erhalten eine Komponentenlasche. Nichtgrafische Komponenten laufen im Hintergrund und werden nur von der Komponentenansicht dargestellt. Die Spalte „Schleife“ zeigt an, ob eine Komponente eine Hauptschleife hat, die dauerhaft durchlaufen wird oder ob die Komponente nur dann aktiv ist, wenn sie z.B. Nachrichten erhält.

Jede Komponente kann das Hauptmenü um eigene Menüpunkte erweitern, im vorliegenden Fall sind das die Menüpunkte „Query“ und „Show“. Die im oberen rechten Bereich des Hauptfensters vorhandenen Schaltflächen dienen zum Schließen und Neuladen der Komponenten. Ebenso kann jede Komponente einen eigenen Konfigurationsdialog und eine Anzeige von Informationen über sie zur Verfügung stellen. Da dies für die Komponentenansicht nicht gilt, sind die dafür vorgesehenen Knöpfe deaktiviert.

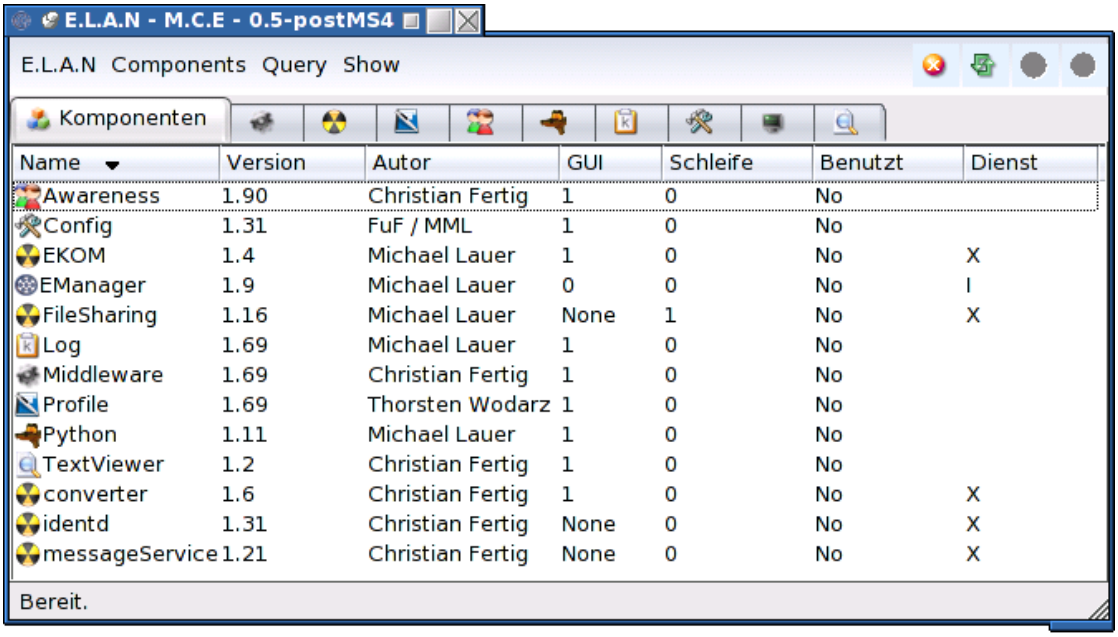

Abbildung 5.1: Komponentenansicht

Damit Komponenten des Framework als solche erkannt werden, müssen sie gewisse Vorgaben erfüllen. Innerhalb der E.L.A.N-Verzeichnisstruktur muß eine Komponente als Python-Modul im Komponentenverzeichnis liegen. Eine Komponente wird von der Klasse `EComponent` abgeleitet, die als abstrakte Basisklasse elementare Komponentenmethoden und Variablen vorgibt. Die in der Komponentenansicht dargestellten Informationen sind über statischen Attribute `name`, `author` und `gui` definiert. Über den Wert `service` kann bestimmt werden, ob eine Komponente einen Dienst zur Verfügung stellt. Entsprechend den Definitionen zu Diensten (Kapitel 3.2.2) werden hierzu die Werte `specified`, `unspecified-implict` und `unspecified-explicit` unterschieden.

Über den Wert des Attributs `spec` wird die Versionsnummer der Komponentenspezifikation definiert, die zusätzlich prüft, ob ein Python-Modul eine gültige Komponente beinhaltet. Neben den vorgegeben Attributen werden aus der Komponentendefinition durch die Basisklasse eine Reihe von Methoden definiert, die das Verhalten der Komponente beeinflussen. So werden nach dem Start der Komponenten Initialisierungen durch den Aufruf der Methoden `bootNonGUI` und, bei grafischen Komponenten, zusätzlich `bootGUI` durchgeführt. Durch Überladen der Methoden `canConfigure`, `canClose` und `canReload` kann der Entwickler das Aktivieren der in Abbildung vorgestellten Komponentenknöpfe festlegen. Im Hauptmenü zusätzlich anzuzeigende Menüpunkte werden über die Methode `getMenuStructure` abgefragt. Definiert eine Komponente die Methode `run`, so wird sie als Komponente gestartet, die eine eigene Hauptschleife enthält. Diese wird durch einen eigenen Thread realisiert.

Die Klasse `EComponent` ist unter anderem von `ECommunicationComponent` abgeerbt. Somit nehmen alle Komponenten automatisch an der ECom-Kommunikation teil (vgl. 4.2.1).

5.2 Die Awareness-Komponente: awarenessView

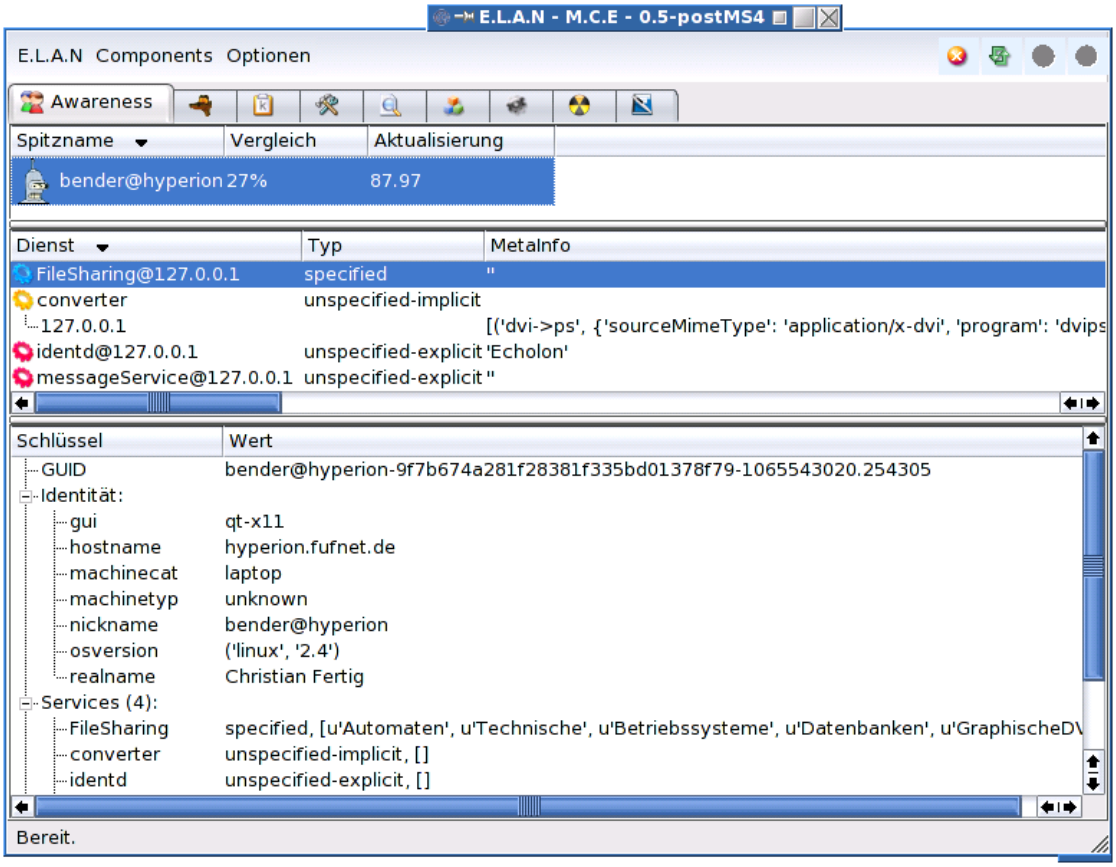

Abbildung 5.2: Die awarenessView-Komponente

Die `awarenessView` stellt die zentrale Awareness-Ansicht zur Verfügung. Diese dient als Benutzer- bzw. Dienstansicht. Sie stellt die für den Benutzer interessanten Kollaborationspartner und Dienste dar. Aktionen, wie das Eröffnen eines Chats mit einem entfernten Benutzer oder das Nutzen eines durch ihn angebotenen Dienstes sollen über sie gestartet werden.

5.2.1 Aufbau von awarenessView

Abb. 5.2 zeigt die dreigeteilte Ansicht der awarenessView-Komponente. Im oberen Bereich sind mögliche Kollaborationspartner zu sehen, also Personen mit interessanten Wilidips. Mögliche Aktionen wie das Eröffnen eines Chats sind, sofern die jeweiligen Dienste auch auf dem Zielgerät verfügbar sind, über das Menü bzw. das Kontextmenü erreichbar. Zu jedem Benutzer kann neben dem Spitznamen das Bild des Benutzers angezeigt werden, sowie das Ergebnis der Korrelationsfunktion in Prozent. Die Spalte „Aktualisierung“ zeigt in Sekunden an, wann der Benutzer sein WiLiDip zuletzt verändert hat.

Im mittleren Bereich des Ausgabefensters werden die im Netz verfügbare Dienste dargestellt. Dabei sind die verschiedenen Diensttypen durch farblich unterschiedliche Symbole gekennzeichnet. Die impliziten Dienste, hier der Konverter, müßten nicht dargestellt werden, da sie nicht durch Benutzereinfluß ausgeführt werden können. Diese Informationen wäre in der `middlewareActivity`-Komponente besser plaziert, doch kann diese zur Zeit noch keine Dienste darstellen.

Der untere Bereich ist für zusätzliche Informationen über einen der angewähltem Kollaborationspartner vorgesehen, z.B. derjenigen, die beim Übertragen des Benutzerbildes zusätzlich vom identd-Dienst, der später erläutert wird, mitgeliefert wurden. Dazu gehören die Informationen über die Geräteklasse und über das Betriebssystem des vom Benutzer verwendeten Geräts.

5.2.2 Steuerung der awarenessView Komponente

Die awarenessView wird durch lego-Nachrichten von der Middleware, bzw. deren WiLiDip-manager-Komponente gesteuert.

Über die Nachricht `awareness_updateUser` wird die awarenessView über einen neuen „interessanten“ Benutzer über dessen WiLiDip informiert. Sie fügt diesen Benutzer ihrer Ansicht hinzu und fordert vom identd der entfernten Station des Benutzers weitere Informationen an. Sobald diese eingetroffen sind, wird das Benutzerbild angezeigt. Innerhalb der Informationsansicht werden auch alle anderen übertragenen Daten angezeigt. Die Nachricht `awareness_removeAllUsers` setzt die awarenessView zurück, d.h. alle Informationen werden gelöscht. Der Eintrag einzelner Benutzer wird über die Nachricht `awareness_removeUser` gelöscht.

Analog werden Dienste über die Nachricht `awareness_updateService` hinzugefügt und über `awareness_removeExplicitServices` werden alle expliziten Dienste zu einem Benutzer gelöscht. `awareness_removeImplicitService` löscht einen impliziten Dienst einer Station. Eine Beschreibung aller lego-Nachrichten befindet sich in A.1.

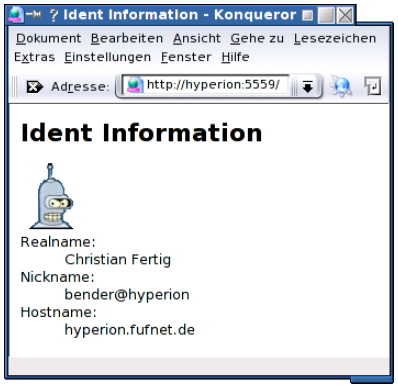

Abbildung 5.3: Der WWW-Server des identd

5.3 Die Identifikationskomponente: identd

Die Identd-Komponente versorgt andere Komponenten mit Informationen über den Benutzer eines entfernten Systems.

Ihre einzige Aufgabe ist es, auf die extIn-Nachricht `requestIdent` zu warten und diese mit einer `updateIdent`-Nachricht zu beantworten. Die aktuellen Benutzerinformationen sammelt der identd aus der Konfiguration und aus Framework-Funktionen. Er speichert sie in einem Schlüssel-Wert-Dictionary. Im laufenden Betrieb ist der identd Empfänger der `updateNickname`-, `updateRealname`- und `updateUserPixmap`-Nachrichten der Middleware. Über diese hält er seine Informationen auf aktuellem Stand.

Der identd diente während der Entwicklung als Beispielkomponente für die Interkomponenten-Kommunikation und zum Entwurf der extIn- / extOut-Schnittstelle (vgl. 4.2.3)

Der identd beinhaltet auch einen WWW-Server, der optional gestartet werden kann. Dessen Implementierung war durch das in Python enthaltende Modul `BaseHTTPServer` problemlos möglich. Der voreingestellte Port zum Beantworten von HTTP-Anfragen über Benutzerinformationen ist Port 5559. Abbildung 5.3 zeigt die Ausgabe einer solchen Anfrage an den identd auf dem Rechner „hyperion“ im KDE-Browser Konqueror. Die Implementierung des WWW-Servers wurde durchgeführt, um während der Entwicklung der Komponenten-Architektur prüfen zu können, ob der Dienst noch verfügbar ist. Der identd entstand als erste diensterbringende Komponente des E.L.A.N-Systems.

5.4 Die Debuggingkomponente middlewareActivity

Die middlewareActivity ist eine Debugging-Komponente zur Darstellung interner Zustände und des Kommunikationsflusses innerhalb der Middleware.

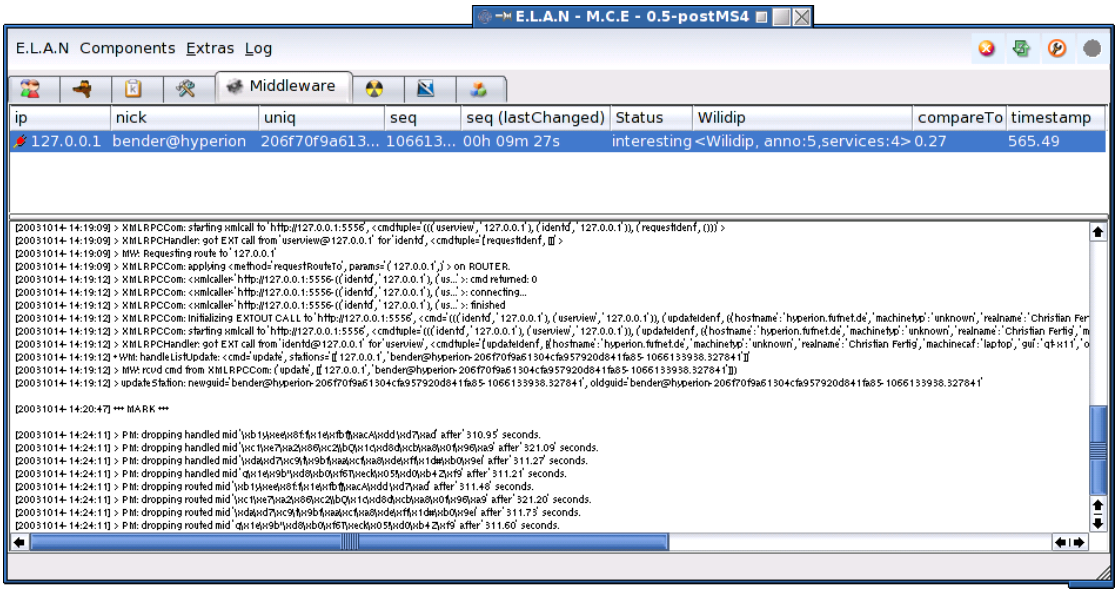

Abbildung 5.4: Die middlewareActivity-Komponente

Abb. 5.4 zeigt eine Ansicht von ihr. Im oberen Bereich werden alle Stationen des Netzes dargestellt. Dazu gehören auch „uninteressante“ Teilnehmer und die eigene Station. In den einzelnen Spalten werden neben den Werten, die auch die Awareness-Komponente anzeigt, die GuID und der Status innerhalb der Zustandsmaschine des WiLiDipmanagers angezeigt. Die Sequenznummer der GuID wird als Zeitstempel und umgerechnet als Zeitpunkt der letzten Änderung angezeigt. Der untere Bereich enthält die Debugging-Ausgaben der Middleware.

Auf den markierten Einträgen können über das Kontextmenü Aktionen durchgeführt werden. Dies beinhaltet alle Aktionen, die auch durch die Awareness-Komponente durchgeführt werden könnten, wie z.B. das Versenden von Chat-Nachrichten an die entfernten Stationen. Darüber hinaus ist es möglich, die Brückenfunktion der Protokollmaschine nachträglich zu aktivieren.

Weiterhin können direkt WilidipRequest-Nachrichten an entfernte Stationen versendet werden. Eine Kernfunktion ist der direkte Eingriff in die Statusmaschine des WiLiDipmanagers. Diese ermöglicht z.B. das Zurücksetzen des Status auf „unbekannt“. Hier läßt sich beobachten, wie die Middleware den Zustand wieder korrigiert. Für Tests innerhalb des Festnetzes ist eine Namensauflösung der IP-Adresse einer Station per DNS möglich.

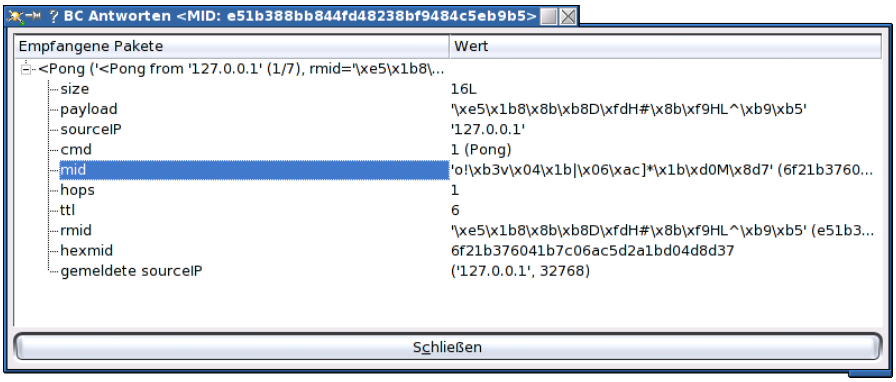

Abbildung 5.5: Pong-Antworten auf ein Ping

Ohne daß eine spezielle Station markiert ist kann das eigene WiLiDip jederzeit gebroadcastet werden. Ebenso ist es zur Evaluierung der Routingfunktionen der Protokollmaschine möglich, ein Ping-Paket ins Netz zu versenden. Alle anderen Stationen sollten daraufhin mit einem Pong antworten. Pongs werden von der Protokollmaschine empfangen und über die lego-Nachricht `receivedPong` dem Framework mitgeteilt. Empfängt die middlewareActivity diese Nachricht, öffnet sie ein Anzeigefenster, in dem der Paketinhalt des Pongs analysiert werden kann (siehe Abb. 5.5). Die Pongs werden einem Anzeigefenster über die Message-ID des Absendepakets zugeordnet.

Im unteren Bereich der `middlewareActivity` werden sämtliche Debugging-Ausgaben der Middleware gesammelt. Es läßt sich hier anschaulich verfolgen, was im System passiert. Außerdem läßt sich die Logdatei leeren, abspeichern und mit einer Markierung versehen, um eine bestimmt Stelle schnell wiederfinden zu können. Das Logfenster verfügt über einen Cache, so daß keine Nachrichten zwischen dem Initialisieren der Komponente und dem Start des grafischen Teils der Komponente verloren gehen können.

Die middlewareActivity wird von der Middleware hauptsächlich über zwei lego-Nachrichten gesteuert: `fullWilidipList` hat die vollständige interne Darstellung der WiLiDips als Argument und `middlewareDebug` die Nachrichten für die Logdatei. Zusätzlich horcht sie noch auf einige Nachrichten. So muß z.B. die GuID zum Erkennen der eigenen Station und das aktuelle WiLiDip für den expliziten Broadcast aktuell gehalten werden.

5.5 Die Konfigurationskomponente elanConfig

Über die in Abb. 5.6 dargestellten `elanConfig`-Komponente läßt sich das Verhalten der Middleware beeinflussen.

Neben den Einstellungen zu Name und Spitzname des Benutzers und dessen Pictogramm gibt es die Möglichkeit zu bestimmen, ab welchem Schwellenwert ein möglicher Kollaborationspartner als interessant eingestuft wird. Dies geschieht über die Eingabe eines Prozentwertes oder über einen Schieberegler. Dieser Wert wird von der Middleware als Schwellenwert für die WiLiDip-Vergleiche verwendet.

Die elanConfig-Komponente sendet veränderte Einstellungen per lego-Nachricht an das Framework und die Middleware, die sich daraufhin durch neue Vergleiche und Anpassen der GuID neu konfiguriert.

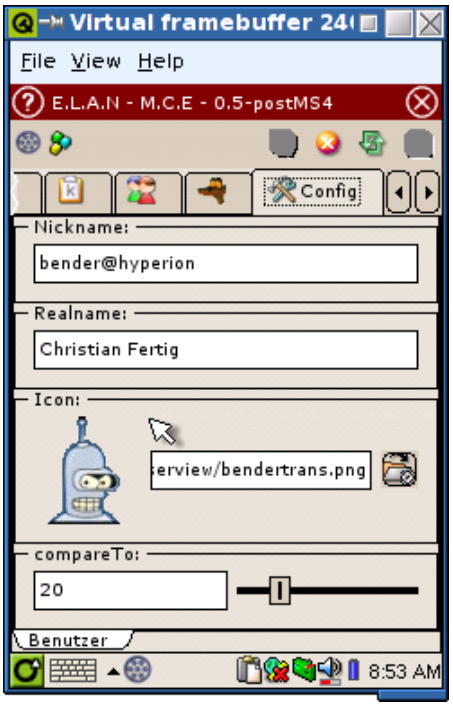

Abbildung 5.6: Die E.L.A.N Config-Komponente

5.6 Nutzung der unterschiedlichen Dienste

Dieser Abschnitt beschreibt die Nutzung der in Kapitel 3.2.2 vorgestellten verschiedenen Diensttypen. Für jeden Diensttyp wurden Beispielkomponenten entworfen. Für die Nutzung eines spezifischen Dienstes wurde die `FileSharing`-Komponente entwickelt. Die Nutzung eines unspezifiziert-expliziten Dienstes erfolgt beim Nutzen des Nachrichtendienstes (`messageService`). Ein Beispiel für eine implizite Dienstnutzung folgt mit dem `converter`-Szenario.

5.6.1 Nutzung spezifischer Dienste: Filesharing

Der Filesharing-Dienst wurde im Rahmen der Entwicklung des Frameworks von Michael Lauer als Prototyp eines spezifizierten Dienstes entwickelt. Er ermöglicht das Verknüpfen von Dokumenten mit einem Wissensgebiet. Diese Verknüpfung wird durch den Benutzer im Profil-Editor vorgenommen.

Einem entfernten Benutzer wird ein an ein WiLiDip verknüpfter spezifischer Dienst durch seine lokale Awareness-Ansicht dargestellt. Um ihn nutzbar zu machen, stellt diese den Menüpunkt „Do something“ zur Verfügung. Ein Ausführen des Menüpunkts startet die Dienstnutzung.

Während ein Dienst auf der entfernten Station unter Umständen im Hintergrund arbeitet, kann seine grafische Nutzung nur durch eine Komponente, die auf der lokalen Station läuft, erfolgen. Dazu muß eine Zugriffskomponente, der sogenannte ServiceAccessor, ausgeführt werden, der dem Benutzer die möglichen nutzbaren Funktionen darstellt.

Um diesen Vorgang zu starten, sendet die awarenessView-Komponente nach Auswählen des Menüpunktes die lego-Nachricht `ServiceUsageRequest`, an das eigene Framework. Dieses kommuniziert über die extIn / extOut-Schnittstelle mit dem entfernten Framework, welches

den Dienst anbietet. Es sendet diesem eine `ServiceInfoRequest`-Nachricht, um Informationen über den entfernten Dienst abzurufen. Das entfernte Framework beantwortet diese Anfrage über die Nachricht `ServiceInfoResponse`. Anhand der übermittelten Informationen kann das Framework erkennen, ob der ServiceAccessor für den entfernten Dienst bereits verfügbar und nicht veraltet ist. Sollte dies nicht der Fall sein, wird die Nachricht `ServiceUsageRequest` an das entfernte Framework gesendet. Dieses überträgt dann innerhalb einer `ServiceUsageResponse`-Antwort den aktuellen Quelltext der Zugriffskomponente. Dieser wird innerhalb der E.L.A.N-Verzeichnisstruktur abgespeichert.

Nachdem sichergestellt ist, daß die Zugriffskomponente vorhanden ist, wird diese vom lokalen Framework geladen und ausgeführt.

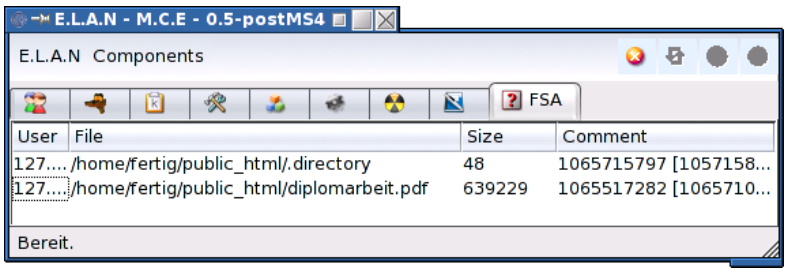

Abbildung 5.7: Der Filesharing-Accessor

Im Falle des Filesharing-Dienstes kann der Benutzer nun Dateien, die ein entfernter Benutzer freigegeben hat, herunterladen. Abb. 5.7 zeigt einen Filesharing-Accessor, über den ein Benutzer die Datei `diplomarbeit.pdf` zum Download anbietet.

Die aktuelle Version des Filesharing-Accessors ermöglicht zur Zeit noch kein Herunterladen von Dateien. Sie stellt die verfügbaren Dateien nur dar. Eine Erweiterung, um Dateien herunterladen zu können, ist in Vorbereitung.

5.6.2 Nutzung unspezifiziert-expliziter Dienste: messageService

Der Nachrichtendienst `messageService` dient als Referenzimplementierung eines unspezifiziert-expliziten Dienstes. Der messageService läuft als grafische Komponente im Hintergrund, d.h. ohne eine eigene Komponentenansicht im Hauptfenster. Nur bei Aktivität, also wenn entweder der Benutzer einer entfernten Station eine Nachricht schicken möchte oder eine Nachricht einer entfernten Station eingetroffen ist, wird ein Nachrichtenfenster geöffnet (siehe Abb. 5.8).

Der Nachrichtendienst ist zur Zeit die einzige Komponente, die mit dem gleichen Komponententyp an der entfernten Station kommuniziert. Es gibt also keine Unterscheidung von Client und Server, wie z.B. bei der Kommunikation zwischen `awarenessView` und `identd`. Zwei Benutzer kommunizieren über ihre jeweiligen gleichberechtigten `messageService`-Komponenten.

Der messageService kann mehrere Nachrichtenfenster verwalten, um mit verschiedenen Benutzern gleichzeitig kommunizieren zu können. Eine eindeutige Zuordnung von Nachrichten zu einem Fenster geschieht über eine Message-ID. Diese wird jeder Nachricht, die zu einem Fen-

ster gehört, mitgegeben. Antwortet ein Empfänger auf eine Nachricht wird die selbe Message-ID verwendet.

Möchte ein Benutzer einem anderen Benutzer eine Nachricht senden, so wird er dies üblicherweise in der Benutzer-Ansicht, der `awarenessView`, durch die Funktion „Nachricht senden“ einleiten. Diese Funktion steht nur zur Verfügung, wenn der entfernte Benutzer einen messageService laufen läßt. Die awarenessView schickt nun die lego-Nachricht `sendMessage`, die der lokale Nachrichtendienst empfängt. Dieser erstellt nun eine neue eindeutige Message-ID und öffnet ein Nachrichtenfenster, dem er diese ID zuordnet. Der Benutzer kann nun Nachrichten verfassen. Er versendet diese durch Drücken des „Nachricht versenden“-Knopfes. Der geschriebene Text wird dadurch in das Nachrichtenfenster übertragen und anschließend (über ein Qt-Signal) dem messageService übergeben.

Abbildung 5.8: Die E.L.A.N-message-Service-Komponente (Chat)

Der messageService sendet die neue Nachricht nun über die extOut-Nachricht `SendMessage` an die Empfängerstation. Deren messageService wartet auf eingehende extIn-Nachrichten und empfängt diese Nachricht. Er öffnet nun entweder ein der Message-ID zugeordnetes existierendes Nachrichtenfenster oder erstellt ein neues und ordnet diesem die Message-ID zu. Der Eingang einer neuen Nachricht wird dem Benutzer durch einen Klang und durch Abbilden der Nachricht im Textfenster signalisiert. Der entfernte Benutzer kann anschließend in diesem Nachrichtenfenster antworten, was den beschriebenen Sendevorgang in umgekehrter Richtung wiederholt. Sollte der Absender der ursprünglichen Nachricht sein Nachrichtenfenster bereits geschlossen haben, so wird dieses automatisch wieder geöffnet, da in jeder versendeten Antwort-Nachricht die gleiche Message-ID verwendet wird.

Die Nachrichtenfenster funktionieren als eigenständige Widgets, d.h. sie besitzen teilweise eigene Funktionalität, wie z.B. das optionale Abspielen eines Signals über den eingebauten Lautsprecher. Auch das Einfärben und die Formatierung der Texte, sowie das Voranstellen der Uhrzeit erfolgt eigenständig, ebenso die History-Funktion, also das Speichern vorangegangener Nachrichten. Um eine Nachricht in der Ansicht hinzuzufügen, muß der Nachrichtendienst nur die vom Nachrichtenfenster zur Verfügung gestellte Methode `append` aufrufen.

Neben den obigen Aufgaben kann der Nachrichtendienst noch auf das Ändern des eigenen Spitznamens (`updateNickname`) reagieren (indem er neue Nachrichten nunmehr mit dem neuen Namen als Absender verschickt) und auf die `awareness_removeUser`-Nachricht, die eigentlich für die awarenessView bestimmt ist und von der Middleware verschickt wird, wenn eine Station das Netz verlassen hat. In diesem Falle wird eine entsprechende Nachricht in das Nachrichtenfenster eingefügt, um den Benutzer zu informieren, daß sein Chatpartner nicht mehr im Netz ist.

5.6.3 Nutzung unspezifiziert-impliziter Dienste: converter/textviewer

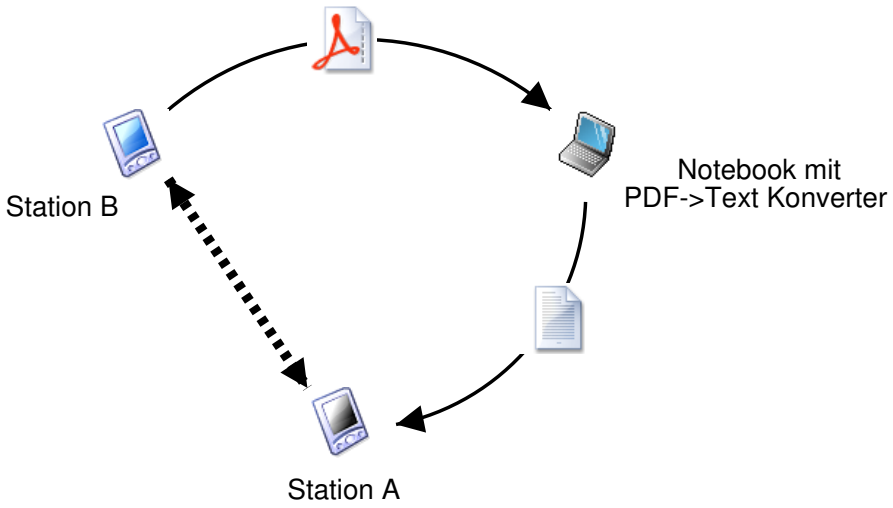

Abbildung 5.9: Implizite Benutzung eines Konverterdienstes

Das in Abb. 5.9 vorgestellte Szenario zeigt zwei PDAs, deren Benutzer miteinander kommunizieren. Benutzer **A** stellt Benutzer **B** ein PDF-Dokument über seinen FileSharing-Dienst (vgl. 5.6.1) zur Verfügung. Benutzer **B** lädt diese Datei auf seinen PDA. Dabei registriert die Middleware seiner Maschine, daß sein Gerät nicht die Ressourcen oder keinen Betrachter hat, um komplexe PDF-Dateien darstellen zu können. Sie weiß jedoch, daß reiner Text über die `TextView`-Komponente angezeigt werden kann. Daher sucht daher eine Möglichkeit die PDF-Datei in Text umzuwandeln. Die Middleware kennt den unspezifiziert-impliziten `converter`-Dienst. Auf einem Notebook im Netz läuft dieser und ist in der Lage, die Konvertierung durchzuführen. Die Middleware veranlaßt nun das entfernte Notebook die Datei abzuholen und zu konvertieren. Anschließend wird ihr vom Notebook die erstellte Textdatei zugesendet.

Um dieses Szenario umzusetzen, wurden die `converter`- und die `TextViewer`-Komponenten entwickelt.

converter-Komponente

Die converter-Komponente kann die ihr übergebenen Dokumente von einem Typ in einen anderen umwandeln. Sie nutzt dazu auf der Maschine vorhandene Hilfsprogramme.

Sie soll als nichtgrafische Komponente, ähnlich wie der in 5.3 vorgestellte `identd`, im Hintergrund laufen. Zu Evaluierungszwecken hat sie zur Zeit noch eine grafische Ausgabe, die die gefundenen Hilfsprogramme und die dadurch möglichen Konvertierungen darstellt (siehe Abb. 5.10).

Zur Identifizierung von Ziel- und Quellformaten verwendet die converter-Komponente MIME-Types[1]. Nach dem Start sucht sie anhand einer Liste von Quell- und Ziel-Typen und einer Liste

[1] Multipurpose Internet Mail Extensions

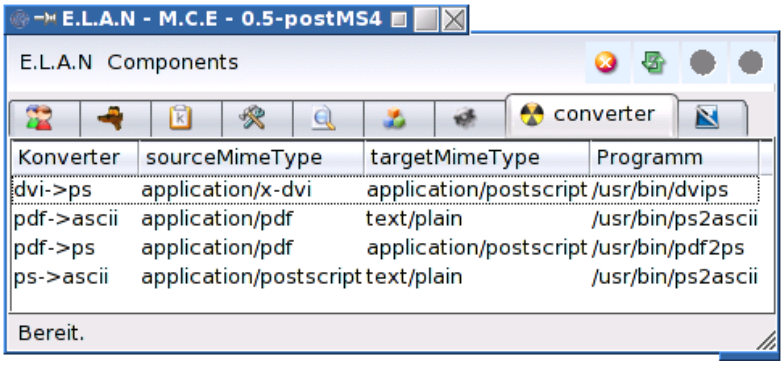

Abbildung 5.10: Debugging-Ausgabe der converter-Komponente

von typischen bekannten Konverterprogrammen unter Linux geeignete Programme heraus und fügt sie der Service-Metainfo hinzu. In der Abbildung sind das die Programme dvips, ps2ascii und pdf2ps. Die Middleware liest diese MetaInfo nach Beendigung der Startphase eines Dienstes aus und verteilt sie innerhalb der Dienstinformation im Netz. Auf diese Weise erhalten andere Stationen die semantisch wichtigen Informationen, welche speziellen Konverter ein entfernter `converter`-Dienst enthält.

Der Konverterdienst wertet die extIn-Nachricht `converterRequest` aus, die als Argumente die Quell- und die Ziel-Mimetypes der zu konvertierenden Datei enthält. Diese wird über den Aufruf des passenden Hilfsprogramms konvertiert und an den Anfragenden über die extOut-Nachricht `converterResponse` zurückgeschickt.

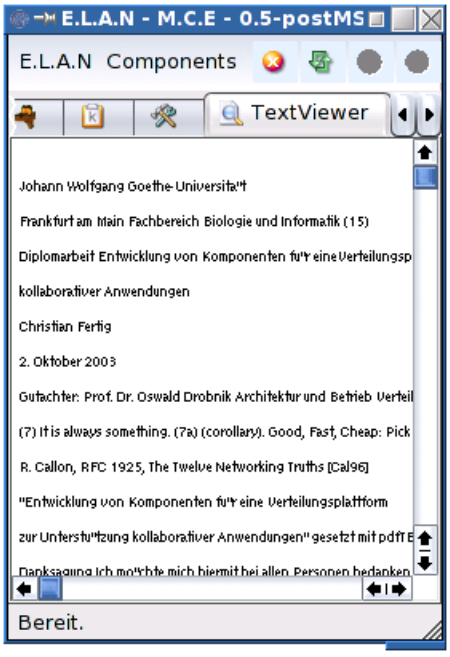

Abbildung 5.11: Die Textviewer-Komponente

TextViewer-Komponente

Die Textviewer-Komponente stellt eine einfache Textansicht zur Verfügung. Sie soll garantieren, daß in E.L.A.N ausgetauschte Textdokumente einsehbar sind. Dazu stellt sie die extIn-Nachricht `viewText` zur Verfügung, deren Argument ein beliebiger Text sein kann. In Abb. 5.11 wird eine Fassung dieser Diplomarbeit, umgewandelt in reinen Text, dargestellt.

Neben dem vorgestellten Konvertierungsszenario wird die TextViewer-Komponente auch vom Framework verwendet, um die Log-Dateien der `middlewareActivity`- und der `debug`-Komponente anzuzeigen.

Komposition des Szenarios

Das in Abb. 5.9 vorgestellte Szenario ist die Darstellung des optimalen Nutzungsfalles. Durch den direkten Versand über die

Konverter-Station soll verhindert werden, daß das Dokument zweimal über das Ad-hoc Netz gesendet werden muß (einmal von **A** nach **B** und anschließend von **B** zum Notebook). Zur Zeit kann die Dienstnutzung des Konverters jedoch nur direkt erfolgen, was ein doppeltes Übertragen der Datei bewirkt. Eine heruntergeladene Datei wird von der Middleware an den entfernten Konverter gesendet. Wenn die umgewandelte Datei eingetroffen ist, veranlaßt sie das Framework über die lego-Nachricht `componentBootupRequest` den TextViewer zu starten. Durch die Nachricht `componentActivationRequest` wird die Komponente anschließend aktiviert, also in den Vordergrund gebracht. Die Datei kann dem `TextViewer` nun zur Darstellung übergeben werden.

6 Zusammenfassung und Ausblick

Es wurden im Rahmen der vorliegenden Arbeit einige wesentliche Komponenten für eine Verteilungsplattform zur Unterstützung kollaborativer Anwendungen entwickelt. Diese Komponenten bilden innerhalb einer Middleware eine Verteilungsplattform für eine spontane Gruppenbildung und einer interessengestützten Kommunikation in Ad-hoc-Netzwerken. Diese Middleware wurde in das E.L.A.N-Framework eingebunden und versorgt dieses mit den gesammelten Informationen.

Weiterhin wurden grafische Plugins in das Framework integriert, die zur Evaluierung der Middleware-Plattform dienten. Mit Hilfe dieser Tools wurde das Verhalten von Middleware, Routingschicht und die Kommunikation verschiedener E.L.A.N-Instanzen in einem Ad-hoc-Netzwerk ausgiebig getestet.

Zur Darstellung der gelieferten Informationen wurde eine Awareness-Ansicht entwickelt, die die im Netz vorhandenen Dienste und Kollaborationspartner anzeigen kann. Mit ihrer Hilfe ist eine flexible Nutzung von Diensten möglich. Zur Evaluierung der Dienstnutzung wurden zusätzliche prototypische Dienste implementiert und angewandt.

Die Screenshots, die in der vorliegenden Arbeit verwendet wurden, sind auf den verschiedenen Testplattformen entstanden. So ist der Sharp Zaurus in Abb. 2.3 auf Seite 12 ein Screenshot des VNC-Viewers auf einer X11 Plattform.

Ein Screenshot, in dem eine Komponente innerhalb eines virtuellen Framebuffers läuft, wird in Abb. 5.6 auf Seite 52 gezeigt. Alle anderen Komponenten-Abbildungen enstanden durch Screenshots der jeweiligen Anwendung, die unter Qt/X11 lief.

Es wurde dabei jeweils die gleiche Anwendung verwendet, ohne manuell Veränderung für die spezifische Zielplattform vorzunehmen. Das Framework stellte sich dynamisch auf das verwendete Geräte ein und war in den verschiedenen Umgebungen problemlos lauffähig.

Dies zeigt, das die Forderungen nach einer plattformübergreifenden Lösung voll erfüllt werden konnten.

Ausblick

Eine Beschleunigung des Programms könnte erfolgen, indem Teile des Frameworks im optimierten Code einer Compilersprache implementiert würden. Durch die definierten Schnittstellen von Framework und Middleware ließe sich eine kompiliertebare Version verhältnismäßig leicht realisieren.

In zukünftigen Versionen könnte die Möglichkeit hinzugefügt werden, Eigenschaften von WiLiDips zu filtern. So ließen sich Kollaborationspartner für ausgewählte Themen wie „Biologie“ finden, obwohl auch Interesse in anderen Bereichen besteht. Diese Funktionalität könnte sich

durch Versenden eines verkleinerten WiLiDips oder durch Vergleichen der WiLiDips mit bestimmten Filteroptionen umsetzen lassen.

Mögliche noch zu entwickelnde Dienste könnten z.B. ein Dienst zur Verbindung von Ad-hoc- und Festnetz sein. Ein Rechner, der sowohl mit einer Netzwerkkarte am Ad-hoc-Netz teilnimmt, als auch über eine feste Internetverbindung verfügt, könnte einen Dienst für den Zugang zum Festnetz anbieten. Somit könnten die Stationen eines Ad-hoc-Netzes auch die Funktionen des Internets nutzen.

Ein Thesaurusdienst könnte zur Vereinheitlichung der Kategorienamen innerhalb der Wissensprofile dienen. Dies ist zukünftig wichtig, da die Benutzer ihre Kategorien selbst festlegen sollen. Da Benutzer für ähnliche Wissensgebiete verschiedene Namen wählen können, wäre mit Hilfe eines Thesaurusdienstes eine Vereinheitlichung der Kategorien zur Erleichterung des Vergleichs möglich. Ebenso könnten über einen solchen Dienst internationale Bezeichnungen automatisch übersetzt werden.

Eine Erweiterung der Middleware könnte die Integration einer Entfernungsmessung in den Algorithmus zur Verwaltung interessanter Kollaborationspartner sein. Eine geringe Entfernung (also wenige Hops) kann interessant sein, weil dadurch die Dienstgüte in bezug auf die Geschwindigkeit erhöht wird und weil so räumlich nahe und interessante Benutzer lokalisierbar wären, mit denen eventuell auch persönlicher Kontakt aufgenommen werden könnte.

Um die Belastung des Netzes zu reduzieren ist angedacht, WiLiDips mit verfeinerten Abstufungen zu verwenden. Stationen könnten zunächst durch Analyse der groben Abstufung überprüfen, ob ein WiLiDip für eine genauere Analyse interessant erscheint. Erst wenn dies erfüllt ist würden die feineren Abstufungen des WiLiDips zur genaueren Analyse angefordert werden.

Die sogenannten *Clusters of Interest* stellen eine Überlegung dar, bestimmte Zonen des Netzes zusammenzufassen. Dies ist zum einen räumlich gemeint, wie etwa eine Gruppe Studierende, die sich in einem Hörsaal befinden. Dort könnten örtlich optimierte Routingverfahren das Fluten von anderen Netzbereichen minimieren. Ebenso könnte man berücksichtigen, daß sich alle Hörer einer Vorlesung für das Thema der Vorlesung interessieren werden. Zum anderen beziehen sich die Cluster auf ähnliche Interessensgebiete innerhalb eines Netzwerks. Es könnte versucht werden, das Verteilen von WiLiDips auf gewisse Interessensgebiete zu optimieren, z.B. durch Broadcast-Schneisen.

Broadcast-Schneise

Die Idee der Broadcast-Schneise ist es, gezielt Daten durch das Netz transportieren zu können. Wegen der Funk-Broadcasts können immer mehrere Stationen mithören, wenn eine Station sendet. Jede Station in Reichweite erhält daher das gesendete Paket. Wenn für dieses keine Netzwerkflutung notwendig wäre, könnte es nur durch diejenigen Stationen weitergeleitet werden, die auf dem direkten Weg zur Zielstation liegen. Zu diesem Zweck könnte die Middleware beim Router nach der nächsten Station in Richtung der Zielstation (`getNextHopTo`) anfragen und diese Information in den Paketheader integrieren. Damit würde man partielle (zellenweise) geroutete Broadcasts erzeugen und die vollständige Flutung des Netzes vermeiden).

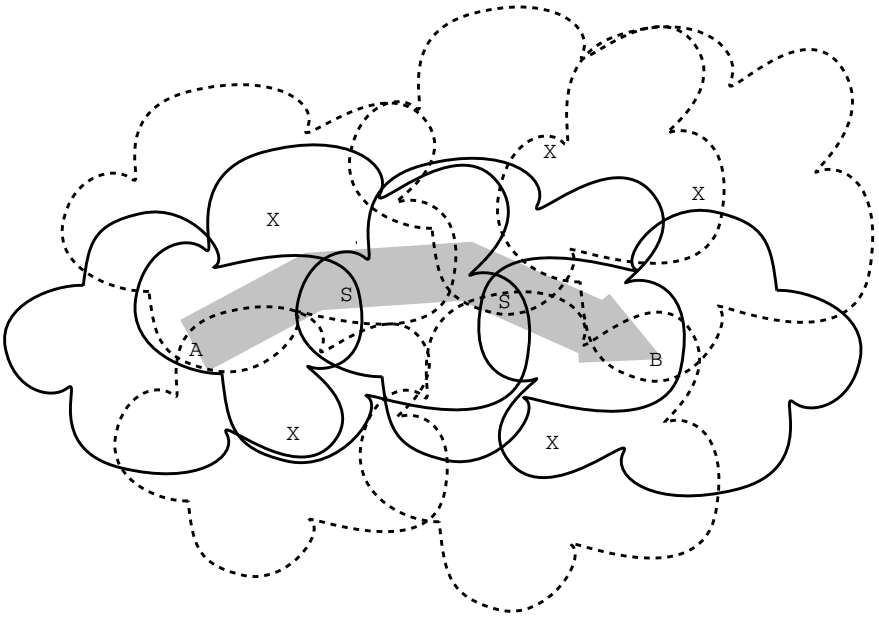

Abbildung 6.1: Broadcast-Schneise: Nur die mit **S** markierten Stationen broadcasten das Paket von **A** nach **B** weiter. Die Stationen **X** in Funkreichweite bekommen dieses zwar auch, senden es aber nicht weiter.

Connected Dominating Sets

In der Arbeit „Power-Aware Broadcasting and Activity Scheduling in Ad Hoc Wireless Networks Using Connected Dominating Sets“ [SWW02] wird ein optimiertes Broadcast-Schema vorgestellt, das „wichtige“ Gateways identifiziert und aus diesen *Connected Dominating Sets* (CDS) erstellt. Ein CDS ist eine Untermenge aller Stationen im Ad-hoc-Netz, die nur diese Gateways enthält. Jeder Gateway leitet ein Paket genau einmal weiter und stellt sicher, daß jede Station das Paket erhält. Die Analyse dieser Methode erscheint im Zuge der Optimierung der Verteilung erstrebenswert.

Lernumgebungen

Zur Zeit wird im Rahmen einer Diplomarbeit an einer Erweiterung von E.L.A.N durch eine lernunterstützende Komponente gearbeitet. Dabei soll ein ein Lehrer-Schüler Modell umgesetzt werden. Die Aktoren sollen dabei nicht von vorne herein auf ihre Rollen festgelegt sein, sondern über ein Belohnungssystem bestimmt werden. Ein Aktor soll nur in bestimmten Fachgebieten die Rolle des Lehrers innehaben, wenn er dort genügend Punkte gesammelt hat. In einem anderen Fachgebiet könnte er ein Lernender sein. Wie das Punktesystem implementiert werden soll ist zur Zeit noch nicht klar, ebenso noch nicht, welchen Reiz es haben soll, ein Lehrer zu sein – wahrscheinlich wird es darauf hinauslaufen, daß man nur durch aktives Beantworten von Fragen und somit sammeln von Punkten in die Rolle des Lernenden kommen kann.

Collaboratives Learning Environment

Das Collaborative Learning Environment (CLE) ist ein Projekt zum rechnergestützen Erarbeiten von Dokumenteninhalten und -strukturen. Durch CLE kann kollaborativ die Erarbeitung und Strukturierung von Wissensinhalten in einer Gruppe erfolgen. Das CLE bietet hierfür ein Vorgehensmodell zur gemeinsamen Arbeit und unterstützt sowohl individuelle asynchrone, wie auch kollaborativ synchrone Arbeitsphasen.

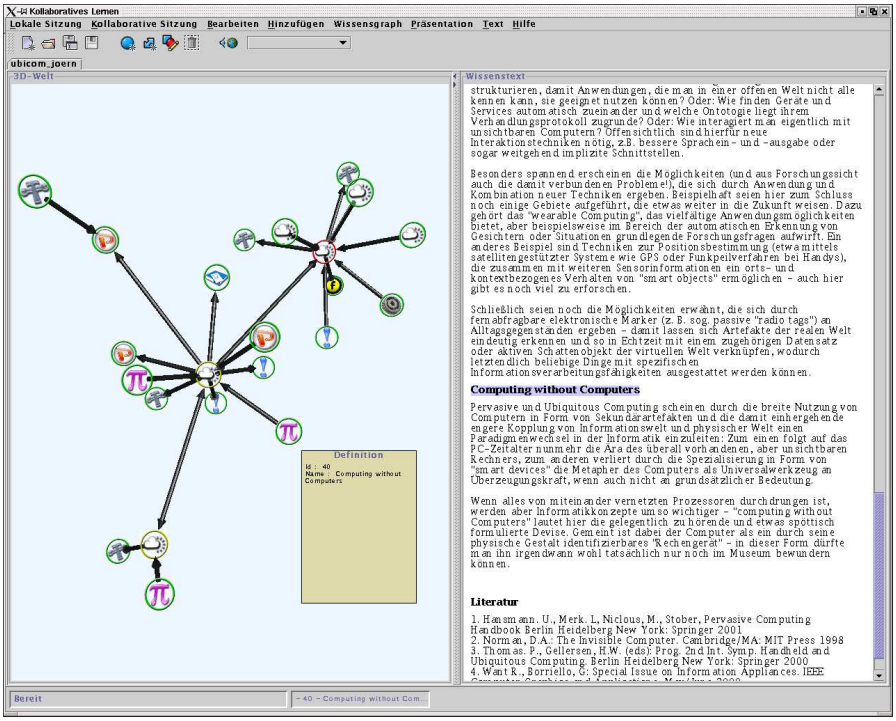

Abbildung 6.2: Collaborative Lerning Environment

Zur Zeit ist CLE noch als Java-basierte Client- / Server-Applikation implementiert. Es soll in den nächsten Monaten auf Python portiert werden und als Peer-To-Peer Anwendung in E.L.A.N integriert werden. Eine Abbildung des CLE ist in Abb. 6.2 dargestellt.

Anhang

A.1 lego-Nachrichten

In diesem Abschnitt werden alle verwendeten lego-Nachrichten und ihre zugehörigen Argumente aufgeführt.

A.1.1 `ServiceUsageRequest`

Nachricht an ein entferntes Framework: eine Dienst-Migration soll initiiert werden.

Parameter

`serviceName` String, Name des Dienstes.
`serviceIP` String, IPv4-Adresse des Dienstes.

A.1.2 `StartServiceUsage`

Nachricht, daß ein Dienst erfolgreich migriert ist und gestartet wurde.

Parameter

`serviceName` String, Name des Dienstes.
`serviceIP` String, IPv4-Adresse des Dienstes.

A.1.3 `categoriesRequest`

Anfrage an die aktuell verwendeten Kategorien des Profils. Vorbereitung für die Anbindung eines Kategorie-Editors (Papito).

Parameter

`bGetMarked` boolean, nur verwendete Kategorien.

A.1.4 `categoriesResponse`

Liefert mögliche Kategorien des aktuellen Profils.

Parameter

`response` die zurückgelieferten Interessenfelder

A.1.5 debug

Gibt eine Debug-Nachricht in der debug-Komponente aus. Der Debuglevel bestimmt, ab welchem voreingestellten Level die Nachricht überhaupt ausgegeben wird. sMögliche Werte sind:

DEBUG_UNKNOWN	000
DEBUG_DEBUG	100
DEBUG_NOTIFY oder DEBUG_INFO	200
DEBUG_WARN	300
DEBUG_ERROR	400
DEBUG_CRITICAL	500

Parameter

debuglevel Integer, Debuglevel.

message String, die zu loggende Nachricht.

A.1.6 updateWilidip

Eine Änderung des eigenen WiLiDips wird dem restlichen System mitgeteilt.

Parameter

wilidip Wilidip-Instanz, das neue Wilidip

A.1.7 requestWilidip

Anforderung eines Wilidips. Die Middleware ist dafür zuständig, es zu finden.

Parameter

key String, Schlüssel des WiLiDips (IP-Adresse).

A.1.8 updateNickname

Nachricht an das Framework, daß der eigene Spitzname verändert wurde. Diese Nachricht wird üblicherweise von der elanConfig-Komponente versendet.

Parameter

nick String, neuer Spitzname.

A.1.9 updateRealname

Nachricht an das Framework, daß der eigene Name verändert wurde. Diese Nachricht wird üblicherweise von der elanConfig-Komponente versendet.

Parameter

realname String, neuer Name.

A.1.10 compareToChanged

Nachricht an das Framework, daß sich der Vergleichskoeffizient für den Korrelationsalgorithmus zum Vergleich zweier WiLiDips geändert hat. Diese Nachricht wird üblicherweise von der `elanConfig`-Komponente versendet.

Parameter

`compareTo` Float, neuer Vergleichskoeffizient.

A.1.11 middlewareDebug

Debug-Nachricht der Middleware an die middlewareActivity. Überträgt die Nachrichten für das Logfile.

Parameter

`message` String, IPv4-Adresse

A.1.12 componentReady

Eine Komponente wurde gestartet und ist bereit. Diese Nachricht wird insbesondere von der Middleware ausgewertet, z.B. um dem Profileditor das aktuelle Profil zu senden.

Parameter

`component` String, Komponentenname

A.1.13 componentShutdown

Eine Komponente wurde beendet.

Parameter

`component` String, Komponentenname

A.1.14 EXTIN

Eingehender extIn-Aufruf.

Parameter

`envelope` Tupel, Adress-Envelope
`cmdtuple` Tupel, Nachricht und Parameter

A.1.15 EXTOUT

ausgehender extOut-Aufruf.

Parameter

`envelope` Tupel, Adress-Envelope
`cmdtuple` Tupel, Nachricht und Parameter

A.1.16 updateGuid

Mitteilung, daß die GuID aktualisiert wurde.

Parameter

guid GuID-Instanz, die neue GuID

A.1.17 fullWilidipList

Nachricht der Middleware an die middlewareActivity mit allen verfügbaren Wildips.

Parameter

list Dictionary, die WiLiDip-Liste

A.1.18 addService

Nachricht des Frameworks an die Middleware: ein neuer Dienst wurde gestartet.

Parameter

name String, Name des Dienstes.

A.1.19 removeService

Nachricht des Frameworks an die Middleware: ein Dienst wurde beendet.

Parameter

name String, Name des Dienstes.

A.1.20 forceStatusChange

Nachricht an die Komponente messageService: Öffne den Chat-Dialog für bezeichnete Station.

Parameter

ip String, IPv4-Adresse

nick Spitzname des Empängers (muß mitgegeben werden, da der message-Service diesen nicht kennt).

A.1.21 forceStatusChange

Nachricht an die Middleware: ändere den Status des WiLiDip zur bezeichneten Station auf den übergebenen, neuen Status. Diese Methode ist dazu da, einzelne Stationen gezielt zu modifizieren.

Parameter

ip String, IPv4-Adresse

newstate WilidipInfo-Instanz

A.1.22 `componentConfigurationRequested`

Benutzeranfrage an das Framework, den Komponenten-Konfigurationsdialog zu öffnen.

Parameter

`params` Tupel, Name der Komponente und Argumente

A.1.23 `componentCloseRequested`

Benutzeranfrage an das Framework, die Komponenten zu schließen.

Parameter

`params` Tupel, Name der Komponente und Argumente

A.1.24 `componentReloadRequested`

Benutzeranfrage an das Framework, die Komponenten neu zu laden.

Parameter

`params` Tupel, Name der Komponente und Argumente

A.1.25 `componentBootupRequest`

Anfrage an das Framework, eine Komponenten zu starten.

Parameter

`params` Tupel, Name der Komponente und Argumente

A.1.26 `componentActivationRequested`

Anfrage an das Framework, eine Komponenten zu aktivieren, also die Lasche der Komponente in den Vordergrund zu bringen.

Parameter

`params` Tupel, Name der Komponente und Argumente

A.1.27 `broadcastMUP`

Nachricht an die Protokollmaschine: verteile das mitgelieferte MUP.

Parameter

`pkg` Das zu verteilende Paket.

A.1.28 `receivedPong`

Die Protokollmaschine hat ein neues Pong empfangen.

Parameter

`pkg` MUP, das unveränderte empfangene Paket.
`addr` String, IPv4-Adresse des Absenders.

A.1.29 `broadcastWilidip`

Nachricht an die Protokollmaschine das mitgelieferte WiLiDip zu broadcasten.

Parameter

`wilidip` WiLiDip-Instanz, das zu verteilen ist.

A.1.30 `setupBridge`

Nachricht an den XMLRPCCommunicator: führe einen Bridge-Request zu einer entfernten Station aus.

Parameter

`ip` String, IPv4-Adresse der entfernten Station.

A.1.31 `bridgeRequest`

Eingehender Bridge-Request. Eine fremde Station möchte mit direkten Unicasts beliefert werden.

Parameter

`ip` String, IPv4-Adresse der entfernten Station.

A.1.32 `BatteryStatusCritical`

Nachricht der Contextawareness an das Framework: der Batteriezustand ist kritsch.

Parameter

`percentRemain` Integer, Restenergie in Prozent

A.1.33 `BatteryStatusOKAgain`

Nachricht der Contextawareness an das Framework: der Batteriezustand wieder in Ordnung.

Parameter

`percentRemain` Integer, Restenergie in Prozent

A.1.34 awareness_updateUser

Aktualisier die Informationen über einen möglichen Kollaborationspartner. Sollte der Partner vorher nicht bekannt gewesen sein, wird ein neues Benutzer-Symbol erstellt und nach weiteren Benutzerinformationen über den identd des Benutzers angefragt.

Parameter

ip String, IPv4-Adresse
wilidipinfo WilidiInfo-Instanz

A.1.35 awareness_removeAllUsers

Zurücksetzen der userView – alle Teilnehmer werden gelöscht.

Parameter keine.

A.1.36 awareness_removeUser

lösche den Teilnehmer mit übergebenem Schlüssel.

Parameter

ip String, IPv4-Adresse, der zu löschende Teilnehmer

A.1.37 awareness_updateService

aktualisiert die Informationen zum übergebenen Service. Sollte dieser noch nicht vorhanden sein, so wird ein neues Service-Icon erstellt.

Parameter

ip String, IPv4-Adresse
service Service-Deskriptor

A.1.38 awareness_removeExplicitServices

lösche alle expliziten Dienste zur übergebenen IP-Adresse

Parameter

ip String, IPv4-Adresse

A.1.39 awareness_removeImplicitService

lösche aus der Liste der Stationen die den übergebenen impliziten Service bereitstellen die übergebenen IP-Adresse heraus.

Parameter

service Service-Deskriptor
ip String, IPv4-Adresse

A.2 Middleware UDP Pakete – MUPs

Die MUPs (*Middleware UDP Pakete*) sind eine Klassenstruktur zum Generieren von binären, komprimierten Inhalten zum Transport in UDP-Datagrammen. Für den Transport in binärer Form stellt Python das Modul `struct` zur Verfügung. Durch dieses werden die einzelnen Bestandteile der Nutzdaten verpackt. Dazu wird eine Formatstring (im Folgenden Kodierung) definiert. Durch Verwendung derselben Kodierung beim Entpacken, lassen sich die ursprünglichen Daten wieder herstellen.

A.2.1 Message

Der Nachrichtenkopf UHeader) enthält neben den Basisinformationen wie Message-ID und Funktions-ID (Typ der Nachricht) noch die Länge der Nutzdaten (Payload), die an das Paket angehängt werden. Diese ist vom jeweiligen Pakettypen abhängig abhängig.

Bytes	Name	Beschreibung	Kodierung
0-14	MUP Prefix	Kontroll-Präfix zur Identifizierung eines MUPs. Zur Zeit: 'MUP (0.3)'	15p
15-30	Message ID	Eindeutige Nachrichten Id. 16 Byte generiert über `unique.unique()` (vgl. A.3.4)	16s
31	Function ID	Nachrichtentyp zur Unterscheidung der 4 MUPs und R-MUPs. `0x00 (0)`: Ping, `0x01 (1)`: Pong (Ping Response), `0x40 (64)`: WilidipRequest, `0x80 (128)`: WilidipTransport (Request Response)	B
32	TTL Left	Time To Live - Anzahl der noch möglichen Hops	B
33	Hops taken	Anzahl der zurückgelegten Hops	B
34-37	Source IP	IP Adresse des Senders	4s
38-41	Data Length	Länge des folgenden, funktionsspezifischen Datenpakets (Payload)	I

encode Die virtuelle Methode `encode` gibt ein binäres Paket zurück, das aus der Summe sämtlicher verwendeter Header und der Nutzdaten besteht. Die zurückgegeben Daten können einem RMessage-Typen im Konstruktoraufruf übergeben werden, um ein Paket zu rekonstruieren.

A.2.2 Ping (0x00)

Die Funktion Ping stellt ein Debugging-Paket zur Verfügung, das durchgeroutet werden soll. Eine Antwort auf ein Ping erfolgt durch ein Pong – ähnlich wie bei GNUtella [Gnu01].

Payload (zusätzliche Daten) Keine.

A.2.3 Pong (0x01)

Antwortpaket auf ein Ping. Zur Erkennung, auf welches Ping geantwortet wurde, wird dessen Message-ID als RMID zurückgeschickt.

Payload (zusätzliche Daten)

Bytes	Name	Beschreibung	Kodierung
0-15	Refering MID	Die ursprüngliche Nachrichten Id des Pings	16s

A.2.4 WilidipRequest (0x40)

Der WilidipRequest ist dazu da, eine Anfrage nach einem Wilidip, an das Netz zu stellen. Jede Station entscheidet, ob sie das Wilidip, wenn sie es schon hat, über einen WilidipTransport zurückschickt, oder ob sie die Nachricht weiterroutet.

Payload (zusätzliche Daten)

Bytes	Name	Beschreibung	Kodierung
0-3	Key (IP)	IP-Adresse des angeforderten WiLiDips	4s (bzw. BBBB)

A.2.5 WilidipTransport (0x80)

Der WilidipTransport ist der älteste und wichtigste Pakettyp. Er enthält in einem zusätzlichem Header eine GuID und ein WiLiDip als Transportdaten.

Payload (zusätzliche Daten)

Bytes	Name	Beschreibung	Kodierung
0-1	guidlen	Länge der transportierten GuId	H
2-3	wilidiplen	Länge des transportierten WiLiDips	H
4+	GuId	Üblicherweise 32 Bytes Pascal-String (Nick), 32 Bytes String (Unique) und 16 Bit unsigned short Integer	32p32sH
x+	WiLiDip	Komprimiertes WiLiDip	%ds

A.3 Wichtige Funktionen

A.3.1 `common.netutils.getHostnameShort`

In einem Ad-hoc Netzwerk steht normalerweise kein Nameserver zur Verfügung.

Daher versucht die Funktion `getHostnameShort` zuerst den Rechnernamen über eine der Linux-Umgebungsvariablen `COMPUTERNAME`, `HOST`, `HOSTNAME`, `host` oder `hostname` zu lesen. Sollte sie hier nicht fündig werden, wird über die Python-Funktion `os.popen` der Unix-Befehl `hostname` ausgeführt, die den Rechnernamen ausgibt. Nur wenn auch dies fehlschlägt wird ein Nameserver-Aufruf versucht. Dieser Fall kam allerdings weder auf dem Zaurus noch auf einer den anderen Testmaschinen vor. Um dennoch Timeouts beim Warten auf einen Nameserver zu vermeiden, wird ein ermittelter Rechnername statisch gespeichert. Nachfolgende Aufrufe auf die Funktion geben nur noch den gespeicherten Wert zurück.

Für den Gebrauch von E.L.A.N ist es in der Praxis nicht notwendig, daß der Rechnername bekannt ist. Es ist jedoch während des Programmierens und Testens eine Hilfe.

Parameter keine.

Rückgabewert den Rechnernamen als String.

A.3.2 `common.netutils.getFQDN`

Entsprechend A.3.1 wird von dieser Funktion der *Full Qualified Domain Name* (FQDN) zurückgegeben, also der Rechnername und sein Domain-Suffix.

Parameter keine.

Rückgabewert den FQDN als String.

A.3.3 `common.netutils.isPrivateIP`

Diese Funktion ermittelt, ob eine gegebene IPv4-Adresse aus einem der nach RFC 1918 reservierten privaten IP-Bereichen stammt [RMK+96].

Die Adressbereiche sind:

Adressbereich	Klasse
10.0.0.0 bis 10.255.255.255	Class-A-Netz
172.16.0.0 bis 172.31.255.255	Class-B-Netz
192.168.0.0 bis 192.168.255.255	Class-C-Netz

Parameter

`ip` String, IPv4-Adresse die zu Prüfen ist.

Rückgabewert `True`, wenn private IP, sonst `False`.

A.3.4 common.unique.unique

Die Funktion `unique.unique` wird an mehreren Stellen des Programms gebraucht. Einmal zur Erzeugung der GuID (vgl. A.3.5) und zur Erzeugung eindeutiger Nachrichten-Nummern (Message-IDs oder MIDs) in versendeten Paketen.

Ihre Ausgabe ist ein 32 Zeichen langer Hexadezimal-String, der zufällig generiert wird: dazu wird ein `md5`-Fingerabdruck (*message digest*) über einen String gebildet, der aus einem beliebigen Prefix, einer Pseudo-Zufallszahl (`random.random()`), der aktuellen Uhrzeit (`time.time()`) und dem aktuellen Prozess-Identifier (`os.getpid()`) gebildet wird. Die von der `hexdigest`-Methode ausgegebenen Strings sind alle gleich lang.

Die md5-Summe wurde gewählt, da sie eindeutig ist und relativ schnell berechnet werden kann [Riv92].

Parameter

`Prefix` String, optionales Prefix

Rückgabewert Ein eindeutiger 32 Zeichen langer Hexadezimal-String.

A.3.5 GuID

Die GuID ist ein *Global unique Identifier*. Wie der Name sagt, ist sie eine global verwendete und einzigartige Identifikationsnummer im Netz. und bezeichnet genau das WiLiDip einer E.L.A.N-Station. Sie besteht aus drei Teilen:

nick: Ein frei definierbarer Nickname (Spitzname).

unique: Ein eindeutiger Teil in der Mitte, generiert durch die `unique()`-Funktion, die auch von anderen Programmteilen verwendet wird.

sequence: Eine Sequenznummer. Anfangs war diese ein einfacher Integer-Wert, der inkrementiert wurde. Im Laufe der Entwicklung wurde sie jedoch von einem Zeitstempel ersetzt. Dieser erlaubt es, sowohl die Erhöhung der Sequenznummer als auch den Zeitpunkt, an dem dies geschah, zu bestimmen.

Eine GuID wird durch die Klasse `common.gumaid.GuMaID` beschrieben. Diese enthält Methoden, um eine GuID zu erzeugen und Ihre einzelnen Bestandteile zu lesen und zu schreiben (`getNick`, `getUniq`, `getSeq`, `setNick`, `setUniq`) und die Sequenznummer zu aktualisieren (`incSeq`).

Außerdem enthält sie die Methode `parseString`, die die einzelnen Bestandteile der GuID aus einer Stringrepräsentation ausliest. Dies wird z.B. für den Transport der Cargo-Daten benötigt.

A.3.6 common.unique.config

Die Klasse `ElanConfig` stellt eine einfach zu bedienende, bequeme Möglichkeit dar, auf gespeicherte Konfigurationswerte zuzugreifen. Diese werden dabei als einfacher strukturierter Text gespeichert und sind somit lesbar und editierbar.

Die ElanConfig basiert auf dem Python-Modul `ConfigParser`, erweitert diesen jedoch um Attribute und Methoden.

Eine wichtige Erweiterung ist, daß das Lesen von noch nicht geschriebenen Konfigurationswerten keinen Fehler produziert, sondern einen vorgegebenen Default-Wert zurückgibt. Eine weitere Erweiterung ist die Zugriffsmethode `getport`, die einen TCP / IP-Port zurückgibt. Mittels einer Umgebungsvariable (`ELAN_PORT_OFFSET`) kann ein Offset angegeben werden, der auf den eigentlichen gespeicherten Wert addiert wird. Auf diese Weise können mehrere getrennte Netze innerhalb eines Festnetzes simuliert werden, da sowohl mehrere Router als auch mehrere E.L.A.N-Instanzen auf einer Maschine laufen können.

Die ElanConfig stellt eine Reihe von statischen Werten zur Verfügung, die von E.L.A.N-Anwendungen genutzt werden können. Z.B. bezeichnet `elandir` den Pfad zur Installation der Umgebung und `userwilidip` den Pfad zum verwendeten Interessensprofil

Wird E.L.A.N mit der gesetzten `__debug__`-Variable gestartet, das passiert immer dann, wenn Python *nicht* mit der Option `-OO` (für „optimiere maximal") gestartet wurde, so werden andere Werte für den Pfad zum Profil und für den Pfad zur Konfigurationsdatei verwendet. Dem Pfad zu den Dateien wird jeweils der Rechnername vorangestellt. Dies ermöglicht das Testen von E.L.A.N mit einem Unix Benutzer-Profil auf verschiedenen Rechnern innerhalb eines Unix-Netzwerks.

A.3.7 LanRouter-Werte

Der LanRouter verwendet aus der Konfigurations-Datei folgende Werte:

Sektion: 'router', Wert: 'routerport' default: UDP, 5557, Listener-Port, horche auf Broadcasts von anderen Routern

Sektion: 'router', Wert: 'xmlrpcport' default: TCP, 5558, Listener-Port, horche auf XML-RPC-Nachrichten (der Middleware)

Sektion: 'middleware.xmlrpccom', Wert: 'listenerport' default: TCP, 5556, Listener-Port der Middleware für die Kommunikation in Gegenrichtung mit XML-RPC.

Sektion: 'global', Wert: 'fakenet' Zu Testzwecken: verwende den letzten Teil der echten IP-Adresse aber in einem Uni-lokalen net (192.168.72.x). Dieses Netz ist ein virtuelles Ad-hoc-Netz, das durch iptables [net] Link-Connection simulieren kann.

Sektion: 'router', Wert: 'ttl' default: 7. Vorgabewert für den TTL-Wert (Time to Live).

A.3.8 Middleware-Werte

Die Middleware, bzw. ihre Komponenten, verwenden aus der Konfigurationsdatei folgende Werte:

Sektion: 'middleware.xmlrpccom', Wert: 'listenerport' default: TCP, 5556, Listener-Port der Middleware für die Kommunikation in Gegenrichtung mit XML-RPC.

Sektion: ‘middleware.xmlrpccom’, Wert: ‘numtasks’ default: 5, Maximale Anzahl gleichzeitiger Threads, die entfernte Aufrufe durchführen.

Sektion: ‘router’, Wert: ‘xmlrpcport’ default: TCP, 5558, Listener-Port des Routers, für die Kommunikation in Gegenrichtung mit XML-RPC.

Sektion: ‘middleware.protocollmachine’, Wert: ‘listenerport’ default: UDP, 5555, Listener-Port, horche auf Broadcasts von anderen Routern

Sektion: ‘middleware.protocollmachine‘, Wert: ‘ttl’ default: 7. Vorgabewert für den TTL-Wert (Time to Live) der MUPs.

Sektion: ‘middleware.wilidipmanager‘, Wert: ‘good_wilidip’ default: 0.75, Vorgabewert für den Vergleich zweier WiLiDips.

Sektion: ‘component.config‘, Wert: ‘nickname’ default: `userhostname` ermittelt über netutils.getUserAtHost(), der Vorgabewert für den Spitznamen innerhalb der GuID.

A.4 XML-RPC-Schnittstellen

A.4.1 LanRouter

`setupBridge(self, ip)`

Initialisiert die Bridge-Funktionalität zur übergeben IP-Adresse. Diese wird in die `additional_ip`-Liste hinzugefügt und beim Broadcasten als zusätzlicher Empänger verwendet.

Parameter

`ip` String, IPv4-Adresse

Rückgabewert `1`, eventuelle Fehler werden erst später bemerkt und die Liste dann automatisch angeglichen

`setCargoData(self, cargo)`

Setzt die Cargo-Daten. Ein beliebiger String, der die maximale Größe eines UDP-Packets (abzüglich der Größe von GuIDund Broadcast-Prefix) nicht überschreiten darf. Es erfolgt diesbezüglich keine Sicherheitsabfrage.

Parameter

`ip` String, die Cargo-Daten

Rückgabewert `0`, wenn erfolgreich, sonst `1`.

`getCargoData(self)`

Holt das aktuelle Cargo-Daten vom Router.

Parameter keine.

Rückgabewert die Cargo-Daten in Form eines Strings.

`retrieveAllStations(self)`

Dem Router wird mitgeteilt, daß die Middleware alle Stationen übertragen bekommen möchte. Aufgrund dieser Nachricht liefert der Router anschließend über die `notifyUpdate`-Methode die Stationen an die Middleware. Es wird dem Router überlassen, ob er dies durch mehrfache Aufrufe der `notifyUpdate`-Methode mit dem Parameter 'add' oder durch einen einzelnen Aufruf mit dem Parameter 'full' durchführt. (vgl. A.4.2).

Parameter keine.

Rückgabewert `0`

`requestRouteTo(self, ip, hold=False)`

Die Middleware veranlaßt den Router eine Route zum Rechner `ip` aufzubauen. `hold` gibt dabei an, ob die Middleware wünscht, ob die Route dauerhaft erhalten bleiben soll. Ist der Wert `False` bzw., kann der Router sie nach eigenem Ermessen wieder abbauen. Ist dieser Wert `True`, so erwünscht die Middleware, daß eine Route, z.B. zu einem wichtigen Dienst, möglichst aufrecht erhalten wird.

Diese Methode hat im LanRouter keine Funktion, da dieser keine Routen-Tabellen des Kernels modifiziert.

Parameter

`ip` String, IPv4-Adresse zu der die Route aufgebaut werden soll.

`hold` Integer, Vorgabewert ist 0.

Rückgabewert `0`, wenn erfolgreich, sonst `1`. Der LanRouter gibt hier 0 zurück, wenn die Station bekannt ist, sonst 0.

`isNeighbour(self, ip)`

Abfrage, ob eine gegebene `ip`-Adresse ein direkter Nachbar des Routers ist oder nicht.

Parameter

`ip` String, IPv4-Adresse

Rückgabewert `True`, wenn die Station als direkter Nachbar markiert ist, sonst `False`.

`getNextHopTo(self, ip)`

Gibt aus, welches die nächste routende Station auf dem Weg zur gegebenen `ip`-Addresse ist. Diese Methode wurde implementiert, um ein Broadcast-Verfahren der Middleware zu ermöglichen, das nicht das ganze Netz flutet.

Da der LanRouter nicht weiß, welches die nächste Station ist, gibt nur dann eine Adresse zurück, nämlich die der Station selbst, wenn die Station direkter Nachbar ist.

Parameter

`ip` String, IPv4-Adresse

Rückgabewert Die `ip`-Adresse des nächsten Hops. `None`, wenn die Zieladresse nicht erreichbar ist oder ein Fehler beim Ermitteln des nächsten Hops aufgetreten ist.

`dummy(self)`

Dummy-Methode. Diese Methode wird ausschließlich zum Testen und zum Herunterfahren des XML-RPC-Servers benötigt.

Parameter keine.

Rückgabewert immer `0`.

START(self)

Mit dem START-Kommando für den Router signalisiert die Middleware, daß sie läuft und empfangsbereit ist. Nach dem Handshake (vgl. 4.3.6) kann der Router an die Middleware senden, ohne Fehler bezüglich ihrer Unerreichbarkeit zu produzieren.

Parameter keine.

Rückgabewert immer 1.

A.4.2 Middleware

notifyUpdate(self, cmd, stations=[])

notifyUpdate ist ein Befehl vom Router, der der Middleware mitteilt, daß sich die Umgebung geändert hat. Er sendet diesen Befehl immer an die Middleware, wenn Knoten aktualisiert oder gelöscht wurden oder neue hinzugekommen sind.

Parameter

cmd String. Mögliche Kommandos sind:

add	Behandle die Stationen als neue Knoten.
remove	Behandle die Stationen als aus dem Netz entfernt.
update	Aktualisiere die Daten der Stationen.
full	Überträgt die vollständige Liste aller Knoten im Netz.

stations stations ist eine (Python-)Liste von Tupeln von IP-Adresse und die Cargo-Daten in Form der GuID. Eine Station ist eine IP-Adresse.

Rückgabewert Der Rückgabewert ist immer 0

post(self, *args)

post ist eine Introspektions-Methode, die es ermöglicht, lego-Nachrichten von außerhalb des Programms senden zu können. Die übergeben Argumente werden direkt in eine lego-Nachrichten umgesetzt und über ECom an das Framework weitergeleitet.

Parameter

cmd String. Die zu sendende lego-Nachricht.

args String. Die mitzugebenden Argumente.

Rückgabewert Der Rückgabewert ist immer 0

extin(self, *args)

Schnittstelle für eingehende extIn / extOut-Kommunikation. Die übergeben Argumente werden direkt durch die lego-Nachricht EXTIN weitergeleitet.

Parameter

envelope Tupel. Versender- und Empfänger-Informationen auf Netzwerk- und Komponentenebene

cmdtuple Tupel. Übertragene Nachricht und Argumente.

Rückgabewert Der Rückgabewert ist immer 0

setupBridge(self, ip)

Initialisiert die Bridge-Funktionalität zur Station mit der übergeben IP-Adresse. Diese wird in die `additional_ip`-Liste hinzugefügt und beim Broadcasten als zusätzlicher Empänger verwendet.

Parameter

ip String, IPv4-Adresse

Rückgabewert `1`, eventuelle Fehler werden erst später bemerkt und die Liste dann automatisch angeglichen

dummy(self)

Dummy-Methode. Diese Methode wird ausschließlich zum Testen und zum Herunterfahren des XML-RPC-Servers benötigt.

Parameter

keine.

Rückgabewert immer `0`.

START(self)

Mit dem `START`-Kommando für die Middleware signalisiert der Router, daß er läuft und empfangsbereit ist. Nach dem Handshake (vgl. 4.3.6) kann die Middleware an den Router senden, ohne Fehler bezüglich seiner Unerreichbarkeit zu produzieren.

Parameter

keine.

Rückgabewert immer `1`.

Abbildungsverzeichnis

2.1 Eine Ad-hoc Wolke mit drei Stationen 4
2.2 Die E.L.A.N Architektur 9
2.3 E.L.A.N auf dem Sharp Zaurus SL-5500 12

3.1 Der E.L.A.N-Kommunikationsfluß zwischen den verschiedenen Komponentenschichten 15
3.2 WiLiDip-Editor-Komponente (wilied) 17
3.3 Aufbau von midas 21
3.4 Aufbau des LanRouters 25

4.1 UML-Ansicht des XMLRPCCommunicators 30
4.2 extIn- / extOut-Kommunikation 31
4.3 Klassenstruktur der MUPs (*Middleware UDP Packets*) 32
4.4 UML-Ansicht der Protokollmaschine 33
4.5 Die Klasse Wilidipinfo 34
4.6 UML-Ansicht des WiLiDipmanagers und des ServiceManagers 35
4.7 Statusmaschine des Wilidipmanagers 36
4.8 Die SocketNotifier-Hierarchie 38
4.9 Kontrollfluß in der Middleware 39
4.10 LanRouter-UML-Ansicht 40
4.11 Programmfluß der Hauptroutine 41
4.12 LanRouter GUI: Knotenansicht 43
4.13 LanRouter GUI: Statusansicht 44
4.14 Middleware-Routing-Handshake 45

5.1 Komponentenansicht 47
5.2 Die awarenessView-Komponente 48
5.3 Der WWW-Server des identd 49
5.4 Die middlewareActivity-Komponente 50
5.5 Pong-Antworten auf ein Ping 51
5.6 Die E.L.A.N Config-Komponente 52
5.7 Der Filesharing-Accessor 53
5.8 Die E.L.A.N-messageService-Komponente (Chat) 54
5.9 Implizite Benutzung eines Konverterdienstes 55
5.10 Debugging-Ausgabe der converter-Komponente 56

5.11 Die Textviewer-Komponente 56

6.1 Broadcast-Schneise 60
6.2 Collaborative Lerning Environment 61

Literaturverzeichnis

[Ahl02] AHLERS, ERNST: *Leinenlos – Leitfaden zum WLAN-Kauf*. c't Magazin für computer technik, 25, 2002.

[Ahl03] AHLERS, ERNST: *Funknetze allerorten – Schnelles WLAN breitet sich aus*. c't Magazin für computer technik, 6, 2003.

[Ami04] AMIROVA, NATALIYA: *Modellierung und Visualisierung von Mobilität und Gruppenbildung in Ad-hoc-Netzen*. Diplomarbeit, Goethe–University Frankfurt/Main, Frankfurt, Germany, 2004.

[Api03] APITZSCH, FELIX: *Entwicklung von anwendungsorientierten, adaptiven Routing-Mechanismen für optimale Selbstorganisation in mobilen Ad-Hoc-Netzwerken*. Diplomarbeit, Goethe–University Frankfurt/Main, Frankfurt, Germany, 2003.

[Arn02] ARNOLD, ALFRED: *WLAN-Reichweite in Werbung und Wirklichkeit*. c't Magazin für computer technik, 15, 2002.

[BC] BLAIR, GORDON S. und GEOFF COULSON: *The Case For Reflective Middleware*. Basislink: `http://citeseer.nj.nec.com/355090.html`.

[BCA+99] BLAIR, GORDON S., GEOFF COULSON, ANDERS ANDERSEN, LYNNE BLAIR, MICHAEL CLARKE, FABIO COSTA, HECTOR DURAN, NIKOS PARLAVANTZAS und KATIA B. SAIKOSKI: *A Principled Approach to Supporting Adaptation in Distributed Mobile Environments*. Technischer Bericht, Distributed Multimedia Research Group, Lancaster University, Januar 1999. (accepte at PDSE'2000, see blair2000a).

[BCA+00] BLAIR, GORDON S., GEOFF COULSON, ANDERS ANDERSEN, LYNNE BLAIR, MICHAEL CLARKE, FABIO COSTA, HECTOR DURAN, NIKOS PARLAVANTZAS und KATIA B. SAIKOSKI: *A Principled Approach to Supporting Adaptation in Distributed Mobile Environments*. In: *5th International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE-2000)*, Limerick, Ireland, Juni 2000.

[blu] *The Official Bluetooth© Wireless Info Site, (Stand Mai 2002)*. URL: `http://www.bluetooth.com/`.

[Cal96] CALLON, R.: *RFC 1925: The Twelve Networking Truths*, April 1996. Status: INFORMATIONAL.

[CBC98] COSTA, FABIO M., GORDON S. BLAIR und GEOFF COULSON: *Experiments with Reflective Middleware*. In: *ECOOP Workshop on Reflective Object-Oriented Programming and Systems (ROOPS'98)*, Brussels, June 1998. Springer-Verlag. Basislink: `http://www.comp.lancs.ac.uk/computing/research/mpg/reflection/publ.html`.

[CBM+02] CAPRA, L., G. BLAIR, C. MASCOLO, W. EMMERICH und P. GRACE: *Exploiting Reflection in Mobile Computing Middleware*. In: *ACM SIGMOBILE Mobile Computing and Communications Review*, Band 6, Seiten 34–44, 2002.

[CEM01a] CAPRA, L., W. EMMERICH und C. MASCOLO: *Exploiting Reflection and Metadata to build Mobile Computing MIddleware*. In: *Workshop on Mobile Computing Middleware*, Heidelberg, Germany, November 2001.

[CEM01b] CAPRA, L., W. EMMERICH und C. MASCOLO: *Reflective Middleware Solutions for Context-Aware Applications*. In: *REFLECTION 2001. The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. LNCS (Springer-Verlag). Basislink: `http://www.cs.ucl.ac.uk/staff/L.Capra/reflection01.html`, Publikationen von L. Capra: `http://www.cs.ucl.ac.uk/staff/l.capra/publications.html`.

[Cha82] CHANG, ERNEST J. H.: *Echo Algorithms: Depth Parallel Operationson General Graphs*. IEEE Transactions on Software Engineering, Seiten SE–8(4):391–401, 1982.

[CM99] CORSON, S. und J. MACKER: *Mobile Ad hoc Networking (MANET): Routing Protocol, Performance Issues and Evaluation Considerations*. RFC 2501, Jan 1999.

[CMZE01] CAPRA, L., C. MASCOLO, S. ZACHARIADIS und W. EMMERICH: *Towards a Mobile Computing Middleware: a Synergy of Reflection and Mobile Code Techniques*. In: *8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001)*, Seiten 148–154, Bologna, Italy, October 2001.

[DJL+01] DROBNIK, PROF. DR. OSWALD, PROF. DR. KLAUS JOBMANN, PROF. DR. PETER LOCKEMANN, PROF. DR. KURT ROTHERMEL, PROF. DR. OTTO SPANIOL und PROF. DR. MARTINA ZITTERBART: *Antrag auf Einrichtung eines neuen Schwerpunktprojekts: „Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Systeme"*, Juli 2001.

[EAB+99] ELIASSEN, FRANK, ANDERS ANDERSEN, GORDON S. BLAIR, FABIO COSTA, GEOFF COULSON, VERA GOEBEL, ØIVIND HANSEN, TOM KRISTENSEN, THOMAS PLAGEMANN, HANS OLE RAFAELSEN, KATIA B. SAIKOSKI und WEIHAI YU: *Next Generation Middleware: Requirements, Architecture, and Prototypes*. In: *Proceedings of 7th Workshop on Future Trends of Distributed Com-*

puting Systems (FTDCS'99), Cape Town, South-Africa, Dezember 1999. IEEE. Basislink: `http://www.unik.no/~tomkri/papers/`.

[ela] *E-Learning in Ad-hoc Netzen (ELAN): Eine adaptive Kollaborationsumgebung für E-Learning in mobilen Ad-Hoc-Netzen*. URL: `http://elan.wox.org/`.

[Eng01] ENGEL, MICHAEL: *Zwergen-Makeup – Grafische Benutzerschnittstelle ohne X-Server*. Linux-Magazin, 03, 2001.

[FB96] FREED, N. und N. BORENSTEIN: *RFC 2046: Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types*, November 1996. Obsoletes RFC1521, RFC1522, RFC1590. Status: DRAFT STANDARD.

[FM99] FISCHER, STEFAN und WALTER MÜLLER: *Netzwerkprogrammierung unter LINUX und UNIX*. München ; Wien : Hanser ; (Unix Easy), 1999. 2., aktualisierte und erw. Auflage.

[Fok02] FOKINE, KLAS: *Key Management in Ad Hoc Networks*. Doktorarbeit, Linköping Univerity, Schweden, September 2002.

[GBB00] GRÄSSLE, PATRICK, HENRIETTE BAUMANN und PHILIPPE BAUMANN: *UML projektorientiert*. Galileo Computing, 2000.

[giF01] *giFT (GNU Internet File Transfer, an Generic Interface to FastTrack), (Stand April 2003)*, 2001. URL: `http://gift.sourceforge.net/`.

[GM01] GOLD, R. und C. MASCOLO: *Use of Context-Awareness in Mobile Peer-to-Peer Networks*. In: *8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001)*, Bologna, Italy, October 2001.

[Gnu01] *Gnutelliums - The comprehensive directory of Gnutella clients, (Stand April 2003)*, 2001. URL: `http://www.gnutelliums.com`.

[Her01] HERMANN, JÜRGEN: *Fork a daemon process on Unix, (Stand Mai 2003)*, 2001.

[iee] *IEEE P802.11, The Working Group for Wireless LANs, (Stand Mai 2002)*. URL: `http://grouper.ieee.org/groups/802/11/`.

[KCBC02] KON, FABIO, FABIO COSTA, GORDON BLAIR und ROY H. CAMPBELL: *The Case for Reflective Middleware*. Communications Of The ACM, 45(6), June 2002.

[LA01] LUTZ, MARK und DAVID ASCHER: *Einführung in Python*. O'Reilly Verlag, 1. korrigierter Nachdruck Auflage, 2001.

[Lau02] LAUER, MICHAEL: *Python und GUI-Toolkits*. mitp-Verlag, Bonn, 2002.

[Lau04] LAUER, MICHAEL: *ECOM: A Message–Passing Communication Framework*. Technischer Bericht, Goethe–University Frankfurt/Main, Frankfurt, Germany, 2004.

[LFCK94] LEAR, E., E. FAIR, D. CROCKER und T. KESSLER: *RFC 1627: Network 10 Considered Harmful (Some Practices Shouldn't be Codified)*, Juni 1994. Obsoleted by BCP0005, RFC1918 [RMK+96]. Status: INFORMATIONAL.

[LM02] LAUER, MICHAEL und MICHAEL MATTHES: *ELAN: An E-Learning Infrastructure for ad-hoc Networks*, 2002.

[LMD03] LAUER, MICHAEL, MICHAEL MATTHES und OSWALD DROBNIK: *A Framework for developing applications for ad-hoc environments*. In: *4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/-Distributed Computing (SNPD'03)*, Lübeck, Germany, Oct 2003.

[LMDW02] LAUER, MICHAEL, MICHAEL MATTHES, OSWALD DROBNIK und THORSTEN WODARZ: *A Framework for developing cooperative ad-hoc aware applications*. Technischer Bericht, Goethe–University Frankfurt/Main, Frankfurt, Germany, 2002.

[Lyn96] LYNCH, NANCY A.: *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[MCE01] MASCOLO, C., L. CAPRA und W. EMMERICH: *An XML-based Middleware for Peer-to-Peer Computing*. In: *In Proc. of the International Conference on Peer-to-Peer Computing (P2P2001)*, Seiten 69–74, Linkoping, Sweden, August 2001. IEEE Computer Society Press. Publikationen von L. Capra: `http://www.cs.ucl.ac.uk/staff/l.capra/publications.html`.

[Mön01] MÖNCH, CHRISTIAN ALEXANDER: *Eine verteilte Infrastruktur für typ- und diensterweiterbare orthogonale Digitale Bibliotheken*. Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften, Johann Wolfgang von Goethe Universität Frankfurt am Main, 2001.

[net] *Homepage of the netfilter/iptables project, (Stand April 2002)*.

[NV02] NARASIMHAN, NITYA und VENU VASUDEVAN: *Migratable Middleware for Mobile Computing*. In: *Autonomous Intelligent Networks and Systems*, 2002.

[opi] *Open Palmtop Integrated Environment (OPIE) - an Open Source based graphical user environment for PDA and other devices running Linux, (Stand Mai 2002)*. URL: `http://opie.handhelds.org/`.

[PC97] PARK, V. und M. S. CORSON: *Temporarlly–Ordered Routing Algorithm (TORA) Version 1 Functional Specification*. Internet Draft, Dec 1997.

[PyQ] *PyQt: A set of Python bindings for the Qt toolkit*. URL: `http://www.riverbankcomputing.co.uk/pyqt/`, (Stand April 2003).

[pyt] *Python Language Website, (Stand April 2002)*. URL: `http://www.python.org/`.

[Qt] *TROLLTECH Documentation: Qt C++ GUI Application Development Toolkit and Qtopia - Embedded Linux Graphical Application Environment*. URL: `http://doc.trolltech.com/`, (Stand April 2003).

[Riv92] RIVEST, R.: *RFC 1321: The MD5 Message-Digest Algorithm*, April 1992. Status: INFORMATIONAL.

[RMK+96] REKHTER, Y., B. MOSKOWITZ, D. KARRENBERG, G. J. DE GROOT und E. LEAR: *RFC 1918: Address Allocation for Private Internets*, Februar 1996. See also BCP0005. Obsoletes RFC1627, RFC1597 [LFCK94, RMKdG94]. Status: BEST CURRENT PRACTICE.

[RMKdG94] REKHTER, Y., B. MOSKOWITZ, D. KARRENBERG und G. DE GROOT: *RFC 1597: Address Allocation for Private Internets*, März 1994. Obsoleted by BCP0005, RFC1918 [RMK+96]. Status: INFORMATIONAL.

[Sch02] SCHMIDT, DOUGLAS C.: *Middleware for real-time and embedded systems*. Communications Of The ACM, 45, No. 6, June 2002.

[Sie01] SIERING, PETER: *WLAN-Wegweiser - Was man zum Aufbau eines 802.11b-Funknetzes braucht*. c't Magazin für computer technik, 18:122, 2001.

[SPP02] *SPP 1140: DFG Schwerpunktprojekt „Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Systeme“*, 2002. URL: `http://www.tm.uka.de/forschung/SPP1140/`, (Stand Mai 2002).

[Ste92] STEVENS, W. RICHARD: *Advanced Programming in the Unix Environment, (Stand Mai 2002)*. Addison-Wesley, 1992.

[SWW02] STOJMENOVIC, I., J. WU und B. WU: *Power-Aware Broadcasting an Activity Scheduling in Ad Hoc Wireless Networks Using Connected Dominating Sets*. In: *Proceedings of the IASTED International Conference „Wireless and Optical Communications (WOC2002)*, Canada, July 2002.

[Tel94] TEL, GERALD: *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[Ven02] VENKATASUBRAMANIAN, NALINI: *Safe 'Composability' of Middleware Services*. Communications Of The ACM, 45(6), June 2002.

[Vil00] VILLAZÓN, ALEX: *A Reflective Active Network Node*. In: *In Proceedings of the Second International Working Conference on Active Networks (IWAN 2000)*, Tokyo, Japan, 2000. Basislink: `http://cui.unige.ch/tios/staff/Alex.Villazon/publications.html`.

[Wod03] WODARZ, THORSTEN: *Entwicklung von editierbaren, korrelierbaren Interessensprofilen und deren Integration in ein Rahmenwerk für Kollaborative Anwendungen*. Diplomarbeit, Goethe–University Frankfurt/Main, Frankfurt, Germany, 2003.