technische universität
dortmund

PG-Endbericht

PG 526

„UbiMuC“
-
Ubiquitous Multimedia-Community

Fakultät für Informatik
Institut für eingebettete Systeme

Autoren:
Björn Bosselmann, Markus Görlich, Thomas Grabowski,
Nils Kneuper, Michael Kupiec, Lutz Krumme,
Fabian Marth, Michael Puchowezki, Markus Straube,
Stephan Vogt, Stefan Wahoff, Jens Wrede

Betreuer:
Dr. Heiko Falk
Constantin Timm

Abgabedatum:
31. März 2009

Inhaltsverzeichnis

1 Einleitung

Die Projektgruppe 526 – *Ubiquitous Multimedia-Community (UbiMuC)* – fand im Sommersemester 2008 und im Wintersemester 2008/2009 am Lehrstuhl 12 der Fakultät für Informatik der Technischen Universität Dortmund statt. Die Aufgabe bestand in der Entwicklung einer dezentral organisierten Multimedia-Community auf einer mobilen Plattform.

1.1 Motivation

Mit der zunehmenden Verbreitung von drahtlosen Netzwerktechnologien und deren Integration in Mobiltelefonen, Smartphones, Netbooks und Internet-Tablets wächst gleichzeitig das Bedürfnis nach entsprechender Software zur Nutzung und Interaktion mit den verschiedenen Technologien. Für den Multimediasektor gibt es derweil Insellösungen, die beispielsweise das Streaming von Inhalten durch einen dedizierten Medienserver zu drahtlosen Teilnehmern ermöglichen. Auf Seiten der drahtlosen Kommunikation mit Mobiltelefonen gibt es dank Smartphones mit WLAN und UMTS nun auch die Möglichkeit der Internet-Telefonie per VoIP, was jedoch mit recht hohen laufenden Kosten verbunden ist. Integrierte Webcams in Notebooks ermöglichen bei vorhandenem Internet-Zugang die Erstellung von Videokonferenzen durch Skype oder andere Programme. Alle der gerade erwähnten Lösungen benötigen jedoch immer eine vorhandene Infrastruktur oder expliziten Internet-Zugang.

Zielsetzung

Im Rahmen der Entwicklung einer mobilen Multimedia-Community sollen die einzelnen Teilnehmer über ein drahtloses Netzwerk verbunden werden und miteinander kommunizieren können. Im Mittelpunkt der Bemühungen steht dabei der konzeptionelle Entwurf einer solchen Softwareumgebung sowie die Integration verschiedener Softwaresubsysteme unter einer einheitlichen Oberfläche. Die Benutzer von *UbiMuC* sollen mit Hilfe der Software in die Lage versetzt werden, spontan untereinander Nachrichten und Informationen austauschen zu können, ohne dabei auf eine statische Netzwerkinfrastruktur angewiesen zu sein. Im Vordergrund des Unterfangens steht also die Dynamik des Netzwerkes und die Mobilität der Benutzer, weshalb in erster Linie Ad-Hoc-Netzwerke in Frage kommen. Statische Netze mit vorhandenen Access-Points sind nachrangig zu betrachten. Inhaltlich sollen dabei die besonders wichtigen Funktionalitäten zur Kommunikation gewährleistet sein: Der textuelle Datenaustausch soll durch ein Chatmodul ermöglicht werden, eine AV-Umgebung soll die integrierte Kamera und das Mikrofon ansprechen können und Videokonferenzen herstellen können. Die Teilnehmer sollen sich individuell in das *UbiMuC*-Netz mit frei wählbaren Pseudonymen anmelden, um so mit anderen Teilnehmern in der Umgebung in Verbindung treten zu können. Dabei sind

insbesondere die Videokonferenz zwischen Benutzern und der Austausch von textuellen Nachrichten von Bedeutung, da diese Funktionen bislang durch *Skype*, *AIM*, *ICQ* oder andere Instant-Messenger-Programme nur über statische Server möglich sind und zwingend Zugang zum Internet erfordern. Auf Basis von Ad-Hoc-Netzwerken sollen genau diese zentralen Funktionen von *UbiMuC* bereitgestellt werden.

Die ausgewählte Plattform

Als Endgerät ist das *Nokia* N810 ausgewählt worden, welches hauptsächlich durch das Linux-Subsystem eine gute Entwicklungsbasis aufweist und zusätzlich über eine Vielzahl von integrierten Funktion verfügt. Besonders erwähnenswert ist die im Gerät enthaltene Webcam, die in Kombination mit dem ebenfalls vorhandenen Mikrofon für den Austausch von audiovisuellen Mitteilungen sorgen kann. Die ausklappbare Daumentastatur ermöglicht zusätzlich das schnelle und komfortable Eingeben von Nachrichten. Auf technischer Seite ist außerdem die Unterstützung von drahtlosen Netzwerken über die WLAN-Standards 802.11b und 802.11g gegeben. Das N810 verfügt über eine Bluetooth-Schnittstelle, die von *UbiMuC* jedoch nicht verwendet wird. Die hohe Konnektivität in verschiedene Drahtlosnetzwerke ermöglicht so die Bildung von Ad-Hoc-Netzen, welche im Weiteren die Basis für *UbiMuC* darstellen. Ein detaillierter Blick auf das N810 soll hier jedoch nicht vorweggenommen werden, sondern wird in Abschnitt 1.4 gegeben.

Die Software-Subsysteme

Aufgrund des eingeschränkten Zeitrahmens wurde ein bestehendes Peer-to-Peer-Framework als Basis für die Entwicklung von *UbiMuC* benutzt. Die Entscheidung der Projektgruppe fiel dabei auf das Open-Source-Projekt *GNUnet*, welches ein dezentrales und hochgradig anonymes Peer-to-Peer-Programm mit Filesharing-Funktionen darstellt. *GNUnet* selbst setzt zwar auf ein eher dezentrales Peer-to-Peer-Konzept. Daher wurde das ebenfalls unter GNU-Lizenz stehende Programm *Avahi* für die Bildung von Ad-Hoc-Netzen mittels *ZeroConf* ausgewählt, welches im weiteren Verlauf in *GNUnet* eingebunden werden muss.

Projektplanung

Um die einleitend erwähnten Ziele und Vorstellungen in Form einer einheitlichen Software-Umgebung durch *UbiMuC* zu gewährleisten, ist eine Vielzahl an Einzelarbeiten und individuellen Lösungswegen notwendig. Als erste Ansatzpunkte und zur Etablierung einer adäquaten Planung müssen auf Basis der geschilderten Zielvorstellungen einzelne Anwendungsfälle ausgearbeitet werden, welche die *UbiMuC*-Umgebung realisieren sollte. Diese Anwendungsfälle müssen natürlich inhaltlich mit Lösungskonzepten und Strukturen gefüllt werden, so dass die geforderten Zielvorstellungen realisiert werden können. Was die Nutzung von externen Diensten und Frameworks betrifft, so ist deren strukturierte Analyse zur Integration der Funktionalitäten in *UbiMuC* unausweichlich.

Im Mittelpunkt stehen dabei besonders die Modifikationen des Peer-to-Peer-Frameworks, die für die Integration und reibungslose Zusammenarbeit mit *UbiMuC* nötig sind sowie die Entwicklung einer Zwischenschicht, welche die Daten zwischen dem Framework und den einzelnen *UbiMuC*-Funktionen vermittelt. Um entsprechende Modifikationen durchführen zu können, muss die ausgewählte Software detailliert betrachtet werden um ein Verständnis der Arbeitsweise zu entwickeln. Was die spätere Anpassung und Integration der benutzten Softwaresysteme betrifft, ist besondere Sorgfalt wichtig, da diese Komponenten für einen reibungslosen Ablauf der Ad-Hoc-Community *UbiMuC* unausweichlich sind.

Im Anschluss an erste konzeptionelle Arbeiten und die Planungsphase sind die eigentlichen Funktionalitäten von *UbiMuC* zu erstellen. Diese müssen dabei im Kontext des mobilen Endgerätes (Details in Abschnitt 1.4) und der geplanten Nutzungsszenarien entsprechend implementiert und getestet werden. Eine Herausforderung stellt dabei auch die Anpassung der Software an das Endgerät dar, da sowohl mit Schwierigkeiten hinsichtlich der vorhandenen Rechenleistung, als auch des Speicherplatzes zu rechnen ist. Weiterhin gilt es ebenfalls die Laufzeit der Batterie und die gesamte Gerätauslastung im Blick zu halten, da die nutzbaren Ressourcen durch die Zielplattform limitiert sind.

Während der Implementierungsarbeiten ist durch geeignete und gut dokumentierte Schnittstellen darauf zu achten, dass andere, parallel entwickelte Programmteile auf den eigenen Modulen und Funktionen aufsetzen können. Nach Abschluss der einzelnen Anforderungen beziehungsweise Fertigstellung von Unterfunktionen muss durch geeignet ausgewählte Tests und Probeläufe sichergestellt werden, dass die entwickelten Programmteile ihre Arbeit korrekt und wie gewünscht durchführen. Zuletzt ist ein Abschlusstest durchzuführen, der die anfänglich definierten Anforderungen mit der tatsächlich entwickelten Software abgleicht.

1.2 Anwendungsfälle

Im Fokus der hier folgenden Anwendungsfälle steht die Realisierung der bereits in der Einleitung umrissenen Nutzungsmöglichkeiten. Besitzer eines N810 sollen mithilfe der *UbiMuC*-Anwendung über ein spontan gebildetes Ad-Hoc-Netzwerk in die Lage versetzt werden, mit anderen *UbiMuC*-Teilnehmern Kommunikationsbeziehungen eingehen zu können. Vorab muss natürlich eine Anmeldung am Netzwerk erfolgen, damit überhaupt Informationen ausgetauscht werden können. Die erkannten Szenarien werden hier nun grob skizziert und nachfolgend weiteren Vorgehensweisen und Lösungskonzepten zugeordnet. Um die einzelnen Anwendungsfälle und deren Implikationen aber nicht aus dem Blick zu verlieren, werden diese den entworfenen Lösungskonzepten gegenübergestellt, die für die Realisierung der Szenarien zuständig sind. Abbildung 1.1 zeigt eine Skizze der zentralen Anwendungsfälle im Kontext der *UbiMuC*-Umgebung.

Im Vordergrund der Bemühungen standen dabei die Hauptanwendungsfälle Videokonferenz und Chat, da diese als Hauptaspekte von *UbiMuC* von zentraler Bedeutung sind und die eigentlichen Kommunikationsmöglichkeiten der Benutzer darstellen. Neben diesen zentralen Nutzungsszenarien mussten natürlich auch die darunterliegenden Konzepte und notwendigen Schnittstellen, wie ein Paketdienst und das Ad-Hoc-Netzwerk über *GNUnet*, implementiert werden, was jedoch keinen echten Anwendungsfall an sich darstellt und deswegen nicht extra abgebildet wurde. Im Zusammenhang mit der folgenden Roadmap in Abschnitt 1.5 für *UbiMuC*

wird auch noch auf ursprünglich geplante, aber dann später gestrichene Szenarien eingegangen und einige Hintergründe für die getroffenen Entscheidungen erläutert.

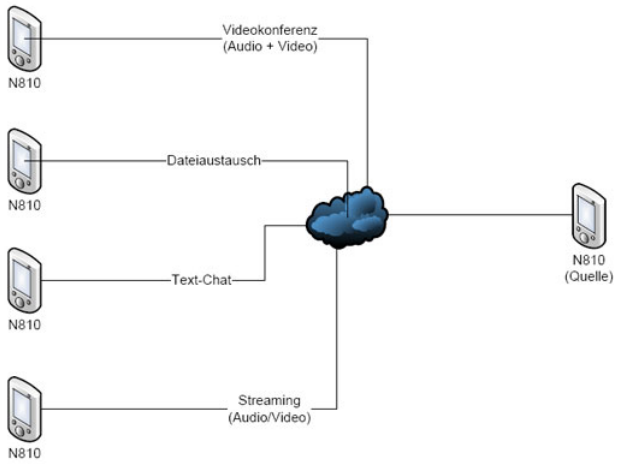

Abbildung 1.1: UbiMuC-Anwendungsfälle

Zentrale Anwendungsfälle

Die beiden anfänglich erwähnten Nutzungszenarien werden anhand der in Abbildung 1.2 gezeigten Videokonferenz exemplarisch dargestellt. Der konzeptuelle Unterschied zwischen Textnachricht und Videostream ist aufgrund der analogen Struktur bei unterschiedlich zu interpretierenden Daten zu vernachlässigen.

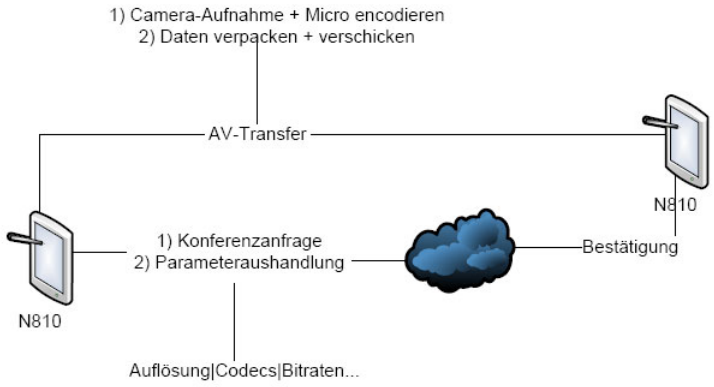

Abbildung 1.2: Anwendungsfall der Audio/Video-Konferenz

In Analogie zu einem Klingelton eines Mobiltelefons ist es vorgesehen, dass eine Videokonferenz einer vorherigen Autorisierung durch den *UbiMuC*-Benutzer bedarf. Die Optionsauswahl des Benutzers ist dabei auf die beiden Punkte „Annehmen“ und „Ablehnen“ beschränkt. Soll eine Konferenz aufgebaut werden, initiiert ein Teilnehmer den Verbindungsaufbau und sendet eine Konferenz-Anfrage an das Ziel ab. Besteht eine Route über das Netzwerk zum Ziel, kann dort die Anfrage akzeptiert oder abgelehnt werden. Im Falle der Akzeptanz werden auf beiden Seiten die integrierte Webcam und das Mikrofon des N810 aktiviert, um anschließend die Audio/Video-Konferenz zu ermöglichen. Nähere Informationen zum N810 und dessen Hardware sind in Abschnitt 1.4 zu finden.

Handelt es sich um einen textuellen Nachrichtenaustausch, so kann das Chat-Modul verwendet werden. Dabei sollen ausnahmslos Texte beliebiger Länge zwischen zwei Teilnehmern ausgetauscht werden. Eine Realisierung von Chat-Räumen oder Mehrteilnehmer-Sitzungen ist nicht vorgesehen.

Zuordnung der Anwendungsfälle zu Lösungskonzepten

Zur besseren Übersicht folgen nun die im einzelnen zu erledigenden Anwendungsfälle sowie deren Zuordnung zu den entwickelten Lösungskonzepten in Form von Tabelle 1.1.

Anwendungsfall	Lösungskonzept
Verbindung zum Netzwerk aufnehmen	*Avahi* und *GNUnet*
Personensuche	DHT und Kontaktliste
Datenweiterleitung	Eigene Datenstrukturen und Paketdienst über *GNUnet*
Audio/Video-Konferenz	Paketdienst, GStreamer
Chat	Paketdienst

Tabelle 1.1: Anwendungsfälle und deren Lösungskonzepte

Die Realisierung eines universell nutzbaren Paketdienstes stellt dabei die Hauptaufgabe dar, da *GNUnet* über keine solche Möglichkeit verfügt. *GNUnet* selbst ist darauf ausgelegt, auf Basis von Queries und Replies Daten auszutauschen, ein selbständiges Absenden von Informationen in das Netzwerk ohne vorher eingetroffene Anfrage ist aufgrund der Routing-Struktur von *GNUnet* gar nicht vorgesehen. Die Module Chat (siehe Abschnitt 4.4) und Audio/Video-Konferenz (siehe Abschnitt 4.5) benötigen allerdings genau diese Möglichkeit, da die Daten hier spontan ausgetauscht werden sollen, ohne dass Anfragen vorhanden sind. Mehr zur Struktur von *GNUnet* folgt jedoch im entsprechenden Abschnitt 2.2. In den späteren Abschnitten folgt zur Lösung dieser Problemstellung ein detailliertes Konzept (siehe Abschnitt 4.2) mitsamt Implementierung, weshalb an dieser Stelle nicht weiter darauf eingegangen wird. Anhand von Tabelle 1.1 ist aber bereits jetzt zu erkennen, dass die Funktionalität und Realisierung des Paketdienstes von essenzieller Bedeutung für das *UbiMuC*-Gesamtprojekt ist. Ohne diese Strukturen lassen sich die erstellten Anwendungsszenarien mithilfe von *GNUnet* nicht realisieren.

1.3 Minimalziele

Das Ziel der Projektgruppe *UbiMuC* ist die Realisierung eines Prototypen für einer mobilen Multimedia-Community zum Austausch von multimedialen Inhalten unter Einsatz des *Nokia* N810. Die Benutzer von *UbiMuC* sollen mit Hilfe der Software in die Lage versetzt werden, spontan untereinander Nachrichten und Informationen austauschen zu können, ohne dabei auf eine statische Netzwerkinfrastruktur angewiesen zu sein. Dabei ist explizit die Nutzung der WLAN-Funktionalität zur Datenübertragung gefordert. Daher soll die Kommunikation hierbei nicht wie in klassischen Ansätzen über einen zentralen Server abgewickelt werden, sondern in Form von Peer-to-Peer-Netzen. Als multimediale Inhalte gelten in unserem Fall insbesondere Instant Messaging, IP-(Video-)Telefonie und Filesharing.

Eine der großen Herausforderungen des Projektes ist es, den bei eingebetteten Systemen verfügbaren geringen Speicher und niedrige Rechenleistungen optimal zu nutzen. Insbesondere macht dies eine Anpassung der Peer-to-Peer-Strukturen bezüglich Algorithmen und Protokolle für mobile Geräte notwendig. Bedingt durch die Umsetzung in WLAN-Netzen ist es unabdingbar, Verschlüsselungsverfahren bei Datenübertragung einzusetzen. Daher müssen effiziente Verfahren gefunden werden, die sich mit den beschränkten Ressourcen vereinbaren lassen.

Als Minimalziel ist hierbei gefordert, dass *UbiMuC* sowohl in einem WLAN mit fester Infrastruktur als auch in einem Ad-Hoc-Netz lauffähig ist. Für die Sicherheit der übertragenen Daten soll eine effiziente Ver- und Entschlüsselung im Framework vorhanden sein. Aufgabe ist die Entwicklung eines Prototyps, der dieses Framework nutzt, sowie die Evaluierung unter den folgenden Gesichtspunkten:

- Geschwindigkeit und Ressourcenverbrauch von Streaming
- Kodierung und Dekodieren von Multimedia-Inhalten
- Ver- und Entschlüsselung der Daten
- Tests der Algorithmen und Protokolle auf Performanz

1.4 Hardware und Arbeitsumgebung

Dieser Abschnitt behandelt die der Projektgruppe 526 zugrundeliegende Hardwareplattform. Auf dieser soll das gesamte Projekt *UbiMuC* bei Abschluss der Projektgruppe lauffähig sein. Es wird sowohl auf Sinn und Zweck der Zielplattform eingegangen, als auch auf die Hardware und Software selbst. Des Weiteren werden für die Softwareentwicklung für dieses Gerät benötigte Grundlagen kurz angeschnitten.

Allgemeines zur Hardwareplattform

Als Zielplattform für die Projektgruppe war das von *Nokia* vertriebene „Internet-Tablet“ N810 vorgegeben. Dieses soll eine allgegenwärtige Internetnutzung ermöglichen. Bedingt durch den

mobilen Ansatz müssen entsprechend den Erfordernissen auch die Hardwareeigenschaften angepasst sein. Das bedeutet, dass eine hohe Akkulaufzeit und möglichst vielfältige Verbindungsmöglichkeiten vorliegen müssen. Des Weiteren muss das Gerät hinreichend klein sein, so dass es möglich ist, es immer bei sich zu tragen, vergleichbar mit einem Handy oder Smartphone. *Nokia* grenzt das Gerät dabei bewusst durch Aussparen des GSM/UMTS Senders/Empfängers von den Mobiltelefonen ab. Somit ist es mit dem N810 nicht möglich, auf klassische Art und Weise zu telefonieren, so dass hier Voice over IP eingesetzt wird.

Im Gerät selbst kommt ein OMAP2420 von *Texas Instruments* zum Einsatz. Diese System-on-Chip Lösung stellt sämtliche Grundfunktionen, wie eine ARM11 basierende CPU, einen 2D/3D Beschleuniger, einen digitalen Signalprozessor und einen „Imaging Video Accelerator“ zur Verfügung. Die ARM11 CPU ist mit 400 MHz getaktet und hat Zugriff auf 128 MB „mobile DDR memory“. Verbindung zur Außenwelt erhält man über das dem IEEE 802.11 b/g Standard folgende WLAN Modul oder via Bluetooth. Als Eingabegerät fungieren sowohl der berührungssensitive Bildschirm, als auch eine Daumentastatur. Weitergehende Informationen über die verbaute Hardware können auf der zugehörigen Internetseite von *Nokia* nachgelesen werden [1].

Softwareplattform N810

Das Betriebssystem des N810 ist linuxbasiert. Die verwendete Distribution baut auf *Maemo-Linux* auf, welches selbst von *Debian* abstammt. Es ist explizit auf die Nutzung mit kleinen, eingebetteten Systemen auf ARM Basis ausgelegt. Die grafische Oberfläche wird mithilfe von X11 dargestellt und sowohl ein Mediaplayer als auch ein Browser und Kommunikationssoftware sind entweder schon vorinstalliert oder durch den Nutzer nachträglich installierbar. Das gesamte System baut dabei weitestgehend auf nur leicht modifizierten Standardkomponenten auf, wie sie auch bei vielen Linuxdistributionen zu finden sind. Entsprechend sind viele der zur nachträglichen Installation angebotenen Programme Portierungen gebräuchlicher Software, wie zum Beispiel *MPlayer*[2] oder *Pidgin* [3], eine Multiprotokoll-Chat-Software.

Die Entwicklung von Software für das N810 geschieht über ein von *Nokia* zur Verfügung gestelltes „Software Development Kit“, das *Maemo 4.0 SDK*. Dieses stellt eine vom normalen System abgekapselte Umgebung namens *Scratchbox* [4] zur Verfügung, die das System eines N810 nachbildet. Von ihren Entwicklern wird sie als „cross compilation toolkit“ bezeichnet und enthält entsprechend sowohl einen Compiler, der X86 Binärdateien erstellt, die das Testen auf dem normalen System erlauben, als auch einen, der ARM Binärdateien ausgibt, so dass in der gelieferten Umgebung auch direkt die Pakete für das N810 gebaut werden können. Dies erlaubt es, Software für das N810 auf normalen Linux Systemen zu entwickeln und testen, ohne besondere Anforderungen an die für die Entwicklung zu nutzende Hardware zu stellen.

Zusammenfassung zur Hardwareplattform

Die Rechenleistung der CPU des N810 ist alles andere als überdimensioniert. Dies ist insbesondere für das Verschlüsseln von Daten, erforderlich für eine sichere Kommunikation, und das Enkodieren von Video- und Audio-Daten, wie es für die Arbeiten im Bereich Multimedia

(siehe Abschnitt 4.5) erforderlich ist, problematisch. Im Rahmen der Projektgruppe muss entsprechend auf dedizierte Einheiten, wie zum Beispiel den verbauten digitalen Signalprozessor, zugegriffen werden, um die gesteckten Ziele erfüllen zu können. Die Softwareentwicklung selbst hingegen ist vergleichbar mit der Entwicklung von Software für gebräuchliche Linux Systeme. So gibt es zum Beispiel ausgereifte Bibliotheken für die Entwicklung grafischer Oberflächen (siehe Abschnitt 2.6), sowie weit entwickelte Multimedia-Frameworks (siehe Abschnitt 4.5), die es ermöglichen auf zuvor geleisteter Arbeit aufzubauen, anstelle alles von Grund auf selbst zu implementieren.

1.5 Struktur des Endberichtes

Dieser Abschnitt behandelt grob die Struktur der noch folgenden Kapitel des Endberichts. Um eine Wissensgrundlage für weiterführende Herangehensweisen zu schaffen, schafft das zweite Kapitel eine Basis an Informationen. Hierbei werden essentielle Konzepte erläutert und deren Realisierung geklärt, welche meistens durch Fremdanbieter-Software gelöst wurde.

Zum besseren Verständnis des zugrunde liegenden Peer-to-Peer-Frameworks werden die Hauptelemente von *GNUnet* und die von *UbiMuC* benutzten Funktionen im dritten Kapitel näher beleuchtet. Dabei ist besonders das Verständnis der eigentlichen Arbeitsweise und Struktur von *GNUnet* ein wichtiger Aspekt, der für die darauf aufbauenden Konzepte unumgänglich ist. Erste Anpassungen des *GNUnet*-Basiscodes und eigens entwickelte Erweiterungen kommen ebenfalls dort zur Sprache.

Nach diesen einführenden Worten folgt innerhalb des vierten Kapitels der Hauptteil der durchgeführten Projektarbeiten. Darin enthalten sind detaillierte Lösungskonzepte und Implementierungswege zur Gewährleistung der gestellten Anforderungen an *UbiMuC*. Der anfängliche Entwurf wird dort um zusätzliche Abstraktionsebenen und passende Schnittstellen zwischen den Schichten ergänzt, damit eine vernünftige Kapselung von Datentypen und Informationsstrukturen sichergestellt ist. Innerhalb des vierten Kapitels befinden sich auch Ausführungen zu der besonders zentralen *UbiMuC*-Transportschicht, welcher ein eigenes Unterkapitel gewidmet wird.

Im fünften Abschnitt befindet sich der abschließend durchgeführte Test der fertigen Software. Das vorletzte Kapitel des Endberichts widmet sich einem abschließenden Fazit, worin die gestellten Anforderungen mit der entgültigen *UbiMuC*-Umgebung verglichen werden. Ebenso wird ein Ausblick auf weiterführende Arbeiten mit *UbiMuC* gegeben.

Der Anhang beinhaltet eine Einführung in *UbiMuC* in Form eines Benutzerhandbuchs und Installationshinweisen. Außerdem werden dort die wesentlichen GUI-Elemente erklärt und aufgezeigt, so dass eine leichte Benutzung von *UbiMuC* im Anschluss der Lektüre möglich ist.

2 Grundlagen

Die Entwicklung von *UbiMuC* stützt sich nicht nur auf selbstentwickelten Code und Konzepte. Vielmehr werden vorhandene Komponenten verwendet, die im ersten Semester der Projektgruppe schon erarbeitet wurden. Dieses Kapitel greift essentielle Themen des Zwischenberichtes noch einmal kurz auf, da sie die Grundlage für das weitere Vorgehen bildeten.

Hierbei handelt es sich sowohl um Konzepte, als auch um deren Implementierungen. Zunächst wird ein grober Überblick über Peer-to-Peer-Netzwerke und deren Umsetzung in *UbiMuC* mit *GNUnet* gegeben. Ein weiteres Konzept bilden Ad-Hoc-Netzwerke mit *ZeroConf*-Unterstützung. Als *ZeroConf*-Implementierung kommt hier *Avahi* zum Einsatz, welches ebenfalls kurz vorgestellt wird.

Nach den grundlegenden Konzepten und deren Umsetzung wird ein Überblick über die Struktur des bisherigen Projektes gegeben. Hierbei werden die einzelnen Komponenten und deren Verbindungen untereinander aufgezeigt. Es folgt eine gesonderte Betrachtung der im ersten Semester entwickelten Benutzeroberfläche.

Abgeschlossen wird dieses Kapitel durch die Projektplanung des zweiten Projektsemesters. Diese Planung zeigt die Abhängigkeiten der einzelnen Projektbestandteile, die in den folgenden Kapiteln beschrieben werden.

2.1 Peer-to-Peer-Konzepte

Das Peer-to-Peer-Konzept ist eine essenzielle Eigenschaft des Projektes *UbiMuc*. Um dieses Konzept zu verstehen, ist eine Definition des Begriffes „Peer-to-Peer“ notwendig. Professor C. Schindelhauer definiert es in seiner Vorlesung „Algorithmen für Peer-to-Peer-Netzwerke“ [5] wie folgt:

> Wenn man also die funktionale Beschreibung eines Netzwerks betrachtet, so gilt die folgende Definition:
>
> **Ein Peer-to-Peer-Netzwerk ist ein Kommunikationsnetzwerk zwischen Rechnern, in dem jeder Teilnehmer sowohl Client als auch Server-Aufgaben durchführt.**
>
> Obgleich diese Definition eine gute Intuition gibt, werden wir sehen, dass es tatsächlich Peer-to-Peer-Netzwerke gibt (z.B. Napster), die nicht unter diese Definition fallen. Daher ist die Peer-to-Peer-Working-Group [6] mit ihrer Definition etwas allgemeiner:
>
> **In einem Peer-to-Peer-Netzwerk werden verteilte Rechenressourcen durch direkte Kommunikation gemeinsam genutzt.**

> Mit dieser Definition ist man sicher zu allgemein. Das Einzige was man mit Sicherheit über Peer-to-Peer-Netzwerke sagen kann, ist dass Peer-to-Peer-Netzwerke nicht Client-Server-Netzwerke sind.

Diese Definitionen lassen sich auf *UbiMuC* übertragen. Alle Geräte übernehmen die gleichen Funktionen. Der Einsatz auf mobilen Geräten impliziert trotz geteilter Rechnerressourcen eine dynamische Netzstruktur, da Teilnehmer zu jeder Zeit dem Netz beitreten oder dieses verlassen können.

Da drei verschiedene Ausprägungen von Peer-to-Peer-Netzwerken existieren, musste eine der Folgenden für das *UbiMuC* Projekt ausgewählt werden:

- Serverbasierte Peer-to-Peer-Netzwerke
- Reine Peer-to-Peer-Netzwerke
- Hybride Peer-to-Peer-Netzwerke

Jede dieser Ausprägungen beinhaltet einige Vor- und Nachteile. Daher werden sie hier genauer betrachtet.

Serverbasierte Peer-to-Peer-Netzwerke

Diese Ausprägung von Peer-to-Peer-Netzen orientiert sich noch sehr stark an traditionellen Client-Server-Strukturen. Hierbei ist ein Server für die Verwaltung von Ressourcen und den Verbindungsaufbau zwischen zwei Peers zuständig. Demnach besteht dieses Netz aus heterogenen Knoten mit unterschiedlichen Aufgaben: Den Peers auf der einen Seite und dem Server auf der anderen Seite.

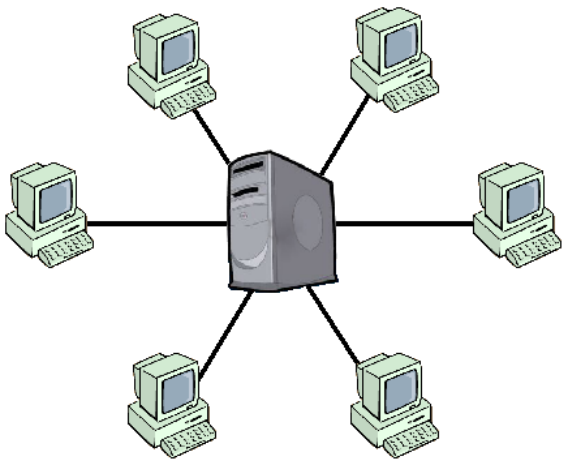

Abbildung 2.1: Aufbau eines serverbasierten Peer-to-Peer-Netzes

Serverbasierende Peer-to-Peer-Netze sind relativ leicht zu implementieren. Sie sind zudem leicht wartbar und einfach zu administrieren. Allerdings haben sie eine eklatante Schwachstelle: Fällt der Server aus, bricht das gesamte Netz zusammen. Dies macht das Netz extrem angreifbar, aber auch eine große Anzahl an Anfragen können den Server stark belasten. Des

Weiteren sind die Speicherressourcen des Servers begrenzt. Da auf diesem sowohl Typ und Beschreibung der der durch die Peers bereitgestellten Ressourcen als auch deren Adresse gespeichert werden müssen, ist die Größe des Netzes durch diesen beschränkt. Ein ebenfalls nicht zu unterschätzender Nachteil liegt in der permanent benötigten Erreichbarkeit und dem damit verbundenen Energiebedarf.

Reine Peer-to-Peer-Netzwerke

In reinen Peer-to-Peer-Netzwerken wird der Grundgedanke des Peer-to-Peer strikt verfolgt. Alle Peers im Netzwerk haben die gleichen Aufgaben, Pflichten und Rechte. Diese Topologie besteht demnach ausschließlich aus homogenen Knoten. Peers können Anfragen stellen und Ressourcen, Dienste oder ähnliches zur Verfügung stellen. Stellt ein nicht unmittelbar benachbarter Peer eine Ressource bereit, so werden Anfragen über die anderen Peers zu diesem weitergeleitet.

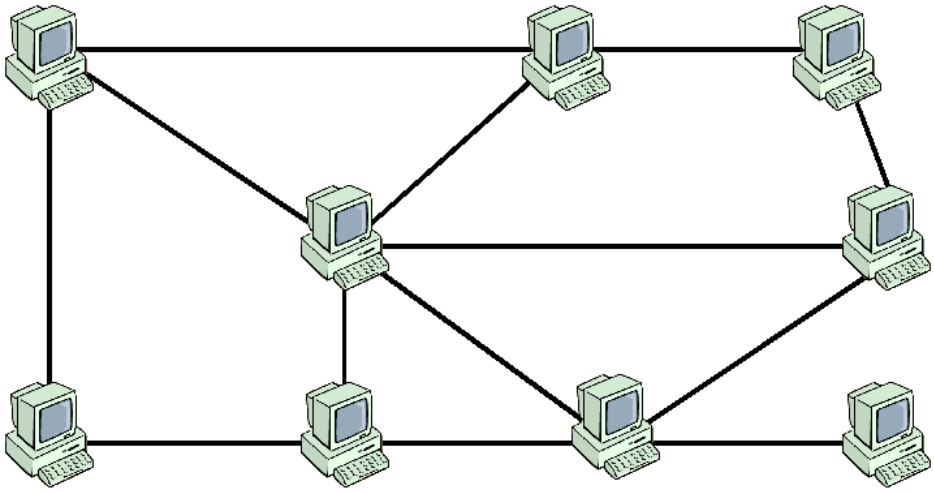

Abbildung 2.2: Aufbau eines reinen P2P-Netzes

Reine Peer-to-Peer-Netzwerke sind zwar weiterhin angreifbar, jedoch können durch die vollkommen dezentrale Struktur nur selten Schlüsselknoten ausgeschaltet werden. Das Wegfallen eines Peers lässt somit das bestehende Netz nicht zusammenbrechen. Eine hohe Ausfallsicherheit ist gewährleistet. Eine hohe Skalierbarkeit ist hierdurch ebenfalls gegeben. Es ist zudem energiesparend, da keine permanent betriebenen Server benötigt werden. Allerdings ist die Wartbarkeit des Netzes nahezu unmöglich. Die Suche nach Ressourcen ist durch die Time-to-live (kurz TTL) der Suchanfrage beschränkt. So sind Ressourcen einzelner Peers nicht im gesamten Peer-to-Peer-Netzwerk verfügbar.

Auch das Finden vorhandener Peers ist eine Herausforderung, für die es bislang keine optimale Lösung gibt. Ist ein Teilnehmer des Netzwerks bekannt, müssen dem beitretenden Peer Adressen weiterer Netzwerkteilnehmer mitgeteilt werden. In kleinen lokalen Netzwerken kann dieses über eine Broadcast-Anfrage realisiert werden. Bei großen Netzen, wie beispielsweise dem Internet, ist eine solche Methode jedoch nicht durchführbar.

Daher wird das sogenannte „Ping-Pong-Protokoll“ verwendet (siehe Abbildung 2.3). Hierbei sendet der beitretende Peer eine Ping-Anfrage mit einer TTL von beispielsweise drei in das Netzwerk. Jeder Peer, der diese Anfrage empfängt, sendet eine Pong-Antwort mit seiner eigenen Adresse an den beitretenden Peer zurück und verringert die TTL der Ping-Anfrage um eins. Zudem trägt er die Adresse des beitretenden Peers in seiner eigenen Hostliste ein. Auf diese Weise wird der beitretende Peer dem bestehenden Netz bekannt gemacht. Ist die TTL der Ping-Anfrage noch nicht abgelaufen, so sendet dieser Peer die Anfrage an ihn bekannte Peers weiter ins Netzwerk. Diese bekannten Peers empfangen die Anfragen und verarbeiten diese ebenso. Ist eine TTL von Null erreicht, senden die betreffenden Peers die Ping-Nachricht nicht weiter, sondern nur eine Pong-Nachricht an den beitretenden Peer. Die Adressen der Pong-Nachrichten werden in der Hostliste des beitretenden Peers eingetragen. So werden Peers des Netzwerks dem beitretenden Peer bekannt gemacht.

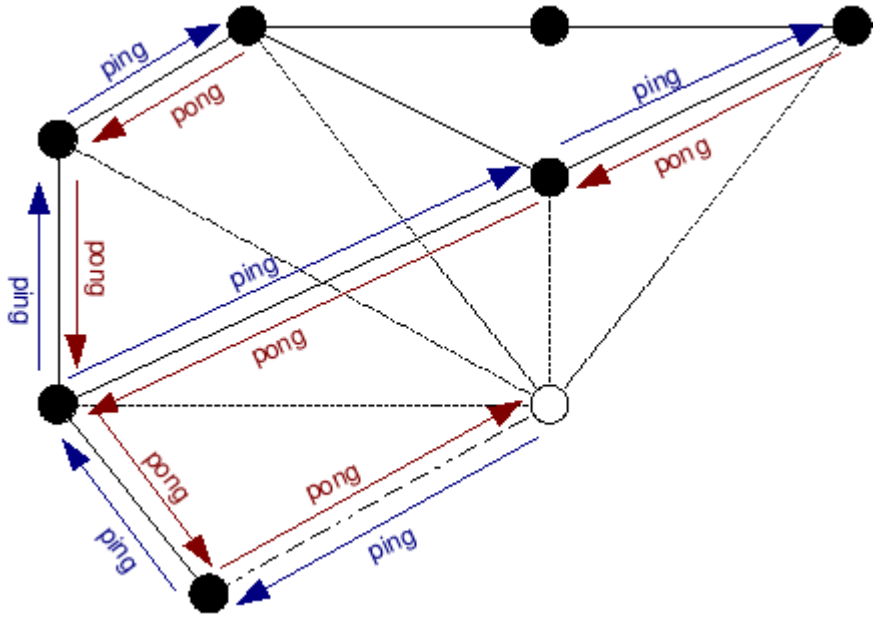

Abbildung 2.3: Bekanntmachen beitretender Peers unter Verwendung des „Ping-Pong-Protokolls“

Dieses Protokoll wird vornehmlich bei reinen Peer-to-Peer-Netzwerken eingesetzt, generell ist aber auch eine Verwendung in hybriden Netzwerken denkbar, die im folgenden Abschnitt beschrieben werden.

Hybride Peer-to-Peer-Netzwerke

Bei hybriden Peer-to-Peer-Netzwerken wurde versucht, die Vorteile von reinen und serverbasierten Peer-to-Peer-Netzen zu vereinen und die Nachteile zu minimieren. Daraus ergaben sich zwei Arten von Peers: Endknoten und Superpeers. Die Superpeers bilden das „innere“ Netz eines hybriden Peer-to-Peer-Netzwerks, welches als reines Peer-to-Peer-Netzwerk aufgefasst werden kann. Jeder Endknoten hat eine Verbindung zu einem Superknoten. Dieser fungiert als eine Art Server für diesen Endknoten.

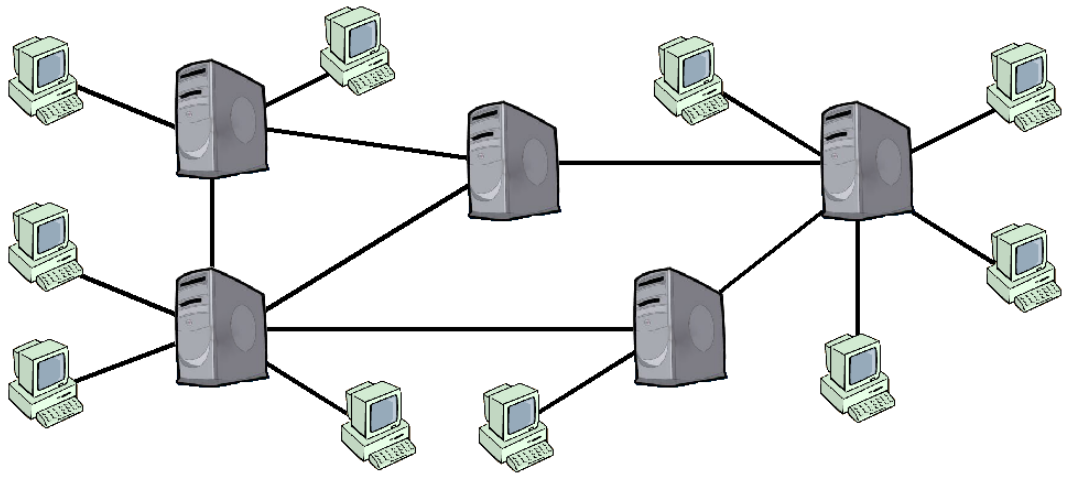

Abbildung 2.4: Aufbau eines hybriden P2P-Netzes

Ein Vorteil eines hybriden Peer-to-Peer-Netzwerks ist die verbesserte Skalierbarkeit im Vergleich zur Skalierung des serverbasierten Peer-to-Peer-Netzwerks. Durch die multiplen Superknoten kann das Netz zwar weiterhin von außen angegriffen, jedoch nicht so leicht komplett zerstört werden. Entfällt ein Superknoten, so bricht nicht das gesamte Netz zusammen. Zudem sind sie relativ leicht zu realisieren. Wie bei einem serverbasierten Peer-to-Peer-Netzwerk können an den Superpeers auch Engpässe entstehen. Weiterhin kann ein Superpeer durch die Vergabe von Prioritäten auch Peers bei der Bearbeitung von Suchergebnissen blockieren. Auch in hybriden Systemen müssen mehrere Superknoten permanent erreichbar sein, was wiederum einen nicht zu vernachlässigenden Energieverbrauch mit sich bringt.

Skalieren von Peer-to-Peer-Netzwerken

Um Peer-to-Peer-Netzwerke zu skalieren, müssen verschiedene Herausforderungen gelöst werden. Informationen müssen ausgewogen über das Netzwerk verteilt werden. Außerdem muss verhindert werden, dass Peers veraltete Daten zur Verfügung stellen. Verlässt ein Peer das Netzwerk, müssen die von ihm verwalteten Daten von anderen Peers zur Verfügung gestellt werden. Um diese Informationskohärenz und Fehlertoleranz zu gewährleisten, gibt es verschiedene Datenstrukturen. Diese sind in der folgenden Liste aufgeführt:

- Distributed Hash Table (DHT)
- Content Addressable Network (CAN)
- CHORD
- Koorde
- Tapestry
- Pastry
- Viceroy
- Distance Halving

Relevanz für *UbiMuC*

Außer Frage steht, dass Peer-to-Peer-Netzwerke die Grundlage für *UbiMuC* bilden. Da das Projekt auf mobilen Geräten realisiert werden soll und somit keine zentralen Server oder Superpeers permanent zur Verfügung stehen, wird das Netz als reines Peer-to-Peer-Netzwerk realisiert.

Um Daten in UbiMuC zu verwalten, wird eine Distributed Hash Table verwendet. Diese wird später genauer beschrieben (siehe Abschnitt 3.1.3.1).

2.2 *GNUnet*

Die *UbiMuC*-Anwendung setzt als Peer-to-Peer-Framework auf *GNUnet*, welches ein hochgradig anonymes, zensurresistentes und dezentrales Filesharing-Netzwerk bereitstellt (vgl. [7]). *GNUnet* stellt seinen Anwendern dabei vorrangig Möglichkeiten zum Download und zur Suche von im Netzwerk veröffentlichten Inhalten bereit. Das *GNUnet*-Netzwerk besteht aus einer Vielzahl einzelner Teilnehmer, den sogenannten Peers. Jeder Peer kann zu jeder Zeit dem Netzwerk beitreten, Daten veröffentlichen und herunterladen oder auch das Netzwerk verlassen.

Besonders zu erwähnen ist an dieser Stelle auch, dass *GNUnet* durch den Einsatz von Verschlüsselungsmethoden (symmetrisch und asymmetrisch) beziehungsweise Sicherheitskonzepten für eine sehr hohe Vertraulichkeit und Datensicherheit innerhalb des Netzwerks sorgt. Strukturell identifiziert *GNUnet* die einzelnen Teilnehmer des Netzwerks anhand von eindeutigen 64 Byte langen Hashwerten, die gleichzeitig als *GNUnet*-Adressen interpretiert werden und für den Datenaustausch unabkömmlich sind. Diese Identifikation ist nötig, um auf dem Public-Key-Prinzip gesicherte Datenkanäle zwischen den einzelnen Teilnehmern aufzubauen. Die Datentransfers werden innerhalb von *GNUnet* grundsätzlich symmetrisch verschlüsselt, so dass nur der designierte Empfänger die Informationen entschlüsseln kann.

Um die Herkunft von Daten und deren Wege durch das Netzwerk zu verschleiern, benutzt *GNUnet* innerhalb des Filesharing-Moduls ein gesondertes GAP-Routing (siehe auch [7]). Dieses Routingverfahren beinhaltet lokale Routing-Tabellen bei den Teilnehmern, da jeder Peer potenziel als Hop fungieren muss, wenn er Anfragen erhält und diese weiterleitet. Außerdem schleust *GNUnet* über das Filesharing-Modul selbstständig Datenpakete in das Netzwerk, um das allgemeine Kommunikationsaufkommen zu maskieren. Näheres zum Routingverfahren folgt in der Passage des Filesharing-Moduls, da das GAP-Routing dort als fester Bestandteil eingebaut ist. All diese Verfahren gewährleisten dabei ein hohes Maß an Anonymität und Datensicherheit über ein potentiell unsicheres Basismedium.

Insgesamt ist die Struktur von *GNUnet* sehr modular gehalten, was die Einbindung oder Deaktivierung einzelner Funktionalitäten begünstigt. Die besonders wichtigen Module werden dabei nachfolgend beschrieben, ebenso wie auf die Gesamtstruktur von *GNUnet* eingegangen wird.

Prozessstruktur

GNUnet selbst besteht aus einem lokalen Server- und einem Client-Prozess, welche beide getrennt voneinander arbeiten. Der *GNUnet*-Server gnunetd, auch als „Core“ bezeichnet, ist für die Verwaltung sämtlicher Peer-to-Peer-Verbindungen und Datenaustausche mit anderen *GNUnet*-Teilnehmern zuständig. Er kapselt darüber hinaus die gesamte Netzwerkebene inklusive der Verschlüsselung von Daten, der Adressverwaltung und dem Bandbreitenmanagement. Die Verbindung zwischen den beiden Prozessen wird über lokale Nachrichten hergestellt, wobei ein vorhandener *GNUnet*-Client die eigentliche Schnittstelle zwischen Benutzer und *GNUnet*-Netzwerk bzw. anderen Teilnehmern bildet. Die durchgeführten Client-Aktionen des Benutzers werden zur Verarbeitung und Durchführung an den Kern weitergeleitet. Näheres zur allgemeinen Kommunikation zwischen *UbiMuC* und *GNUnet* findet sich in Abschnitt 3.2.

Das Filesharing-Modul

Da *GNUnet* im Wesentlichen als reines Filesharing-Programm entworfen ist, realisiert es die Kommunikation zwischen den Teilnehmern hauptsächlich auf der Basis des PULL-Prinzips. Stellt ein Teilnehmer also eine Such- oder Downloadanfrage mit Hilfe des Filesharing-Moduls von *GNUnet*, so wird die Suchanfrage an umliegende Peers weitergeleitet. Diese prüfen nun, ob sie die Anfrage beantworten können oder leiten die Suche an andere Peers weiter. Im Falle der Weiterleitung speichern sie jedoch, dass sie eine Anfrage weitergeleitet haben, von welchem Peer sie eintraf und an welche Peers sie weitergeleitet wurde. Nach dem Ablauf einer festen Zeitspanne verfallen diese lokalen Routingtabellen jedoch. Sendet ein Peer nun Antworten auf die Anfrage, ist jeder der einzelnen Peers in der Lage, die Daten weiterzuleiten, so dass diese bei der Anfragequelle ankommen können. Auf diese Weise wird ein sehr flexibles Routing der Daten über das Netzwerk ermöglicht, ohne das einem externen Betrachter der Ursprung der Anfrage oder die eigentliche Datenquelle offenbart wird. Das Routingskonzept hat allerdings als Konsequenz, dass Nutzdaten zwischen den Teilnehmern nur dann ausgetauscht werden können, wenn es eine Anfrage und darauf passende Antworten gibt, da ohne dieses Vorgehen keine passenden Zuordnungen bei den Zwischenpeers vorliegen und kein Routing der Daten stattfinden kann. Die eigentlichen Anfragen des Filesharing-Moduls werden inhaltlich in Form von Hashwerten und dazu korrespondierenden Datenblöcken gestellt.

Um den anfänglich erwähnten hohen Grad der Anonymität innerhalb des *GNUnet*-Netzes zu gewährleisten, werden zwei zentrale Konzepte angewendet: Auf der einen Seite werden die beschriebenen Anfragen und Antworten auf eine einheitliche Größe gebracht, bevor sie abgesendet werden. Zusätzlich wird durch sog. „Covertraffic“ sämtliches Anfrage- und Antwortverhalten der Teilnehmer maskiert. Zur Gewährleistung eines hohen Sicherheits- und Integritätsmaßes innerhalb von *GNUnet* wird die Kommunikation darüber hinaus durch Nutzung von asymmetrischer Kryptographie abgesichert. Für einen externen Zuschauer ist also nur ersichtlich, dass eine Kommunikation stattfindet, nicht jedoch deren syntaktischer oder semantischer Inhalt. Das GAP-Routing ist fest ins das Filesharing-Modul integriert und steht auch nur für die damit verbundenen Teilfunktionen bereit.

Das Hostlisten-Modul

Jeder *GNUnet*-Client verfügt in der Basiseinstellung über eine hartkodierte Adresse, die er bei Verbindungsaufname abfragt. Unter dieser Adresse ist kontinuierlich eine sogenannte Hostliste zu finden, welche die zum aktuellen Zeitpunkt dort vorhandenen HELLO-Pakete enthält. *GNUnet* benötigt zum späteren Aufbau von Kommunikationskanälen diese HELLO-Nachrichten der Teilnehmer, mit denen ein Datenaustausch stattfinden soll.

Innerhalb eines HELLO-Pakets befindet sich unter anderem der öffentliche Schlüssel des Absender-Peers und ein Zeitstempel. Mit Hilfe dieses Schlüssels lassen sich auf Basis des Public-Key-Prinzips verschlüsselte Verbindungen zum Besitzer des zugehörigen privaten Schlüssels aufbauen. Dieser gesicherte Kanal wird allerdings nicht zum Austausch von Massendaten benutzt, sondern nur für die Abstimmung eines symmetrischen Sitzungsschlüssels verwendet, der dann für die eigentliche Verschlüsselung benutzt wird.

Durch eine periodisch angestoßene Weiterleitung von vorhandenen HELLO-Paketen sorgt das Hostlisten-Modul selbstständig für eine Verbreitung der Teilnehmeradressen. Diese Nachrichten werden von allen verbundenen Peers zufällig ins Netzwerk gesendet, um anderen Peers von der eigenen Existenz zu berichten. Auch werden empfangene HELLO-Pakete von anderen Peers weitergeleitet. Um eine Verbindung zu einem anderen Peer aufzubauen, ist für *GNUnet* daher das Vorhandensein eines besagten HELLO-Pakets notwendig, da sonst keine Abstimmung eines Sitzungsschlüssels stattfinden kann.

Das DHT-Modul

GNUnet besitzt eine verteilte Datenstruktur in Form einer verteilten Hashtabelle, abgekürzt DHT. Innerhalb dieser DHT können Daten über mehrere Peers hinweg bereitgestellt werden, ohne dass eine zentralisierte Datenstruktur benötigt wird. Außerdem kann die DHT anhand von Verteilungs- und Migrationsfunktionen dafür sorgen, dass darin gespeicherte Inhalte dynamisch und nicht vorhersagbar unter den *GNUnet*-Teilnehmern verbreitet werden. *GNUnet* selbst bietet über das DHT-Modul eine Schnittstelle an, die das Eintragen von Daten in die DHT sowie das spätere Abfragen erlaubt. Details zur DHT und den Schnittstellen werden in den Abschnitten 3.1.3.1 und 4.3.1.2 gesondert behandelt.

Das Statistik-Modul

Zur statistischen Auswertung der Datentransfers steht in *GNUnet* ein gesondertes Modul bereit. Dort können unter anderem die versendeten und empfangenen Datenpakete in Form eines Graphen visualisiert werden. Im Wesentlichen dient das Modul jedoch nur einer recht oberflächlichen Einordnung der Netzwerkauslastung, um dem Benutzer einen groben Eindruck über den vorliegenden Datendurchsatz zu geben.

Nachrichtentypen und Protokolle

Zur Nutzung des *GNUnet*-Netzwerkes für *UbiMuC* war es notwendig, eigene Nachrichtentypen und dazu passende Callback-Funktionen innerhalb des *GNUnet*-Serverprozesses und des Clients zu registrieren. Der *GNUnet*-Kern behandelt Nachrichten anhand ihres Typflags und ordnet sie den dazu passenden Behandlungsmethoden zu und ruft diese auf. Die Zuordnung zwischen Nachrichtentyp und Callback-Funktion muss daher stattgefunden haben, da sonst keine Behandlung eben dieser neuen Typen möglich ist. Diese neuen Nachrichtentypen werden von *UbiMuC* dazu benutzt, um eigene Nachrichtenformate zu erstellen und selbstentwickelte Protokolle über *GNUnet* zu betreiben.

Der modulare Aufbau von *GNUnet* erleichterte dabei die Realisierung der eigenen *UbiMuC*-Nachrichtentypen. Diese wurden per Namensgebung und eindeutiger Nummerierung innerhalb des Core in die entsprechende Protokolldatei eingetragen und standen dann für die allgemeine Nutzung bereit. Ohne diese vorherige Registrierung und Bekanntmachung der Typen würden von *UbiMuC* an *GNUnet* geschickte, eigene Nachrichten verworfen, da *GNUnet* über keine passende Handlermethode verfügt bzw. den Nachrichtentyp gar nicht kennt und entsprechend verwirft. Details zu den einzelnen Nachrichtentypen und konkreten Implementierungen werden dabei im Unterabschnitt 4.2.1.1 behandelt.

Abschlussbetrachtung

Nach dieser ersten Betrachtung von *GNUnet* ist bereits ersichtlich, dass einige Module angepasst und erweitert werden müssen, während auf der anderen Seite einige Module garnicht benötigt werden. Im nachfolgenden Kapitel 3 werden die notwendigen *GNUnet*-Modifikationen erläutert sowie genauer beschrieben. Auch wird dort der auf *GNUnet* aufsetzende Paketdienst in den Grundzügen behandelt.

2.3 Ad-Hoc-Netzwerke

Bei Ad-Hoc-Netzwerken handelt es sich um einen speziellen Typ von Funknetzen. Die einzelnen Netzteilnehmer stehen dabei über ein vermaschtes Netzwerk in Verbindung. Es besteht keine feste Struktur, insbesondere können die Teilnehmer auch mobil sein. Die Größe des Netzes bestimmt sich über die Reichweite der Funksignale.

Durch die fehlenden Strukturen müssen die Teilnehmer eines Ad-Hoc-Netzwerkes zusätzlich zur eigenen Netznutzung die Routing-Funktionalitäten eines festinstallierten Netzwerkes übernehmen. Die üblichen Routing-Mechanismen für festinstallierte Netzwerke greifen bei Ad-Hoc-Netzwerken in der Regel nicht, da einige weitere Bedingungen berücksichtigt werden müssen, die in statischen Netzwerken nicht auftreten. Zum einen ist bei den beteiligten Geräte nicht immer Wissen über die Netzwerktopologie vorhanden. Die Routing-Entscheidungen sind also gegebenenfalls ohne dieses Wissen zu treffen.

Darüber hinaus muss jedes einzelne Gerät eine eigene Routing-Tabelle verwalten, da es keine übergeordnete Instanz gibt, die dieses übernimmt. Durch die Mobilität der einzelnen Netzteilnehmer muss ständig mit einer Topologieänderung gerechnet werden. Das heißt, dass die

in den Routing-Tabellen vorhandenen Daten keine besonders lange Gültigkeitsdauer haben sollten. Schließlich sollte das Routing trotz allem möglichst effizient gestaltet werden, damit einerseits die durch das Routing entstehenden Latenzzeiten möglichst gering bleiben und damit andererseits sowohl Prozessor als auch Energieversorgung nicht übermäßig beansprucht werden.

Bei den möglichen Routingkonzepten wird grundsätzlich zwischen zwei Ansätzen unterschieden. Zum einen existieren proaktive Verfahren, die schon vor einer eventuellen Datenübertragung die Netztopologie analysieren und Routing-Pfade ausarbeiten. Beim Versand von Nutzdaten muss damit nicht auf den Aufbau einer Route gewartet werden. Allerdings besteht so beim initialen Aufbau des Netzes ein erhöhter Aufwand und es kann passieren, dass Routen gebildet werden, die mangels Nachfrage nicht benutzt werden.

Bei Netzen mit mobilen Teilnehmern kann es zusätzlich dazu kommen, dass eine zu Beginn ermittelte Route zum Zeitpunkt der Nutzung nicht mehr existiert. Ist dies der Fall, muss als Reaktion auf den Datentransfer eine neue Route aufgebaut werden. Wird dies grundsätzlich durchgeführt, spricht man von reaktiven Verfahren, dem Gegenstück zu proaktiven Verfahren. Diese bilden die Routen ausschließlich bedarfsorientiert. Das heißt, dass erst bei Nutzdatenversand eine Route zum Zielpeer gesucht wird. Der Versand der Nutzdaten wird dadurch zwar verzögert, aber es werden keine unnötigen Routen aufgebaut. Wir haben uns im Rahmen unseres Projektes für ein reaktives Verfahren entschieden. Näheres dazu im Kapitel 4.2.1.2.

2.4 *Avahi/ZeroConf*

Um innerhalb von Ad-Hoc-Netzwerken TCP/IP-basierte Kommunikation zu ermöglichen, muss die Verteilung von IP-Adressen dezentral bewerkstelligt werden. Bei Netzwerken mit statischer Benutzerzahl und bekannten Benutzern ist es möglich, diese im Voraus fest zuzuteilen. In unserem Fall ist weder das eine noch das andere gegeben, weswegen wir auf dynamische Zuteilungsverfahren zurückgreifen.

Für eine dynamische IP-Adressvergabe existiert ein Standard, *Zero Configuration Networking* oder auch kurz *ZeroConf*. Dieser regelt die Zuteilung von IP-Adressen aus dem Raum 169.254.0.0/16. *Avahi* ist eine unter Linux lauffähige und freie Implementierung des *ZeroConf*-Standards.

Da dieser Standard prinzipbedingt ohne zentrale Kontrollinstanz auskommen muss, werden im Standard ebenfalls Mechanismen zur Konfliktbewältigung bei mehrfach vergebenen Adressen festgeschrieben. Die Adresse wird pseudozufällig gewählt, damit bei jedem Neuverbinden mit dem Netzwerk nach Möglichkeit die selbe Adresse ausgewählt wird.

Weiterhin sieht der Standard eine dezentrale Namensauflösung und einen Mechanismus zum Bekanntgeben und Auffinden von Netzdiensten vor. Die dezentrale Namensauflösung verläuft analog zum bekannten DNS-System des Internets, allerdings kommt hier kein Server zum Einsatz, sondern die Anfragen werden an eine Multicast-Adresse versendet und vom Namensinhaber beantwortet.

Die Zuteilung von IP-Adressen wird bereits vor dem Start von *UbiMuC* bei der Auswahl der Verbindung am N810 durchgeführt. Die beiden anderen beschriebenen Dienste, die Namensauflösung und das Dienstesystem, werden durch *UbiMuC* genutzt. Durch unsere Implementierung wird sichergestellt, dass alle Geräte innerhalb ihres Empfangsradius unterschiedliche Namen besitzen. Dies wird zwar vom Standard nicht vorgesehen, da es durchaus Anwendungsfälle für solche Szenarien geben kann, wie zum Beispiel die Lastverteilung auf gleichartigen Netzwerk-Ressourcen. Deswegen müssen wir unterschiedliche Namen durch unsere Implementierung sicherstellen.

Weiterhin wird über Avahi der *GNUnet*-Dienst bekanntgegeben. Alle Peers, die sich im aktuellen Empfangsradius befinden, erhalten unmittelbar nach Bekanntgabe des Dienstes diese Information und bauen direkt eine Verbindung zum neu hinzugekommenen Peer auf. Der Peer, der sich neu ins Netz einbucht und den *GNUnet*-Dienst bekannt gibt, sendet seinerseits auch eine Anfrage nach bereits vorhandenen Diensten und baut eine Verbindung zu allen Peers auf, die auf diese Anfrage geantwortet haben.

Zum Verbindungsaufbau werden per http-Anfrage die Hostlisten der anderen Peers angefordert. In diesen stehen dann unter Umständen auch schon weitere Peers, die sich momentan außerhalb des eigenen Empfangsradius befinden, die er aber über *GNUnet* trotzdem erreichen kann. (Näheres zu den Hostlisten unter 3.1.3.2.) Nach Herunterladen der Hostliste kann der Peer beginnen, sich an der Verteilten Hashtabelle zu beteiligen, um die Personensuche aufnehmen zu können.

2.5 Struktur von *UbiMuC*

Dieses Kapitel behandelt die Schicht zwischen *GNUnet* und der GUI, dem sogenannten WIR Interaction Relay, kurz *WIR*. Damit die Basisfunktionen von *GNUnet* von unserer Anwendung genutzt werden können, ist es erforderlich, die zentralen *GNUnet*-Programmteile zu kapseln und so für uns nutzbar zu machen. Durch die Kapselung sollen weitergehende Eingriffe in den *GNUnet*-Quellcode umgangen werden. Im Falle von späteren Weiterentwicklungen oder Codeveränderungen durch die *GNUnet*-Community ist es dann höchstens notwendig, unsere Schnittstellen anzupassen. Alle Anfragen unserer Anwendung an das *GNUnet*-Netzwerk sollen in diesen Interface-Klassen übersetzt und angepasst werden. Auf der anderen Seite müssen dort ebenfalls die Antworten von *GNUnet* interpretiert und unter Umständen aufbereitet werden.

Weiterhin ist in der WIR-Schicht unsere eigene Programm-Logik untergebracht. Es werden von dort aus Nutzer und Kontaktdaten zu anderen Peers verwaltet, sowie unsere Paketdienstverwaltung realisiert. Die von der Schicht bereitgestellten Informationen werden schließlich von der GUI lediglich dargestellt, während die eigentliche Verarbeitungsintelligenz innerhalb der Adapterklassen liegt. Auf diese Weise sollte ein Ersetzen der GUI in der Zukunft deutlich vereinfacht werden, da lediglich die Kommandoweiterleitung und Darstellung der Ergebnisse angepasst werden müssen.

Interaktion mit der WIR-Schicht

Neben der oben genannten Schnittstellen-Funktion der WIR-Schicht, die einzelne Komponenten wie GUI, *GStreamer* und *GNUnet* verbindet, hat die WIR-Schicht insbesondere noch die Aufgabe, dafür Sorge zu tragen, dass die aufgenommenen Informationen von Webcam und Mikrofon passend kodiert und für *GNUnet* verpackt werden.

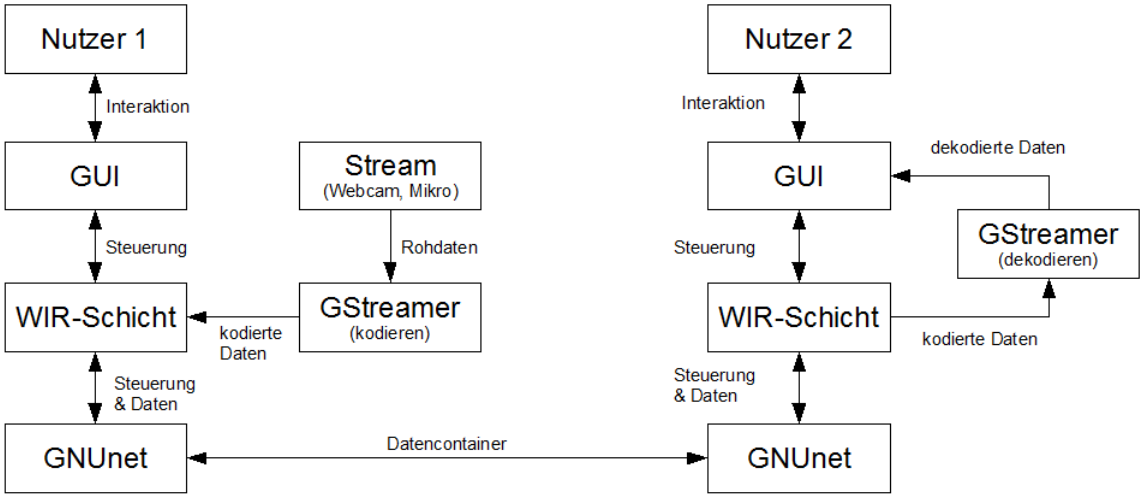

Abbildung 2.5: Schema der *UbiMuC*-Struktur

In Abbildung 2.5 wird das Zusammenspiel der einzelnen Komponenten der *UbiMuC*-Plattform bei einem Datenaustausch zwischen zwei Nutzern graphisch dargestellt. Die Pfeilrichtung gibt dabei die Flussrichtung der Befehle und Daten an.

Wählt der Nutzer beispielsweise die Videokonferenz-Option in der GUI aus, wird eine *GStreamer*-Instanz starten. Diese steuert selbstständig die im N810 eingebaute Webcam und das Mikrofon an. Die daraus gewonnenen Video- und Audiorohdaten werden vom *GSteamer*-Prozess in Echtzeit in ein Format konvertiert, das sich relativ leicht über ein Netzwerk streamen lässt. Auf die Details der Multimediaverarbeitung wird genauer in Kapitel 4.5 eingegangen. Die kodierten Multimediadaten werden in der WIR-Schicht an den Paketdienst weitergereicht, der sie für den Versand auf *GNUnet*-Ebene vorbereitet. In *GNUnet* sind unsere selbst definierten Nachrichten durch unsere Anpassungen registriert, so dass anschließend die Pakete über die *GNUnet*-Schicht im Netzwerk übertragen werden können, ohne das *GNUnet* deren Inhalte kennen muss.

Auf der Empfängerseite verarbeitet die WIR-Schicht auf analoge Weise die Pakete von *GNUnet*, allerdings in umgekehrter Reihenfolge. Der Paketstrom wird von unserem Paketdienst wieder zu einem kontinuierlichen Datenstrom zusammengesetzt und an den *GStreamer* weitergereicht. In der GUI läuft schließlich der *GStreamer* eingebettet und zeigt den Multimedia-Stream an. Chatnachrichten können auf ähnliche Weise verarbeitet und übertragen werden. Jedoch entfällt hierbei die Kodierung und Dekodierung der Daten, da der Text auch direkt in unseren Paketen als Nutzdaten verpackt werden kann.

Inhalte der WIR-Schicht

Die WIR-Schicht fasst, wie bereits dargestellt, alle selbst entwickelten *UbiMuC*-Module zusammen. Die übergeordnete Verwaltung dieser eigenen, sowie der eingebunden Komponenten, übernimmt dabei der so genannte WIR-Core-Bereich. Allgemein umfasst dieser alle Funktionalitäten die zum Initialisieren, Starten und Beenden unserer anderen Module benötigt werden. Es wird beim Programmstart sichergestellt, dass Werte aus einer Konfiguration-Datei geladen werden, oder, wenn keine sinnvollen Daten gefunden werden, Standardwerte gesetzt werden, so dass *UbiMuC* grundsätzlich funktionsfähig ist. Anschließend wird sichergestellt, dass der *GNUnet*-Dienst gestartet ist. Sollte er nicht bereits laufen, so wird er nun mit den aus der Konfiguration ermittelten Werten gestartet.

Sobald *GNUnet* läuft, werden unsere eigenen Dienste gestartet und die Benutzeroberfläche initialisiert. Zur Laufzeit des Programms findet im WIR-Core auch die Ausnahmebehandlung für den Fall statt, dass *GNUnet* unerwartet terminiert, oder dass andere Fehlerfälle auftreten. Beim Beenden werden schrittweise die einzelnen Komponenten wieder freigegeben und der *GNUnet*-Dienst wird beendet, falls er von *UbiMuC* gestartet wurde. Über die bisher beschriebenen Verwaltungsaufgaben hinaus enthält die WIR-Schicht die Module für die Nutzerverwaltung, den Paketdienst sowie die Multimedia-Ansteuerung und den Chat. Auf diese wird in Kapitel 4 genauer eingegangen.

2.6 Benutzeroberfläche

Bereits vor dem eigentlichen Implementieren wurden gewisse Anforderungen an die GUI formuliert. Als oberstes Ziel wurde dabei gewählt, dass sämtliche wichtigen (oft benutzten) Funktionen leicht zugänglich sind. Des Weiteren sollte die GUI das von anderen Programmen des N810 gewohnte „look-and-feel“ bieten, so dass sich der Nutzer sofort zurecht findet. Dieser Ansatz wurde durch folgenden Aufbau realisiert.

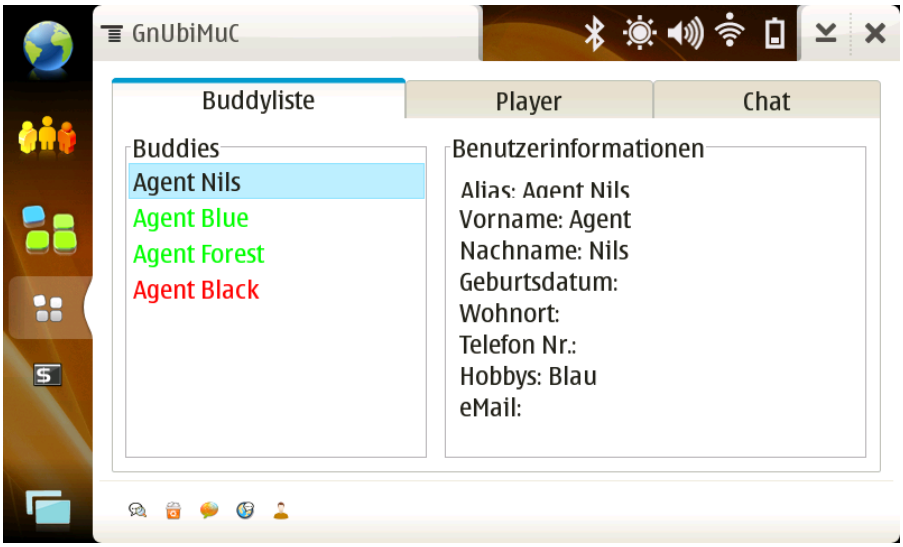

Abbildung 2.6: Ein Bild der finalen GUI

Der größte Teil des Bildschirms wird von dem jeweils aktiven Programmteil - Buddyliste, Chat oder Videokonferenz - genutzt. Das Umschalten zwischen den einzelnen Programmteilen ist über einige beschriftete Reiter am oberen Bildschirmrand möglich. Eine Leiste am unteren Bildschirmrand bietet dem Nutzer - durch bebilderte Icons - die Möglichkeit, den gerade aktiven Programmteil zu steuern.

Weniger häufig benutzte Programmteile sind über einen Klick auf den Programmnamen erreichbar. Über das so aufrufbare Optionsmenü kann der Nutzer bestimmte Programmeinstellungen wie die für die Videokonferenz benutzten Ports modifizieren, ohne eine Datei von Hand editieren zu müssen.

Beim Design der GUI musste letztendlich nicht nur auf den gewünschten Benutzerkomfort, sondern auch noch auf die Hardwaregegebenheiten geachtet werden. Sämtliche Optionen und Schaltflächen durften weder zu klein - da sie sonst nicht mehr lesbar sind - noch zu groß - da sie sonst Platz wegnehmen, der anderweitig gebraucht wird - sein.

Zur Implementierung wurde das Hildon-Framework [8] benutzt, welches auf GTK+2.0 [9] aufsetzt. Durch den Einsatz von Hildon konnte das N810 spezifische „look-and-feel“ erreicht werden.

2.7 Projektplanung des zweiten Semesters

Die bisher in diesem Kapitel genannten Themen stellten die Basis für die weiterführende Entwicklung des Programms *UbiMuC* dar. Diese Themenbereiche wurden überwiegend im ersten Halbjahr der Projektgruppe herausgearbeitet. Nachdem entschieden wurde, sich auf reine Peer-to-Peer-Netze zu beschränken, wurde *GNUnet* als grundlegendes Framework für die Entwicklung unserer Software gewählt.

Da reine Peer-to-Peer-Netze serverlos arbeiten, wurden Ad-Hoc-Netze als bevorzugte Netzwerkinfrastruktur festgelegt. Deren Umsetzung wird von Avahi übernommen, welches bereits als Applikation in *GNUnet* integriert wurde. Als Bindeglied zwischen *GNUnet* und der grafischen Oberfläche wurde ein erstes Konzept der sogenannten WIR-Schicht entwickelt, welche die Steuerzentrale von *UbiMuC* darstellt.

Während sich die Arbeit in der ersten Hälfte der Projektgruppe eher auf eine Recherche in breit gefächerten Themen bezog, sollte im Folgenden die Arbeit an Kernthemen forciert werden. Um die weitere Planungs- und Implementierungsarbeit klarer zu strukturieren, wurde deshalb beschlossen, einen Projektplan (nachfolgend auch als Roadmap bezeichnet) für das zweite Halbjahr zu entwickeln.

Diese Roadmap dient vor allem der genauen Übersicht des Projektfortschritts, der Festlegung von Deadlines und der sinnvollen Untergliederung in Arbeitsgruppen. In mehreren Iterationen wurde diese Roadmap an unsere Designentscheidungen angepasst. Im Folgenden wird allerdings nur die endgültige Version dargestellt. Somit gibt dieses sehr genau den Ablauf des 2. Semesters wieder.

Die in der Abbildung 2.7 auf den oberen Ebenen dargestellten Themenkomplexe stellen die zuerst angegangen Aufgabenbereiche dar. Die umrahmten Rechtecke stellen die Aspekte dar,

die sehr eng miteinander verknüpft sind. In hellgrau markierte Kästen umfassen grob die geforderten Minimalziele des Projekts. Diese Herangehensweise hat letztlich zu einer Aufteilung in vier übergeordnete Themenbereiche geführt, die von verschiedenen Kleingruppen bearbeitet wurden. Bei diesen vier Bereichen handelt es sich um:

- Transportschicht
- Nutzerverwaltung
- Chat
- Multimedia

Alle Themen auf tieferen Ebenen wurden folglich angegangen, sobald die notwendigen Vorarbeiten geleistet wurden. Im Weiteren soll kurz beschrieben werden wie die einzelnen „Knoten“ der Roadmap zu den großen Themengebieten zugeordnet werden.

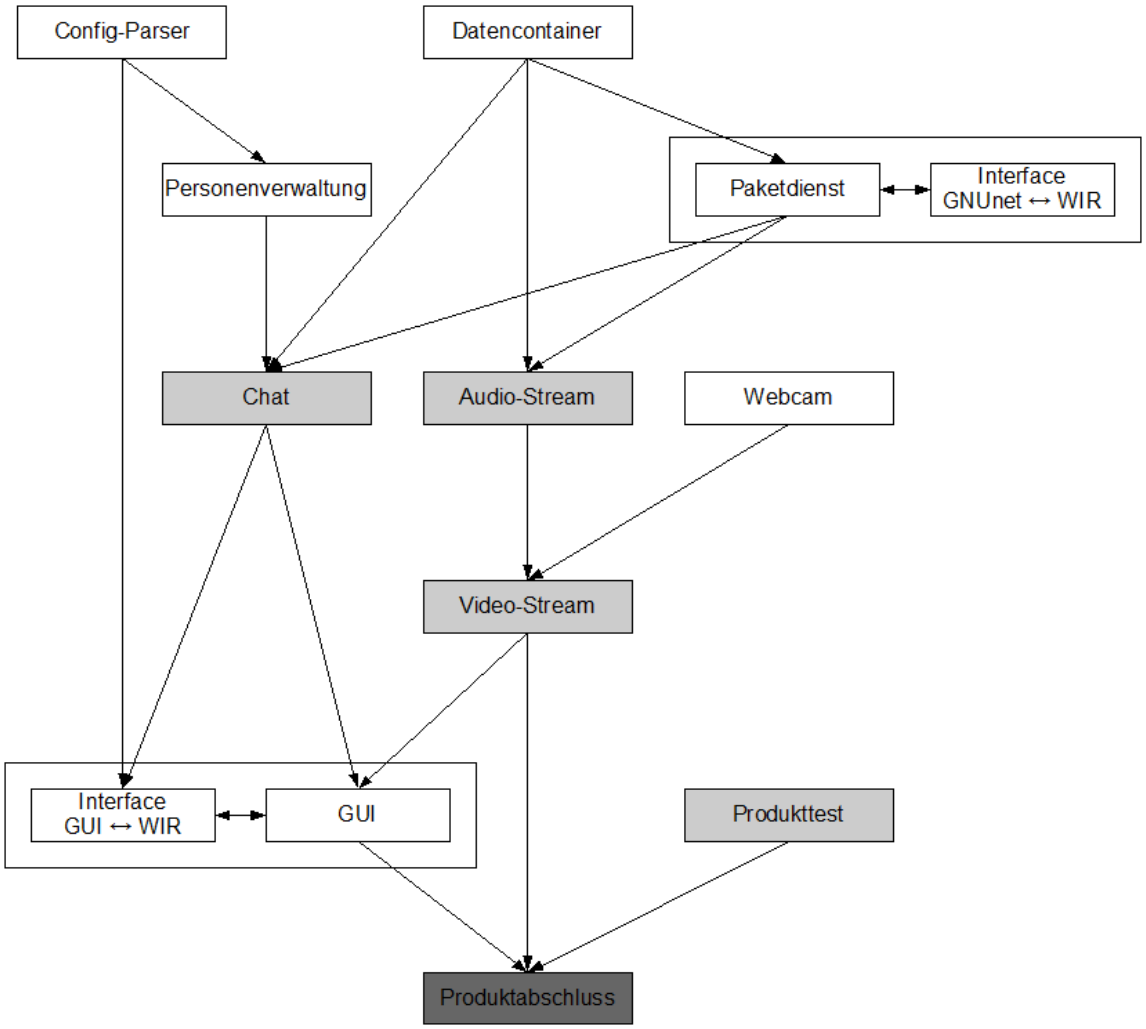

Abbildung 2.7: Die endgültige Roadmap

Zu den Aufgabenbereichen der Transportschicht-Gruppe (siehe Abschnitt 4.2) zählen Datencontainer, Paketdienst, die Interface-Struktur zwischen WIR-Schicht und *GNUnet* sowie das Interface zwischen der grafischen Oberfläche und der WIR-Schicht. Die Nutzerverwaltung (siehe Abschnitt 4.3) wurde von der gleichnamigen Gruppe übernommen. Zusätzlich wurde von dieser der Config-Parser implementiert. Nach Vollendung des Paketdienstes wurde ein Teil des Teams mit der Entwicklung des Chats (siehe Abschnitt 4.4) beauftragt. Die Multimedia-Gruppe (siehe Abschnitt 4.5) war für die Umsetzung der Audio- und Video-Streams, sowie die Integration und Ansteuerung der Webcam und des Mikrofons zuständig. Während des Entwicklungsprozesses wurde die grafische Oberfläche von den jeweiligen Gruppen um deren Bedürfnisse erweitert. Die erfolgreiche Entwicklung dieser Teilaspekte führt zur Fertigstellung des Projekts *UbiMuC*.

3 Transport und Kommunikation

Nachdem die Grundlagen von *GNUnet* bereits im Abschnitt 2.2 erläutert wurden, ist das Kapitel der Transport- und Kommunikationsschicht ein tieferer Blick auf *GNUnet* und die Kopplung an *UbiMuC*. Im Gegensatz zum vorher gewährten allgemeinen Einblick auf das Framework, werden in diesem Kapitel unter Anderem die in *UbiMuC* weiterverwendeten Konzepte und Module beschrieben, ebenso wie die genutzten Funktionen von *GNUnet*. Zu den zentralen Konzepten gehören Sachverhalte, die beim Verständis von *GNUnet* weitergeholfen haben und eine Basis für weiterführende Arbeiten bilden. Hierzu gehört unter anderem die Nutzerverwaltung von *UbiMuC* mittels der *GNUnet*-DHT, die Sende- und Empfangsmechanik von *GNUnet* über Handlen und Sendefunktionen sowie der Aufbau von sogenannten CRON-Jobs. Weiterhin wird in den *GNUnet*-Services die Anbindung von *GNUnet*-Applikationen beschrieben. Dabei wird verstärkt auf die Module der Hostlisten und der DHT eingegangen, da diese die für die Entwicklung von *UbiMuC* hauptsächlich verwendeten Applikationen darstellen. Im Anschluss daran wird erläutert, wie die Verknüpfung zwischen *UbiMuC* und *GNUnet* umgesetzt wird.

3.1 *GNUnet*-Konzepte

Basierend auf der in Abschnitt 2.2 geschilderten Struktur von *GNUnet* haben die Entwickler des Frameworks einige zentrale Module entwickelt, die die Gesamtfunktionalität von *GNUnet* realisieren. Für *UbiMuC* ist jedoch die Nutzung aller Module überhaupt nicht notwendig, so dass sich dieser Abschnitt dabei vor allen Dingen mit den von *UbiMuC* verwendeten oder angepassten Modulen von *GNUnet* befasst. Für *UbiMuC* steht im Gesamtbild die Anpassung der von *GNUnet* bereitgestellten und tatsächlich benötigten Funktionalitäten und deren Integration und Anpassung an die *UbiMuC*-Anforderungen im Vordergrund.

Benutzer-Anmeldung

Die erstmalige Anmeldung an das *GNUnet*-Netzwerk wird hauptsächlich durch das Hostlisten-Modul geleistet, wie in Abschnitt 2.2 beschrieben wurde. Der Austausch der HELLO-Pakete wird in *UbiMuC* durch die Nutzung einer angepassten DHT-Struktur realisiert. Baut ein Peer eine Verbindung in das *GNUnet*-Netzwerk auf, sendet er gleichzeitig sein HELLO-Paket an die modifizierte DHT und signalisiert damit seinen Onlinestatus. Empfängt ein Peer diese Daten aus der DHT, kann er für ein Zeitfenster von einigen Minuten (sofern kein neues HELLO eintrifft) annehmen, dass der besagte Peer „online“ und kommunikationsfähig ist. Da HELLO-Pakete einen Zeitstempel besitzen, verfallen veraltete Pakete und können nicht kontinuierlich benutzt werden. Es ist also ein stetiges Auffrischen der HELLO-Pakete zur Aufrechterhaltung von Kommunikationsmöglichkeiten nötig.

Datenaustausch zwischen Benutzern von *UbiMuC*

Damit einzelne Peers gezielt Daten miteinander austauschen können, setzt *UbiMuC* auf den *GNUnet*-internen Strukturen zum Versenden von Daten auf. Allerdings wird dabei nicht das Filesharing-Modul benutzt, sondern werden vielmehr die unmittelbar über der Netzwerkebene von *GNUnet* arbeitenden Methoden direkt mit *UbiMuC*-spezifischen Nutzdaten aufgerufen. Besonders zentral ist dabei die `cyphertext_send`-Funktion, welche das gezielte Senden von beliebigen Datenstrukturen an andere Peers ermöglicht. Dafür wird lediglich eine gültige *GNUnet*-Adresse als Ziel benötigt sowie ein *GNUnet*-spezifischer Header vor den eigentlich zu sendenden Nutzdaten. Mithilfe des für den Sendevorgang notwendigen HELLO-Pakets des Zielpeers findet eine Schlüsselabstimmung statt, so dass der Datenaustausch gesichert stattfinden kann. Auf Empfängerseite ist es notwendig, dass geeignete Handler-Funktionen für die Behandlung der eingehenden Pakete eingebunden werden, was im folgenden Unterkapitel separat behandelt wird.

3.1.1 *GNUnet*-Handler

Alle per *GNUnet* empfangenen Pakete werden zunächst in einer Queue zwischengespeichert. Mittels der Funktion „GNUNET_CORE_p2p_inject_message“ wird jede eingegangene Nachricht innerhalb von *GNUnet* durch einen speziellen Thread untersucht.

Jede Nachricht enthält im Header unter anderem eine wohldefinierte *GNUnet*-Typnummer. Falls diese nicht bekannt sein sollte, werden die ankommenden Pakete verworfen. Je nach Typnummer werden die entsprechenden Handler-Funktionen zur Behandlung des Nachrichteninhalts aufgerufen. Dabei ist es möglich, eine Vielzahl von Nachrichtentypen zu definieren, ebenso wie mehrere Handler-Funktionen pro Nachrichtentyp registriert werden können. Dies wird über ein mehrdimensionales Array realisiert, welches eine Spalte pro Nachrichtentyp besitzt, in der sämtliche Handler-Funktionen abgelegt werden können.

Schlägt die Behandlung durch eine Handler-Funktion fehl, wird automatisch die nächste, sofern vorhanden, aufgerufen. Die Nachrichtentypen werden in „gnunet_protocols.h“ definiert. Hierbei muss darauf geachtet werden, dass die Nummerierung eindeutig ist. Handler werden mit „GNUNET_CORE_p2p_register_handler“ registriert. Diese Funktion ergänzt hauptsächlich das Array um den selbst geschriebenen Handler, der mit den *GNUnet*-Konventionen konform sein muss.

3.1.2 CRON

Innerhalb des *GNUnet*-Frameworks existieren eine Reihe von Aufgaben, die in bestimmten Zeitabschnitten wiederkehrend ausgeführt werden müssen. Sei es das Versenden von HELLO-Nachrichten, oder im Rahmen der Initialisierung der verteilten Hashtabelle. Für solche Aufgaben bietet *GNUnet* einen Cron-Manager an, der einzelne Methoden nach bestimmten Zeitschemata ausführen kann. Dieser Cron-Manager ist an das Cron-System angelehnt, welches unter unixoiden Systemen Programme automatisiert ausführen kann, arbeitet aber auf Methodenebene. Mit Hilfe dieses Cron-Managers wird eine threadbasierte Nebenläufigkeit ermöglicht. Nützlich ist diese Funktion in unserem Falle insbesondere für Timeouts, da der Kontrollfluss so,

während die „Wartezeit“ abläuft, weiter fortschreitet. Der Cron-Manager ermöglicht sowohl die einmalige Ausführung einer Methode nach einer bestimmten Zeit, als auch die wiederkehrende Ausführung einer Methode in gewissen Zeitabständen. Die auszuführenden Funktionen dürfen keinen Rückgabewert haben. Weiterhin müssen die Funktionen einen Zeiger als Parameter übergeben bekommen.

3.1.3 *GNUnet*-Applikation / -Services

GNUnet ist ein Peer-to-Peer Framework, welches verschiedene Services und Applikationen unterstützt. Es basiert auf mehreren Schichten, wie in Abbildung 3.1 zu sehen ist, wobei die Hauptschicht *gnunetd* als Server ausgeführt wird. Diese kann mittels TCP eine Verbindung mit seinen Clients aufbauen. Weiterhin enthält *gnunetd* eine *GNUnet*-Core-Schicht, in der sich die Services befinden. Diese Services können von den Clients, die den Applikationen entsprechen über eine Applikation-Service-API angefragt werden.

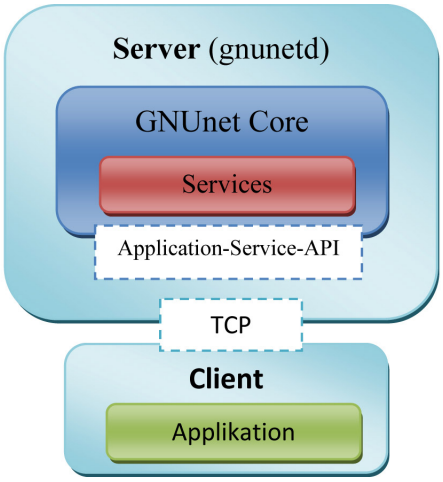

Abbildung 3.1: Schichtenmodell *GNUnet*

Die Services und Applikationen die beim *UbiMuC* benutzt werden, sind in der Tabelle 3.1 zusammengefasst. Im Folgenden werden diese kurz vorgestellt und erläutert.

Services	Applikationen
Identity	Advertising
Transport	Getoptions
Stats	Traffic
PingPong	Datastore
Topology	Avahi
Fragmentation	DHT
State	Hostliste
Boostrap	

Tabelle 3.1: Services und Applikationen

Services

Ein Service ist ein interner Dienst von *GNUnet*. Er ist nicht direkt sichtbar für einen Benutzer, kann aber von einer oder mehreren Applikationen genutzt werden. Weiterhin wird ein Service nur dann geladen, wenn er von einer Applikation gebraucht wird.

Der erste Service, Identity ist für die Verwaltung einer Liste zuständig, die aus bekannten Peers besteht. Weiterhin enthält er Informationen über die Peers, die sich auf einer Blacklist befinden, und über die HELLO-Nachrichten. Für das Versenden von Paketen steht in *GNUnet* eine Transportschicht zur Verfügung. Der Zugriff auf diese Schicht wird durch den nächsten Service, Transport ermöglicht. Stats kann verwendet werden, um bestimmte statistische Informationen, wie die Anzahl von empfangenen Bytes, gesendeten Nachrichten und gespeicherten Datenmengen zu speichern. Mittels des Werkzeug `gnunet-stats` hat man dann die Möglichkeit, die gespeicherten Informationen auszulesen.

Der Service PingPong pingt einen Host an und löst eine Aktion aus, sobald eine Antwortnachricht empfangen wurde. Topology ist ein Service, der für den Aufbau einer Mesh-Topologie zustängig ist. Hierbei handelt es sich um ein Netz, in dem jeder Netzwerkknoten mit einem oder mehreren verschiedenen Knoten verbunden ist. Fragmentation erlaubt das Senden und Empfangen von Nachrichten, die größer sind als die „Maximum Transmission Unit“. Dabei werden alle Nachrichten vor dem Versand auf die maximale Größe von 65535 Bytes zerlegt. Beim nächsten Service State handelt es sich um eine kleine Datenbank, welche den internen Zustand von *GNUnet* protokolliert. Der letzte Service Boostrap ist für den Download von HELLO-Nachrichten mittels HTTP zuständig.

Applikationen

Eine Applikation in *GNUnet* ist ein Dienst, die eine Funktionalität dem Benutzer bereitstellt. Dazu baut sie zuerst eine Verbindung zum gnunetd auf und verbindet sich danach mit der *GNUnet*-Core. Weiterhin enthält sie auch eine Eingabemöglichkeit damit der Benutzer auf ihre Funktionalität zugreifen kann. Diese Funktionalität wird dann durch die Services unterstützt, die aus der GNUnet-Core angefordert werden können. Die Abhängigkeiten zwischen den Applikationen und den Services sind in der Abbildung 3.2 zu sehen.

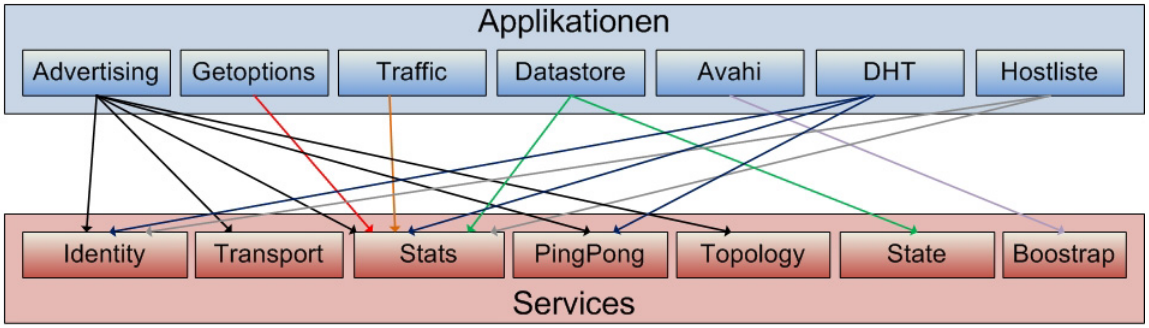

Abbildung 3.2: Abhängigkeiten zwischen Applikationen und Services

Bei der ersten Applikation Advertising handelt es sich um einen Cron-Job. Dieser stellt durch den Austausch von HELLO-Nachrichten sicher, dass sich die Knoten gegenseitig kennen und somit ein Netzwerk bilden. Durch die zweite Applikation Getoptions bekommt der Client die Möglichkeit, den Wert einer *GNUnet*-Option erfragen zu können. Die nächste Applikation Traffic verwaltet die Menge des entstehenden P2P-Traffics in einem lokalen *GNUnet* Knoten. Bei Datastore handelt es sich um die Verwaltung von Inhalten. Dabei muss entschieden werden, welcher Inhalt und wie lange zur verfügung stehen soll.

Die Applikation *Avahi* realisiert die von uns genutze Ad-Hoc-Netzwerkstruktur. Da wir keine zentrale Verwaltungsinstanz haben, wird eine verteilte Datenstruktur benötigt. Diese erhalten wir durch das DHT-Modul. Um eine Kommunikation zwischen den Clients aufzubauen, benutzen wir die Hostliste, die Adressen von den *GNUnet*-Clients enthält.

Nähere Erläuterungen zu den Funktionsweisen des DHT- und des Hostlisten-Moduls werden in den nächsten Abschnitten gegeben.

3.1.3.1 DHT-Modul

GNUnet bietet über das DHT-Modul eine verteilte Datenstruktur an, welche ihren Inhalt mit gewisser Redundanz auf alle Clients, die dieses Modul aktiviert haben, verteilt. DHT steht dabei für „Distributed Hash Table“, ins Deutsche übersetzt „verteilte Hashtabelle“. Ursprünglich diente dieses Konzept als eine Möglichkeit, Informationen über von Peers zur Verfügung gestellten Inhalt im Netzwerk zu verbreiten, ohne eine zentrale Daten- und Infrastruktur zu benötigen. Grundsätzlich baut die in *GNUnet* verwendete DHT-Implementierung, wie sie ursprünglich in einem Paper beschrieben wurde [10], auf Kademlia [11] auf.

Bei einer verteilten Hashtabelle werden die zu speichernden Werte über die verschiedenen Knoten im Netzwerk verteilt. Beim Suchen nach Einträgen muss eine Anfrage an das Netzwerk verschickt werden, so dass im Anschluss Antworten von verschiedenen Knoten eingehen. Eine Besonderheit der DHT ist dabei, dass nicht deterministisch vorhergesagt werden kann, welche Antworten eingehen werden und von wo diese kommen werden. Dies hat zur Folge, dass durchaus nicht immer alle Werte, die in der Datenstruktur im gesamten Netz vorhanden sind, auch als Antwort auf die Anfrage geliefert werden. In *UbiMuC* wird die hier beschriebene verteilte Datenstruktur genutzt, um die in Abschnitt 4.3 näher erläuterte Nutzerverwaltung zu realisieren, ohne dafür einen zentralen Server zu benötigen.

Struktur der *GNUnet*-DHT

Jeder *GNUnet*-Client mit aktiviertem DHT-Modul sendet regelmäßig „P2P_DHT_Discovery“-Nachrichten an das Netz, um so mit anderen Knoten der Hashtabelle bekannt gemacht zu werden. An diese bekannten anderen Knoten werden dann Anfragen weitergeleitet. Die Anzahl der Weiterleitungen ist zur Kompilierzeit festgelegt, genau wie die Redundanz der Einträge, welche bestimmt, wie oft ein Wert im gesamten Netz gespeichert werden soll. Dies gilt sowohl für Anfragen für das Ablegen von Daten, sogenannte „DHT Put Messages“, als auch für Anforderungen von Daten, den „DHT Get Messages“.

Generell besteht jeder DHT-Eintrag aus einem Daten-Tripel. Zum einen verfügt jeder Eintrag über eine „Lebenszeit“, die festlegt, wie lange diese Daten bei GET-Nachrichten noch gefunden werden können. Dieser Wert ist eine zur Kompilierzeit festgelegte Konstante und standardmäßig auf zwölf Stunden voreingestellt. Pakete verfallen nach dieser Zeit und werden bei GET-Anfragen nicht mehr geliefert, wenn die Lebenszeit abgelaufen ist. Der zweite Wert, über den DHT-Einträge verfügen, ist ihre „Kategorie“. Anfragen an die DHT fordern immer alle Pakete einer Kategorie an. Es ist nicht möglich, explizit ein einziges Paket zu erhalten. Der dritte Block ist für die „Nutzdaten“ des Pakets. Diese können nahezu beliebige Form haben.

Speichern von Werten in der DHT

Das Speichern von Daten geschieht bei Nutzung der DHT entweder in einer MySQL oder einer SQLite basierten Datenbank. Für die DHT wird in der Konfigurationsdatei vom *GNUnet*-Dienst eine Maximalgröße an verfügbarem Speicherplatz festgelegt, welche auf 64KB voreingestellt ist.

Beim Eingehen einer PUT-Nachricht wird anhand der zu nutzenden Redundanz und der Nähe der von Kademlia vorgegebenen XOR Metrik [11] bestimmt, ob der Wert auch lokal gespeichert werden soll, oder ob er nur weitergeleitet wird. Hat der XOR-Algorithmus sich dazu entschieden, einen Wert von einem PUT-Request in der eigenen Datenbank zu speichern, so wird überprüft, ob maximal 90% des verfügbaren Speicherplatzes genutzt wird. Wird mehr Platz genutzt, so werden jene Pakete aus der DHT entfernt, deren Lebenszeit abgelaufen ist. Anschließend wird der Wert in der gewählten Datenbank mit der zugehörigen Lebenszeit gespeichert.

Es gibt keine explizite Operation zum Löschen von veralteten Einträgen. Dies wird nur durchgeführt, wenn der maximal für die DHT zu nutzende Speicherplatz nahezu komplett verwendet wird. Geht eine erneute PUT-Anfrage für ein Paket ein, dessen Kategorie und Nutzdaten identisch zu einem bereits gespeicherten Paket sind, wird nur dessen Lebenszeit aktualisiert, da das Paket ja bereits vorhanden ist.

Auslesen von DHT Werten

Das Auslesen von Daten geschieht durch das Versenden von „DHT Get Messages“. Diese werden vom *GNUnet*-Dienst an eine festgelegte Anzahl anderer Nutzer weitergeleitet, welche selbst wiederum bis zu einer festgelegten Netztiefe weiterleiten. Es werden dabei immer alle Einträge einer bestimmten Kategorie von der Datenstruktur erfragt. Jene Knoten, die eine Anfrage für auf ihnen verfügbare Pakete empfangen, antworten mit einer „DHT Result Message“.

Mögliche Probleme der DHT und *UbiMuC* spezifische Anpassungen

Generell ist die DHT keine verlässliche Datenstruktur. Es ist nicht sichergestellt, dass die GET-Anforderungen auch an all jene Nutzer weitergeleitet werden, die die entsprechenden Daten haben. Dieses Problem wird dadurch verringert, dass Anfragen öfter weitergeleitet werden, als dies mit den Standardeinstellungen der Fall ist. Des Weiteren kann es helfen, die Redundanz von DHT-Einträgen im Netz zu erhöhen.

Da die DHT im Falle von *UbiMuC* nur für sehr kleine Einträge genutzt wird, haben wir uns dafür entschieden, eingehende PUT-Nachrichten immer zu speichern. Dies erlaubt bei den vorhandenen 64 KB Speicherplatz in der DHT fast 1000 *UbiMuC*-Nutzer, was eine sehr große Netzgröße darstellt, die so erst einmal nicht zu erwarten ist und entsprechend nicht zu Problemen führen wird. Außerdem sollten die PUT Nachrichten öfter weitergeleitet werden, so dass eine bessere Verteilung über das Netzwerk gewährleistet ist. Hierfür wurde die entsprechende Konstante im *GNUnet*-DHT-Modul angepasst.

Ein weiteres Problem ist, dass DHT-Nachrichten normalerweise mit einer recht geringen Priorität vom *GNUnet*-Dienst behandelt werden. Dies führt dazu, dass bei hoher System- oder Netzwerklast DHT-Nachrichten bevorzugt verworfen werden. Entsprechend musste von unserer Projektgruppe die Priorität der DHT-Nachrichten derart erhöht werden, dass sie von *GNUnet* nie verworfen werden.

Auch mit diesen Änderungen ist die DHT keine sichere Datenstruktur, da Nachrichten beim Transport noch verloren gehen können. Die Verlässlichkeit ist aber durch diese Änderungen generell verbessert, so dass die DHT für die in Abschnitt 4.3 beschriebene Kontaktliste genutzt werden kann.

3.1.3.2 Hostlisten-Modul

Die *GNUnet*-interne Bekanntmachung von Peers verläuft über HELLO-Nachrichten. Beim initialen Betritt zum *GNUnet*-Netzwerk werden Hostlisten ausgetauscht, die aus derartigen HELLO-Nachrichten bestehen. In unserem Fall wird das Finden der Peers über den Abruf von Dienst-Informationen über *Avahi* (siehe auch Abschnitt 2.4) durchgeführt. Die Verbindung auf *GNUnet*-Ebene wird unmittelbar im Anschluss über den Download von Hostlisten eingerichtet.

Das Hostlisten-Modul stellt einen kompakten HTTP-Server zur Verfügung, über den dieser Austausch erfolgen kann. Der genutzte HTTP-Server stammt aus der *libmicrohttpd*-Bibliothek, die ebenfalls dem *GNUnet*-Projekt entstammt, aber auch als eigenständige Bibliothek genutzt werden kann.

Eine eingehende HTTP-Anfrage nach einer Hostliste wird vom HTTP-Server an eine Callback-Funktion übergeben. Diese erstellt dann aus den vorhandenen HELLO-Nachrichten eine Liste, die dann an den HTTP-Server zurückgegeben wird. Der Server überträgt dann die Hostliste an den Anfragenden.

3.2 Kommunikation zwischen *UbiMuC* und *GNUnet*-Daemon

Wie in Abschnitt 2.2 bereits erwähnt wurde, ist das *GNUnet*-Framework als Client/Server-Architektur organisiert. Demnach gibt es einen eigenständigen Serverprozess mit dem Namen *gnunetd*, der vollkommen unabhängig von den Clientprozessen und damit auch von *UbiMuC* läuft. Der getrennte Adressraum von Prozessen erfordert es daher, einen geordneten Nachrichtenaustausch zwischen *UbiMuC* und *gnunetd* zu implementieren. Sämtliche dafür notwendigen Funktionen stellt *GNUnet* bereits in einer Bibliothek zur Verfügung, die nur noch hinzugelinkt werden muss. Damit kann man ähnlich zu *UDP-Sockets* Daten paketorientiert austauschen.

Jedes übertragene Datenpaket beginnt mit einem kurzen Header, der jeweils 16 Bit für Nutzdatenlänge und -typ enthält. Anhand des Datentyps wird entschieden, welche Handlerfunktion aufgerufen werden soll. Unsere Aufgabe war es also, geeignete Handler sowohl innerhalb von *gnunetd* als auch *UbiMuC* zu schreiben. Dazu haben wir bis dahin noch unbenutzte Typnummern definiert, die beim Initialisieren unserer *GNUnet*-Plugins zusammen mit den entsprechenden Handlerfunktionen bei *gnunetd* registriert werden.

Beim Eintreffen eines Paketes mit passender Typnummer wird von nun an die Handlerfunktion aufgerufen, die das vollständige Paket inkl. Header als Parameter übergeben bekommt. Auf der Seite der *WIR*-Schicht wurden eigenständige Threads angelegt, die immer dann aufgeweckt werden, wenn Pakete vom *GNUnet-Daemon* eintreffen. Anschließend findet auch dort eine Überprüfung der Typnummer statt. Dabei wird auch entschieden, wie das Paket weiter zu behandeln ist. Auf diesen Vorgang wird in Abschnitt 4.2.2.2 genauer eingegangen.

Mit diesem Vorgehen können allerdings nur reine Bytearrays verarbeiten werden. Um auch komplexere Datenstrukturen handhaben zu können, war es notwendig, *Marshalling*-Methoden (auch *Serialisierer* genannt) zu entwerfen, die ganze Objekte in einen Bytestrom umwandeln können, so dass sie sich später wieder vollständig rekonstruiert lassen. Dabei gilt es auch zu beachten, dass C++-Objekte der *WIR-Schicht*, die auch im *GNUnet*-Plugin genutzt werden, eine passende C-Repräsentation haben. Allgemein betrachtet, werden beim *Marshalling* sämtliche Objekt-Attribute der Reihe nach in ein Bytearray geschrieben, wobei man ggf. *Litte-Endian-Big-Endian*-Konvertierungen durchführen muss, um die Portabilität zu wahren.

4 Design

Im Anschluss an die mit *GNUnet* gelegten Grundlagen der Transportschicht orientiert sich das Kapitel Design an der von der Projektgruppe geleisteten Arbeit.

Das Ziel von *UbiMuC* ist es, einen Prototypen einer Peer-to-Peer-Kommunikationsplattform zu implementieren. Mit *GNUnet* ist zwar eine Basis für den Austausch von Datenpaketen im Netz geschaffen, allerdings müssen darauf aufbauend noch einige zusätzliche Funktionalitäten erarbeitet werden. Dazu gehören unter anderem der in *GNUnet* bereits geplante, jedoch noch nicht verwirklichte Chat, der Austausch von Bild- und Tonmaterial sowie verschiedene Erweiterungen auf niedrigeren Programmschichten.

Hierbei richtet sich die Gliederung nach der Projektplanung (siehe Abschnitt 2.7), welche zuvor ausgearbeitet wurde. Daraus resultieren fünf Unterkapitel, die Bottom-up, wie in der Struktur von *UbiMuC* bereits beschrieben, vom Transport der Datenpakete bis zur eigentlichen Anwendung angeordnet sind.

Zunächst wird ein Überblick über den selbstentwickelten Datencontainer gegeben, gefolgt von der Transportschicht-Erweiterung, welche mit Hilfe des Datencontainers Pakete an andere Peers verschickt. Ein weiteres Konzept bildet die Nutzerverwaltung, die es erlaubt bekannte Netzteilnehmer in einer Freundesliste zu speichern und verwalten. Aufbauend darauf bilden der Chat und die Audio-Video-Konferenz die Möglichkeit, mit bekannten Teilnehmern in Kontakt zu treten.

Bei jedem einzelnen dieser Themen wird zunächst das Konzept erläutert und danach werden gegebenenfalls die zu Grunde liegenden Algorithmen genauer erklärt. Im Anschluss daran wird jeweils auf deren Implementierung eingegangen.

4.1 Datencontainer

Der modulare Aufbau von *UbiMuC* erfordert es, eine einheitliche Schnittstelle für Datenpakete zu entwickeln, die zwischen den Applikationsteilen ausgetauscht werden. Unter Beibehaltung dieser Schnittstelle kann man einzelne Module, wie beispielsweise den „Splitter“, der für das Vorsortieren von empfangenen Paketen zuständig ist, ohne großen Aufwand nach Belieben austauschen.

GNUnet enthält zwar bereits ein Paketformat, das für unsere Daten verwendet werden kann, allerdings beinhaltet dieses einige Nachteile:

- Da *GNUnet* für das eigene Datenformat „C-Strukturen“ einsetzt, die mit globalen Funktionen manipuliert werden, würde das objektorientierte Paradigma gebrochen.
- Das Verwenden von *GNUnet*-spezifischen Teilen innerhalb der oberen Schichten von *UbiMuC* würde zudem die Modularität gefährden, da die Transportschicht damit nicht mehr einfach ausgetauscht werden könnte.
- Im Nachrichtenheader fehlen einige Felder, die wir zwingend zur Umsetzung einiger Dienste benötigen.

Daher haben wir uns dafür entschieden, die Klasse „Datacontainer“ zu entwickeln, die für alle in *UbiMuC* anfallenden Datenübertragungen genutzt werden soll.

Format

Der „Datacontainer“ enthält alle notwendigen Informationen, um eine zuverlässige Rekonstruktion der Nutzdaten gewährleisten zu können. Wie in Abbildung 4.1 dargestellt ist, wurden jeweils 32 Bit lange Felder im Header eingefügt, um eine Sequenznummer zur Wiederherstellung der Paketreihenfolge, eine Verbindungsnummer zur Zuordnung von Datenströmen, eine Typ-Enumeration zur Klassifikation der enthaltenen Nutzdaten, sowie die Datengröße abzubilden.

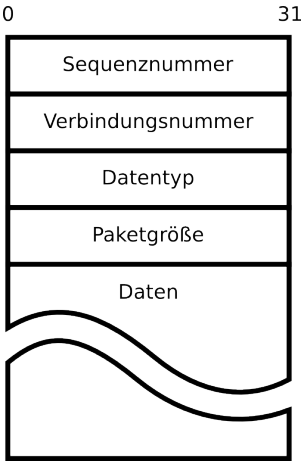

Abbildung 4.1: Datacontainer

Auffällig ist, dass im Gegensatz zum Header von anderen Netzwerkprotokollen wie TCP/IP keine Adressinformationen enthalten sind, da diese ausschließlich zum Versand von Paketen benötigt werden und „Datacontainer“ innerhalb von *UbiMuC* auch zum Zwischenspeichern von Daten (beispielsweise innerhalb des „Splitters“) genutzt werden, wo Adressdaten, die jeweils 64 Byte belegen, nur unnötig Speicherplatz belegen würde.

Bei den Headern wurde darauf geachtet, dass das *Marshalling* portabel bleibt, insbesondere bei Übertragungen zwischen Architekturen mit unterschiedlicher „Endianess“.

Besonderheiten

Um die öffentlichen Schnittstellen des Datacontainers besonders einfach zu halten, wurden einige Operatoren überladen. Mit den üblichen Vergleichsoperatoren kann man nun gegen die Sequenznummer vergleichen, um die Reihenfolge der Datenpakete zu ermitteln. Abschließend wurden noch sechs Konstruktoren eingefügt, die besonders häufig vorkommende Daten wie Textdaten effizient behandeln.

4.2 *UbiMuC* Transportschicht

Die eigentliche Transportschicht von *UbiMuC* besteht aus einem bei jedem Peer verfügbaren „Paketdienst“. Dieser sorgt dafür, dass Verbindungen zwischen den *UbiMuC*-Teilnehmern über das *GNUnet*-Netzwerk ohne großen Aufwand hergestellt und verwaltet werden können. Aufgrund der Beschränkungen durch die vorliegende *GNUnet*-Architektur und den allgemeinen Aufbau des Frameworks (siehe Abschnitt 3.1) ist die Bereitstellung einer allgemeinen Schnittstelle zum Datenversand und -empfang unausweichlich.

GNUnet verfügt über kein natives Multi-Hop-Routing (ausgenommen das Filesharing-Modul), daher können Daten zwischen Peers nur dann ausgetauscht werden, wenn diese über eine direkte Verbindung zueinander verfügen. Ist diese direkte Verbindung nicht vorhanden, müssen die Daten über andere Teilnehmer weitergeleitet werden.

Um jedoch diese Weiterleitung zu ermöglichen, muss eine Pfadliste zwischen Quell- und Zielpeer vorliegen, die für den Datentransfer benutzt werden kann. Diese Pfadliste wird mittels der in Abschnitt 4.2.1.1 erläuterten „GetRoute“- und „GetRouteReply“-Nachrichten aufgebaut. Mit Hilfe dieser Pfadliste kann anschließend eine Weiterleitung der Daten über jeden einzelnen Peer durch Verwendung von „UbiMuC-DataBlock“-Nachrichten erfolgen. Abbildung 4.2 verdeutlicht den geschilderten Vorgang.

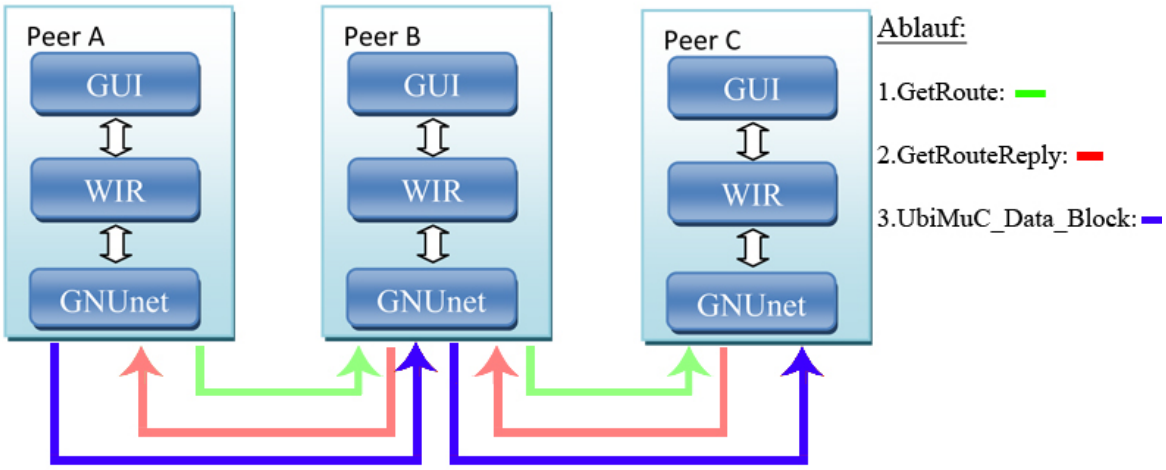

Abbildung 4.2: Struktur des Routings

Ein weiterer wichtiger Aspekt liegt außerdem in der Kapselung der *GNUnet*-internen Datenstrukturen, so dass diese nicht bis zu den Routinen für AV-Konferenz und Chat weitergeführt werden müssen. Zu diesem Zweck wurde ein eigener Datencontainer entwickelt, der sämtliche Nutzdaten kapselt und vom Paketdienst zur Kommunikation mit anderen Teilnehmern des Netzwerkes benutzt wird. Der Paketdienst wird dabei an dieser Stelle kurz erwähnt, wobei eine detaillierte Erläuterung der Routing-Konzepte und Vorgehensweisen in den nachfolgenden Abschnitten beschrieben wird.

Paketdienst

Da *GNUnet* nativ keine extern nutzbare Schnittstelle bereitstellt, um auf direktem Wege unkompliziert und ohne vorherige Anforderung Daten zwischen zwei Teilnehmern auszutauschen, war es nötig, einen solchen Datendienst für Nutzdaten der Multimedia- beziehungsweise Chat-Module selbst einzubauen. Die eigentliche Verbindung zwischen den *UbiMuC*-Teilnehmern wird über ein eigens dafür entworfenes Routing-Protokoll, „GetRoute“, mittels Erkundung des Netzwerkes aufgebaut und bei den jeweiligen Endpunkten in Form einer Pfadliste durch das Netzwerk für spätere Verwendungszwecke gespeichert und vorgehalten. Abschnitt 4.2.1.2 widmet sich im Detail dem Entwurfskonzept der Routingstruktur.

Die wesentliche Aufgabe des Paketdienstes liegt also in der Bereitstellung einer Sende- und Empfangsmöglichkeit von Nachrichten für *UbiMuC*-Module. Dabei hat der Paketdienst selbstständig für die Verarbeitung der Daten und den Routenaufbau zu sorgen, damit die darauf aufbauenden Module sich nicht mit besagten Details beschäftigen müssen und die benutzten Datenstrukturen gut gekapselt werden. Details und Struktur des Paketdienstes sind im Abschnitten 4.2.2 verzeichnet, ebenso wie die Routing-Konzepte und Vorgehensweisen in den Abschnitten 4.2.1.2 und 4.2.1.3 erläutert sind.

4.2.1 Konzepte und Spezifikationen

Wie schon in vorherigen Kapiteln erwähnt, basiert unser Projekt auf dem *GNUnet*-Framework. Dieses Framework gibt uns jedoch nicht die Möglichkeit, selbständig Pakete ins Netzwerk zu senden, da es nach dem PULL-Prinzip arbeitet. Um jedoch unsere Ziele zu erreichen, ist ein Mechanismus notwendig, der uns die Möglichkeit gibt, einzelne Peers im Netzwerk zu adressieren. Für die Lösung dieses Problems wurde ein Konzept entwickelt, welches in diesem Abschnitt vorgestellt wird.

Im Folgenden ein einfaches Beispiel-Szenario: Ein Peer A möchte eine Nachricht an Peer B schicken. Welche Informationen werden benötigt, um dieses Problem lösen zu können? Zunächst wird ein eigenes Routing-Protokoll in *GNUnet* gebraucht. Dieses verwendet zwei Nachrichtenformate „GetRoute“ und „GetRouteReply“, die bei der Aufbau der Route zwischen Peer A und Peer B benötigt werden. Sobald die Verbindung aufgebaut ist, können über den Paketdienst die eigentlichen Nutzdaten versendet werden. Die Nutzdaten werden in der vordefinierten Datenstruktur „UbiMuC-DataBlock“ anschließend von Peer A zu Peer B transportiert. In den folgenden Kapiteln wird näher auf den Aufbau und die Funktionsweise der Nachrichtenformate „GetRoute“ und „GetRouteReply“ (siehe Abschnitt 4.2.1.2), sowie auf *UbiMuC*-DataBlock eingegangen.

4.2.1.1 Nachrichtenformate

Im Folgenden werden die wichtigsten Nachrichtenformate der Transportschicht zuerst durch ihre Funktionsweise und danach durch ihren Aufbau erläutert. Dabei handelt es sich um die Formate „GetRoute“ und „GetRouteReply“ für den Aufbau einer Route. Und das Format „UbiMuC-DataBlock“ für den anschließenden Transport von Nutzdaten. Für jedes Nachrichtenformat wurde innerhalb von *GNUnet* ein *GNUnet*-Pakettyp definiert und mit einer zugehörigen Handler-Funktion versehen.

Die „GetRoute“-Nachricht wird vom Sender genau dann gesendet, wenn dieser eine Verbindung zu einem anderen Peer herstellen möchte und keine Route zu ihm kennt. Per Broadcast wird die „GetRoute“-Nachricht an alle Teilnehmer des *UbiMuC*-Netzes gesendet, wobei alle auf dem Weg liegenden Peers in einer Liste innerhalb des „GetRoute“-Paketes gespeichert werden.

Sobald das „GetRoute“-Paket beim Empfänger angekommen ist, wandelt der Empfänger das empfangene „GetRoute“-Paket in ein „GetRouteReply“-Paket um und schickt es entlang der gespeicherten Route an den ursprünglichen Sender zurück. Somit bestätigt der Empfänger den Verbindungsaufbau zum Sender.

Abbildung 4.3: Aufbau von GetRoute und GetRouteReply

Im Folgenden werden die beiden Nachrichtenformate „GetRoute“ und „GetRouteReply“ vorgestellt. Der Aufbau ist für beide gleich und wird in Abbildung 4.3 dargestellt. Das Format besteht immer aus zwei Teilen - dem Header und der Nutzlast (Payload).

Das erste Feld („Länge“) gibt die Gesamtlänge der Nachricht an. Im nachfolgendem Feld wird der Typ der Nachricht spezifiziert. Innerhalb dieses Feldes wird zwischen einer „GetRoute“- und „GetRouteReply“-Nachricht unterschieden. In den nächsten beiden Feldern befinden sich die jeweiligen Adressen der Peers, die miteinander kommunizieren möchten. Das Feld Hop-Anzahl gibt die Anzahl an Hops an, die zwischen dem Sender und Empfänger liegen. Das letzte Feld, welches dem Payload entspricht, speichert die Adressen der Hops, die auf der Route vom Sender zum Empfänger liegen.

Der „UbiMuC-DataBlock“ ähnelt vom Aufbau her den beiden zuvor beschriebenen Nachrichtenformaten. Der wesentliche Unterschied dabei ist, dass im „UbiMuC-DataBlock“ auch die eigentlichen Nutzdaten gesendet werden. Weiterhin beinhaltet der „UbiMuC-DataBlock“ auch den zuvor definierten Datacontainer.

Diese beiden Nachrichtenformate befinden sich jedoch auf unterschiedlichen Ebenen, der Datacontainer auf der WIR-Schicht und der „UbiMuC-DataBlock“ auf der GNUnet-Ebene. Der Datacontainer ist dafür da, um aus komplexen Datenstrukturen einfache Bytearrays zu erzeugen, die sich dann leicht in *GNUnet*-Datenblöcke verpacken lassen.

Der „UbiMuC-DataBlock“ beinhaltet den Datacontainer, die Absender- und Empfängerdaten und alle Informationen, die zum routing benötigt werden. Als Typ der Nachricht kann hier „Undefiniert“, „Kontrolldaten“, „Datei“, „Text“ oder „Multimedia“ definiert werden. Am Ende des Headers befindet sich das Feld Datenlänge, in dem die Länge der nachfolgenden Nutzdaten angegeben wird. Der Payload-Bereich besteht aus den Nutzdaten, die an den Empfänger adressiert sind. Der „UbiMuC-DataBlock“ ist in der Abbildung 4.4 dargestellt.

Abbildung 4.4: Aufbau des UbiMuC-DataBlock

4.2.1.2 Routingkonzepte

UbiMuC verwendet zum Aufbau und zur Verwaltung von Pfaden zwischen den einzelnen Peers des *GNUnet*-Netzwerks ein „Source-Routing“. *GNUnet* selbst bietet keine Möglichkeit, gezielt Daten in das Netzwerk zu senden, da es nach dem *Query-Response-Konzept* aufgebaut ist. Man hat als Teilnehmer also nur die Option, Daten anzufragen und bei Vorhandensein der Daten eine Antwort vom Anbieter zu erhalten. Anderenfalls ist *GNUnet* nativ nicht in der Lage, einen Pfad durch das Netzwerk bereitzustellen, so dass kein selbständiger Aufbau einer gezielten Verbindung oder ein gerichteter Datenaustausch möglich ist. Zur Behebung dieses Problems musste eine eigene Routing-Struktur erschaffen werden, die sowohl den gerichteten Aufbau von Datenkanälen über das *GNUnet*-Netzwerk ermöglicht, als auch das gezielte Versenden von eigenen Daten an andere Peers (Paketdienst) anbietet. Um Nutzdaten zwischen zwei Teilnehmern auszutauschen, müssen beide über eine Pfadliste verfügen, welche die Schritt-für-Schritt-Weiterleitung von Daten ermöglicht.

Routing-Nachrichtentypen

Um den Pfad-Aufbau durch ein eigenes Routing-Protokoll in *GNUnet* zu integrieren, musste der *GNUnet*-Kern um zwei weitere Nachrichtentypen, „GetRoute“ und „GetRouteReply“, ergänzt werden. Diese beiden neuen Nachrichtentypen erhalten eigene Callback-Funktionen, so dass die selbstentwickelten Routingmethoden umgehend nach Empfang einer entsprechenden Nachricht vom Kern abgearbeitet werden. Zum Aufbau einer Verbindung zu einem anderen Peer benötigt der Quellpeer zuerst einmal die eindeutige *GNUnet*-Adresse des Ziels. Deshalb ist ein vorhandenes HELLO-Paket des Zielpeers zwingend notwendig. Ohne dies kann keine Route aufgebaut werden.

Bevor jedoch „GetRoute“ angestoßen wird, findet eine Prüfung statt, ob nicht bereits eine gültige Route zum Ziel existiert. Besitzt der Quellpeer keinen Routingpfad zum Zielpeer, so startet er erst dann das „GetRoute“-Protokoll. Mithilfe der vorliegenden Zieladresse wird ein „GetRoute“-Pakete erzeugt, welches an alle erreichbaren Nachbarn verschickt wird. Diese müssen dann gemäß des nun folgenden Routings für die Bearbeitung des „GetRoute“-Pakets sorgen.

„GetRoute“

Erhält ein Peer eine „GetRoute“-Nachricht, prüft er zuerst, ob seine eigene *GNUnet*-Adresse mit der angegebenen Zieladresse innerhalb des „GetRoute“-Paketes übereinstimmt. Ist dies der Fall, antwortet der betroffene Peer mit einem „GetRouteReply“ auf die Nachricht, welches er an den Peer schickt, von dem er die „GetRoute“-Nachricht erhalten hat. Außerdem speichert der Empfänger die im „GetRoute“-Paket enthaltene Peerliste als Routingpfad zur Quelle ab.

Stimmt die Zieladresse hingegen nicht mit der eigenen Adresse überein, so muss das Paket zwangsläufig weitergeleitet werden. Bei der Weiterleitung eines „GetRoute“-Pakets muss jeder Peer dabei seine eigene *GNUnet*-Adresse an die im Paket enthaltene Source-Route anhängen. Beginnend beim ursprünglichen Quell-Peer entsteht über die Zeit eine Routing-Liste aller Peers, die das Paket weitergeleitet haben.

Um Zyklen zu verhindern, prüft jeder Peer beim Empfang des „GetRoute“-Pakets, ob seine eigene Adresse nicht bereits in der Liste enthalten ist. Ist dies der Fall, verwirft er das Paket bzw. leitet es nicht weiter. Auf diese Weise werden Routing-Zyklen verhindert, und das Netzwerk wird durch die schnelle Auslöschung von doppelten „GetRoute“-Anfragen nicht unnötig überflutet.

„GetRouteReply“

Das Verhalten beim Empfang einer „GetRouteReply“-Nachricht ist dem von „GetRoute“ sehr ähnlich, wobei die Peers hier nur dafür sorgen, dass das Paket vom Ziel zur ursprünglichen Quelle zurückkehrt. Nach Empfang einer „GetRouteReply“-Nachricht sucht der jeweilige Peer also nach der eigenen Adresse innerhalb der mitgelieferten Source-Route und sendet das Paket an den dort verzeichneten unmittelbaren Vorgänger. Sofern dies durch kurzfristige Veränderungen der Netzwerktopologie nicht möglich ist, wird der Routingvorgang durch ein Timeout abgebrochen.

Jeder Peer auf dem Pfad der Source-Route zwischen den Quell- bzw. Ziel-Peers ist also als Teil der Route für die Weiterleitung der Pakete verantwortlich. Diese Weiterleitung findet solange statt, bis das „GetRouteReply“-Paket bei der ursprünglichen „GetRoute“-Quelle angekommen ist.

Bei den einzelnen Peers auf der Route findet keine Speicherung der Pfadliste statt, da dies das Routing bzw. die Prüfung auf vorhandene Pfade erhebliche verkompliziert hätte. An der Quelle erkennt der Peer, dass er das Request vor einiger Zeit initiiert hat und entnimmt die enthaltene Source-Route. Diese Route speichert er nun als Pfadliste ab und merkt sich, dass er zum Zielpeer eine Route besitzt. Spätere Routenanfrage des Paketdienstes können dann mit der vorliegenden Source-Route beantwortet werden, so dass keine neue Route aufgebaut werden muss.

Datenaustausch per Paketdienst

Nachdem durch die anfangs erwähnten Routinen „GetRoute“ und „GetRouteReply“ gezielt ein Netzwerkpfad zwischen zwei Teilnehmern aufgebaut werden kann, können diese mithilfe des Paketdienstes anschließend Daten austauschen.

Zur Abgrenzung der Nutz- und Routingdaten wurde ein weiterer Nachrichtentyp zur Behandlung von Nutzdaten in *GNUnet* integriert und mit einer eigenen Callback-Funktion versehen. Diese sorgt dafür, dass eine Nachricht nach dem Empfang entweder an höhere *UbiMuC*-Schichten weitergegeben wird, oder aber eine Weiterleitung der Daten zum nächsten Peer auf der Pfadliste stattfindet.

Die Unterteilung in verschiedene Nachrichtentypen war notwendig, da *GNUnet* die Routingdaten mit höherer Priorität verarbeiten soll. Dies ist nur über eigene Typen möglich, die mit statischen Prioritätswerden im Quellcode bevorzugt gesendet und verarbeitet werden.

Inhaltlich stellt der Paketdienst dabei einfache Sende- und Empfangsschnittstellen bereit, die lediglich einen Datencontainer (für die Nutzdaten) sowie eine Empfangs-Adresse benötigen. Auf Anwendungsebene muss also nur ein passend typisierter Datencontainer erstellt werden, der über den Paketdienst zum Ziel transportiert wird.

Details wie Routenaufbau oder konkrete Behandlung der Nachrichtentypen etc. sind auf Anwendungsebene völlig unerheblich, da dies durch den Paketdienst gekapselt wird.

Auswirkungen des Routings

Mit Hilfe der eigens entwickelten „GetRoute“- und „GetRouteReply“-Methoden ist es möglich, eine Verbindung zwischen zwei Teilnehmern an beliebigen Stellen des *UbiMuC*-Netzwerkes aufzubauen. Durch die automatische Weiterleitung von „GetRoute“ ist außerdem sichergestellt, dass alle momentan erreichbaren Peers benachrichtigt werden.

Ausnahmen stellen hier Szenarien dar, in denen der Ziel-Peer erst kurz nach der Bearbeitung einer „GetRoute“-Anfrage dem Netzwerk beitritt und so in erster Instanz nicht berücksichtigt werden kann. Zum Zeitpunkt der Bearbeitung der Anfrage ist jedoch der Netzwerkzustand korrekt wiedergegeben worden, da besagter Peer nicht online war.

Im Hinblick auf die mögliche Netzwerkgröße ist allerdings zu erwähnen, dass das vorliegende Konzept der SourceRoute nicht für große, ausgedehnte Netze geeignet ist. Mit zunehmender Pfadlänge steigt zum einen die Länge der eigentlichen SourceRoute, außerdem erhöht sich gleichzeit die Anzahl der „GetRoute“-Pakete, die von jedem einzelnen Peer losgeschickt werden. Da *UbiMuC* aufgrund der Reichweitenbeschränkung von Ad-Hoc-Netzwerken auf kleine Topologie ausgelegt ist, sollte dieser Fall eigentlich keine Probleme bereiten.

4.2.1.3 Vermittlungskonzepte

Durch das auf *GNUnet* aufbauende Netzwerk sind wir nun in der Lage, Daten zwischen Peers auszutauschen. In Sachen Funktionalität befinden wir uns auf der dritten Schicht, der „Netzwerkschicht“, des ISO/OSI-Referenzmodells. Analog zu diesem Modell ist es nun das Ziel, die „Transportschicht“ zu implementieren, die es zusätzlich ermöglicht, einzelne Dienste auf einem Peer zu unterscheiden.

Transportdienste

Die Transportschicht von *UbiMuC* sah ursprünglich zwei Dienste vor, den „Paketdienst“ und die darauf aufbauende „Bidirektionale Pipeline“. Von den Funktionen her ähneln diese Dienste dem *User Datagram Protocol* (UDP) und dem *Transmission Control Protocol* (TCP). Der Paketdienst von *UbiMuC* ist hierbei das verbindungslose Pendant zu *UDP* und für das Senden und Empfangen von unabhängigen Datencontainern zuständig.

Die Typ-Enumeration der „Datacontainer“-Objekte übernimmt in diesem Fall die Aufgabe der Portnummer und dient der Unterscheidung von unterschiedlichen Diensten. Abgesehen von der Vorsortierung nach Pakettyp und Zwischenspeicherung von empfangenen Paketen, die in Abschnitt 4.2.2.2 genauer beschrieben wird, hat der Paketdienst keine weiteren Aufgaben.

Insbesondere findet keine Sicherung der Paketreihenfolge statt. Das ist die Aufgabe der verbindungorientieren „Bidirektionalen Pipeline“, die dazu die restlichen Header des Datencontainer nutzt. Neben der Reihenfolge, die die Sequenznummer nutzt, wird mit der Verbindungsnummer eine eindeutige Zuordnung zwischen zwei Endpunkten vorgenommen. Zusätzlich werden nicht mehr nur ganze Pakete zwischengespeichert, sondern die gesamten Nutzdaten als Bytestrom, um bei Bedarf auch zeichenweisen Zugriff zu ermöglichen.

Im Laufe der Zeit hat sich herausgestellt, dass die Zusatzfunktionen der „Bidirektionalen Pipeline“ für die Implementierung der in *UbiMuC* enthaltenden Anwendungen nicht benötigt wurden. Für die Video- und Audioübertragung wurde ein UDP-Tunnel auf Basis des Paketdienstes entwickelt, der in Abschnitt 4.2.2.3 genauer beschrieben wird. Chat-Nachrichten werden jetzt ebenfalls über den Paketdienst abgewickelt. Und auch für die Übertragung der Informationen der Nutzerverwaltung wurde eine Lösung mit der Hilfe der „Kontrollnachrichten“ gefunden, auf die in Abschnitt 4.2.2.4 eingegangen wird. Die Weiterentwicklung der „Bidirektionalen Pipeline“ wurde damit gestoppt.

4.2.2 Implementierung

Der Paketdienst bzw. das „BidiPipe“-Modul wurden, wie auch der *Avahi*-Dienst, als Applikationen in *GNUnet* eingebunden. Die Kommunikation mit der WIR-Schicht läuft über einen TCP-Socket, über den die zu sendenden bzw. die empfangenen Daten als normale *GNUnet*-Nachrichten aus der WIR-Schicht nach *GNUnet* bzw. in die entgegengesetzte Richtung transportiert werden. Auf der *GNUnet*-Seite werden Handler registriert, die die vorsortierten Nachrichten entsprechend ihres Inhalts weiterverarbeiten.

Da *GNUnet* selbst bereits beim Eingang der Daten den Typ über ein Header-Byte identifiziert und an die entsprechenden Handler-Routinen weiterleitet, erschien es uns sinnvoll, keinen generischen *UbiMuC*-Typ einzuführen, sondern gleich verschiedene Datentypen bzw. deren Handler zu registrieren. Da die eingehenden Daten ohnehin identifiziert werden, sparen wir somit eine zusätzliche Identifizierungsfunktion und können die Daten unmittelbar und typspezifisch verarbeiten. Diese Handler werden beim Start von *gnunetd* für die einzelnen Datentypen registriert und dann beim Eingang eines Pakets von *GNUnet* aufgerufen.

Die durch „GetRoute“-Anfragen entstandenen Routen werden am Ausgangspunkt zwischengespeichert und nach dem Ablauf von fünf Minuten nach dem letzten erfolgreichen Datentransfer gelöscht. Kommt aus der WIR-Schicht ein zu versendendes Paket an, wird zunächst überprüft, ob für den Ziel-Peer eine gültige Route besteht. Ist dies der Fall, wird die Route den Daten vorangestellt und an den ersten Hop in der Route versendet. Andernfalls wird ein „GetRoute“ angestoßen. Sobald eine Route aufgebaut werden konnte, werden die Daten unmittelbar versendet. Wurde innerhalb von zehn Sekunden keine Route gefunden, wird das Paket verworfen.

Um eine abgebrochene Verbindung erkennen zu können, ohne das Netzwerk zu stark zu belasten, wurde ein rudimentäres Schema zum Versenden von ACK- bzw. NAK-Nachrichten implementiert. Beim Eingang eines Datenpakets wird aus den letzten 4 Byte der Nutzdaten ein ACK-Paket erstellt, welches an den unmittelbar vorhergehenden Hop gesendet wird.

Trifft andererseits bei einem Hop nach Versenden bzw. Weiterleiten eines Pakets innerhalb von drei Sekunden kein ACK-Paket ein, so wird ein NAK-Paket generiert, welches an den ursprünglichen Absender des Pakets zurückgesendet wird. In diesem NAK-Paket ist das komplette nicht zugestellte Datenpaket enthalten, damit eine erneuter Zustellversuch gestartet werden kann.

Durch die Nutzung des in *GNUnet* integrierten CRON-Dienstes (siehe Abschnitt 3.1.2) können die oben angegebenen Zeitschranken überprüft werden, ohne das der Programmfluss angehalten werden muss.

4.2.2.1 Handler

In der *UbiMuC*-Transportschicht gibt es vier verschiedene Handler, die sich um die Verarbeitung der unterschiedlichen Nachrichtentypen kümmern:

- Datencontainer
- „GetRoute“
- „GetRouteReply“
- WIR-Schicht

Handler für Datencontainer

Der Handler für den Datencontainer empfängt die Datencontainerpakete (inklusive Route), verarbeitet sie und sendet sie danach weiter. Nach dem Empfangen wird das Paket deserialisiert und überprüft, an welchen Peer es adressiert ist. Falls es an den aktuellen Peer geht, wird es

für die WIR-Schicht verpackt, serialisiert und an die WIR-Schicht gesendet. Wenn ein anderer Peer das Ziel ist, wird es zum nächsten Hop weitergeleitet und dem vorherigen Knoten ein ACK gesendet.

Handler für „GetRoute“

Nach dem Deserialisieren vergleicht der Handler für die „GetRoute“-Pakete die Zieladresse mit der eigenen Adresse. Sind die beiden Adressen gleich, wird ein „GetRouteReply“-Paket angestoßen. Bei unterschiedlichen Adressen trägt der Handler die Adresse des Peers in die Routingtabelle ein und sendet das aktualisierte Paket an seine benachbarten Hosts weiter.

Handler für „GetRouteReply“

Der Handler für die „GetRouteReply“-Pakete nimmt das Paket entgegen, deserialisiert es und überprüft die Zieladresse. Sofern der Peer nicht selber das Ziel ist, leitet er es an seinen Vorgänger in Routingtabelle weiter und speichert die Source-Route für kommende Routings. Wenn der Peer das Ziel war, fügt er die erhaltene Source-Route in die lokale Routingstruktur ein.

Handler für die WIR-Schicht

Der Handler für die WIR-Schicht nimmt die Pakete aus der WIR-Schicht entgegen und startet das „GetRoute“. Wenn eine Source-Route gefunden wurde, werden die Nutzdaten mit dem Routingpfad versehen und anschließend versendet.

4.2.2.2 Paketdienst-Schnittstelle

Wie man in den vorherigen Abschnitten zur Transportschicht bereits erkannt haben sollte, ist es alles andere als trivial, Daten mit *GNUnet* zu übertragen. Um diese Komplexität nicht noch in die Anwendungsschicht zu bringen, benötigen wir eine klar definierte Schnittstelle, die sämtliche Elemente und Strukturen von *GNUnet* sicher kapselt. Dies ist die Aufgabe der Paketdienst-Schnittstelle.

Die Schnittstellenklassen sind modular aufgebaut, wobei hier alle Module auf der Klasse *PacketService* aufbauen, die eine abstrakte Schnittstelle zum Empfangen und zum Senden von Datencontainern darstellt. Eine umfassende Implementierung des *PacketService* auf der Basis von *GNUnet* ist die Klasse NetworkLayer. Dort werden ausgehende Datencontainer, wie sie in Abschnitt 4.1 beschrieben sind, serialisiert und an *gnunetd* weitergereicht. Parallel dazu wartet ein spezieller Lese-Thread auf eintreffende Datencontainer, die nach dem Deserialisieren in einem Empfangspuffer zwischengespeichert werden und dort auf Abholung warten.

Da für *UbiMuC* verschiedenste Daten übertragen werden müssen, mussten wir auf der Basis des PacketService und des NetworkLayers einen Multiplexer entwickeln, der eingehende Datencontainer in verschiedene Eingangswarteschlangen vorsortiert. Die Klasse Splitter übernimmt diesen Part und wertet dazu das Typfeld der Datacontainer-Objekte zur Unterscheidung der

eintreffenden Pakete aus. Neben der Sortierfunktion ist der Splitter auch noch Ansatzpunkt für verschiedene andere Dienste, wie beispielsweise der Kontroll-Nachrichten, die im Abschnitt 4.2.2.4 genauer beschrieben werden.

4.2.2.3 Erstellung der UDP-Sockets

Um für die AV-Konferenz die Daten an das Netzwerk zu schicken, wurde eine Schnittstelle benötigt. Dazu wurde eine Methode implementiert, die bei Bedarf Sockets aufbaut, welche dann genutzt werden um Daten an das Netzwerk zu schicken.

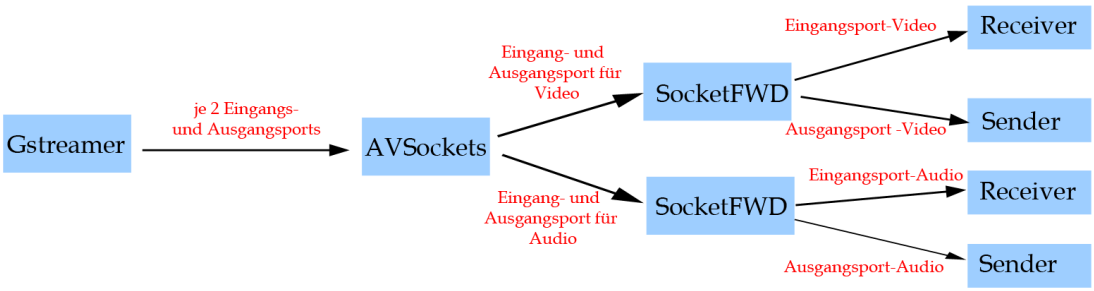

Abbildung 4.5: Aufbau der Socketverbindungen

Dafür wurden die zwei Methoden „AVSockets“ und „SocketFWD“ implementiert. Wie in Abbildung 4.5 zu sehen ist, kann die Methode „AVSockets“ von dem *GStreamer* aufgerufen werden und erwartet die Ein- und Ausgangsports für die Videoschnittstelle, die Ein- und Ausgangsports für die Audioschnittstelle und die Peeradresse des Ziels. Sie dient im Prinzip nur dazu, einen vereinfachten Aufruf für die Sockets zu ermöglichen. Sie macht nichts weiteres, als jeweils für Audio und Video die entsprechenden Methoden in „SocketFWD“ aufzurufen.

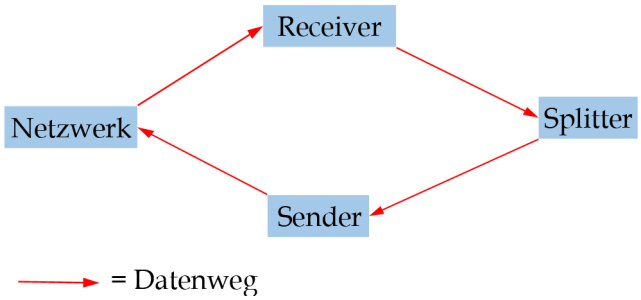

Abbildung 4.6: Datenwege über die UDP-Sockets

„SocketFWD“ erstellt für den Eingangsport einen Socket vom Typ „Receiver“, der die Daten aus dem Netzwerk in Empfang nimmt und sie an den Splitter weiterleitet. Für den Ausgangsport wird der Socket vom Typ „Sender“ erstellt, der die Daten an sein Ziel im Netzwerk schickt. Abbildung 4.6 verdeutlicht hierbei noch einmal den Datenstrom zwischen Splitter und Netzwerk über die Sockets.

4.2.2.4 Kontroll-Protokoll

Der Paketdienst ist gut geeignet, um größere Datenmengen zu übertragen. Die Pakete, die auf der einen Seite versendet werden, kommen beim Empfänger nach und nach an und können zu beinahe beliebigen Zeitpunkten aus dem Empfangspuffer gelesen werden.

Dieses Prinzip impliziert allerdings, dass auf Empfangsseite regelmäßig die Ankunft von Paketen abgefragt wird. Meistens ist dazu ein eigener Thread nötig, da längere Lesevorgänge die Benutzeroberfläche für die Zeit einfrieren lassen.

Oft benötigt man aber keine aufwändige Datenverarbeitung und es reicht eine kleine Behandlungsroutine aus, die prinzipiell auch direkt vom Paketdienst aufgerufen werden kann. Hierzu wurde das „Kontroll-Protokoll“ entwickelt.

Umsetzung

Das „Kontroll-Protokoll“ sieht vor, dass man im Splitter sogenannte Callback-Funktionen zu einer bestimmten Signatur registrieren kann. Die Signatur besteht in unserem Falle aus einer einfachen Zeichenkette und ermöglicht es, verschiedene Kontrollnachrichten zu unterscheiden.

Die Callback-Funktion wird vom Splitter aufgerufen, sobald eine Kontrollnachricht mit passender Signatur empfangen wird und enthält eine Kopie der Nutzdaten des zugehörigen Datenpaketes. So lassen sich kurze (Steuer-) Nachrichten austauschen, ohne dafür eigene Threads anzulegen.

Der Austausch der erweiterten Benutzerinformationen der Kontaktliste wurde beispielsweise damit implementiert. Hierbei sollte man allerdings beachten, dass die Callback-Funktionen nicht zu lang werden, da der Splitter in dieser Zeit blockiert und keine weiteren Pakete empfangen kann.

4.2.3 Auslastungs und Geschwindigkeitstests von *GNUnet*

Testaufbau

In diesen Tests wurde unser Paketdienst in Bezug auf die Geschwindigkeit getestet. Dazu wurden mehrere Geräte wie in Abbildung 4.7 vorgestellt aufgebaut. Die beiden Geräte A und C können nur das Gerät B erreichen, sehen sich aber nicht gegenseitig. Das Gerät B steht in der Mitte und hat Verbindung zu beiden Geräten. Auf diese Weise wird sichergestellt, dass bei einer Kommunikation zwischen Gerät A und C ein echtes „Multihop“-Routing zustande kommt.

Diese Tests wurden durchgeführt, um sicherzustellen, dass unser Paketdienst eine ausreichende Geschwindigkeit für die Multimediaübertragung zur Verfügung stellen kann. Zusätzlich konnte damit die Zuverlässigkeit des Paketdienstes überprüft werden.

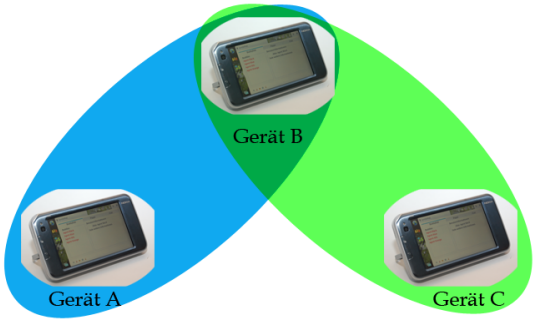

Abbildung 4.7: Grafische Darstellung des Testaufbaus

Abbildung 4.8 zeigt hier den Testaufbau mit fünf Geräten. Die Geräte wurden hier, wie in Abbildung 4.7 schematisch dargestellt, aufgebaut, so dass jeder nur seinen Vorgänger sowie seinen Nachfolger erreichen kann.

Abbildung 4.8: Aufbau des Multihoptests mit fünf Geräten

Testprogramme

Als erstes wurde ein Testprogramm geschrieben, welches versucht eine Route aufzubauen, um dann Nachrichten zu versenden. Der Aufruf hierzu ist `./chat-client peerid`, wobei peerid für die Peeradresse des Chat-Partners steht.

Nach dem Starten des Programms konnte man Nachrichten eintragen, die beim Chat-Partner dann auf der Konsole dargestellt wurden. Dieses Programm diente nur dazu, herauszufinden, ob der Paketdienst es schafft, eine Verbindung aufzubauen und ob er auch über mehrere Hops routen kann.

Um die Datenrate zu testen, wurde ein Benchmarkprogramm geschrieben, welches als Server oder Client gestartet werden kann. Der Aufruf des Benchmarkprogramms geschieht mit `./benchmark c|s peerid`, wobei c oder s für Client oder Server steht und peerid für die Peeridentity des Empfängers.

Das Testprogramm lässt als Server *GNUnet* eine Verbindung aufbauen und schickt dann dauerhaft Daten an den Client. Auf dem Client wird die Uhrzeit des ersten empfangenen Paketes festgehalten und gemessen, wie viele Daten ankommen. So kann man nach dem Beenden des Tests ablesen, wie viele Daten in welcher Zeit übertragen worden sind.

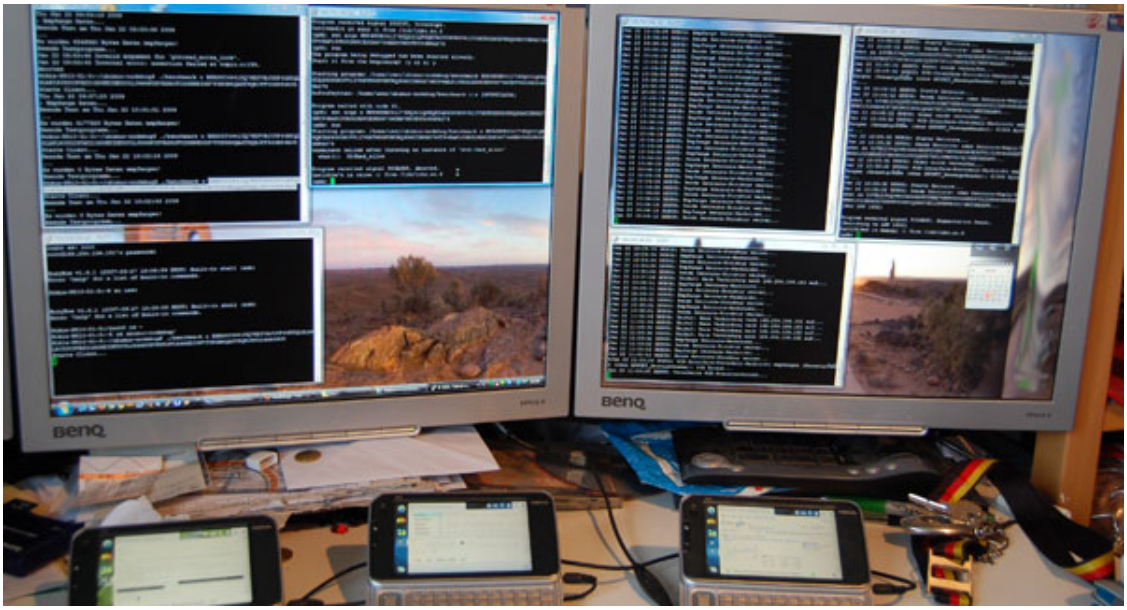

Abbildung 4.9: Testaufbau mit Konsolenverbindungen

Um die Handhabung der Geräte zu vereinfachen, wurde auf den Geräten ein SSH-Server installiert und an einem Desktoprechner mittels Putty SSH-Verbindungen zu den einzelnen Geräten geöffnet. Dies wird in Abbildung 4.9 gezeigt, wo für drei N810 auf einem Desktoprechner pro N810 jeweils zwei SSH-Verbindungen geöffnet wurden. Eine Verbindung war für den Benchmark und die andere um *GNUnet* zu starten. Dadurch hatte man eine viel bessere Übersicht und konnte auch einfach mit Copy & Paste die langen Peeridentitys von einer Konsole in die nächste kopieren.

Testergebnisse

Im Verlauf der Projektgruppe wurden drei *GNUnet*-Versionen getestet, die jeweils den aktuellen Entwicklungsstand wiederspiegelten und eine fortschreitende Entwicklung des Projektes darstellen.

In Tabelle 4.1 wurde das Benchmark Programm mit zwei Geräten unter *GNUnet* getestet; also ohne einen Hop zwischen den Geräten. Hier wurden Geschwindigkeiten zwischen 50 und 67 KB/s erzielt.

Dauer (min)	Gesendete Daten (Bytes)	KB/s
03:16	13516800	67,35
19:46	60518400	49,38
05:24	19169280	57,78
03:16	10137600	50,51
10:15	30904320	49,07

Tabelle 4.1: Test mit *GNUnet* Revision 804, zwei Geräte

Beim zweiten Test wurde die *GNUnet* Revision 804 mit drei Geräten getestet. Die Anmerkung *iptables* deutet hier darauf hin, dass die Reichweite der Geräte künstlich durch einen *iptables*-Eintrag eingeschränkt wurde. Die beiden Geräte A und C sind durch *iptables*-Einträge gegenseitig gesperrt, so dass sie nicht mehr direkt miteinander kommunizieren können und das Gerät B als Hop benutzen müssen. Bei der Anmerkung „echter Hop“ wurde die Geräte so weit voneinander entfernt, dass sie sich wirklich nicht mehr sehen konnten.

Dauer (min)	Gesendete Daten (Bytes)	KB/s	Anmerkungen
01:35	3686400	37,89	iptables
04:22	13639680	50,84	iptables
02:40	5529600	33,75	echter Hop, loud
00:33	1781760	52,73	echter Hop, loud auf Senderseite
05:48	18923520	53,1	echter Hop, loud auf Senderseite

Tabelle 4.2: Test mit *GNUnet* Revision 804, drei Geräte

Die Ergebnisse zeigen, dass sich hierbei kein großer Unterschied zwischen den beiden Testkonstellationen ergibt. Beide Testkonstellationen schwanken circa zwischen 34 KB/s und 53 KB/s. In Tabelle 4.2 sind die Ergebnisse dargestellt, wobei die Anmerkung „loud“ bedeutet, dass *GNUnet* mit Debug-Ausgaben gestartet wurde.

Dauer (min)	Gesendete Daten (Bytes)	KB/s	Anmerkungen
03:31	22179840	102,65	loud
05:06	33300480	106,27	loud
01:48	11243520	101,67	
02:59	18493440	100,89	
04:00	25067520	102	

Tabelle 4.3: Test mit *GNUnet* Revision 1179, drei Geräte

Die Tabelle 4.3 zeigt die Ergebnisse der Tests mit der *GNUnet*-Revision 1179, bei der sich der Datendurchsatz deutlich erhöht hat. Hier wurde wieder mit drei Geräten getestet und Ergebnisse um die 104 KB/s erzielt. Bis zu dieser Version war die Verbindung noch nicht stabil, so dass die Tests nicht über längere Zeit durchgeführt werden konnten.

Dauer (min)	Gesendete Daten (Bytes)	KB/s	Anmerkungen
02:59	19353600	105,59	loud
02:36	16465920	103,08	loud
02:34	15421440	97,79	loud
05:02	31764480	102,72	loud
24:49	138117120	90,58	loud

Tabelle 4.4: Test mit GNUnet Revision 1187, drei Geräte

Beim letzten Test wurde die *GNUnet*-Revision 1187 mit drei Geräten getestet. In dieser Version wurde ein Bugfixing am Paketdienst vorgenommen und es sollte überprüft werden, ob sich die Performanz verändert hat. Tabelle 4.4 stellt diese Ergebnisse dar. Da die Verbindung in dieser Version viel stabiler geworden ist, konnte ein circa 25 minütiger Test durchgeführt werden, um zu schauen wie sich der Durchsatz über längere Zeit verhält.

Auswertung

Die Tests haben gezeigt, dass es bei der Geschwindigkeit zwischen einem Versuchsaufbau mit zwei und einem mit drei Geräten keinen großen Unterschied gibt. Es lässt sich auch die Entwicklung der *GNUnet*-Versionen ablesen. Während die Verbindung bei der *GNUnet*-Revision 804 noch ziemlich instabil war und deshalb keine Langzeittests möglich waren, haben die Änderungen in Revision 1179 eine deutliche Leistungssteigerung gebracht.

Da sich bei dem realen Test die Geschwindigkeit gegenüber der künstlichen Testumgebung mit *iptables* nicht geändert hat, läßt sich daraus schließen, dass die Begrenzung der Geschwindigkeit softwareseitig ist und nicht durch die Hardware hervorgerufen wird. Durch die Änderungen am Paketdienst in Revision 1187 ist die Verbindung dann viel stabiler geworden, so dass auch Langzeittests möglich waren. Insgesamt wurde gezeigt, dass der Paketdienst eine ausreichende Geschwindigkeit für *UbiMuC* bereitstellt, wobei die Geschwindigkeit dennoch nach oben begrenzt ist.

4.3 Nutzerverwaltung

In diesem Unterkapitel wird auf den Teil unseres Programms eingegangen, der die Verwaltung von Nutzern regelt. Der Zweck diese Teils unserer Software ist, die Kommunikation verschiedener *UbiMuC*-Nutzer innerhalb eines gemeinsamen Netzes komfortabler zu gestalten, indem wir jedem Nutzer eine Liste bekannter anderer Peers inklusive deren Status bereitstellen. Auf diese Weise ist es dem Nutzer leichter möglich, den Überblick über aktuell im Netz erreichbare Peers zu behalten und mit diesen eine Verbindung aufzubauen.

Die Module für Chat und Multimedia-Streaming nutzen daher auch die von der Nutzerverwaltung bereitgestellten Daten zum Verbindungsaufbau. Darüber hinaus hilft die zentrale Sammlung all dieser Informationen auch, eine einfache Schnittstelle zur Anzeige in der GUI zu definieren. Die Idee lehnt sich an Kontaktlisten als zentrale Organisationsform der Gesprächspartner an, wie sie in den meisten Messaging-Programme üblich sind.

Zuerst werden die entwickelten Konzepte vorgestellt, mit welchen Abstraktionsebenen die Verbindung zwischen *GNUnet* und unseren Programmschichten hergestellt werden und wie diese mit der GUI verknüpft sind. Dabei wird insbesondere auf die Integration in der *WIR-Core*-Struktur und die Schnittstellen zwischen dieser und der GUI beziehungsweise der DHT in *GNUnet* eingegangen. Im zweiten Teil des Abschnittes werden schließlich die Implementierungsdetails genauer erläutert.

4.3.1 Konzepte und Spezifikationen

Die Nutzerverwaltung stellt innerhalb von *UbiMuC* das zentrale Element für die Suche von Nutzern und die persistente Speicherung der Nutzerdaten dar. Deshalb ist sie das Bindeglied zwischen der Anzeige der Nutzer in der GUI, der von *GNUnet* bereitgestellten DHT, dem Paketdienst und dem *WIR-Core*-Konstrukt.

Ziel dieses Moduls ist es, dem Nutzer von *UbiMuC* eine einfach zu bedienende Oberfläche zu bieten, mit der rudimentäre Aufgaben einer Nutzerverwaltung, wie man sie aus Messaging-Programmen kennt, erledigt werden können. Konkret bedeutet dieses, dass man andere im Netz verfügbare *UbiMuC*-Nutzer suchen und in einer Liste von Kontakten (im Weiteren auch „Buddyliste“ genannt) abspeichern kann. Diese Kontakte werden dauerhaft gespeichert und können mit einem selbst gewählten Alias versehen und wieder gelöscht werden.

Außerdem sind über die GUI weitere Informationen zu den Nutzern abrufbar, und über den Online-Status ist jederzeit einsehbar, ob die anderen Nutzer gerade innerhalb des Netzes verfügbar sind. Letztlich wird für die aktuell verfügbaren Nutzer die Optionen angeboten, eine Konferenz oder Chat-Session zu starten.

Die Nutzerverwaltung stellt dabei alle zur Verwaltung der Informationen benötigten Datenstrukturen bereit und verwaltet diese. All diese Elemente sind folglich im *WIR-Core* definiert, damit auch ein zentraler Zugriff auf diese Programmteile von allen Modulen, sofern nötig, möglich ist. Sie stellt ebenfalls die Speicherung der Daten über das Beenden das Programms hinaus persistent sicher, indem alle relevanten Daten automatisch in eine Config-Datei geschrieben werden. Das korrekte Lesen und Schreiben dieser Datei wird von einem Parser übernommen, auf den im Anhang eingegangen wird (siehe Anhang 7.1).

Die Informationen, wann Nutzer das lokale Netz betreten oder verlassen, bezieht *UbiMuC* aus der *GNUnet*-DHT. Jedoch müssen diese Daten zunächst für unsere Zwecke aufbereitet werden. Daher arbeitet die Nutzerverwaltung mit mehreren Abstraktionsebenen, die die Kapselung der DHT gegenüber unseren Programmteilen sicherstellt. Weiterhin ist durch diese Kapselung sichergestellt, dass die verschiedenen *UbiMuC*- und *GNUnet*-Module bei Bedarf unabhängig von einander gestartet werden können.

4.3.1.1 Abstraktionsebenen

Wie bereits in Abschnitt 3.1.3.1 detailliert dargestellt wurde, hat die *GNUnet*-DHT einige nachteilige Eigenschaften. Daher ist sie für die direkte Verwaltung unserer Nutzerdaten nicht geeignet. Insbesondere das nicht-deterministische Antwortverhalten bei Anfragen und die Tatsache, dass es sich um eine verteilte Struktur handelt, die lokal nicht vollständig verfügbar ist,

machen eine übergeordnete Verwaltungsebene notwendig. Weiterhin wollten wir eine ausreichende Kapselung unserer Software-Module gegenüber den *GNUnet*-Modulen gewährleisten, nicht zuletzt weil unsere Software in C++ geschrieben ist und *GNUnet* auf C basiert. Die Konsequenz ist, dass die Verwaltung der Nutzerdaten sich über 3 Ebenen erstreckt:

- dem DHT-Modul von *GNUnet*
- der *UbiMuC*-Schnittstelle zur DHT
- der Buddyliste der WIR-Schicht

DHT-Ebene

Auf der DHT-Ebene haben wir keinen Einfluss auf die genaue Verwaltung unserer Daten. Mit Hilfe von *GNUnet*-Funktionen werden der DHT Tripel von Nutzdaten übergeben, deren Speicherung und Verteilung komplett *GNUnet* überlassen ist. Diese Daten umfassen nur die Informationen, die zur Zuordnung der *GNUnet*-Nutzerdaten zu unseren Nutzerdaten nötig sind, und werden daher von *GNUnet* in keiner Weise inhaltlich verarbeitet, sondern ausschließlich von der darüber liegenden Schicht erstellt und verarbeitet.

DHT-Einträge haben, wie bereits in Abschnitt 3.1.3.1 unter Struktur beschrieben, einen Typen, nach dem sie aus der DHT angefordert werden können. Es können nur alle Einträge mit dem gleichen DHT-Typ gemeinsam angefragt werden, deshalb nutzen wir einen eigenen Typ namens „UbiMuC“ und versehen alle unsere Einträge mit diesem Typ. Damit haben wir die Möglichkeit, auf einfache Weise auf den für uns relevanten Teil der DHT-Einträge zuzugreifen.

DHT-Schnittstelle

Diese Ebene stellt die Abbildung der DHT in *WIR*-Datenstrukturen dar. Daher kann man auch von einem lokalen Abbild der DHT in Listenform sprechen. Die erwähnten Daten-Tripel werden hier zusammengestellt und für die Übergabe an die DHT formatiert. Diese Tripel umfassen den Benutzernamen, eine in *UbiMuC* eindeutige Hash-ID und die dem Nutzer aktuell zugeordnete *GNUnet*-ID.

Für die darüber liegende Schicht werden Funktionen angeboten, die zum Arbeiten mit der DHT verwendet werden können. Dazu zählen unter anderem das Auffrischen der lokalen DHT-Informationen, das Einfügen neuer Daten und das Abfragen der aktuell lokal gespeicherten Inhalte. Ein explizites Löschen der Daten ist nicht nötig, da die DHT diese selbstständig Timeout-basiert entfernt, wenn diese nicht regelmäßig aufgefrischt werden.

Buddylisten-Ebene

Auf dieser Ebene wird unsere persistente Liste von Nutzern realisiert. Im Gegensatz zum Ansatz der DHT sollen in die auf dieser Ebene verwaltete Liste nur auf expliziten Wunsch des Nutzers neue Einträge aufgenommen und vorhandene gelöscht werden. Daraus resultiert die persönliche Buddyliste, die in der GUI angezeigt werden kann. Außerdem werden die

rudimentären Nutzerdaten der DHT zu vollständigen Benutzerprofilen erweitert. Dazu werden den *UbiMuC*-IDs die separat verwalteten Zusatzdaten der Nutzer und ein Online-Status-Flag zugeordnet. Regelmäßige Überprüfungen, ob alle Nutzer noch in der DHT gelistet sind, stellen sicher, dass dieser Status aktuell gehalten wird. Zusätzlich werden die Daten für die Abspeicherung in der Config-Datei aufbereitet.

Auf Strukturen dieser Ebene greifen die GUI und die anderen Module des Programms zu, wenn Nutzerdaten benötigt werden. Daher werden alle Funktionen bereitgestellt, die zum Bearbeiten und Speichern der Buddyliste in den anderen Modulen benötigt werden.

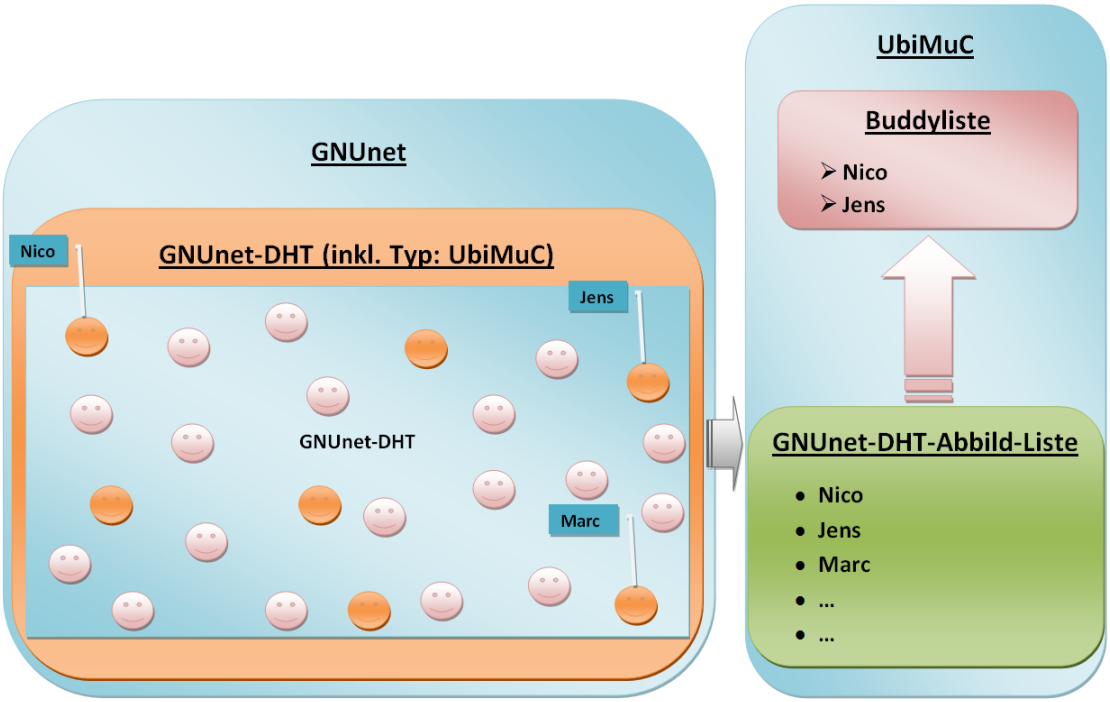

Abbildung 4.10: Schema der Abstraktionsebenen

In Abbildung 4.10 wird das Zusammenspiel dieser drei Ebenen nochmal schematisch dargestellt. Der rechte Teil stellt die Einträge des DHT-Moduls in *GNUnet* dar. Die roten Smileys repräsentieren dabei die mit dem Typ „UbiMuC„ versehenen Einträge, also alle im Netz gelisteten UbiMuC-Nutzer. Auf der rechten Seite sieht man unten die DHT-Schnittstelle in *UbiMuC*. In dieser werden nur noch die DHT-Einträge mit dem Typ „UbiMuC“ verwaltet. Darüber ist die daraus generierte Buddyliste abgebildet, in der wiederum nur noch die in der individuellen Nutzerverwaltung angelegten, bekannten Nutzer erscheinen.

Buddylisten-Caching

Die Abstraktion der DHT-Information zu einer persistenten Liste bekannter Nutzer macht die Verwaltung und Nutzung der Nutzerdaten für unsere anderen Module wie den Chat und das Streaming deutlich komfortabler. Ein Großteil der Nachteile der DHT könnte auf diesem Wege ausgeglichen werden. Jedoch kann sich das nicht-deterministische Antwortverhalten der

DHT noch immer bei den periodisch ausgeführten DHT-Rückfragen zur Aktualisierung des Online-Status aller Einträge in der Buddyliste negativ bemerkbar machen. Unvorhergesehene Änderungen in der DHT könnten sich dadurch direkt in spontanen Wechseln zwischen Online- und Offline-Status niederschlagen, was gerade bei Versuch eines Verbindungsaufbaus zwischen zwei Peers zu Komplikationen führen kann.

Unsere Lösung für dieses Problem ist das Cachen der Einträge in unserer Buddyliste. Das lokale Abbild der DHT in der WIR-Schicht wird nicht nach jeder Überprüfung der DHT-Inhalte vollständig neu erstellt, sondern zwischengespeichert, und zusätzlich wird mit abgespeichert, wie oft der Eintrag eines Nutzers nun schon nicht mehr vorgefunden wurde.

Erst nachdem ein solcher Nutzer nach mehrfachen Anfragen nicht mehr in der DHT vorgefunden wurde, wird er aus der Liste entfernt. Dies führt auf der darüberliegenden Ebene wiederum dazu, dass der Status in der Buddyliste auf „Offline“ gesetzt wird. Auf diese Weise ist ein eventuelles Fehlerverhalten der DHT ausgleichbar und es können sehr rasche und ungewollte Wechsel des Online-Status abgefangen werden.

4.3.1.2 Schnittstellen zwischen DHT und Core sowie Core und GUI

Die Schnittstelle zwischen DHT und Core befasst sich vor allem mit der Ansteuerung der DHT über *GNUnet* Befehle. Hauptsächlich werden dabei zwei Befehle benutzt: zum Lesen von Daten aus der DHT und zum Schreiben von Daten in die DHT. Die durch das Lesen der Daten erhaltenen Einträge werden in einer lokalen Liste zwischengespeichert und können von dort aus weiterverwendet werden. Sie stellen ein Abbild der aktuell (lokal) vorhandenen DHT dar.

Die Schnittstelle zwischen dem Core und der GUI dient dazu, die in der Buddylist-Datenstruktur im Core abgelegten Informationen graphisch zu veranschaulichen. Dabei wird eine Liste der aktuell betrachteten „Buddies“ - inklusive deren „Onlinestatus“ - an die GUI weitergeleitet. Dort werden die einzelnen Einträge in einer Liste angezeigt und entsprechend ihres „Onlinestatus“ eingefärbt. Die eigentliche Verwaltung des „Onlinestatus“ und das Sortieren nach Onlinestatus bzw. Alias geschieht dabei im Core. Die GUI kümmert sich nur um die graphische Darstellung.

4.3.2 Implementierung

Bei der Implementierung können drei Teile unterschieden werden:

- das Ansteuern der DHT in *GNUnet*
- das eigentliche Verwalten der einzelnen Kontakte
- das Anzeigen in der GUI

Ansteuern der DHT

Das Ansteuern der DHT in *GNUnet* befasst sich in erster Linie mit der *GNUnet*-API. Hier wird eine Verbindung zum eigentlichen *GNUnet*-Daemon aufgebaut, und mit Hilfe dieser Verbindung werden die Befehle zum Lesen beziehungsweise Schreiben von Werten aus der - respektive in die - DHT realisiert.

Da es sich bei *GNUnet* um ein in C geschriebenes Tool handelt, geschieht dies über einzelne Funktionsaufrufe, die den Erfolg beziehungsweise Misserfolg mittels eines Status-Codes mitteilen. Daher haben wir die einzelnen Routinen in eine C++-Klasse gekapselt, welche die einzelnen C-Funktionen benutzt und die Ergebnisse in Form einer Liste an die nächste Schicht (das Verwalten der einzelnen Buddies) weiterleitet.

Das Verwalten der einzelnen Buddies

Das Ergebnis der DHT-Ansteuerung ist eine Liste, die ein Abbild unserer lokalen DHT darstellt. In dieser Liste sind jedoch nicht nur Informationen zu den einzelnen Personen in der Buddyliste, sondern auch zu vielen weiteren Buddies vorhanden. Daher muss die von der DHT erhaltene Liste gefiltert und mit den Einträgen in der jeweilig aktuellen Buddyliste verknüpft werden. Dies geschieht dadurch, dass - entsprechend der Einträge in der von der DHT stammenden Liste - der Onlinestatus der Einträge der spezifischen Buddyliste aktualisiert wird.

Zusätzlich wird hier eine Art „Caching“ realisiert, um der Unzuverlässigkeit (genauer gesagt: dem nicht-deterministischen Verhalten) der DHT entgegen zu wirken. Die resultierende Buddyliste - inklusive Onlinestatus der einzelnen Buddies - wird zur Weiterleitung an die GUI zusätzlich noch sortiert (primär nach Onlinestatus, sekundär lexikalisch).

Anzeigen in der GUI

Die so bearbeitete Liste wird daraufhin zum Anzeigen an die GUI weitergeleitet. Dort werden die einzelnen Einträge in eine GTK-kompatible Datenstruktur konvertiert und in einer Liste angezeigt. Der Onlinestatus wird dabei durch eine entsprechende Farbe (Grün für Online, Rot für Offline) realisiert.

Als Schwierigkeit hat sich hierbei die Prozess-Synchronisation erwiesen, da GTK nicht „threadsafe“ ist. Daher musste über das Sperren beziehungsweise Lösen eines Mutex sichergestellt werden, dass die Datenstruktur zur Speicherung der darzustellenden Einträge konsistent bleibt.

Des Weiteren stellt die GUI eine gewisse Funktionalität zum Hinzufügen und Löschen von Buddies, sowie zum Umbenennen eben dieser bereit. Dabei handelt es sich um einfache Dialoge, die die entsprechenden Befehle an den Core weiterleiten, um die dort befindlichen Datenstrukturen dementsprechend anzupassen.

4.4 Chat

Das *UbiMuC*-Chatmodul soll den Benutzern der Software im Rahmen der in Abschnitt 1.2 vorgestellten Anwendungsfälle und Anforderungen eine intuitiv zu bedienende und einfach gehaltene Möglichkeit bieten, beliebige textuelle Nachrichten untereinander auszutauschen. Die Kommunikationsbeziehungen beschränken sich dabei immer auf genau zwei Teilnehmer des Netzwerkes, die miteinander in Kontakt treten möchten.

Es sind keinerlei Multicast-Optionen oder Chat-Räume vorgesehen, der Datenaustausch findet ausschließlich zwischen zwei Personen statt. Inhaltlich soll es dabei keine Rolle spielen, an welchen Punkten des Netzwerkes sich die beiden Teilnehmer befinden, oder wieviele Knoten die Nachricht auf der Route zwischen Quelle und Ziel traversieren muss.

Die Realisierung des Chat-Nachrichtendienstes für die *UbiMuC*-Anwendung wurde in insgesamt zwei Schritten durchgeführt: Zu Anfang wurde ein Konzept für das Chat-Modul erstellt, auf dessen Basis später die Implementierung durch Klassen und GUI-Methoden folgte. Innerhalb des Konzepts wurden die Spezifikation und die Rahmenbedingungen erfasst sowie die eigentliche Zielsetzung für das Chat-Modul.

4.4.1 Konzepte und Spezifikationen

Rahmenbedingungen

Um gezielt Nachrichten innerhalb des *UbiMuC*-Netzwerkes austauschen zu können, muss eine Differenzierung zwischen den einzelnen *UbiMuC*-Teilnehmern vorgenommen werden können. Das Chat-Modul benötigt also eine Möglichkeit, die im Netzwerk vorhandenen Benutzer von *UbiMuC* auseinander zu halten, um so die Nachrichten gezielt behandeln zu können. Diese Identifikation und Unterscheidung zwischen den verschiedenen Benutzern ist durch deren eindeutige Benutzeradressen und die Funktionen der Buddyliste gewährleistet, wie sie in Abschnitt 4.3 angesprochen wurde. Dort kann ein *UbiMuC*-Benutzer anhand seines Aliasnamens und dessen eindeutigem Hashwert erkannt werden, so dass sich zu ihm eine Chat-Sitzung aufbauen lässt.

Aufbauend auf der Netzwerkebene durch den Paketdienst (bekannt aus Abschnitt 4.2.2.2) benötigt das Chat-Modul also nur die eigentlich zu übermittelnde Nachricht und die Adresse des Zielpeers in Form seiner *UbiMuC*-Adresse.

Folgen der Architektur

Aufgrund der dezentralen Netzwerkarchitektur und dem flüchtigen Charakter des Ad-Hoc-Netzwerkes kann eine Chat-Sitzung ausschließlich zu Benutzern aufgebaut werden, die in der zugrundeliegenden DHT-Struktur (vergleiche Abschnitt 4.3.1) als „online“ verzeichnet sind. Ist ein Benutzer nicht online, liegt für ihn keine gültige Adresse vor, so dass an ihn gesendete Nachrichten niemals ihr Ziel erreichen würden. Da es im Vergleich zu anderen Instant-Messenger-Programmen wie ICQ bei *UbiMuC* strukturbedingt keinen zentralen Anmelde- und Pufferserver gibt, können keine Offline-Nachrichten für die Benutzer hinterlassen werden. Das

Chat-Modul ist demnach so entworfen worden, dass die Sitzungen nur dann aufgebaut werden, wenn eine gültige Adresse für den Zielteilnehmer vorliegt und ein Nachrichtenaustausch überhaupt möglich ist. Was die Zustellung und konkrete Bearbeitung (Senden und Empfangen von Daten) betrifft, baut das Chat-Modul vollständig auf dem Paketdienst auf.

Integration in die GUI

Der Chat ist über einen eigenen Tab in der GUI erreichbar und lässt sich außerdem durch Anwählen von Freunden aus der Buddyliste durch eine Chat-Funktion direkt starten. Die verschiedenen Chat-Sitzungen sind für den Benutzer in Form von Karteikarten separiert, wobei die Reiter mit den jeweiligen Aliasnamen zur besseren Übersicht beschriftet sind.

In einem Tab ist ein Textfenster enthalten, welches die gesendeten und empfangenen Nachrichten anzeigt. Die effektive Abgrenzung der einzelnen Chat-Tabs untereinander wird über die *UbiMuC*-Adressen der einzelnen Buddies sichergestellt, so dass durchaus zwei Tabs mit gleicher namentlicher Beschriftung angezeigt werden können, welche aufgrund der abweichenden Adressen jedoch zu verschiedenen Benutzern gehören.

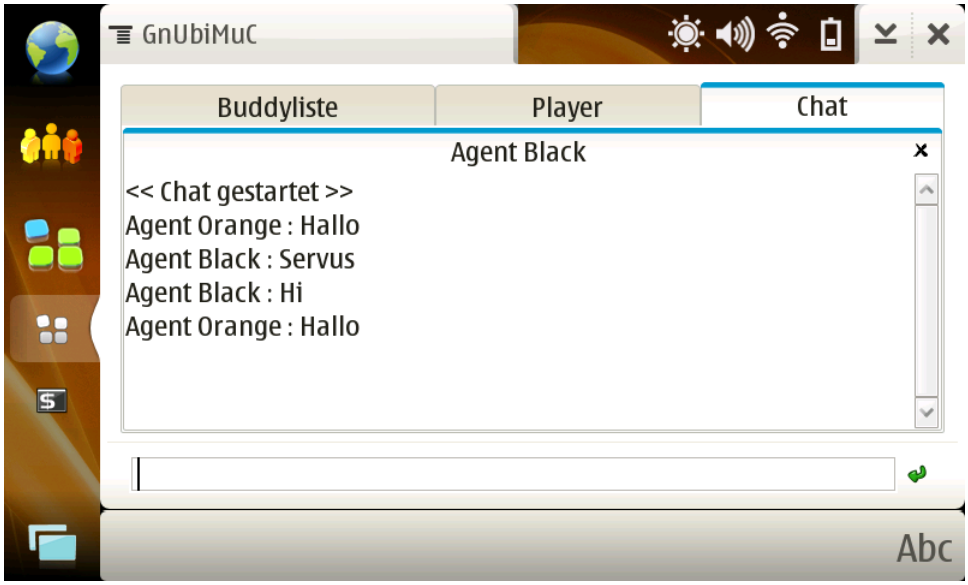

Abbildung 4.11: Exemplarische Chat-Sitzung

Diese Tabs werden entweder durch Aktionen des Benutzers erstellt oder extern durch den Empfang einer Chat-Nachricht angestoßen. Die Tabs können zu beliebigen Zeitpunkten geschlossen werden, wobei dann alle dargestellten Informationen verloren gehen. Es findet keine automatische Speicherung der Sitzungen statt und es ist auch nicht vorgesehen, dass Sitzungen manuell durch den Benutzer gespeichert werden können. Die Abbildung 4.11 zeigt einen Screenshot im laufenden Betrieb, wobei hier eine einzige Chat-Sitzung exemplarisch dargestellt wurde.

Wie bereits erwähnt und in der Abbildung 4.11 visualisiert, sind die erstellten Chat-Tabs mit dem Aliasnamen des Nutzers beschriftet, mit dem die Chat-Sitzung aufgebaut wurde.

Erhält ein Nutzer erstmalig eine Nachricht von einer unbekannten Person, so können zwei Fälle auftreten: Der Absender ist in der DHT-Struktur verzeichnet und der Empfänger kann dessen Adresse, Hashwert und Aliasnamen mittels der DHT-Schnittstelle abfragen und in Erfahrung bringen. Sofern dies möglich ist, wird das Chat-Tab normal aufgebaut und mit dem erhaltenen Aliasnamen gekennzeichnet. Die empfangene Nachricht wird anschließend im Textfenster angezeigt.

Ist über die Buddyliste anfänglich kein Aliasname bestimmbar, wird dies durch die Beschriftung „Unbekannte Person“ hervorgehoben. Falls auch keine Antwort von der DHT eintrifft, bzw. die Suche erfolglos endet und demnach kein korrespondierendes *GNUnet*-HELLO vorliegt, wird die Nachricht verworfen. Dies hat den Grund, dass der Empfänger der Nachricht ohne DHT-Eintrag und *GNUnet*-HELLO nicht auf den ihm zugesandten Text antworten kann. Nähere Details zu den Gründen wurden in den Abschnitten 3.1.3.1 sowie 3.1.3.2 behandelt.

Darstellung der Textnachrichten

Inhaltlich werden die Nachrichten innerhalb des Textfensters in der jeweiligen zeitlichen Reihenfolge dargestellt, auf optische Hervorhebungen oder Zeitstempel ist aufgrund der eingeschränkten Displaygröße des N810 verzichtet worden. Zur besseren Übersicht bei länger andauernden Chat-Sitzungen verfügt jeder Tab über eine vertikale Scrollbar, und die Texte werden automatisch an die vorliegende Fensterbreite innerhalb des N810-Displays angepasst. Ein Zeilenumbruch wird automatisch vorgenommen, wenn der darzustellende Inhalt eine vordefinierte Länge überschreitet. Dabei spielt es keine Rolle, wie lang der Text effektiv ist, da der Text so oft wie nötig umgebrochen wird.

4.4.2 Implementierung

Um eine Chat-Sitzung so allgemein wie möglich zu halten, wurde der Konstruktor der Chat-Klasse nur mit den zwei Parametern „networklayer“ und „wir_buddy_entry“ versehen. Der networklayer stellt dabei die Verbindung zur Netzwerkschicht (dem Paketdienst) und dem dahinterliegenden *GNUnet*-Client her. Der hinter dem „networklayer“ gekapselte Paketdienst sorgt automatisch für den Aufbau einer Route und für die Versendung der Datenpakete, so dass lediglich dessen send-Methode zum Datenaustausch mit anderen *UbiMuC*-Benutzern verwendet werden muss.

Die Struktur „wir_buddy_entry“ impliziert im weiteren Verlauf die Unterscheidung der einzelnen Nutzer bei mehreren offenen Chat-Sitzungen anhand der dort hinterlegten eindeutigen Adressen auf der Basis von Hashwerten. Innerhalb eines Objekts der Klasse „wir_buddy_entry“ (basierend auf Abschnitt 4.3) ist unter anderem der vergebene Aliasname des Benutzers gespeichert, sowie die interne *UbiMuC*-Adresse und der aktuelle Online-Status.

Format von Chat-Nachrichten

Um Chat-Nachrichten abzusenden wird ein Datencontainer vom Typ „Text“ benötigt, dessen Inhalt aus der zu übermittelnden Nachricht des Benutzers besteht. Zusätzlich muss der Text-Nachricht noch der Hashwert des Absenders vorangestellt werden. Dies ist nötig, damit auf der

Empfängerseite mehrere Chat-Sitzungen unterschieden werden können. Da eine eintreffende Nachricht keinerlei weitergehenden Informationen über deren Ursprung oder Quelle erhält, war dieser Schritt zur eindeutigen Identifikation und Abgrenzung der Sitzungen notwendig.

Senden von Chat-Nachrichten

Da ein Objekt der Chat-Klasse immer zwingend mit dem Paketdienst und einem Eintrag aus der Buddyliste beziehungsweise der DHT verbunden ist, können Chat-Nachrichten unmittelbar vom erstellten Chat-Objekt durch Aufruf der send-Methode abgeschickt werden. Das eigentliche Übermitteln der Textnachricht wird dabei vom Paketdienst übernommen, der mit einem vom Chat-Objekt erstellten Datencontainer des Typs „Text“ versorgt wird und die nötigen Inhalte in Form von Nutzdaten enthält.

Empfang von Chat-Nachrichten

Auf der Empfangsseite sorgt ein eigener Listener-Thread dafür, dass die Datenpakete mit Text-Flag entnommen und weiterverarbeitet werden können. Innerhalb des Threads werden die Nutzdaten aus den erhaltenen Datencontainern entnommen, außerdem wird die Absenderadresse zur Herstellung des Kontextes extrahiert. Anhand der Absenderadresse erfolgt die Zuordnung zu bereits bestehenden Chat-Sitzungen. Wurde das Chat-Tab zwischenzeitlich geschlossen oder existierte noch nicht, sorgt die GUI automatisch für die Erzeugung und Darstellung des Nachrichteninhalts durch ein neues Tab.

Alle Chat-Tabs aus der GUI werden in Form von einer gesonderten Mapping-Struktur den *UbiMuC*-Adressen zugeordnet, so dass die eintreffenden Nachrichten eindeutig den offenen Chat-Tabs zugeordnet werden können. Das manuelle Schließen eines Tabs sorgt abschließend für die Zerstörung des lokalen Chat-Objekts sowie die Entfernung des zum Tab korrespondierenden Eintrags innerhalb der Mapping-Struktur. Das lokale Schließen des Tabs hat dabei keine Auswirkungen auf die Objekte und Strukturen des entfernten Chat-Partners.

Abgleich zu den Anforderungen

Die einleitend erwähnten Anforderungen an das Chat-Modul sind erfüllt worden. Das Modul gewährleistet die komplette Kapselung von Sende- und Empfangsdetails und stellt den *UbiMuC*-Benutzern eine einfach zu verwendende Schnittstelle zum Austausch von Textnachrichten bereit. Vor der Erstellung einer Chat-Sitzung wird sichergestellt, dass der Zielteilnehmer erreichbar ist, damit eine Kommunikation überhaupt möglich ist.

Beim Empfang von Nachrichten wird gewährleistet, dass diese den korrespondierenden GUI-Tabs zugeordnet werden, damit es nicht zu fälschlichen Anzeigen kommt. Aufgrund der sehr komfortablen Schnittstelle des Paketdienstes ist es grundsätzlich kein Problem, auch andere, nicht-Chat-Nachrichten, zu versenden. Das hier benutzte Nachrichtenformat wird vom Paketdienst lediglich durch ein Text-Typflag von anderen Datenarten (wie beispielsweise den Multimediadaten des folgenden Abschnitts) unterschieden.

4.5 Multimedia

Multimedia-Funktionalität bildet einen Kernpunkt des Projektes. Dieser Abschnitt beschäftigt sich zunächst mit Konzepten und Spezifikationen. Hier werden das genutzte Framework sowie genutzte Codecs und Protokolle vorgestellt. Es folgen Implementierungsdetails einzelner Komponenten, sowie besondere Eigenheiten und aufgetretene Probleme.

4.5.1 Konzepte und Spezifikationen

Vor der Implementierung mussten einige wesentliche Fragen geklärt werden:

- Sollte ein integrierbares Programm oder ein Multimedia-Framework verwendet werden?
- Welche Funktionen muss diese gewählte Software unterstützen?
- Welche Codecs und Protokolle sollten benutzt werden?
- Wie sollten bestimmte Konzepte wie Verbindungsaufbau und -abbau umgesetzt werden?

Diese Fragen werden im Folgenden genauer untersucht und mögliche Lösungen beschrieben.

4.5.1.1 *GStreamer*

Zur Nutzung von multimedialen Komponenten in *UbiMuC* war es notwendig, ein geeignetes, integrierbares Programm oder Multimedia-Framework zu finden. Mit dem unter LGPL entwickelten *GStreamer* [12] fand sich ein derartiges Framework, welches nativ auf der *Maemo*-Plattform des N810 unterstützt wird.

GStreamer selbst verfolgt ein recht simples Konzept: Dieses Framework besteht aus modularen Komponenten, wie Multimediaquellen und -senken, Codecs und Filtern. Diese können in so genannten Pipelines zusammengesetzt werden. Die Modularität des Frameworks erlaubt eine leichte Erweiterbarkeit durch externe Komponenten, die als weitere Plugins hinzugefügt werden können. Als Sourcen werden Protokolle, wie TCP und UDP, aber auch Hardware über Standard-Schnittstellen, wie *Video4Linux* [13] und ALSA [14], unterstützt.

ALSA ist eine Schnittstelle für Audio-Hardware, *Video4Linux* eine Schnittstelle für Video-Hardware, wie TV-Karten oder Webcams. Die auf dem N810 verbaute Webcam unterstützt *Video4Linux* und kann somit durch das *GStreamer*-Plugin angesprochen werden. ALSA hingegen wird nicht verwendet, die Audio-Hardware ist jedoch über spezielle Audio-Sinks und -Sourcen ansprechbar. Diese Plugins sind teilweise DSP-gestützt. Durch ihren Einsatz ergeben sich erhebliche Performanzvorteile zur Laufzeit.

Heutzutage ist *GStreamer* schon die Grundlage einiger Multimedia-Anwendungen und essenzieller Systembestandteil vieler Linux-Distributionen, da es in der weit verbreiteten Desktopumgebung GNOME [15] verwendet wird. Auch das auf dem N810 verwendete *Debian*-Derivat setzt *GStreamer* ein. Zudem stellt *GStreamer* APIs für eine Vielzahl von Programmiersprachen, insbesondere C und C++, zur Verfügung.

Letztendlich waren die DSP-Unterstützung und die native Anbindung der Audio-Hardware an *GStreamer* der Anlass, diesen anstelle des zuerst favorisierten *MPlayers* zu verwenden.

4.5.1.2 Genutzte Codecs

Zur effektiven Übertragung von Multimediadaten ist eine Kompression dieser Daten nötig. Doch gerade bei eingebetteten Systemen stellt dies ein Problem dar. Die Rechenleistung ist sehr begrenzt, so dass nur ressourcensparende Codecs verwendet werden sollten. Speziell bei Streaminganwendungen sollte die Kompression möglichst zeiteffizient stattfinden. Somit mussten Codecs für die Bild- bzw. Tonübertragung in *UbiMuC* gewählt werden. Hierbei wurden die mitgelieferten Codecs untersucht und getestet.

Testumgebung

Zunächst wurde getestet, ob mittels *GStreamer* eine performante Multimedia-Pipeline erzeugt werden kann. Dazu wurde *gst-launch*, ein mitgeliefertes Kommandozeilen-Tool, verwendet. Bei den Tests wurden zwei N810 eingesetzt.

Abbildung 4.12: Aufbau der Testumgebung

Dabei wurde auf dem einen Gerät eine Sendepipeline per *gst-launch* realisiert, die auf einen lokalen UDP-Port leitete. Als Quellen dienten das Mikrofon und die ebenfalls im Gerät verbaute Webcam.

Auf dem Empfangsgerät arbeitete *gst-launch* als Receiver-Pipeline. Diese Pipeline musste so konstruiert werden, dass eine Wiedergabe des gesendeten Streams möglich war. Die Wiedergabe selbst erfolgte in diesem Test über zugehörige Multimedia-Senken. Videodaten wurden über die „xvimagesink“ wiedergegeben. Diese Senke stellt die Daten auf dem Display in einem separaten, neu erzeugten Fenster dar. Audiodaten wurden in die Audiosink weitergeleitet, welche den jeweils getesteten Codec DSP-gestützt verarbeiten kann.

Eine weitere Instanz simulierte das zwischenliegende Netzwerk, indem sie die Datenpakete vom durch die Sendepipeline bedienten Port des sendenden Gerätes auf einen Port des Empfangsgerätes weiterleitete. Diese Instanz simulierte somit die *UbiMuC*-Transportschicht, wie sie im Abschnitt 4.2 beschrieben wird.

Im Folgenden werden die überprüften Codecs sowie die Ergebnisse der durchgeführten Tests vorgestellt. Als Vergleichskriterien dienen die von ihnen erzeugte Systemlast und die Qualität der wiedergegebenen Streams.

Video-Codecs

Es wurden zwei Video-Codecs für den Einsatz in *UbiMuC* getestet. Der *Smoke*-Codec ist ein JPEG-basierender Video-Codec. Er steht unter GPL und ist Teil des Flumotion-Projektes [16] Die Pipelines dieses Codecs wurden für die Tests wie folgt (siehe Abbildung 4.13) umgesetzt:

```
Sende-Pipeline:
   gst-launch
      v4l2src !
      video/x-raw-yuv,width=176,height=144,framerate=\(fraction\) 8/1 !
      ffmpegcolorspace !
      smokeenc keyframe=8 qmax=40 !
      udpsink host=192.168.178.27 port=5000

Empfangs-Pipeline:
   gst-launch
      udpsrc port=5001 !
      smokedec !
      xvimagesink
```

Abbildung 4.13: Sende- und Empfangspipeline mit *Smoke*-Codecs

Der zweite Enkoder ist der proprietäre *Hantro4200* [17] von On2 Technologies Inc. Es handelt sich um einen MPEG4-Codec für ARM-basierende Systeme, der verschiedene Profile unterstützt. Auch der Standard *H.263* wird von ihm unterstützt, ein Standard, der vor allem für Videokonferenzen ausgelegt ist.

Als Dekoder wird der *Hantro4100* [18] von On2 Technologies Inc. verwendet. Der oben beschriebene Testaufbau wurde durch folgende Kommandozeilen (Abbildung 4.14) umgesetzt:

```
Sende-Pipeline:
   gst-launch
      v4l2src !
      video/x-raw-yuv,width=176,height=144,framerate=\(fraction\) 8/1 !
      hantro4200enc !
      rtph263pay !
      udpsink host=192.168.178.27 port=5000

Empfangs-Pipeline:
   gst-launch
      udpsrc port=5000 caps="application/x-rtp,clock-rate=90000" !
      rtph263depay !
      hantro4100dec !
      xvimagesink
```

Abbildung 4.14: Sende- und Empfangspipeline mit *Hantro*-Codecs

Die Systemauslastung beider untersuchter Video-Codecs im Testsystem wurde durch das Linux-System-Werkzeug *top* ermittelt. Tabelle 4.5 veranschaulicht CPU- und Speicherlast der beiden Codecs.

Die Werte beider Codecs ähneln sich. Im Vergleich zum *Hantro*-Codec erzeugte der *Smoke*-Codec jedoch eine etwas geringere Systemlast. Allerdings war die Bildqualität beim *Hantro*-Codec eindeutig besser, da beim *Smoke*-Codec kein Motiv im Bild zu erkennen war.

Da die Auslastung des Gerätes keinen der beiden Codecs klar herausheben konnte und die Wiedergabe des Bildes so gravierende Unterschiede aufwies, findet der *Hantro4200*-Codec Verwendung in *UbiMuC*.

	Codec	VIRT	RES	SHR	%CPU	%MEM	COMMAND
Sender:	smokeenc	22784	3072	2164	14.9	2.4	gst-launch
	hantro4200enc	23432	3316	2256	17.8	2.6	gst-launch
Empfänger:	smokedec	31716	3140	2272	2.9	2.5	gst-launch
	hantro4100dec	32160	3344	2436	3.3	2.6	gst-launch

Tabelle 4.5: Auswertung der Video-Codec-Tests

Audio-Codecs

Wie auch bei den Video-Codecs bietet das N810 standardmässig mehrere Audio-Codecs an. Bei einigen Codecs bietet der verbaute DSP sogar hardwareseitige Unterstützung an, die in Form von *GStreamer*-Plugins nutzbar gemacht wird.

So können beispielsweise MP3s zwar hardwareseitig dekodiert, aber nicht enkodiert werden. Deshalb wurde dieses Format nicht weiter betrachtet.

Da gerade Multimedia-Anwendungen viel Rechenleistung benötigen, wurde versucht, hardwaregestützte Codecs zu verwenden. Namentlich sind dies die auf Komprimierung von Sprache optimierten Standards *G.711* und *G.729*.

```
Sende-Pipeline:
   gst-launch
      dspg729src dtx=3 !
      rtpg729pay  !
      udpsink host=192.168.178.27 port=5000 sync=fals

Empfangs-Pipeline:
   gst-launch
      -v udpsrc port=5000 caps="application/x-rtp,clock-rate=(int)8000,
                encoding-name=(string)G729" !
      rtpg729depay !
      dspg729sink
```

Abbildung 4.15: Sende- und Empfangspipeline mit G.729

Der Standard *G.711*, wie er von der Internationalen Fernmeldeunion (ITU) [19] verabschiedet wurde, beschreibt die Analog-Digital-Wandlung von Sprachsignalen per Puls-Code-Modulation

(PCM). Dieses Verfahren kommt beispielsweise bei der ISDN-Telefonie zum Einsatz. Für das N810 existiert ein DSP-gestützter *GStreamer*-Codec, der diesen Standard implementiert.

Auch der Standard *G.729* wurde von der ITU [20] verabschiedet. Er beschreibt die Kodierung von Sprache per CS-ACELP („conjugate-structure algebraic-code-excited linear prediction"). Dieses Verfahren wird vor allem bei Voice-over-IP-Telefonie verwendet. Auch hierfür existiert ein DSP-gestützter *GStreamer*-Codec für das N810. Die Realisierung dieser Pipeline wurde wie in Abbildung 4.15 umgesetzt.

Zum Zeitpunkt der Audio-Codec-Tests stand bereits fest, dass RTP als Protokoll der Multimedia-Pipeline eingesetzt wird. Da im Gegensatz zum *G.729*-Codec kein RTP-(De-)Payloader für den *G.711*-Codec auf dem Gerät existiert, wurde nur noch der *G.729*-Codec betrachtet und auf Effizienz überprüft. Analog zu den Video-Codecs erfolgte ein Test der Systemauslastung. Auch hier wurde die Auslastung durch das Linux-System-Werkzeug *top* ermittelt. Da CPU- und Speicherlast sehr gering waren und die Qualität auf Empfängerseite mehr als zufriedenstellend war, gab es keine Notwendigkeit, weitere Codecs zu testen.

Ergebnisse

Nach den beschriebenen unidirektionalen Tests wurde mit den gewählten Codecs (*Hantro4200*, *Hantro4100* und *G.729*) ein bidirektionaler Test durchgeführt. Beide Geräte agierten sowohl als Sender als auch als Empfänger. Zudem wurden gleichzeitig Audio- und Video-Pipelines verwendet. Da auch hier keine weiteren Performanzprobleme auftraten und die wiedergegebenen Streams eine annehmbare Qualität erreichten, wurden diese Codecs für *UbiMuC* festgelegt.

4.5.1.3 Genutzte Protokolle

Für die Übertragung der Nutzdaten ist es einfacher, auf ein bestehendes Protokoll aufzusetzen, als von Grund auf ein Neues zu implementieren. Insbesondere macht es Sinn, ein Protokoll zu verwenden, welches durch das Multimedia-Framework direkt unterstützt wird. Hierzu werden gewisse Anforderungen an das Protokoll gestellt:

So ist es beispielsweise nicht notwendig, dass bei AV-Streams alle Datenpakete empfangen werden. Allerdings ist dafür deren Reihenfolge wichtig. Würde auf ein Datenpaket gewartet werden müssen, erhöht sich die Latenz zwischen dem Senden und der Wiedergabe des Datenstreams. Das *User Datagram Protocol* (UDP) bietet diese Funktionalitäten und wird auch weitestgehend in Streaminganwendungen für ähnliche Zwecke eingesetzt. Zudem unterstützt das in *UbiMuC* verwendete Multimedia-Framework *GStreamer* dieses Protokoll.

Des Weiteren ist sicherzustellen, dass die empfangenen Pakete dekodierbar sind. Dies gewährleistet in *UbiMuC* das *Realtime Transport Protocol* (RTP). Sowohl für den verwendeten Video-Codec *Hantro/H.263*, als auch für den verwendeten Audio-Codec *G.729* liefert das Framework *GStreamer* direkt geeignete RTP-Payloader, die Nutzdaten in Pakete entsprechender Größe packen. Diese Pakete werden auf der Empfangsseite problemlos identifiziert und durch den ebenfalls mitgelieferten RTP-Depayloader entpackt. Die Nutzdaten können danach weiterverarbeitet werden.

Demnach werden RTP-Pakete erzeugt, die mittels UDP auf einen gewählten Port des lokalen Systems gelenkt werden. An diesem Port können sie durch einen geeigneten Mechanismus, wie beispielsweise Sockets (siehe 4.2.2.3), gelesen und zum eigentlichen Empfänger weitergeleitet werden. Dieser Mechanismus lässt sich variabel ersetzen, beispielsweise durch einen *GNUnet*-Dienst. Somit wurde die folgende Pipelinestruktur in *UbiMuC* realisiert (Abbildung 4.16)

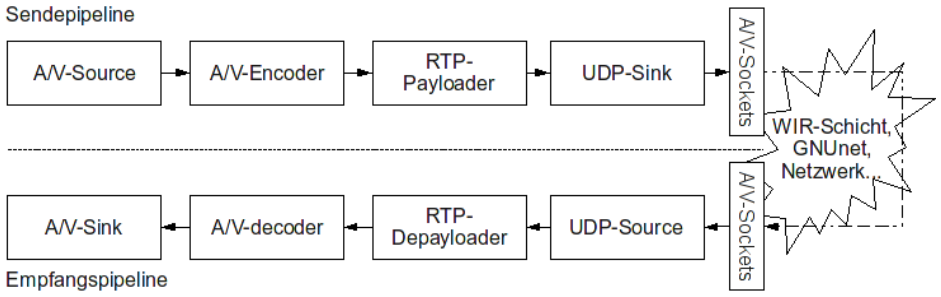

Abbildung 4.16: Sende- und Empfangs-Multimediapipeline in *UbiMuC*

4.5.1.4 Auf- und Abbau einer Konferenz

In *UbiMuC* soll es auch die Möglichkeit einer Audio-/Video-Konferenz zwischen verschiedenen Nutzern geben. Derartige Konferenzen sind bekannt von Programmen wie zum Beispiel *Skype*. Letztlich handelt es sich dabei um eine Form der Videotelefonie. Diese dient in *UbiMuC* dazu, zu zeigen, dass es nicht nur möglich ist, textuell mit anderen Nutzern zu kommunizieren, wie im vorhergehenden Kapitel 4.4 beschrieben, sondern dies auch anderweitig ermöglicht wird.

Um eine Konferenz zwischen zwei Teilnehmern zu starten, müssen einige Voraussetzungen erfüllt werden: So ist es zunächst notwendig, dass beide online sind und sich derzeitig in keiner anderen Konferenz befinden. Nun ist es problematisch, wenn von einer Seite eine Konferenz initiiert wird und diese auf der Empfangsseite nicht abgelehnt werden kann, so dass diese Konferenz bei Initiierung erzwungenermaßen gestartet wird. Daher ist ein Protokoll unumgänglich, das dem Empfänger die beiden Auswahlmöglichkeiten bietet, diese Konferenz zu akzeptieren oder abzulehnen. Dieses Protokoll kann zudem technische Abläufe, wie Einschalten des Mikrofons oder der Kamera, zur Durchführung einer Konferenz anstoßen.

Ein weiteres Problem existiert, wenn ein Partner die Konferenz verlässt und der andere Teilnehmer weiterhin sendet: Datenpakete fluten ungenutzt das Netzwerk und erreichen den Empfänger, ohne durch diesen verarbeitet zu werden. Unter Umständen kann dies zu einem Absturz des Empfangsclients führen. Zudem sollte eine Konferenz nur von ihren Teilnehmern beendet werden dürfen. Deshalb ist neben dem Protokoll für den Verbindungsaufbau ebenfalls ein Protokoll für den Verbindungsabbau nötig. Auch hier können notwendige Vorgänge, wie Abschalten der Kamera und des Mikrofons, eingebunden werden.

All diese Funktionen müssen beim Verbindungsaufbau bzw. -abbau gewährleistet werden. Im Folgenden sollen die hierzu bei *UbiMuC* zum Einsatz kommenden Konzepte näher beschrieben werden und auf die Umsetzungen von selbigen eingegangen werden.

Protokoll des Verbindungsaufbaus

Dem Konzept für den Verbindungsaufbau, wie es in Abbildung 4.17 schematisch dargestellt wird, liegt ein simples 3-Wege-Handshake zu Grunde, und es ist auf ein einfaches Szenario ausgerichtet. Es ist darauf ausgelegt, mit möglichst wenigen verschiedenen Anfragen eine Verbindung aufzubauen, um so die Menge an benötigten Paketen und Funktionen im Handler einfach zu halten. Es wird davon ausgegangen, dass immer nur eine Audio-/Video-Konferenz mit genau einem Gesprächspartner existiert. So ist es weder möglich, zwei verschiedene Konferenzen gleichzeitig zu betreiben, noch eine Konferenz unter mehr als zwei Nutzern zu initiieren.

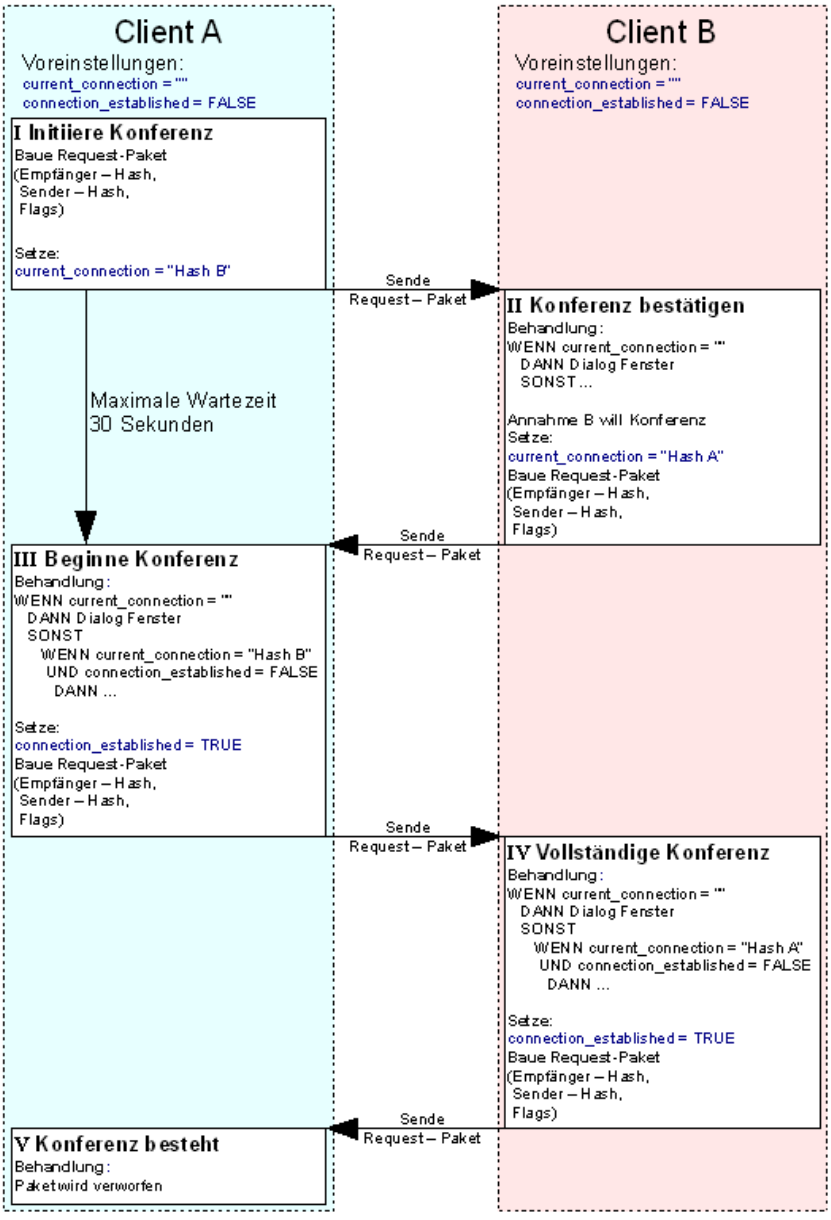

Abbildung 4.17: Handshake beim Verbindungsaufbau einer Audio-/Video-Konferenz

Diese Beschränkungen wurden zum einen eingeführt, um den Verbindungsaufbau nicht zu verkomplizieren, zum anderen, um die vorhandene Hardwareplattform nicht zu überlasten, da die Auslastung des N810 auch bei nur einer einzigen Verbindung schon recht hoch ist. Generell arbeitet der Verbindungsaufbau in der Art, dass Nachrichten nur dann beantwortet werden, wenn das eigene System gerade in einem Zustand ist, der einen Verbindungsaufbau zulässt. Das Abweisen von Verbindungsanfragen oder von Paketen, die momentan nicht erwartet werden, geschieht nur implizit, das heißt, sie werden nicht beantwortet, sondern einfach ignoriert.

Generell wird davon ausgegangen, dass bisher keine aktive Verbindung besteht und bisher auch keine Anfrage abgeschickt wurde. Der aktuelle Status des Clients wird mit zwei Variablen gespeichert. Zum einen wird in der Variable „connection_established“ festgehalten, ob man momentan in einer vollständig etablierten Konferenz ist. Sie ist mit FALSE vorinitialisiert und sollte nur dann, wenn gerade eine Verbindung aktiv ist, auf TRUE gesetzt werden.

In der Variablen „current_connection“ wird der *UbiMuC*-Hash des Nutzers festgehalten, mit dem man gerade versucht eine Verbindung aufzubauen, oder mit dem man bereits eine Konferenz hat. Dies ist nötig, um sicherzugehen, dass die Pakete dieses Nutzers nicht einfach verworfen werden. Hier ist nur dann ein Wert eingetragen, wenn gerade eine Verbindung aufgebaut werden soll oder eine Verbindung besteht. Ist dies nicht der Fall, muss dieser String leer sein, um sicherzugehen, dass eine neue Verbindung aufgebaut werden kann.

Generell wird fast alles mit einem simplen Request-Paket abgearbeitet. Dieses besteht dabei aus dem *UbiMuC*-Hash des Empfängers, dem eigenen Hash sowie optionalen Flags. Um einen funktionsfähigen Handshake zu ermöglichen, ist des Weiteren ein Listener nötig, der eingehende Request-Pakete passend behandelt.

Im Folgenden werden nun die einzelnen Schritte des Handshake für den Verbindungsaufbau beschrieben. Dabei steht „Client A“ für jenen, der die Konferenz initiiert und „Client B“ für denjenigen, der die Verbindung annimmt.

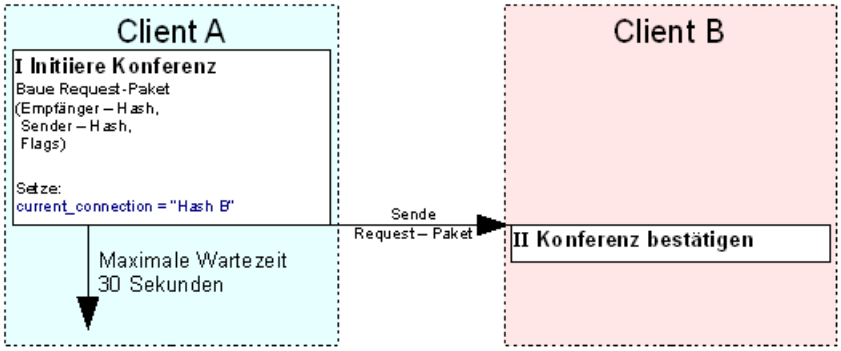

Abbildung 4.18: Ablauf beim Aufbau einer AV-Konferenz (Schritt 1)

Schritt 1:
Von Client A wird, wie in Abbildung 4.18 zu sehen ist, ein Request-Paket erstellt und mit unserem Datencontainer als „AV-Konferenz-Paket“ an den gewählten Konferenzteilnehmer Client B geschickt. Außerdem wird bei Client A der *UbiMuC*-Hash von Client B in der Variablen „current_connection“ vermerkt. Nun beginnt ein 30 Sekunden langes Zeitfenster, in dem auf

eine Antwort von Client B gewartet wird. Letztlich wird beim Warten durchgehend geprüft, ob „connection_established“ auf TRUE gesetzt ist. Geht innerhalb der festgelegten Wartezeit keine Antwort ein, so wird die Variable „current_connection“ wieder zurückgesetzt und alles erscheint für den Pakethandler so, als ob nie eine Anfrage verschickt wurde.

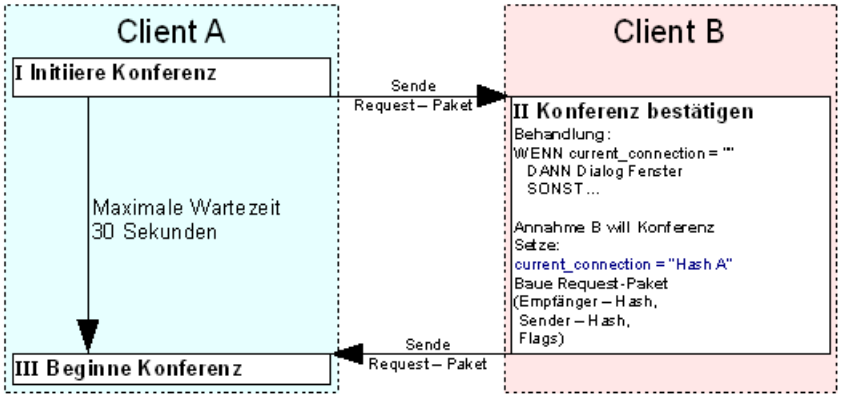

Abbildung 4.19: Ablauf beim Aufbau einer AV-Konferenz (Schritt 2)

Schritt 2:
Der Pakethandler bei Client B stellt ein eingehendes Paket von Client A fest und sieht anhand der eigenen Variablen, dass momentan keine Verbindung besteht. Der Nutzer wird nun per Dialog-Fenster gefragt, ob eine Verbindung zu Client A aufgebaut werden darf. Stimmt der Nutzer dem zu, wird ein eigenes Request-Paket gebaut und an Client B verschickt. Außerdem wird hier die Variable „current_connection“ auf den *UbiMuC*-Hash von Client A gesetzt. Dieser Schritt ist in Abbildung 4.19 visualisiert.

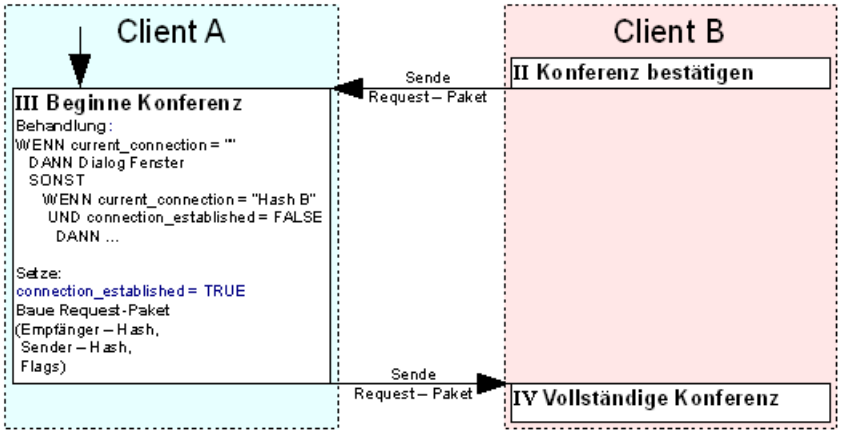

Abbildung 4.20: Ablauf beim Aufbau einer AV-Konferenz (Schritt 3)

Schritt 3:
Ist das Zeitfenster bei Client A noch nicht abgelaufen, so wird beim Eingehen des Requests von Client B „connection_established“ auf TRUE gesetzt und ein weiteres Request-Paket an Client B verschickt, um die Verbindung zu bestätigen. Nun werden, wie Abbildung 4.20 zu

entnehmen ist, die Ports für die Konferenz geöffnet und Daten von Kamera und Mikrofon ab jetzt an Client B verschickt. Sollte das Zeitfenster dagegen abgelaufen sein und entsprechend der Wert von „current_connection“ nicht dem Hashwert von Client B entsprechen, so wird hier ein Dialog-Fenster angezeigt, um zu bestätigen, dass eine Verbindung aufgebaut werden soll. Bei einer Bestätigung wird analog zu Schritt 2 fortgefahren.

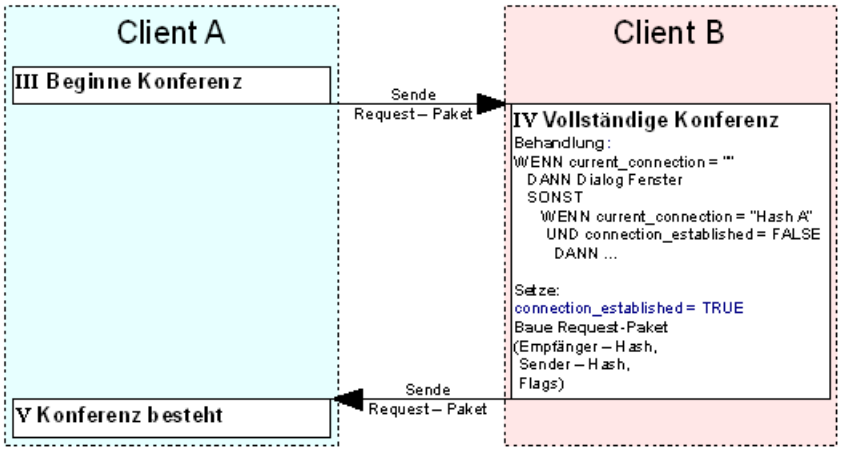

Abbildung 4.21: Ablauf beim Aufbau einer AV-Konferenz (Schritt 4)

Schritt 4:
Bei Eingang des bestätigenden Request-Paketes von Client B wird auch hier die Variable „connection_established“ auf TRUE gesetzt. Abbildung 4.21 veranschaulicht graphisch die Reihenfolge der Prüfungen. Ebenfalls werden die Ports für die Konferenz geöffnet und Daten von Kamera und Mikrofon verschickt.

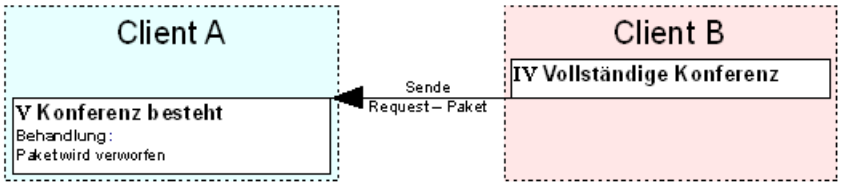

Abbildung 4.22: Ablauf beim Aufbau einer AV-Konferenz (Schritt 5)

Schritt 5:
Da der Handler sehr einfach gehalten ist und nur anhand des Zustands der beiden Variablen für den Handshake arbeitet, wird nun ein weiteres Request-Paket abgeschickt, welches dann allerdings vom Empfänger verworfen wird (siehe Abbildung 4.22).

Für den Verbindungsaufbau ist es letztlich nur relevant, welche Werte „current_connection“ und „connection_established“ haben, wenn eingehende „AV-Konferenz-Pakete“ festgestellt werden. Dabei wird immer wenn bei Eingang eines „AV-Konferenz-Pakets“ die Variable „current_connection“ Null ist der Nutzer gefragt, ob eine Verbindung zum anfragenden Nutzer aufgebaut werden soll. Wenn die Variable nicht Null ist, so wird geprüft, ob sie gleich dem

UbiMuC-Hash des Absenders des empfangenen „AV-Konferenz-Paketes“ ist, und ob außerdem die Verbindung noch nicht hergestellt wurde.

Ist dies der Fall, so werden die Schritte eingeleitet, die nötig sind, um die Ports zu öffnen und Daten zu versenden. Durch Setzen der Variablen „connection_established“ auf TRUE wird notiert, dass die Ports offen sind, um sicherzustellen, dass nicht mehrfach versucht wird, Ports zu öffnen. Sollte man sich in keinem der oben genannten Zustände befinden, so wird keine Audio-/Video-Konferenz eröffnet.

Gibt es keinen Eintrag im Handler für den aktuellen Zustand, so werden die eingehenden Pakete verworfen und es tritt keine Reaktion ein. Mit diesem simplen Mechanismus wird sichergestellt, dass es nicht zu Deadlocks beim Verbindungsaufbau kommen kann, wenn mehrere Nutzer nahezu gleichzeitig versuchen, Verbindungen zueinander aufzubauen.

Nach erfolgtem Aufbau einer Verbindung sind auf beiden Seiten die Ports geöffnet und Daten von Kamera und Mikrofon können ausgetauscht werden. Dieses einfache Schema ermöglicht es einen Handshake leicht zu realisieren, doch ist es schwierig, ihn so zu erweitern, dass Konferenzen mit mehr als zwei Teilnehmern oder mehrere Konferenzen gleichzeitig ermöglicht werden. Allerdings wird dieses in den Anforderungen von *UbiMuC* nicht gefordert und würde die vorhandene Hardwareplattform höchstwahrscheinlich überlasten.

Protokoll des Verbindungsabbaus

Der Verbindungsabbau ist im Vergleich zum Verbindungsaufbau noch einfacher gehalten. Es wird das gleiche Datenpaket wie beim Verbindungsaufbau verwendet. Dieses besteht aus dem Hashwert des Empfängers, dem Hashwert des Senders und Flags. Im Gegensatz zum Verbindungsaufbau sind hier nun die Flags gesetzt. So werden beim Verbindungsabbau keine neuen Datentypen verwendet, sondern nur die Datenformate des Verbindungsaufbaus wiederverwendet. Daher muss an dieser Stelle nur der Handler angepasst werden, damit die entsprechenden Datenpakete mit den gesetzten Flags korrekt behandelt werden.

Beendet ein Benutzer nun die Konferenz, an der er teilnimmt, so wird an den Konferenzpartner eben diese Nachricht geschickt. Der Sender schließt nun seine Empfangs- und Sende-Ports. Zudem werden sowohl die Kamera als auch das Mikrofon abgeschaltet.

Beim Eintreffen der Nachricht öffnet sich nach dem Überprüfen des Hashwertes des Absenders auf der Empfangsseite ein Dialog-Fenster. In diesem wird dem Benutzer mitgeteilt, dass der Konferenzpartner die Sitzung beendet hat.

Gleichzeitig wird derselbe Prozess auf der Empfänger-Seite angestoßen wie zuvor auf der Sender-Seite: Die Ports werden geschlossen, Kamera und Mikrofon in den Ausgangszustand versetzt. Das Abbauprotokoll gewährleistet die an den Verbindungsabbau gestellten Anforderungen.

Zusammenfassung

Die beiden vorgestellten Protokolle zum Verbindungsaufbau und -abbau sind einfach gehalten. Trotzdem garantieren sie aber alle Anforderungen, wie sie zuvor an die Konzepte der jeweiligen Protokolle gestellt wurden. Mit der Umsetzung dieses Gesamtkonzeptes wird, wie zu Anfang dieses Kapitels angesprochen, vor allem sichergestellt, dass die Kamera und das Mikrofon auf Empfängerseite nur dann aktiviert werden, wenn der Nutzer dies auch wünscht, was der Privatsphäre des Anwenders entgegen kommt und damit einen essenziellen Punkt von *UbiMuC* auch bei Nutzung von Audio-/Video-Konferenzen sichert.

4.5.2 Implementierung

Die Implementierung der Multimedia-Funktionalitäten erfolgte in mehreren Schritten. Nach der Evaluierung, welche Komponenten auf welche Weise verwendet werden sollten, wurden diese modular, zum Beispiel in Testprogrammen, auf Brauchbarkeit getestet.

Im Folgenden wurden diese in einem separaten Branch, in eine Kopie der bis dato erstellten Benutzeroberfläche, integriert. Hier ergaben sich einige Probleme, die vor der Wiedervereinigung mit dem Hauptentwicklungszweig beseitigt wurden. Implementierungsdetails der einzelnen Komponenten sowie die Beschreibung der aufgetretenen Probleme werden in diesem Abschnitt behandelt.

4.5.2.1 Audio-Implementierung

Im Gegensatz zu den Vorgängermodellen lassen sich beim N810 die Audio-Komponenten nicht durch die Linux-Audio-Standard-Schnittstelle *ALSA* ansprechen. Der DSP als Soundkarte muss demnach auf eine andere Weise angesteuert werden. Auch das Mikrofon ist hiervon betroffen. Der Zugriff ist hier allerdings über die mitgelieferten *GStreamer*-Plugins möglich, die explizit Ausgänge des DSP als Audioquellen zulassen.

Neben einem PCM-Signal bietet der DSP auch mehrere hardwareseitig enkodierte Streams an. Werden diese Streams genutzt, entlasten diese spürbar den Hauptprozessor des Systems, der in dem Fall keine softwareseitige mehr Enkodierung durchführen muss. Um dieses Potenzial zu nutzen, wird das G.729-Ausgangssignal des DSP genutzt.

Weiteres Optimierungspotenzial bieten die verschiedenen Konfigurationsmöglichkeiten des verwendeten *GStreamer*-Plugins. Eine dieser Konfigurationsoptionen erlaubt Datenpakete nur dann zu erzeugen, wenn ein gewisser Eingangspegel am Mikrofon anliegt. Mit diesem Wissen wurde eine effiziente Enkodierung des Mikrofonsignals in *UbiMuC* implementiert.

4.5.2.2 Kamera-Integration

Die im N810 verbaute Kamera unterstützt *Video4Linux2* (V4L2) und ist demnach über standardisierte Methoden ansprechbar. Auch hier verwendet *UbiMuC* das Multimedia-Framework *GStreamer*. Als Codec kommt der vorher beschriebene *Hantro 4200* zum Einsatz. Um eine optimale Performanz zu erreichen, wurden Abstriche bei Qualität und Darstellung in Kauf

genommen. So ist die Aufnahmeauflösung softwareseitig auf 176x144 Pixel begrenzt, während der Stream auf der Empfängerseite mit einer Auflösung von 352x288 Pixel wiedergegeben wird. Die Aufnahme selbst erfolgt mit acht Bildern pro Sekunde.

Als problematisch erwies sich die Integration in die Benutzeroberfläche. Referenzprogramme verbinden die Ausgabefläche mit der Ausgabesenke, sobald die Ausgabefläche sichtbar wird. Da Streams in *UbiMuC* nur gestartet werden können, wenn diese nicht angezeigt werden, war dieses Vorgehen nur über Umwege möglich: Beim Starten einer Konferenz wird noch vor Senden und Empfangen der Streams auf den Konferenz-Tab des Benutzerinterfaces mit dieser Ausgabefläche gewechselt.

Ein weiteres Problem bildete die Anzeige selbst: Wenn während der Wiedergabe eines Streams die Wiedergabefläche teilweise verdeckt wurde, war selbst nach dem Wiedereinblenden keine Anzeige auf den zuvor verdeckten Teilen dieser Anzeigefläche mehr möglich. Stattdessen blieben diese Flächen weiß. Eine Lösung brachte das Abschalten des Double-Bufferings für das Anzeigeelement.

Durch diese Implementierungen sind das Streamen eines Videobildes sowie das Anzeigen eines Streams in *UbiMuC* möglich.

5 Evaluierung

In diesem Kapitel werden die Tests dargestellt, die zur Bewertung der fertigen Software gefahren wurden. Dabei liegt der Schwerpunkt auf Testszenarien, die das Zusammenspiel aller fertig entwickelten Komponenten überprüft, die in den vorangegangen Kapiteln beschrieben wurden, indem reale Anwendungsfälle mit Hilfe der *UbiMuC*-GUI durchgespielt wurden. Insbesondere wurde dabei die ordnungsgemäße Interaktion zwischen der GUI, der WIR-Schicht und den *GNUnet*-Modulen verifiziert. Auf die autonomen Tests, mit denen die korrekte Funktionsweise der einzelnen Module geprüft wurde, wird hier nicht eingegangen, da auf diese bereits in Kapitel 4 in den zugehörigen Abschnitten eingegangen wurde.

UbiMuC Systemtest

Der Fokus lag hier auf der Kommunikation zwischen den Geräten und der echten Nutzung des Programms. Dazu wurden der Chat und die AV-Konferenz in verschiedenen Netzwerktopologien getestet und die Auslastung sowie die Qualität der AV-Konferenz dabei beobachtet.

Es wurden, wie in Abschnitt 4.2.3, die *iptables* genutzt, um sicherzustellen, dass die Geräte nur ihren Vorgänger und ihren Nachfolger in einer Kette sehen können. So wird ein echtes Multi-Hop-Netzwerk aufgebaut, bei dem die Kommunikation über mehrere Zwischenknoten zum Ziel gelangt. Es wurden drei unterschiedliche Testaufbauten genutzt.

Reihentest mit drei Geräten

Hier wurden drei Geräte genutzt, bei dem die Kommunikationsteilnehmer über einen Zwischenknoten kommunizieren konnten. Die AV-Konferenz wurde erwartungsgemäß gestartet, und auf beiden Geräten erschien nach einem kurzem Verbindungsaufbau das Multimediafenster. Beim Video fiel auf, dass das Bild wegen der Wahl eines Video-Codecs mit recht niedriger Auflösung ziemlich grobkörnig dargestellt wurde. Die Framerate des Videos war ebenfalls sehr gering, so dass es zu deutlichen Verzögerungen kam. Ähnlich sah es bei der Sprachqualität aus, bei der die Audioübertragung nicht sonderlich überzeugte. Die CPU-Auslastung des Gerätes lag während der Übertragung im Schnitt bei 70%, wobei 30% für *GNUnet* und 40% für die GUI anfielen.

Zusätzlich wurde getestet, die Tonübertragung abzuschalten. Dabei wurde eine merkliche Verbesserung der Qualität bei der Videoübertragung bemerkt. Umgekehrt hatte eine Abschaltung der Videoübertragung eine deutliche Verbesserung der Audioqualität zur Folge.

Zusätzlich wurde in dieser Konstellation der Chat überprüft. Alle Geräte konnten problemlos miteinander chatten, und die Nachrichten wurden sofort übertragen.

Reihentest mit fünf Geräten

Dieser Test erweiterte unseren bisherigen Aufbau um zwei weitere Geräte, zur Überprüfung der Skalierbarkeit des Konzepts. Ziel war es, eine AV-Konferenz zwischen zwei Endgeräten aufzubauen, deren Daten über drei Hops weitergeleitet werden. Wie beim vorherigen Test musste hierbei sichergestellt werden, dass für alle Geräte nur ihre direkten Nachbarn innerhalb des Netzes sichtbar sind. Nur so konnte die Kommunikation über eine lange Kette von Weiterleitungen sichergestellt werden.

Die Performanz war bei diesem Test genauso wie die vom vorherigen Test, die zusätzlichen Hops hatten hier zu keiner Verschlechterung der Qualität geführt. Die Verbindung blieb auch stabil, als die Geräte mit größerer Distanz zueinander aufgebaut wurden. Die Geräte wurden in verschiedenen Räumen auf einer Länge von über 20 Metern platziert.

Auch hier wurde der Chat getestet. Alle Geräte konnten problemlos miteinander chatten.

Kreuztest mit fünf Geräten

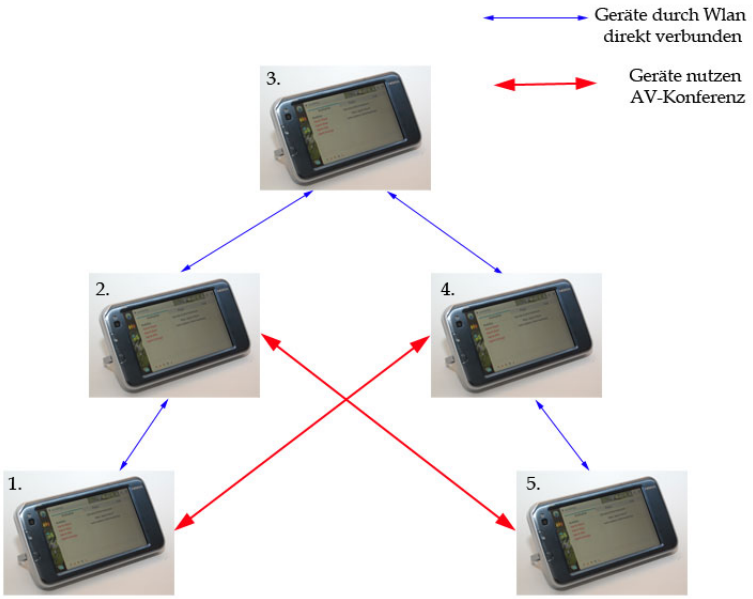

Abbildung 5.1: Aufbau des Kreuztestes mit fünf Geräten

Bei dem Kreuztest wurden, wie im vorherigen Aufbau, fünf Geräte so aufgebaut, dass sie nur mit ihrem Vorgänger und Nachfolger kommunizieren können. Dadurch ergibt sich eine Kette wie in Abbildung 5.1 dargestellt. Ziel des Aufbaus war es, zwei Verbindungen aufzubauen, um die Doppelbelastung als Konferenzteilnehmer und als Hop auf den Geräten zu testen. Dafür haben wir das Gerät Nr. 2 mit dem Gerät Nr. 4, sowie die Geräte Nr. 1 und Nr. 5 untereinander eine Konferenz aufbauen lassen. Dadurch ergab sich bei den Geräten Nr. 2 und 4 eine Doppelbelastung als Konferenzteilnehmer und als Hop, und über das Gerät Nr. 3 wurden zwei Konferenzen transportiert. Aus den Erfahrungen der vorherigen Tests wurde die Audioübertragung direkt ausgeschaltet.

Während der Testläufe wurde eine deutliche Überlastung der GUI festgestellt. Die Bildqualität hatte sich nochmal verschlechtert und der Hop in der Mitte reagierte auf jede Interaktion auf dem Gerät nur noch mit mehr als 5 Sekunden Verzögerung. Das Gerät Nr. 2 hatte sich aufgrund der Überlastung nach 5 Minuten Laufzeit selbst neu gestartet.

Ergebnisse

Der Chat funktionierte einwandfrei, auch wenn die Geräte über mehrere Hops miteinander kommunizieren mussten. Die AV-Konferenz funktionierte ordnungsgemäß, jedoch war die Qualität nicht zufriedenstellend. Die Anzahl der Hops hatte jedoch keinen Einfluss auf die Qualität der Übertragung. Mehrere Konferenzen im Netz führten zu Doppelbelastungen bei den Geräten und ergaben sehr hohe CPU-Auslastungen, die zu Instabilitäten führte.

6 Fazit

Dieses Kapitel soll als kurze Zusammenfassung der vorhergehenden Kapitel dienen, um einen kurzen Überblick der Ergebnisse zu liefern. Nach diesem Überblick werden diese Resultate mit den anfangs beschriebenen Minimalzielen verglichen. Abschließend wird auch ein Ausblick gegeben, bei dem aufgezeigt wird, wo noch Potenziale zur Weiterentwicklung der Software bestehen.

6.1 Projektzusammenfassung

GNUnet

UbiMuC setzt auf dem *GNUnet*-Framework auf und benutzt dieses für den eigenen Datentransfer. Die Vorteile von *GNUnet* liegen in der Konzeption zur anynomen Peer-to-Peer-Kommunikation und dem relativ geringen Ressourcenbedarf. Außerdem wurde direkt auf Transportebene die effiziente Verschlüsselung der Daten mitgeliefert. Das Auffinden anderer Peers und die Kommunikation zwischen den einzelnen Peers wird ebenfalls von *GNUnet* ermöglicht.

Das Framework ist modular aufgebaut, so dass die benötigten Teile unabhängig voneinander eingesetzt werden können und sich neue Module sowie Adapter zu unseren Programm-Modulen leicht integrieren lassen.

UbiMuC Transportschicht

Da das von *GNUnet* gebotene, anonyme Routing nicht zur Verwirklichung unserer Zwei-Punkt-Verbindungen geeignet war, wurden von der Projektgruppe zwei Konzepte entwickelt, mit denen Pakete selbständig und ohne Aufforderung an einen befreundeten Peer gesendet werden können. Essenziell war hierfür der Aufbau des Pfade durch das Netzwerk. Für den eigentlichen Datentransfer wurde der Paketdienst entwickelt, der den von uns entwickelten *UbiMuC*-Datencontainer verwendet, um die Nutzdaten zu verpacken.

Der Paketdienst ermöglicht die Sende- und Empfangsfunktionalität in Bezug auf die von *UbiMuC* neu generierten Nachrichtentypen. Diese sind „GetRoute“ und „GetRouteReply“, mit denen es möglich ist, eine Route zwischen zwei Peers festzulegen. Für die Konnektivität zwischen Paketdienst und der Multimedia-Anbindung wurden die AV-Sockets entwickelt, mit denen es einfach möglich ist, bestehende Streaming-Daten zwischen den Anwendungen der entsprechenden Peers, über *GNUnet* geleitet, zu versenden.

Nutzerverwaltung

Die Nutzerverwaltung ermöglicht dem Nutzer von *UbiMuC*, mit seinen Einträgen in der Kontaktliste in Kontakt zu treten, sobald diese als verfügbar angezeigt werden. Die von *GNUnet* bereitgestellte DHT liefert die Basis für diese Nutzerverwaltung. Da diese jedoch nicht zur lokalen Speicherung der benötigten Informationen genutzt werden konnte, wurden weitere *UbiMuC*-Strukturen als zusätzliche Abstraktion der DHT-Informationen entworfen.

Mittels der entwickelten DHT-Schnittstelle wurde es ermöglicht, auf einem lokalen Abbild der DHT zu arbeiten und die *GNUnet*-eigenen Strukturen unberührt zu lassen. Als zusätzliches Feature wurde die Kontaktliste hinzugefügt, mit der der Nutzer selbständig bekannte Netzteilnehmer zu einer eigenen Freundesliste hinzufügen und dies dauerhaft abspeichern kann.

Chat

Der Chat bietet eine direkte Punkt-zu-Punkt-Kommunikation zwischen Nutzern. Dieser stellte die Vorarbeit für Audio/Video-Konferenzen da, weil der Transfer von Text-Segmenten einfacher zu realisieren war, als die Übertragung großer Datenmengen in Echtzeit. Um den Chat komfortabler nutzen zu können, wurde er mit der Nutzerverwaltung verknüpft und eine intuitive Benutzeroberfläche hierfür erstellt.

Multimedia

Im Zusammenhang mit Multimedia-Streams wurde sich anfangs hauptsächlich damit beschäftigt, wie die Audio-/Video-Eigenschaften des N810 effizient genutzt werden können. Mittels *GStreamer* konnte ohne große Hardwareeinschränkungen auf die Webcam und das Mikrofon zugegriffen werden. Jedoch wurde die größte Einschränkung der Bildqualität immer noch durch die geringen Datentransferraten erzeugt. Deshalb war die Auswahl eines geeigeneten Videocodecs von großer Bedeutung, um die Übertragung des Bildes so flüssig und deutlich wie möglich zu gestalten. Zum Auf- und Abbau einer Konferenz wurde zusätzlich eine Handshake-Routine entwickelt.

Zusammenfassung

Durch die Kombination der von *GNUnet* angebotenen Peer-to-Peer-Funktionaliäten mit der eigens entwickelten *UbiMuC*-Software wurde ein funktionsfähiger Prototyp geschaffen, mit dem innerhalb von WLAN-Netzen Audio/Video-Konferenzen und Chat-Session abgehalten werden können. Außerdem können Freunde durch die Kontaktliste abgespeichert und damit leicht wiedergefunden werden. In Tests wurde gezeigt, dass die von uns entwickelten Konzepte umgesetzt wurden.

Allerdings stößt man bei Verwendung der Software schnell an die technischen Grenzen des verwendeten Internet-Tablets. Daher bleibt die Qualität bei Video-Übertragungen teils hinter den Erwartungen zurück. Trotzdem wurden die im Zusammenhang mit dem Datentransfer angestrebten Funktionen umgesetzt.

Es ist möglich, Ad-Hoc-Netzwerke mit den Geräten aufzubauen, und auch Übertragungen zwischen Geräten inklusive Weiterleitung über mehrere Zwischengeräte sind umsetzbar. Die Adaptierung dieser Infrastruktur für Netze mit festen IPs ist ebenfalls leicht möglich.

6.2 Vergleich mit Minimalzielen

An dieser Stelle wird noch einmal auf die in Abschnitt 1.3 beschriebenen Minimalziele der Projektgruppe eingangen und diese mit den Ergebnissen verglichen.

Gefordert war die Entwicklung eines Prototypen für ein mobiles Peer-to-Peer Framework unter Berücksichtigung von Sicherheitsaspekten. Dieses dient vornehmlich zum Austausch von Multimedia-Daten auf einem Internet-Tablet für bestimmte vorgegebene Anwendungsszenarien.

In Tabelle 6.1 werden diese Anforderungen mit unseren in den vorherigen Kapiteln erläuterten Lösungsansätzen gegebenübergestellt.

Minimalziel	Lösungskonzept
Ad-Hoc Netz	*Avahi*
Sicherheitsaspekte	durch *GNUnet* abgedeckt
IP-(Video-)Telefonie	Paketdienst, Nutzerverwaltung, *GStreamer*
Instant Messaging	Paketdienst, Nutzerverwaltung
Filesharing	nicht implementiert
UbiMuC Prototyp	GUI, WIR-Schicht & *GNUnet*
Evaluation	siehe Kapitel 5

Tabelle 6.1: Kurzübersicht der Minimalziele

Das Suchen und Finden der Peers in Ad-Hoc-Netzen wurde mit Hilfe von *Avahi* gelöst und in *GNUnet* integriert. Durch die Wahl von *GNUnet* als *UbiMuC* zugrundeliegendes Peer-to-Peer-Framework auf Transportebene musste sich nicht mehr um die Implementierung von Sicherheitsfeatures gekümmert werden, da *GNUnet* auf Grund seiner Konzeption bereits über ausgereifte Sicherheitsmechanismen bei der Datenübertragung verfügt. Die Realisierung der gegebenen Anwendungsfälle wird durch das Zusammenspiel von Paketdienst, Nutzerverwaltung und *GStreamer* umgesetzt. Die Videokonferenz liefert ein annehmbares Ergebnis und funktioniert auch über größere Entfernungen. Das Instant-Messaging verhält sich dank eines geringeren Datenaufkommens etwas robuster.

Auf eine Filesharing-Funktion musste jedoch verzichtet werden, da die Implementierung zusammen mit den restlichen Anforderungen aus Zeitmangel nicht mehr möglich war. Die Fertigstellung eines funktionsfähigen Paketdienstes und Streamings hatte insgesamt höhere Priorität. Schlussendlich wurde die Leistungsaufnahme und der Umfang an Daten in verschiedenen Tests geprüft. Bis auf das Filesharing wurden folglich alle im Projektgruppenantrag geforderten Ziele erfüllt. Dies ist bereits in *GNUnet* enthalten und erhält deswegen eine geringere Gewichtung im Gegensatz zu den anderen Anforderungen.

Über die minimalen Anforderungen hinaus wurden jedoch auch mehrere zusätzliche Features in der *UbiMuC*Software realisiert. Eine rudimentäre Kommunikationsmöglichkeit zwischen den Benutzern war von Anfang an vorgesehen. Jedoch wurde dieser Punkt in Form der Nutzerverwaltung inklusive Kontaktliste erweitert. Hierdurch ist gerade in einem Netz mit vielen Teilnehmern ein einfacher Verbindungsaufbau zu bekannten Peers möglich. Außerdem wurde für die Kommunikation auf Transport-Ebene neben dem Paketdienst eine Abstraktion der Verbindungen als bidirektionale Pipeline entwickelt. Diese ist jedoch nur ein Prototyp und wird in der *UbiMuC*-Software nicht verwendet, da sich diese am Ende des zweiten Projektgruppensemesters als sehr fehlerbehaftet herausstellte. Die Teile des Programms, die ursprünglich diese Pipeline nutzen sollten, verwenden daher ausschließlich den Paketdienst.

6.3 Ausblick

Als weitere Arbeit an *UbiMuC* bietet sich für zukünftige Projekte natürlich die Verbesserung der Audio-/Videoqualität sowie die Reduzierung der Prozessorlast an. Darauf aufbauend könnte man Multiuser-Konferenzen entwickeln, um mehrere Benutzer gleichzeitig miteinander kommunizieren zu lassen.

Da sich *UbiMuC* zur Zeit noch auf Multimedia und Kommunikation beschränkt, sind als Erweiterungen auch klassische Peer-to-Peer-Anwendungen wie zum Beispiel Filesharing denkbar. Hier könnten die Nutzer über *UbiMuC* das Netz nach interessanten Inhalten durchsuchen, um diese danach herunterzuladen. Ebenfalls denkbar wären auch Streams, die mit Hilfe von *UbiMuC* bereitgestellt werden und an bestimmten Orten einfach von den Benutzern abgerufen werden können. So könnten Touristeninformationen, aktuelle Abfahrpläne oder ähnliche Informationen durch fest installierte WLAN-Geräte über *UbiMuC* zur Verfügung gestellt werden. Diese Dienste könnten auch mit dem GPS-Gerät des N810 gekoppelt werden, um die Standpunkte für die einzelnen Informationen besser zu lokalisieren.

Da heutzutage immer mehr mobile Geräte mit WLAN ausgerüstet werden, wäre eine Portierung von *UbiMuC* auf andere Systeme auch sehr interessant. Hier bieten sich unter anderem Mobilfunkgeräte, Spielekonsolen, Pocket PC's oder ähnliche an. Die Mobilfunkgeräte bieten zusätzlich noch die Möglichkeit, für das lokale Ad-Hoc-Netz eine Schnittstelle über die Mobilfunkleitung zum Internet bereitzustellen.

7 Appendix

7.1 Config Parser

Der Configparser hat die Aufgabe, die in einer Konfigurationsdatei abgelegten Werte in vom Programm nutzbare Datenstrukturen umzuwandeln. Dabei handelt es sich um die einzelnen Einträge der Kontaktliste, die Konfigurationseinstellungen des Programms selbst und die eigens hinterlegten Zusatzinformationen.

Die Konfigurationsdatei liegt als „plaintext“ Datei im „Home-Verzeichnis“ des Benutzers und könnte somit auch von Hand editiert werden. Die Datei ist - wie oben bereits beschrieben - in drei Bereiche aufgeteilt. Neben Überschriften (zum Beispiel [owninfo]) sind nur Einträge der Form „<Key> = <Value>“ vorhanden. Dies vereinfacht die Dekodierung der einzelnen Werte extrem. Der Configparser liest die Datei zeilenweise ein und interpretiert die gelesene Zeile abhängig vom Bereich der Datei, in dem er sich aktuell befindet. Sollte eine gelesene Zeile nicht zugeordnet werden können, wird eine Fehlermeldung ausgegeben. Diese sagt genau aus, was für eine Art von Eintrag erwartet wird und gibt zudem die Nummer der Zeile aus, die nicht verarbeitet werden konnte.

Die resultierenden Datenstrukturen werden von einem globalen Objekt des Typs „ubimuc _config“ verwaltet. Dieses Objekt besitzt ein Interface, mit dem sowohl einzelne Werte (im Fall Konfigurationseinstellungen) geschrieben und gelesen werden können, stellt diese Funktionalität aber ebenso für ganze Listen (im Fall Kontaktlisteneinträge) beziehungsweise einzelne Objekte (im Fall eigene Informationen) zur Verfügung. Falls Werte schreibend verändert werden, wird auch dafür gesorgt, dass diese nicht nur in den Datenstrukturen des „ubimuc_config“ Objektes sondern ebenfalls in der Config Datei angepasst beziehungsweise hinzugefügt werden.

Beispiele für vom Programm benutzte Einträge sind die Ports für die Videokonferenz. Diese werden bei Bedarf aus dem Konfigurations-Objekt ausgelesen und verwendet.

7.2 Installation von *UbiMuC*

Die Installation von *UbiMuC* verläuft vollständig über den beim N810 bereits mitgelieferten Paketmanager. Es ist jedoch ein zusätzlicher Eintrag vom *UbiMuC*-Repository nötig. Um diesen hinzuzufügen, öffnet man den N810-Programmmanager und wählt unter dem Menüeintrag „Optionen“ den Punkt „Programmkatalog“. Im Programmkatalog klickt man auf „Neu“, um einen weiteren Eintrag zu generieren. Im sich daraufhin öffnenden Fenster „Katalogdetails“ sind die einzelnen Felder mit folgenden Werten zu füllen:

Adresse: **http://ls12-www.cs.tu-dortmund.de/ubimuc/repository/**
Komponenten: **main**

Einige Pakete im *UbiMuC*-Repository hängen von Paketen des *Maemo*-Repository ab, das standardmäßig nicht eingebunden wird. Dieses wird auf die selbe Art wie das vorige Repository eingebunden:

Adresse: **http://repository.maemo.org/**
Komponenten: **free non-free**

Nun kann man *UbiMuC* mittels des grafischen Paketmanagers oder mit root-Rechten per Kommandozeile mittels des Befehls „apt-get install ubimuc“ installieren.

7.3 Benutzerhandbuch

Programmstart

Um das Programm zu starten, muss zuerst das entsprechende Paket mittels Paketmanager installiert werden (siehe Abschnitt 7.2). Danach genügt die Eingabe des Befehls „ubimuc“ auf der Konsole.

```
user@nokia-N810-51-3:~$ ubimuc
```

Das Starten des Programms kann einige Sekunden in Anspruch nehmen. Sobald der Initialisierungsvorgang abgeschlossen ist, wird der Startdialog angezeigt. Hier hat man die Möglichkeit, seinen Nutzernamen nach Belieben zu wählen.

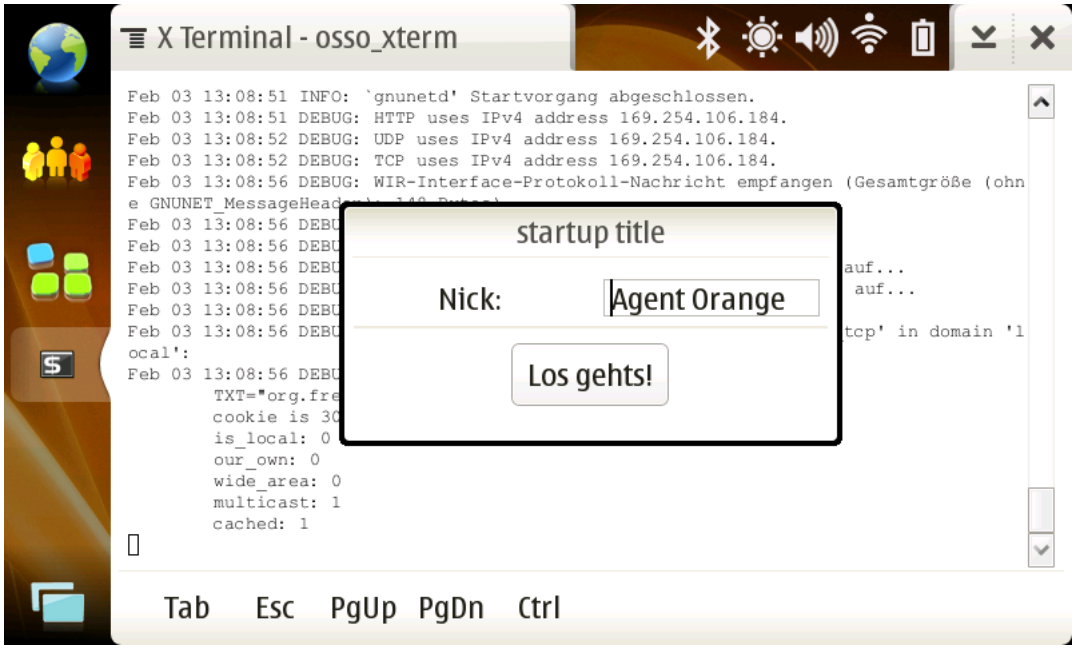

Abbildung 7.1: Nickname Eingabe beim Programmstart (GUI Screenshot)

Wurde das Programm bereits benutzt, wird der zuletzt verwendete Name angezeigt. Ein Klick auf „Los gehts!“ übernimmt den eingetragenen Namen und öffnet die eigentliche Benutzeroberfläche.

Die Benutzeroberfläche

Die Benutzeroberfläche besteht aus drei Teilen:

- Der Buddyliste
- Dem Chatfenster
- Dem Playerfenster

Zwischen diesen kann mit den Reitern am oberen Bildschirmrand umgeschaltet werden.

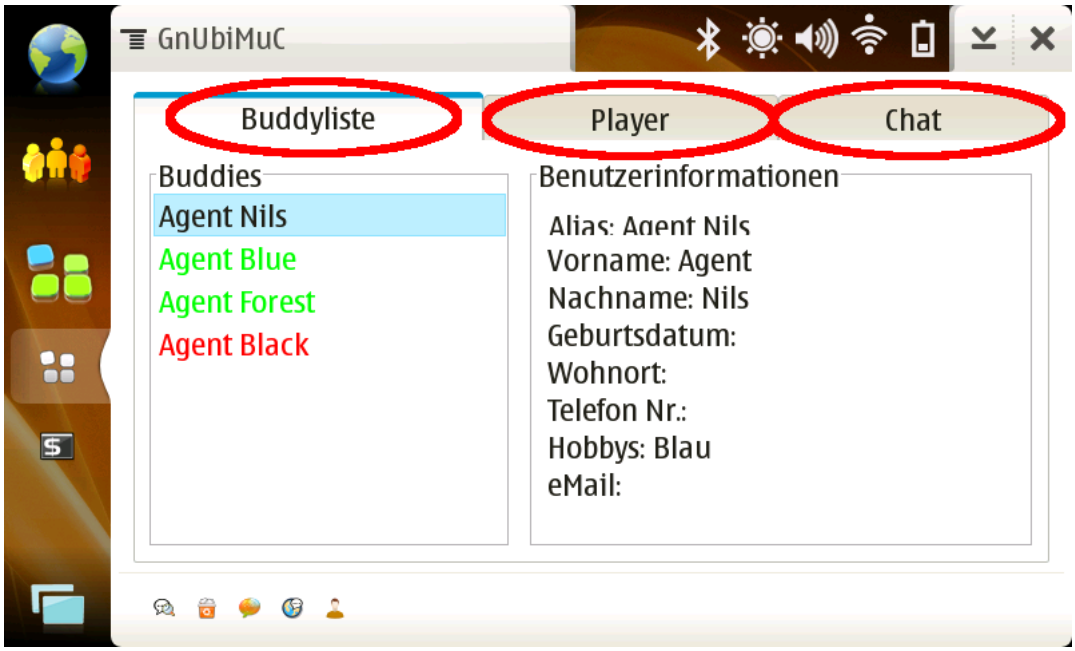

Abbildung 7.2: Navigationsreiter in der UbiMuC-Benutzeroberfläche (GUI Screenshot)

Bei Programmstart ist die Buddyliste standardmäßig ausgewählt.

Die Buddyliste

Die Buddyliste ist das Herzstück des Programms. Von hier aus können sämtliche Funktionen angestoßen werden. Sämtliche Kontakte werden in einer Liste gespeichert. Die Farbe der Einträge gibt dabei deren Online Status wieder. Rot steht für „offline“ und Grün für „online“.

Wenn man einen Kontakt auswählt, erscheint auf der rechten Bildschirmseite ein Kasten mit Informationen. Angezeigt wird hier der Alias des ausgewählten Kontaktes. Ist dieser momentan online, werden automatisch weitere Informationen (wie Geburtsdatum, Hobbys etc.) abgerufen und angezeigt, sofern diese von der Person hinterlegt wurden.

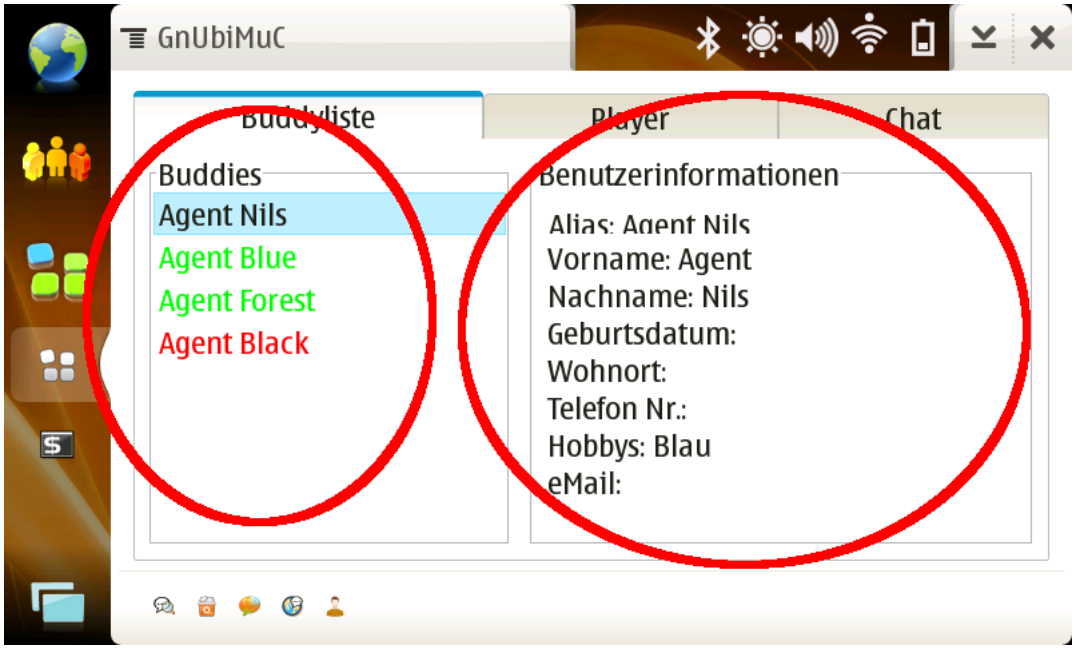

Abbildung 7.3: Die Buddyliste (GUI Screenshot)

Die einzelnen Funktionen lassen sich über Icons am unteren Bildschirmrand steuern. Dabei können folgende Funktionen aufgerufen werden:

Abbildung 7.4: Funktionsmenü für das Manipulieren der Buddyliste (GUI Screenshot)

- Hiermit ist es möglich, nach weiteren Kontakten zu suchen. Bei einem Klick öffnet sich ein Dialog mit einer Liste aktuell verfügbarer Kontakte. Mit einer Eingabe im Feld „Filter“ ist es möglich, die Suche einzugrenzen.

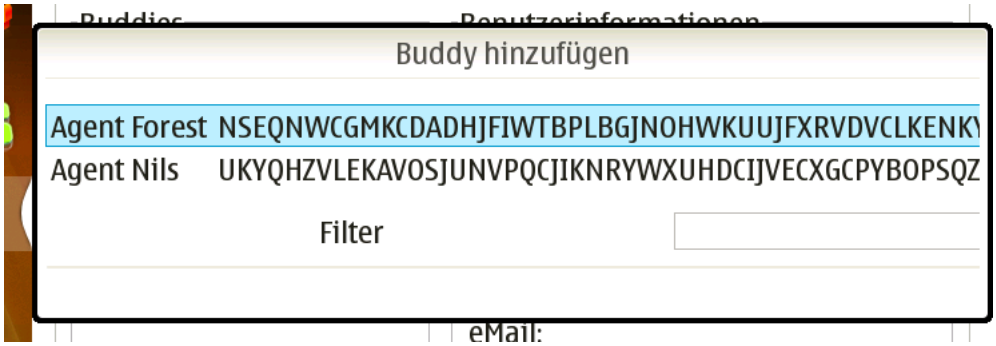

Abbildung 7.5: Dialog für das Hinzufügen einzelner Kontakte (GUI Screenshot)

- Ein Klick auf dieses Icon sorgt dafür, dass der angewählte Kontakt aus der Liste der bekannten Kontakte gelöscht wird.

- Dieses Icon bietet die Möglichkeit, mit dem aktuell angewählten Kontakt eine Chat-Session zu eröffnen. Dabei wird automatisch auf das „Chat“ Fenster gewechselt.

- Wenn ein einfacher Chat nicht reicht, kann auch eine Videokonferenz mit dem aktuell angewählten Kontakt gestartet werden. Dieses Icon ermöglicht dies.

- Mit dieser Option kann man den angezeigten Alias des ausgewählten Kontaktes verändern.

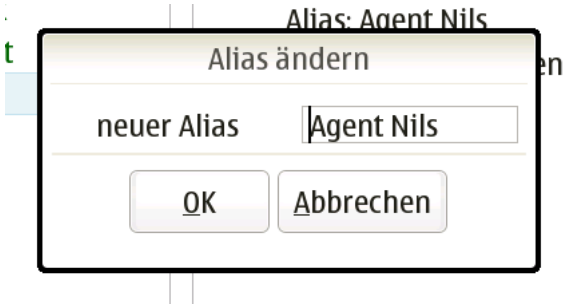

Abbildung 7.6: „Alias ändern“ Dialog (GUI Screenshot)

Das Player-Fenster

Das Playerfenster wird im Falle einer Videokonferenz benutzt.

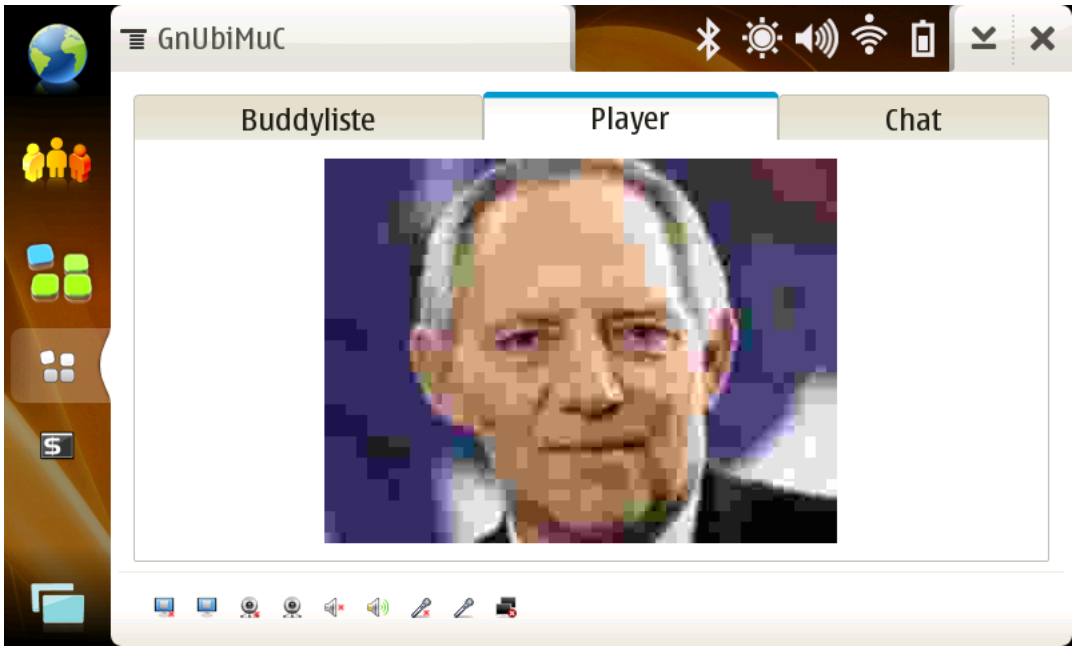

Abbildung 7.7: Eine laufende Videokonferenz (GUI Screenshot)

Über die Icons am unteren Bildschirmrand ist es möglich, verschiedene Funktionen an- und auszuschalten.

Abbildung 7.8: Funktionsmenü für das Manipulieren der Videokonferenz (GUI Screenshot)

Hiermit ist es möglich, das angezeigte Bild an- und auszuschalten.

Analog dazu lässt sich auch die Kamera an- und ausschalten.

Mit diesen Icons lässt sich die Tonausgabe an- und ausschalten.

Analog dazu lässt sich auch das Mikrofon an- und ausschalten.

Hiermit kann man die gesamte Videokonferenz beenden.

Der Chat

Im Chatfenster werden aktive Chat-Sitzungen angezeigt.

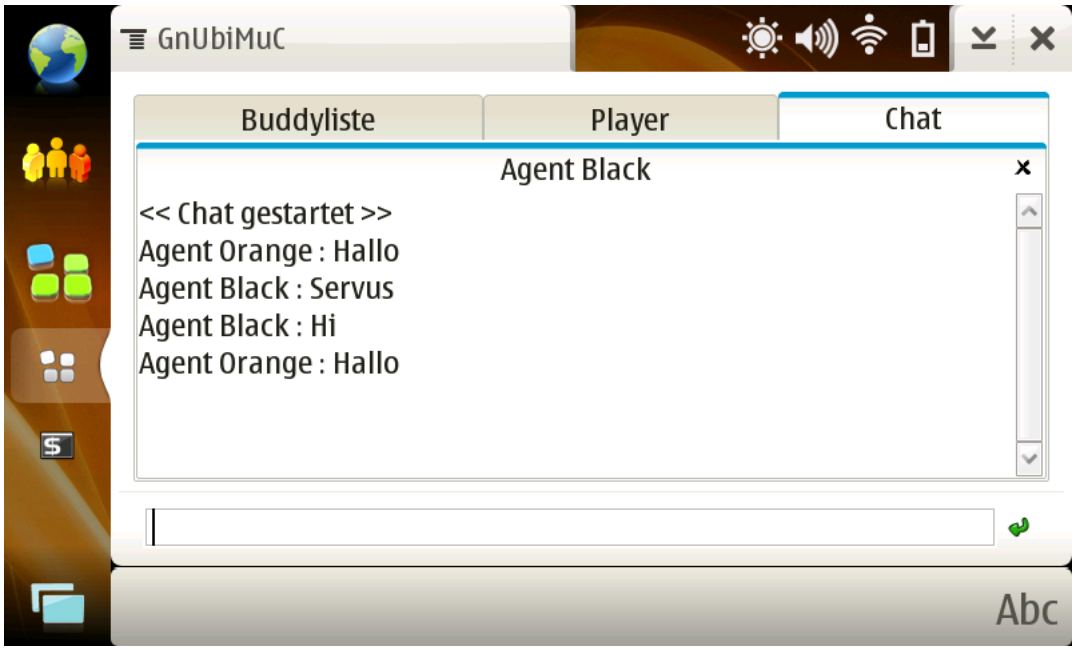

Abbildung 7.9: Aktive Chatsitzung in der UbiMuC-Benutzeroberfläche (GUI Screenshot)

Es ist möglich, mit mehreren Personen parallel zu chatten. Dabei wird jede Chat-Sitzung als einzelner Reiter am oberen Bildschirmrand angezeigt. Um Nachrichten an den aktuell ausgewählten Nutzer zu schicken, muss diese Nachricht in das Textfeld am unteren Bildschirmrand eingegeben und das „Senden“-Icon angeklickt werden. Das Senden mittels drücken der „Enter“-Taste ist ebenfalls möglich.

Abbildung 7.10: Eingabefläche und Senden-Button in der UbiMuC-Benutzeroberfläche (GUI Screenshot)

Optionsmenüs

Um die Optionenmenüs aufzurufen, genügt ein Klick auf den Programmnamen mit anschließender Auswahl des entsprechenden Eintrags.

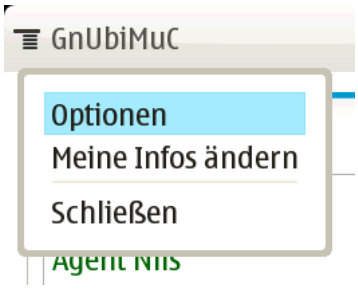

Abbildung 7.11: Aufruf des Optionsmenüs (GUI Screenshot)

Im eigentlichen Optionsmenü lassen sich diverse Einstellungen (wie der Startbefehl des *GNUnet*-Daemons oder die von der Videokonferenz benutzten Ports) ändern. Ein Klick auf „Anwenden“ sorgt dafür, dass die Einstellungen übernommen und gespeichert werden.

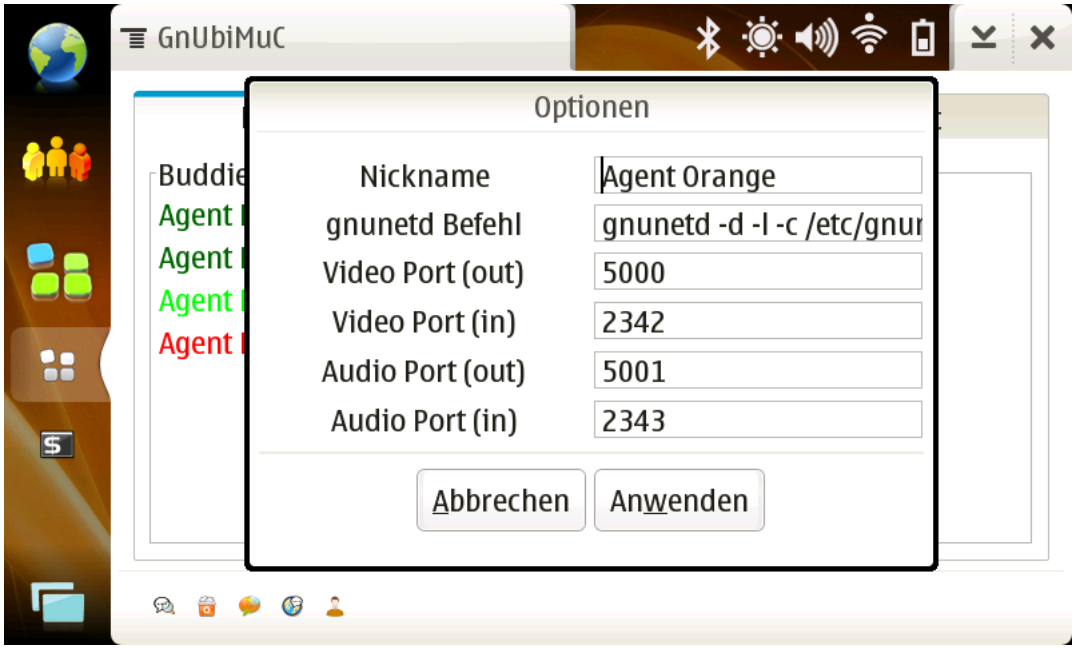

Abbildung 7.12: Das Optionsmenü (GUI Screenshot)

Im Dialog „Meine Infos ändern“ kann man weitere Informationen zu sich selbst angeben. Diese Informationen werden bei anderen Nutzern angezeigt, wenn diese weitere Informationen abrufen (z.B. durch Anwählen eines Kontaktes in ihrer Buddyliste).

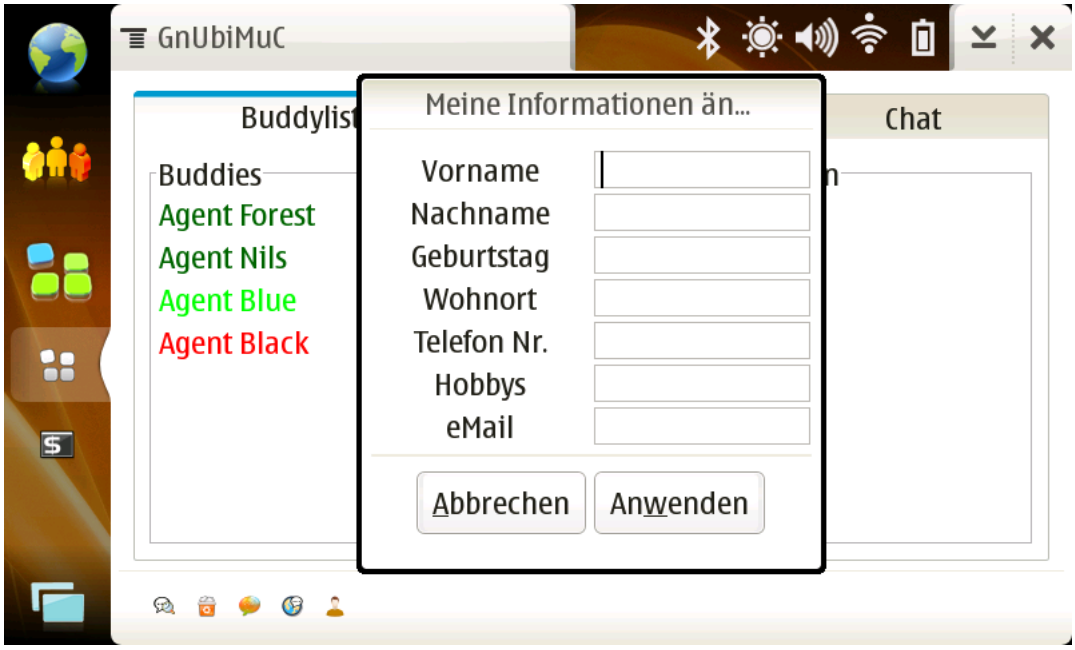

Abbildung 7.13: Dialog zur Angabe weiterer persönlicher Informationen (GUI Screenshot)

Programm beenden

Um das Programm zu beenden, genügt ein Klick auf das „X“ in der oberen rechten Bildschirmecke oder die Auswahl des Menüeintrags „Schließen“.

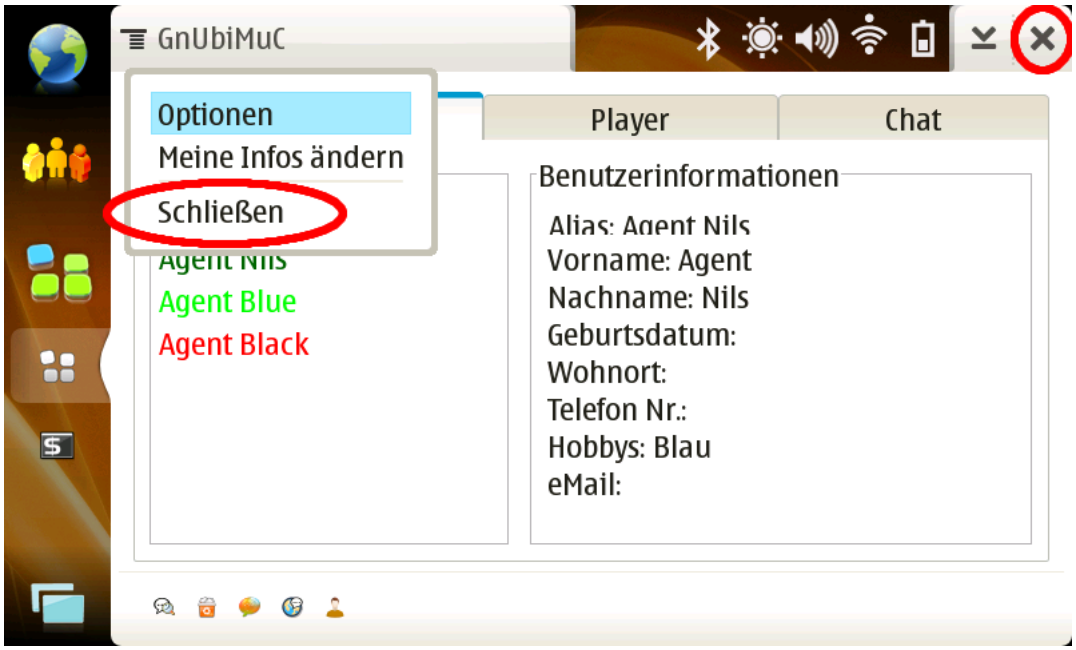

Abbildung 7.14: Beendemöglichkeiten von UbiMuC (GUI Screenshot)

7.4 Bekannte Fehler

In diesem Abschnitt werden die bekannten Probleme (Bugs) beschrieben und gegebenenfalls auf mögliche Lösungsansätze hingewiesen. Die folgenden Bugs wurden aus Zeit- und Ressourcenmangel nicht behoben.

In dem Modul, das die Kontaktliste umsetzt, ist ein Bug bekannt.

- Gelegentlich wird ein Nutzer, der sich im *UbiMuC*-Netz befindet, von der Kontaktliste nicht als „online“ angezeigt. Der Grund hierfür liegt wahrscheinlich in der „Unzuverlässigkeit“ der auf der *GNUnet*-DHT aufbauenden Kontaktliste, die die zugehörigen Antworten auf DHT-Anfragen nicht erhält. Das heißt, der eigentliche Fehler liegt nicht im *UbiMuC*-Modul, sondern in der DHT von *GNUnet*, die teilweise nicht korrekt auf DHT-Anfragen antwortet. Dieses Problem könnte mit einem Patch für *GNUnet* eventuell behoben werden.

Literaturverzeichnis

[1] Funktionsbeschreibung des N810 auf der deutschen Nokia-Homepage:
http://www.nokia.de/A4630299

[2] Homepage des MPlayer Port für Maemo:
http://mplayer.garage.maemo.org/

[3] Homepage des Pidgin Port für Maemo:
http://pidgin.garage.maemo.org/

[4] Homepage des „cross compilation toolkits“ Scratchbox:
http://www.scratchbox.org/

[5] Skript zur Vorlesung „Algorithmen für Peer-to-Peer-Netzwerke“ von P. Mahlmann und C. Schindelhauer, Universität Paderborn, Sommersemester 2004
http://wwwcs.upb.de/cs/ag-madh/WWW/Teaching/2004SS/AlgoP2P/

[6] Homepage der Peer-to-Peer-Working-Group
http://p2p.internet2.edu/index.html

[7] GNUnet-Homepage:
http://www.gnunet.org/

[8] Hildon-Framework auf Gnome.org:
http://live.gnome.org/Hildon

[9] GTK+-Homepage:
http://www.gtk.org/

[10] Whitepaper der DHT in GNUnet: The Design and Implementation of a Distributed Hash Table for a Peer-to-Peer Network
https://gnunet.org/svn/GNUnet-docs/papers/dht/whitepaper.pdf

[11] Maymounkov, P. and Mazières, D., Kademlia: A peer-to-peer information system based on the xor metric. Proceedings of the 1st International Work-shop on Peer-to-Peer Systems (IPTPS).
http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf

[12] GStreamer Homepage:
http://gstreamer.freedesktop.org

[13] Video4Linux Homepage:
http://linux.bytesex.org/v4l2/

[14] ALSA Homepage:
http://www.alsa-project.org/main/index.php/Main_Page

[15] GNOME Homepage:
http://www.gnome.org/

[16] Flumotion Homepage:
http://www.flumotion.net/

[17] Hantro 4200 Homepage:
http://www.hantro.com/index.php?140

[18] Hantro 4100 Homepage:
http://www.hantro.com/index.php?98

[19] G.711 auf der Homepage der ITU-T:
http://www.itu.int/rec/T-REC-G.711/en

[20] G.729 auf der Homepage der ITU-T:
http://www.itu.int/rec/T-REC-G.729/en

[21] RFC 1889:
http://www.ietf.org/rfc/rfc1889.txt

[22] Maemo Testprogramme:
http://maemo.org/development/documentation/manuals/4-0-x/

Hinweis:

Referenzen auf Internetseiten entsprechen - wenn nicht anders angegeben - dem Stand vom 08. März 2009.

Abbildungsverzeichnis

Tabellenverzeichnis