Johann Wolfgang Goethe-Universität
Frankfurt am Main

Fachbereich Biologie und Informatik

Diplomarbeit

Entwicklung von Gruppenkommunikationsdiensten in einem Rahmenwerk für spontan vernetzte Systeme

Kai-Marcel Teuber

22. Juni 2005

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

Danksagung

Ich möchte mich hiermit bei allen Personen bedanken, die mich bei der Anfertigung dieser Arbeit unterstützt haben. Mein besonderer Dank gilt dabei meiner Familie, Christine Kirchmeier und Peter Schminke, die mir stets hilfreich zur Seite standen und mich tatkräftig unterstüzt haben.
Für die ausgezeichnete Betreuung, Freiheiten bei der Erarbeitung der Lösungen und die äußerst wertvollen Anregungen möchte ich mich bei Dipl.-Inf. Michael Lauer und Prof. Dr. Oswald Drobnik herzlichst bedanken.

Kai-Marcel Teuber

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

Frankfurt am Main, den 22. Juni 2005

K a i - M a r c e l T e u b e r

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Innerhalb des Schwerpunktprogramms "Basissoftware für selbstorganisierende Infrastruktur für vernetzte mobile Systeme" der Deutschen Forschungsgesellschaft wurde an der Professur "Architektur und Betrieb Verteilter Systeme / Telematik" ein Rahmenwerk für E-Learning in Ad-Hoc Netzen, kurz ELAN, entworfen und realisiert. Durch immer stärker aufkommende kleine und leistungsfähige mobile Endgeräte werden kognitive und kollaborative Anwendungen in Ad-hoc Netzen umsetzbar und erfreuen sich gesteigerten Interesses. Ad-Hoc Netze entstehen spontan zwischen den verschiedenen Endgeräten und sind aufgrund der Mobilität der Benutzer starken Veränderungen unterworfen. Dies führt zu Problemen, die in fest verdrahteten Netzwerken nicht existieren. Z.B. können sich die Übertragungswege für Daten innerhalb kurzer Zeit verändern oder Gruppenteilnehmer können ihre Endgeräte ausschalten und so für Übertragungen nicht mehr zur Verfügung stehen. Daher wurde im ELAN-Rahmenwerk eine Middleware erarbeitet, die diese Probleme löst und eine zuverlässige Kommunikation zwischen einzelnen Teilnehmern ermöglicht. Das Ziel von ELAN ist die Unterstützung von lerninteressierten Personen mit geeigneten Angeboten. Diese erfolgt derart, daß Interessens- und Wissensprofile vom Anwender eingegeben werden. Mit Hilfe der Nutzerprofile können andere Personen innerhalb des Ad-Hoc Netzes gefunden und angezeigt werden. Durch angebotene Dienste können zusätzlich Leistungen erbracht werden, damit bildet ELAN eine Grundlage für das gemeinsame Lernen von Anwendern. Das Rahmenwerk bietet grundsätzlich Punkt-zu-Punkt Kommunikation an, dabei sind jedoch vor allem Gruppen an gemeinsamer Kommunikation interessiert.

Gruppenkommunikationsdienste bilden im Internet die Grundlage für viele Anwendungen. Dabei sind synchrone als auch asynchrone Anwendungen zu finden. Asynchrone Dienste verwenden dazu meist einen oder mehrere zentrale Server, die Daten vorhalten. Verbindet sich ein interessierter Anwender, bekommt er die gewünschten Daten zeitnah übermittelt. Die populärsten Beispiele für asynchrone Kommunikation sind E-Mail und das Usenet. Synchrone Dienste sind oftmals Chaträume, in denen Personen kommunizieren, Dateiaustauschprogramme, über die verschiedenste Inhalte ausgetauscht werden oder die zurzeit sehr aktuelle Internettelefonie [Man05].
Für E-Learning in Ad-Hoc Netzen sind vor allem die synchronen Dienste von Interesse, da die Anwender zeitnah miteinander interagieren. Dies geschieht genau zum Zeitpunkt des Lernens. Auf diese Art erfährt ein Anwender, daß gerade in seiner Nähe eine Lerngruppe existiert, die sich mit Themen beschäftigt, die für ihn von Interesse sind. Er kann dann

spontan mit der Lerngruppe in Kontakt treten. Diese Leistung können asynchrone Dienste nicht erbringen und sind daher von untergeordnetem Interesse für ELAN. Synchrone Dienste bieten sich wegen ihrer zeitnahen Bindung für Gruppenkommunikation an und sind von zentraler Bedeutung für das Rahmenwerk.

Gruppenkommunikation stellt in Ad-Hoc Netzen eine besondere Herausforderung dar. Durch das Fehlen fester Stationen sind Client-Server Modelle nur bedingt verwendbar, dies erschwert die Verwaltung von Gruppen in Ad-Hoc Netzen. Zusätzlich existiert auf Hardwareebene kein Mechanismus für Multicastübertragungen. Das Versenden von Nachrichten verhält sich wie ein Broadcast[1], der lokal durch die Ausbreitung des Funksignals begrenzt ist.

1.1 Ziele der Arbeit

Das ELAN-Rahmenwerk bildet die Grundlage für die Implementierung von Komponenten. Es kümmert sich dabei um die Verbreitung von Profilen im Ad-Hoc Netz. Kommunikation zwischen Komponenten wird durch die Middleware angeboten und zuverlässig über das Ad-Hoc Netzwerk bereitgestellt. Eine Grundlage zur Entwicklung von Gruppenkommunikationsdiensten fehlt bisher. Im Rahmen der Diplomarbeit sollen die vorhandenen Kommunikationsmöglichkeiten des ELAN-Rahmenwerks auf Gruppentauglichkeit geprüft und gegebenenfalls eine gemeinsame Grundlage für Gruppenkommunikationsdienste geschaffen werden. Zur Realisierung notwendige Änderungen am Rahmenwerk sollen dabei vorgenommen werden. Abschließend soll durch die Entwicklung von Gruppenanwendungen gezeigt werden, daß die entwickelten Verfahren für Gruppenkommunikaton und -verwaltung probate Abstraktionen darstellen. Mögliche Optimierungen der Gruppenkommunikation sollen besprochen werden und auf ihre Realisierbarkeit hin untersucht werden.

1.2 Aufbau der Arbeit

Die Arbeit ist in sechs Kapitel gegliedert. Kapitel 2 beschäftigt sich mit den Grundlagen und erläutert den vorgefundenen Entwicklungsstand vom ELAN-Rahmenwerk. Außerdem wird die Entwicklung einer ersten Komponente beschrieben. Kapitel 3 behandelt die Entwicklung eines ersten Gruppendienstes, mit dessen Hilfe weitere Ansätze motiviert werden. Anschließend wird in Kapitel 4 auf Basis der erkannten Probleme die Entwicklung einer Gruppenkommunikationsschicht für das ELAN-Rahmenwerk beschrieben. Um zu zeigen, daß die Gruppenkommunikationsschicht als Grundlage für Gruppenanwendungen funktioniert, werden zwei Anwendungen entworfen und implementiert. Kapitel 5 stellt anschließend alternative dezentralisierte Verfahren zur Gruppenverwaltung und -kommunikation vor. Abschließend wird aus einer der vorgestellten Alternativen ein Entwurf konzeptionell erarbeitet. Kapitel 6 fasst als Abschluss die gesamte Arbeit zusammen und unterbreitet Vorschläge möglicher Erweiterungen für das ELAN-Rahmenwerk.

[1] Alle Empfänger werden benachrichtigt

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen kurz erläutert, welche für das Verständnis der Arbeit wichtig sind. Um den Rahmen dieser Arbeit überschaubar zu halten, wird gegebenenfalls auf Details in der Literatur verwiesen. Zuerst werden kurz Ad-hoc Netze erläutert, auf denen das ELAN-Rahmenwerk basiert. Dann folgt ein kurzer Abschnitt über die Sprache Python, in der ELAN realisiert wurde. Das GUI-Toolkit Qt3, welches in ELAN verwendet wurde, wird anschließend angesprochen. PyQt und SIP, welche die Bindung zwischen PyQt und den entsprechenden C++ Klassen leisten, werden kurz eingeführt und anschließend das ELAN-Rahmenwerk selbst. Das Kapitel endet mit der Entwicklung einer ersten einfachen Komponente, die bereits Gruppenkommunikation in einfacher Form verwendet.

2.1 Gruppenanwendungen

Synchrone Gruppenanwendungen sind für diese Arbeit von besonderem Interesse. Diese Anwendungen zeichnen sich dadurch aus, daß gleichzeitig mehrere Anwender aktiv teilnehmen können. Daher wollen wir kurz zwei wichtige Anwendungen von Ablaufprinzip her erläutern.

2.1.1 Dateifreigabe - Filesharing

Dateifreigabeanwendungen bieten die Möglichkeit eine durch den Anwender festlegbare Menge von Dateien für andere Nutzer bereitzustellen. Dies kann kontrolliert und unkontrolliert stattfinden. Kontrollierte Freigaben sichern die Daten derart, daß nur Anwender die z.B. das Passwort kennen, die Dateien empfangen können. Unkontrollierte Freigaben sind für jeden Nutzer zugänglich, der das Filesharing Programm verwendet, dabei interessiert in dieser Arbeit vor allem diese Art der Dateifreigabe.
Zur Übertragung von Dateien setzten Filesharing Programme oftmals geschickte Übertragungsprotokolle ein. Diese müssen den Transport der Daten durch das Netz sicher abwickeln. Dies ist gerade in Ad-hoc Netzen schiewiger zu realisieren. Um die Sicherheit der Übertragung zu erhöhen, werden diese blockweise organisiert. Dabei werden Dateien in mehreren einzelnen Blöcken übertragen. Der Empfänger muß die Blöcke wieder

zusammenfügen und erhält dadurch die vollständige Datei. Zerlegen in Blöcke wird als Marshalling sowie das Zusammensetzen als Unmarshalling bezeichnet.

2.1.2 Gemeinsame Zeichenfläche - Shared Whiteboard

Eine gemeinsame Zeichenfläche bietet einen Malbereich, auf dem durch jeden Teilnehmer mit festgelegten Werkzeugen gearbeitet werden kann. Solche Werkzeuge umfassen in der Regel Texte, Rechtecke, Ellipsen und vieles mehr.
Gemeinsame Zeichenflächen-Programme verwenden oftmals eine Zugriffssteuerung. Dies regelt, wann welcher Nutzer auf der Zeichfläche ein Werkzeug verwenden darf. Meist kann der aktive Nutzer die Steuerung bewußt an den nächsten Teilnehmer weitergeben. Diese Art der Zugrifssteuerung widerspricht allerdings dem kollaborativen E-Learningansatz, der gemeinschaftliches Lernen unterstützen soll. Durch das Entfernen der Zugriffssteuerung ist es möglich, daß alle Nutzer gleichzeitig Werkzeuge verwenden können. Zusätzlich kommt dies Ad-hoc Netzen entgegen, denn beim Ausfall von Stationen kann es vorkommen, daß einzelne Teilnehmer nicht mehr erreichbar sind. Dies würde eine zuverlässige Zugriffssteuerung unterbinden.

2.2 Ad-hoc Netze

Ad hoc bedeutet soviel wie spontan, dies ist ebenfalls die Art und Weise wie solche Netze entstehen. Ad-hoc Netze basieren auf spontaner Vernetzung bei mobilen Endgeräten. WLAN-Karten kennen hierfür den Ad-hoc Betriebsmodus. Wird dieser ausgewählt, verbinden sich die Netzwerkkarten mit gleichgesinnten Anderen in ihrer Reichweite. Es entsteht dabei eine Punkt-zu-Punkt Verbindung innerhalb der Reichweite. Jede Netzwerkkarte kann allerdings gleichzeitig mit mehreren Teilnehmern verbunden sein. Unter Ausnutzung dieser Eigenschaft lassen sich Netze spannen. Denn steht ein Knoten mit mehreren anderen Knoten in Verbindung, ist es möglich, daß dieser eine Vermittlerrolle übernehmen kann. Dabei überträgt er Daten derart weiter, daß die ihm bekannten Knoten miteinander kommunizieren können. Er vermittelt dabei die Kommunikation zwischen diesen Knoten.

Auf Grund der spontanen Vernetzung besitzen Ad-hoc Netze eine sehr hohe Flexibilität. Diese sorgt dafür, daß ebenfalls schnell Probleme eintreten können.
Fallen entscheidende Knoten aus, entstehen Teilnetze, auch Partitionen genannt.
Da keine zentralen fixen Punkte im Netz existieren, müssen viele Aufgaben von den Knoten direkt geleistet werden. Client-Server Modelle funktionieren nur stark eingeschränkt in Ad-hoc Netzen. Durch fehlende zentrale Anlaufstellen müssen Serverrollen ebenfalls spontan vergeben werden. Dadurch sind klassische Store-and-Forward[1] Dienste, wie z.B. E-Mail nicht oder nur eingeschränkt realisierbar.
Weitere ausführliche Informationen über Ad-hoc Netze und deren mögliche Probleme können in [Per00] gefunden werden.

[1] Speichern bis an den Empfänger weiterleitbar

2.3 Python

Python liegt derzeit in Version 2.41 vor und kann unter [Pyt05] kostenlos aus dem Internet bezogen werden. Binärpakete für verschiedene Plattformen, als auch compilierbarer C++ Quellcode stehen zur Auswahl. Python wurde von Guido van Rossum im Februar 1991 das erste Mal der Öffentlichkeit zur Verfügung gestellt und war als kleine objektorientierte Skriptsprache entworfen worden. Dabei brachte Python allerdings entscheidende Vorteile mit, die den großen und weiterhin steigenden Erfolg begründen. Ein paar dieser Vorteile von Python sind:

Python bietet eine klare Syntax. Dabei wird auf überflüssige Symbole wie geschweifte Klammern anderer Programmiersprachen verzichtet und Einrückung wird als Teil der Syntax verwendet. Dies führt langfristig zu übersichtlicherem und wartbarem Quellcode.

Python ist durchgängig objektorientiert. Dadurch bietet Python die gleichen Möglichkeiten wie 'große' Sprachen. Polymorphismus, Mehrfachvererbung oder Operator-Überladungen sind einfach anwendbar und das trotz dynamischer Typisierung.

Python besitzt einen modularen Aufbau. Module können mit Import-Anweisungen bekannt gemacht und anschließend verwendet werden. Dies sorgt für große Erweiterbarkeit. Python bringt in der aktuellen Version 2.41 eine nahezu unüberschaubare Vielzahl von Modulen mit, die einen Großteil möglicher Programmieraufgaben erfüllen können. Darunter sind z.B. Webserver, Netzwerkkommunikation, Datenbanken und vieles mehr. Eine ausführliche Beschreibung über Python 2.2[2] sowie aller enthaltenen Module geben [Mar03, Wei03].

Python ist eine interpretierte Sprache. Vor Ausführung von Programmen ist kein Compilerlauf notwendig und Programme können zu jedem beliebigen Zeitpunkt problemlos angehalten werden. Dynamische Typisierung ist dadurch möglich, d.h. auch wenn ein Objekt erst zur Laufzeit in seiner Ausprägung tatsächlich bekannt ist, kann Python Quellcode dieses universell verwenden. Dieser Vorteil kann aber auch zum Nachteil werden, denn durch strenge Typisierung können Compiler bereits Fehler vor der Ausführung finden. Ein Programmierer sollte beim Entwurf seines Quellcodes bereits darauf achten, welche Objekte von einer möglichen Methode entgegen genommen werden sollen.[3] Zusätzlich ist durch den interpretierten Charakter von Python ein Geschwindigkeitsverlust in der Ausführung hinzunehmen.

2.4 GUI-Toolkit Qt

Qt ist kostenlos für Linux und Mac Os unter [qt305] in der GPL-Lizenz verfügbar, derzeit in Version 3.34. Eine Version für Microsoft Windows kann derzeit nur käuflich erworben

[2] und höher

[3] Ab Python 3 soll die optionale Möglichkeit eingeführt werden, Quellcode streng zu typisieren.

werden. Nicht-kommerzielle Versionen liegen verschiedenen Büchern, wie z.B. [BS04] bei. Eine für Ende des Jahres 2005 angekündigte Version 4 von Qt soll für Linux, Mac Os und Windows als kostenlose GPL und als kommerzielle Version erscheinen.[onl05] Qt ist eine C++ Klassenbibliothek und kann daher nicht nativ in Python verwendet werden. Zu diesem Zweck werden zwei zusätzliche Module benötigt, die eine Bindung zwischen Qt und Python erzeugen. GUI-Toolkits liegen oberhalb der API der Betriebssysteme und bieten so eine einheitliche Schnittstelle zur Programmierung plattformübergreifender, grafischer Anwendungen. Ein einheitliches, durchgängig logisches Konzept sowie eine ausführliche Dokumentation[Qt05] unterstützen dabei den Programmierer. Qt3 bietet für nahezu alle gängigen Situationen vordefinierte Klassen und nimmt so dem Programmierer eine Menge Arbeit ab. So kann z.B. ein Dialog zum Erfragen einer Farbe mit wenigen Zeilen Quellcode erzeugt werden. Zusätzlich existiert ein UI-Builder[4], um die grafischen Elemente leichter entwerfen zu können. Bei Qt ist dies der so genannte Qt-Designer. Der Qt-Designer verfügt über die Möglichkeit grafische Oberflächen zusammenzusetzen und kennt die gängigsten Qt-Klassen. Außerdem können sogenannte Signale und Slots verbunden werden.

2.4.1 Qt-Designer

Der Qt-Designer ist ein eigenständiges Tool, das es erlaubt, das User-Interface zu entwickeln. Dabei bietet der Qt-Designer eine Vielzahl von Möglichkeiten für den Entwurf. Allerdings verfügt er nicht über die Möglichkeit, alle Qt-Klassen grafisch abzubilden und kann keine fremd erstellten Klassen verwenden. Dies soll ab Version 4 möglich sein. Der Qt-Designer bietet für die gängigsten Widgets Vorlagen, die das Arbeiten erleichtern. Um z.B. einen Dialog zu erstellen, muß der Entwickler nur eine der möglichen Dialogvorlagen wählen. Darunter finden sich Dialoge mit und ohne Druckknöpfe. Die in Dialogen nötigen Methoden `accept()` und `reject()` sind in allen Vorlagen bereits enthalten. Außer der Möglichkeit grafische Elemente zusammenzubauen, bietet der Qt-Designer weiter die Möglichkeit Signale und Slots zu verbinden. Slots können angelegt werden, die später im Quellcode realisiert werden. Diese Slots tauchen im erzeugten Quellcode alle mit einem leeren Methodenkörper auf. Auf diese Art und Weise kann schnell die Funktionalität eines Widgets erzeugt werden. Der Qt-Designer speichert erzeugte Widgets unter der Endung `.ui` ab. Diese Dateien können mit Hilfe der Kommandozeile `pyuic` in Python Quellcode übersetzt werden, `pyuic` ist Bestandteil von PyQt. Abbildung 2.2 zeigt den Qt-Designer, dabei ist das User-Interface für den ELAN Relay Chat - Abschnitt 2.7 - geöffnet. Innerhalb der Abbildung sind links eine Auswahlmöglichkeit für Widgetelemente, in der Mitte das erstellte User-Interface und rechts die Eigenschaften des ausgewählten Elements des User-Interfaces zu erkennen.

2.4.2 Signale und Slots

Das Konzept der Signale und Slots innerhalb von Qt ist einfach und bietet einen wirkungsvollen Interaktionsmechanismus. "Ein Signal ist eine Nachricht, die von einem Objekt an einen Slot eines anderen Objekts geschickt wird, falls ein bestimmtes Ereignis eintritt" - vergleiche [Lau02]. Dieses Ereignis muß nicht zwangsläufig die Benutzeroberfläche betreffen. Das Signal kann dabei beliebig viele Parameter verschiedenster Typen beinhalten. Zur

[4] User Interface - Anwenderschnittstelle

Verarbeitung von Signalen und Slots werden die drei zusätzlichen Schlüsselworte `SIGNAL`, `SLOT` und `emit` eingeführt. Zusätzlich kennt `QObject` die Methode `connect()`, mit welcher Signal-zu-Slot Verbindungen zur Laufzeit festgelegt werden können. Innerhalb dieser Methode werden die genannten Schlüsselworte verwendet. Zusätzlich ist es möglich, ein Signal mit beliebig vielen Slots zu verbinden, die Aufrufreihenfolge ist dabei allerdings nicht festgelegt. Mit Hilfe von `emit` können eigene Signale gesendet werden, dies ist sehr nützlich zum Entwurf eigener Elemente. Eigene Signale können natürlich ebenfalls an Slots gebunden werden.

2.5 SIP und PyQt

Das Werkzeug SIP kann unter [sip05] bezogen werden. Dabei steht SIP als Open Source Software kostenlos zur Verfügung. SIP erzeugt Bindungen zwischen C++ und Python und wurde ursprünglich für PyQt entwickelt. Seine Nutzung ist aber nicht an PyQt gebunden. Mit Hilfe von SIP können C++ Klassen innerhalb von Python verwendet werden.

Auch PyQt ist kostenlose Open Source Software und kann unter [PyQ05] bezogen werden. PyQt erzeugt Bindungen der Qt-Klassen für Python mit Hilfe von SIP. Erzeugte Bindungen stehen für Python innerhalb der Module `qt`, `qtcanvas`, `qtgl`, `qtnetwork`, `qtsql`, `qttable`, `qtui` und `qtxml` zur Verfügung. Diese beinhalten weit über 300 Klassen mit nahezu 6000 Methoden. Diese Module werden innerhalb von Python mit dem `import` Befehl bekannt gemacht und können anschließend verwendet werden. Werden innerhalb eines Programms Qt-Klasse verwendet, sind dies i.d.R. entsprechende vorkompilierte C++ Objekte, die über SIP gebunden werden. Erst wenn Spezialisierungen von Qt-Klassen verwendet werden, wird Python interpretiert. Dies sorgt für einen Geschwindigkeitsvorteil für Qt-Klassen und erklärt, warum grafische Anwendungen unter Python mit einer ausreichend guten Geschwindigkeit realisierbar sind.

2.6 Elan-Rahmenwerk

Das Elan-Rahmenwerk wurde innerhalb des Schwerpunktprogramms "Basissoftware für selbstorganisierende Infrastruktur für vernetzte mobile Systeme" der Deutschen Forschungsgesellschaft realisiert. Dabei wurden Python als Programmiersprache und Qt als Gui-Toolkit gewählt. Im Rahmenwerk ist eine Entwicklerschnittstelle zum Entwerfen eigener Komponenten enthalten. Dabei unterscheidet Elan zwischen Komponenten, Diensten und Accessoren. Komponenten sind Programme, die innerhalb von Elan ablaufen. Sie besitzen meistens eine eigene GUI und sind bei allen Anwendern vorhanden.
Dienste sind Programme, die gezielt von Anwendern innerhalb von Elan gestartet werden. Dienste erbringen eine Leistung, die andere Anwender im Netz verwenden können. Dienste werden innerhalb des jeweiligen Anwenderprofils vermerkt.

Innerhalb von Elan werden Dienste in `spezifiziert` und `unspezifiziert` unterteilt. Für die unspezifizierten Dienste ist eine weitere Unterteilung in `Unspezifiziert-Implizite` und `Unspezifiziert-Explizite` Dienste vorhanden.

Die Klassifizierungen haben folgende Bedeutung:

- "`Spezifizierte Dienste` sind Dienste, die von einem bestimmten Mitglied des Netzes entweder explizit angeboten werden oder deren Inhalt von Anbieter zu Anbieter variiert." [Fer03]
- "`Unspezifizierte` Dienste sind Dienste, die mehrmals vorhanden sein können, wie z.B. ein allgemeiner Druckdienst oder ein Konverter. Ihr besonderes Merkmal ist die automatische Verteilung ihrer Information im Netz, die nur davon abhängt, ob ein Dienst läuft oder nicht." [Fer03]
  - "`Unspezifiziert-Implizite` Dienste sind austauschbare Dienste: Ein Konverter, der z.B. PostScript in reinen Text umwandelt, kann auf mehreren Maschinen laufen." [Fer03]
  - "`Unspezifiziert-Explizite` Dienste bieten im Netz jeweils die gleiche Funktionalität, sind aber auf ihre spezielle Maschine beschränkt." [Fer03]

In dieser Arbeit werden ausschließlich spezifizierte Dienste von ELAN verwendet, da Gruppenanwendungen eine gleiche Funktionalität bereitstellen. Aber auf eine Maschine beschränkt sind.

Accessoren sind Programme, die auf Dienste zugreifen. Sie starten, wenn ein Anwender einen Dienst gezielt nutzen möchte. Dies geschieht aus der *AwarenessView* heraus. Der Zusammenhang von Dienst und Accessor wird als das 'Dienst und Accessor Konzept' bezeichnet.

2.6.1 Lanrouter

Der *Lanrouter* ist ein eigenständiger Prozess und wird außerhalb von ELAN verwendet. Er bildet ein Ad-hoc Netz auf lokaler Netzwerkebene nach. Dazu sucht er nach anderen *Lanroutern* innerhalb seines Netzbereiches. Findet er einen solchen, kann zwischen den einzelnen Stationen kommuniziert werden. Der *Lanrouter* kann als Alternative für das ELAN-Rahmenwerk eingesetzt werden, dadurch ist es nicht nötig ein Ad-hoc Netz zu realisieren. Anwendungen können dadurch schneller entwickelt werden. Der *Lanrouter* wurde im Rahmen der Diplomarbeit [Fer03] entwickelt und kam für sämtliche Entwicklungen dieser Arbeit zum Einsatz.

2.6.2 Architektur von Elan

Abbildung 2.1 zeigt die einzelnen Schichten, in denen ELAN entworfen wurde. Zu oberst befindet sich die Anwendungsschicht. Innerhalb dieser werden die verschiedenen Komponenten, Dienste und Accessoren von ELAN ausgeführt. Anwendungen stehen den Anwendern durch Nutzung des *Application Component Framework*, das grafische Rahmenwerk, zur Verfügung. Innerhalb dieser Schicht sind die zentralen Anwendung von ELAN angesiedelt. Die Anwendungsschicht bildet die Grundlage für die in dieser Arbeit entwickelten Applikationen. Der Vollständigkeit halber soll einer kurzer Überblick über die weiteren Schichten der ELAN-Architektur gegeben werden.

Die *Middleware* kümmert sich um den Versand der sogenannten Awareness-Profile. Awareness-Profile spiegeln das Interesse und das Wissen eines Anwenders wieder. Das Awareness-Profil wird durch die Middleware im Netz verteilt und mit anderen Profilen verglichen. Erscheint auf Grund des Vergleichs ein Profil als interessant, wird es an die Anwendungsschicht übergeben.
Die Netzwerkschicht *Application Triggered Routing* ist dafür verantwortlich, Routen zwischen Sender und Empfänger einer Nachricht zu finden. Ist eine Route gefunden wird die Nachricht mit Hilfe des `Linux - IP Network Stack` versendet.

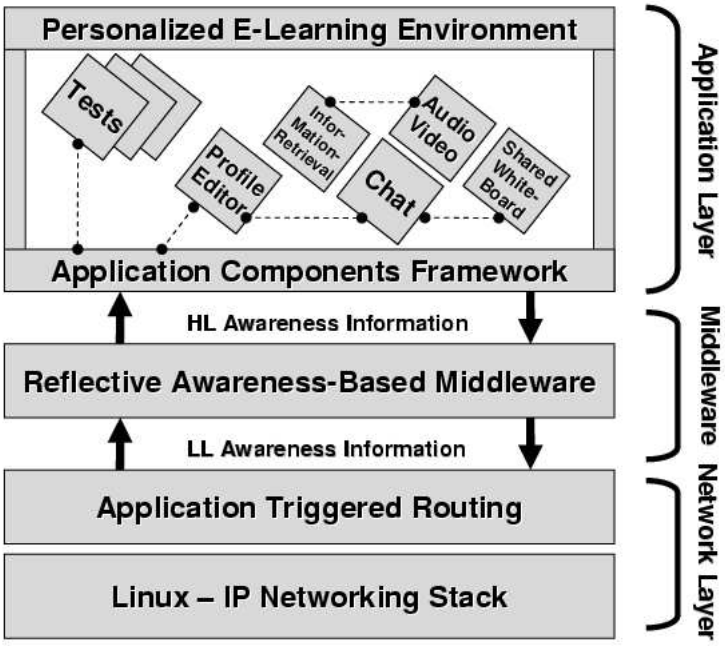

Abbildung 2.1: Die Architektur von ELAN

2.6.3 Wilied

Der *Wilied* ist eine Komponente von ELAN und steht jedem Nutzer zur Verfügung. Im *Wilied* kann der Nutzer sein Interessens- und Wissensprofile erzeugen und hinterlegen. Dies geschieht derart, daß der Anwender den *Wilied* als aktive Komponente auswählt und in einer Baumstruktur die möglichen Gebiete angezeigt bekommt. In jedem Bereich kann der Nutzer dann Wissen oder Interesse mit einem Prozentwert auswählen. Die Summe aller Werte der verschiedenen Bereiche bilden das Interessens- und das Wissensprofil innerhalb von ELAN. Profile werden durch das Netz verschickt. Dabei werden eintreffende Profile mit dem eigenen verglichen und entschieden, ob das fremde Profil für den Anwender von Interesse sein könnte. Ist dies der Fall, wird es in der *AwarenessView* angezeigt. In einem Profil können zusätzliche Informationen über verfügbare Dienste enthalten sein. Zu diesem Zweck können, innerhalb des Profileditors, Dienste gestartet und somit angehängt werden. Jeder Dienst läuft als eigenständige Komponente innerhalb von ELAN. Detaillierte Informationen über den Profileditor, das Vergleichen von Profilen und den Transport dieser ist

unter [Wor03, Fer03] zu finden.

2.6.4 AwarenessView

Die *AwarenessView* ist ebenfalls eine Komponente von ELAN und steht jedem Nutzer zur Verfügung. Die Aufgabe der *AwarenessView* ist es, auf mögliche interessante Anwender im eigenen Umfeld aufmerksam zu machen. Trifft ein fremdes Profil ein und wird dieses als interessant eingestuft, zeigt es die *AwarenessView* in übersichtlicher Art an. Dabei löst die *AwarenessView* einen Signalton aus, um auf die eingetretene Veränderung aufmerksam zu machen. Alle interessanten Anwender, auch der Eigene, werden durch ihr Bild, ihren Nickname (Rufnamen), einen Prozentwert, der die Übereinstimmung der Profile wiederspiegelt und die Zeitspanne der letzten Aktualisierung in Sekunden dargestellt.
In einem zweiten Fensterabschnitt werden alle gefundenen Dienste nach Servicename, Typ und enthaltener Metadaten angezeigt. Dies ist innerhalb der *AwarenessView* eine wichtige Darstellung, denn auf diese Art und Weise erfahren Anwender über angebotene Dienste innerhalb des Netzes. Auch kann hier gezielt auf einen Dienst zugegriffen werden. Dazu drückt der Anwender die rechte Maustaste über dem gewünschten Dienst und wählt den Menüpunkt 'mache etwas' aus. Das ELAN-Rahmenwerk kontaktiert dann den Dienst, mit Hilfe seiner IP-Adresse, sucht nach dem aktuellen Accessor-Programm, überträgt dieses gegebenenfalls über das Netz und startet es lokal. Anschließend steht der Accessor dem lokalen Anwender als Zugriffsprogramm zur Verfügung. Der detaillierte Ablauf inkl. aller Nachrichten und der Aufbau der *AwarenessView* kann bei [Wor03, Fer03] nachgelesen werden.

2.6.5 Weitere Elan-Komponenten

Innerhalb des ELAN-Rahmenwerks sind zusätzlich weitere Komponenten enthalten. Diese sind für die Entwicklung und das Verständnis der entstandenen Arbeit nur bedingt von Bedeutung. Die wichtigsten sollen aber der Vollständigkeit halber kurz hier erläutert werden. Als hilfreich haben sich für Verständnis und Entwicklung erwiesen:

- *About* - zeigt eine Informationsseite über ELAN,
- *Log* - zeigt ein Logfile für Aktivitäten im Rahmenwerk,
- *Config* - der Anwender kann Informationen über sich hinterlegen und einen Schwellwert einstellen, ab wann andere Profile für ihn interessant sind,
- *Python* - ein Python-Interpreter innerhalb von ELAN für Tests,
- *Middleware* - zeigt alle Aktivitäten der Middleware an,
- *Komponenten* - zeigt alle laufenden Komponenten und Informationen über diese an.

2.6.6 Entwicklerschnittstelle

Im folgenden Abschnitt werden die verwendeten Schnittstellen beschrieben, die zur Entwicklung durch das ELAN-Rahmenwerk bereitgestellt werden. Diese Schnittstellen wurden

verwendet, um die in Kapitel 3 und Kapitel 4 beschriebenen Anwendungen zu realisieren. Mit Hilfe dieser Schnittstellen können Komponenten, Dienste und Accessoren erzeugt werden.

Die Klasse EComponent

Die Klasse `EComponent` bildet die gemeinsame Grundlage für Komponenten, Dienste und Accessoren. Schnittstellen werden in Python erzeugt, indem von der definierenden Klasse eine Spezialisierung erzeugt wird und dabei Methoden überladen werden. Diese Methoden besitzen innerhalb der Basisklasse meist nur Quellcode mit einem einzigen Befehl - `pass`. Erbt ein Programm von der Klasse `EComponent`, kann es dabei verschiedene Variablen verändern. Diese sind im Einzelnen:

- `name` - ein String, der den in ELAN verwendeten Namen enthält,
- `spec` - ein Float, der die Spezifikation enthält, nach der das Programm realisiert wurde,
- `version` - ein Float, der die Versionsnummer des Programms enthält,
- `author` - ein String, der den Namen des Autors enthält,
- `purpose` - ein String, der eine Beschreibung des Programms enthält,
- `gui` - ein String mit den Werten 0 oder 1, der angibt, ob das Programm ein GUI verwendet und
- `service` - ein String, mit den Werten unspecified-explicit, unspecified-implicit oder specified, legt die Dienstart fest.

Der Sinn der Variablen sollte durch die Kurzbeschreibung ausreichend erklärt sein. Die in Kapitel 2 und Kapitel 3 entwickelten Komponenten sind spezifizierte Dienste.
Die wichtigsten Methoden der Entwicklungsschnittstelle sind:

- `bootGui` - wenn das Programm eine GUI verwenden möchte, erzeugt es in dieser Methode sein Widget und gibt dies mit Hilfe der `return` Anweisung zurück,
- `shutdown` - wenn das Programm Aufgabe vor dem Beenden erledigen möchte,
- `canConfigure` - falls das Programm einen Konfigurationsdialog anbietet,

Zusätzlich sind folgende Punkte zu beachten, um eine Komponente in ELAN einzubinden:

- Im Unterverzeichnis 'components' muß ein eigenes Verzeichnis für jedes Programm existieren,
- der Dateiname des Programms, mit der zusätzlichen Endung `.py`, muß mit dem Verzeichnisnamen der Komponente übereinstimmen.

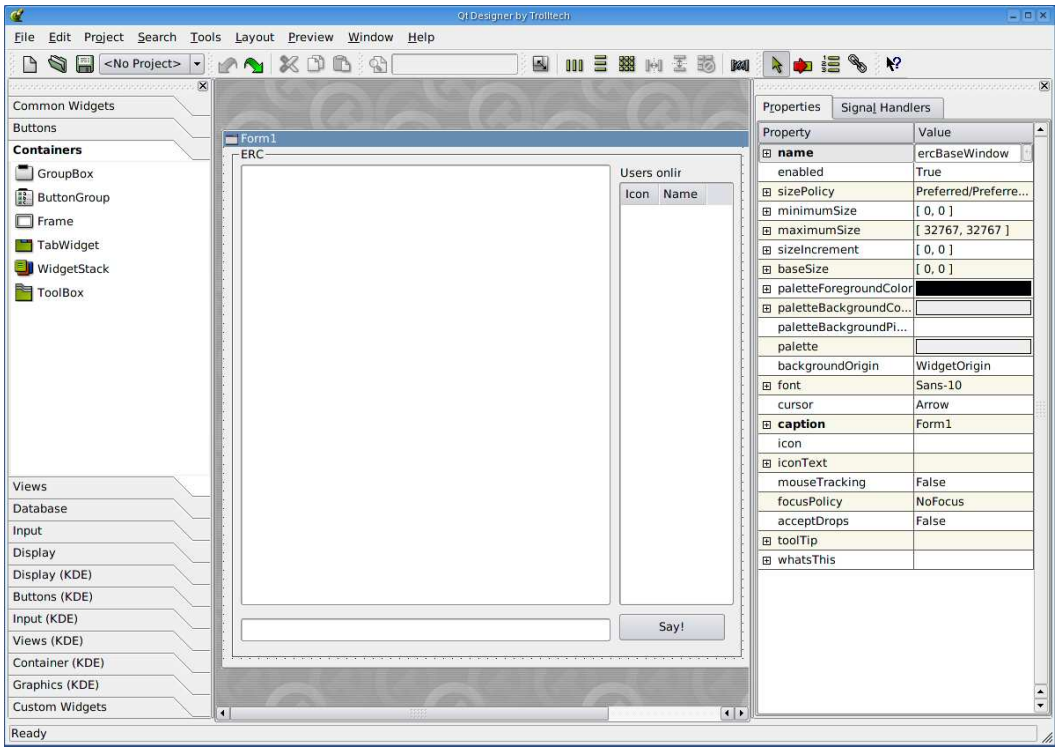

Abbildung 2.2: Qt-Designer mit geöffnetem ERC User-Interface

Ext In&Out Kommunikation

Innerhalb von ELAN kümmert sich das *Lego*[5] um die Zustellung von Nachrichten. *Lego* kennt verschiedene Nachrichten, davon ist vor allem die `Ext In&Out` Kommunikation für das Verständnis von Interesse. Eine vollständige Auflistung und Erklärung aller *Lego*-Nachrichten und deren Bedeutung ist unter [Fer03] zu finden.
Enthält eine Komponente im ELAN-Rahmenwerk eine Methode mit dem Namen `lego_EXTIN`, wird dies vom Rahmenwerk automatisch bemerkt. Diese Methode dient als Schnittstelle zum Zustellen von Nachrichten, die von anderen Komponenten gesendet wurden. Auf der Schnittstelle treffen dabei sämtliche Nachrichten ein, auch solche die nicht für die Komponente bestimmt sind. Möchte ein Programm über die Schnittstelle Nachrichten empfangen, muß es die Methode implementieren und dabei die Nachrichten für sich herausfiltern.
Zum Versenden von Nachrichten wird die Methode `post()` verwendet. Diese erwartet als Parameter den Empfängernamen, die Empfänger IP-Adresse, den Absendernamen, die Absender IP-Adresse und eine Nachricht. Schickt ein Programm eine Nachricht mit Hilfe der Methode `post()` an ein anderes Programm, trifft die Nachricht beim Empfänger über die `lego_EXTIN()` Methode ein. Kann die Nachricht von der Middleware nicht zugestellt werden, geht sie verloren.

[5] Lightweight Extensible Group Objects

Econf

Econf bietet innerhalb von ELAN die Möglichkeit an, Daten zu speichern und wieder herzustellen. Auf diese Art und Weise können Konfigurationen der einzelnen Programme erzeugt werden. Dazu legt *Econf* im Heimatverzeichnis des Nutzers das unsichtbare Unterverzeichnis `.elan` an. In diesem Verzeichnis legt *Econf* für jedes Programm von ELAN eine eigene Datei ab. Diese wird nach dem Namen des Programms benannt und besitzt die Endung `.pkl`. Innerhalb dieser Datei werden alle Daten abgespeichert, die vom Programm bei *Econf* bekannt gemacht werden. Intern verwendet *Econf* den so genannten Pickler von Python, der eine Serialisierung von Standarddatentypen erlaubt. Auf diese Art und Weise können Variablen inkl. der enthaltenen Daten in einer Datei abgespeichert werden und mit Hilfe von Unpickle zu einem beliebigen Zeitpunkt wieder hergestellt werden. Eine genaue Beschreibung über Pickle und Unpickle kann unter [Mar03] gefunden werden.
Econf stellt Daten automatisch beim Start der Komponente durch ELAN wieder zur Verfügung und speichert diese, sobald das Programm oder ELAN geschlossen werden. Daher kann der genaue Ablauf von Pickle und Unpickle vernachlässigt werden. *Econf* steht zum Zugriff als Dictionary[6], unter dem Namen `Econf`, zur Verfügung. Möchte eine Komponente eine Variable dauerhaft abspeichern, muß dazu ein Eintrag innerhalb des `Econf`-Directory vorgenommen werden.
Econf wird für Konfigurationsdialoge innerhalb der Programme verwendet. Die in *Econf* gespeicherten Werte werden dazu in einem Dialogfenster angezeigt, durch den Anwender verändert und wieder in *Econf* abgespeichert.

2.7 Eine erste Elan-Komponente - ERC

In diesem Abschnitt wird die Entwicklung einer ersten ELAN-Komponente beschrieben. Dabei wurde ein einfacher Kommunikationsansatz gewählt. Ziel der Entwicklung war, erste Erfahrungen im Umgang mit dem Rahmenwerk zu sammeln, als auch erste Gruppenkommunikationsversuche zu unternehmen. Daher wurde als Anwendung ein Multiuserchat ausgewählt - der Elan Relay Chat, kurz ERC. Dabei können mehrere Anwender miteinander kommunizieren und sich auf dieser Basis untereinander austauschen. Anforderungen an den ERC:

- Es gibt eine Anwenderliste, die darstellt, welche Personen aktiv teilnehmen,
- eine Historie zeigt das bisher Geschriebene an,
- Texte werden in einer Eingabezeile eingegeben,
- eine einfache Lösung für den Nachrichtenversand soll verwendet werden.

2.7.1 Entwurf

Die Anforderungen wurden bewusst einfach gehalten, um die Einarbeitungszeit in das ELAN-Rahmenwerk überschaubar zu halten. Die drei beschriebenen Elemente *Anwenderliste*, *Historie* und *Eingabezeile* beschreiben das User-Interface der Anwendung. Eine

[6] eine assoziative Datenstruktur in Python

Realisierung dieser Elemente im Qt-Designer erfolgte. Gibt ein Anwender, der den ERC gestartet hat, einen Text in die Eingabezeile ein und drückt die Taste 'Return' oder drückt alternativ den Druckknopf 'Say!', wird der Text durch das Rahmenwerk verschickt und taucht in der Historie aller Teilnehmer auf.
Als schwieriger erwies sich der Nachrichtenversand, denn die einzelnen Instanzen des ERC kennen sich untereinander nicht und müssen trotzdem in der Lage sein, Nachrichten zu verschicken bzw. zu empfangen. Die `post()` Methode des Rahmenwerks kennt allerdings nur die Möglichkeit, Nachrichten von einem Absender an einen Empfänger zu verschicken - siehe Abschnitt 2.6.6. Als zusätzliche Adressierung mehrerer Empfänger wurde daher '<all>' als gültige Adresse realisiert. Die Idee dabei ist, daß eine Nachricht, die an die Adresse '<all>' versendet wird, an alle vom Rahmenwerk als *interessant* (im Sinn der Interessensprofile) eingestuften Teilnehmer verschickt wird. Dabei ist es unwichtig, ob diese den ERC verwenden.

2.7.2 Implementierung

Das User-Interface des ERC wurde mit Hilfe des Qt-Designer, wie in Abbildung 2.2 dargestellt, zusammengestellt. Dabei wurden verschiedene Instanzen von Qt-Klassen erzeugt. Für die *Historie* eine Instanz von der Klasse `QTextEdit`, für die *Eingabezeile* von der Klasse `QLineEdit` und für die *Anwenderliste* von `QListView`. Zusätzlich wurde ein Druckknopf, beschriftet mit 'Say!', eine Instanz von `QPushButton` hinzugefügt. Dieses Widget wurde abgespeichert und die entstandene Datei mit der Endung `.ui` mit dem Kommando `pyuic` in Python Quellcode übersetzt. Dadurch entstand die Klasse `ercBaseWindow`, welche die Grundlage für das Hauptfenster der Anwendung bildet. Bei späteren Programmen - siehe Kapitel 2 sowie Kapitel 3 - wird von der durch den Qt-Designer erstellten Klasse eine zusätzliche Klasse geerbt, um den Widgets zusätzliche, nicht mit dem Qt-Designer realisierbare, Eigenschaften zu geben. Dies ist beim ERC nicht nötig.
Die Hauptklasse `erc` erbt von der Klasse `EComponent` und überlädt dabei die Methode `bootGUI`. Innerhalb dieser Methode wird eine Instanz von `ercBaseWindow` erzeugt, das Hauptwidget vom ERC. Die einzelnen Elemente des Widgets müssen noch mit passenden Signalen und Slots verbunden werden. Die Signale `clicked()` des Druckknopfs und `returnPressed()` der *Eingabezeile* werden durch die Methode `connect()` von Qt mit dem Slot `slotSendMsg` verbunden.
Wird der Slot `slotSendMsg` gerufen, liest dieser die eingegebene Nachricht aus der *Eingabezeile* aus und schickt diese zusammen mit dem Rufnamen des Anwenders, mit Hilfe der Methode `post()`, an alle anderen Teilnehmer. Dabei wird der ERC als Absender- und Empfängerprogramm eingetragen, als Empfänger Adresse '<all>' verwendet und der Text und Rufname, in einem Tupel verpackt, als Nachricht angehängt. Als Nachrichtenkommando wird `newMessage` angegeben.
Die Nachricht wird durch das Rahmenwerk zugestellt und trifft über die `lego_EXTIN` Methode bei ERC Instanzen ein. Diese überprüfen jede eintreffende Nachricht mit Hilfe des Empfängernamen und rufen, falls die Nachricht für sie ist, intern das Nachrichtenkommando als Methode. Dies bedeutet beim ERC, daß die Methode `newMessage` gerufen wird. Dieser Vorgang wird dispatchen genannt und wird in Kapitel 3 und 4 ebenfalls verwendet.
Die Methode `newMessage` extrahiert aus dem Tupel die enthaltene Nachricht und den Rufnamen des Absenders. Der Rufname wird fett geschrieben und vor der Nachricht pla-

ziert. Beides zusammen wird anschließend als neueste Nachricht in die *Historie* angehängt. Der Rufname des Absenders wird dabei in die *Anwenderliste* aufgenommen. Dies sieht so aus, als wären die Anwender dem Programm bekannt, sind sie aber in Wirklichkeit nicht.

2.7.3 Diskussion

Der ERC ist eine sehr einfache Anwendung und diente nur dem Zweck der Einarbeitung in das ELAN-Rahmenwerk. Die erläuterte Implementierung geht dabei nur auf die wichtigsten Aspekte ein. Es wurden zusätzlich weitere Dinge wie z.B. eine `ErcUser` Klasse implementiert, diese dienten aber nur für eigene Tests und sind von keiner Bedeutung für die nachfolgenden Kapitel.
Die gewählte Kommunikationslösung kann nicht gezielt mit einer Gruppe kommunizieren. Zwar können Anwender miteinander kommunizieren, aber nur wenn sie durch das Rahmenwerk und ihre Interessensprofile gegenseitig bekannt sind. Das Kommunikationsprotokoll des ERC besteht aus einer einzigen Nachricht. Dadurch kann keine Sicherheit in der Zustellung und kein Marshalling zur sinnvollen Bandbreitennutzung realisiert werden. Diese Aspekte sollen aber berücksichtigt werden und machen komplexere Protokolle nötig. Der gewählte Kommunikationsansatz erzeugt außerdem einen Overhead an Nachrichten. Die Anzahl resultierender Nachrichten, die durch den `post()` Methodenaufruf im Netz verschickt werden, kann sehr groß werden. Denn Nachrichten werden an alle ELAN Anwender verschickt, die als interessant eingestuft wurden. Also auch an diejenigen, die den ERC nicht einmal gestartet haben. Es gilt daher einen effizienteren Weg zu finden, Nachrichten zwischen Teilnehmern zu verteilen.

2.8 Weiterführende Begriffe

Die Entwicklung des ERC hat verschiedene Aspekte aufzeigen können. Dabei treten zwei zentrale Begriffe in den Vordergrund, die im weiterem Verfaul der Arbeit immer wieder von Bedeutung sein werden. Daher soll hier bereits eine kurze Erläuterung dieser Begriffe gegeben werden.

Gruppenkommunikation Unter dem Begriff Gruppenkommunikation verstehen wir im Nachfolgenden, daß eine Menge gleicher Programme innerhalb des ELAN-Rahmenwerks gezielt miteinander Informationen austauschen können. Dieser Austausch erfolgt durch Nachrichten, die zwischen den einzelnen Instanzen der Programme verschickt werden. Dabei ist eine gezielte Verbreitung der Nachrichten, bei geringem Overhead, ebenfalls Aufgabe der Gruppenkommunikation.

Gruppenverwaltung Der Begriff Gruppenverwaltung beschreibt, daß die einzelnen Mitglieder zur Gruppenkommunikation bekannt sein müssen. Nur dadurch ist es möglich, daß eine Grupppe kommunizieren kann. Dabei können verschiedene Ansätze zur Gruppenverwaltung genutzt werden. Ein Gruppenverwalter, der alle Mitglierder kennt oder aber Gruppernmitglieder die ausgewählte Teile der Gruppen kennen. Die Gruppenverwaltung bildet die Grundlage der Gruppenkommunikation.

Das folgende Kapitel beschäftigt sich mit dem Entwurf und der Realisierung eines Dateifreigabedienstes. Dieser verwendet nur indirekt eine Gruppenverwaltung, indem der implementierte Dienst alle Accessoren kennt, die auf ihn zugreifen. Gruppenkommunikation findet auch bei der Dateifreigabe statt, ist aber auf 1:N Kommunikation beschränkt.

Kapitel 3

Dateifreigabedienst

In diesem Kapitel soll eine erste Gruppenanwendung innerhalb des ELAN-Rahmenwerks entworfen und realisiert werden. Der Fokus der Entwicklung liegt dabei auf dem Dienst und Accessor Konzept, welcher vom Rahmenwerk bereitgestellt wird.

3.1 Anforderungen an die Dateifreigabe

Ziel ist der Entwurf einer Komponente für das ELAN-Rahmenwerk, die es ermöglicht, Dateien zwischen einzelnen ELAN Anwendern auszutauschen. Dabei sollen die zuvor erkannten Probleme aus der Entwicklung des ERC in Kapitel 2 umgangen bzw. gelöst werden. Um dies zu erreichen, wird zur internen Kommunikation ein eigenes Protokoll implementiert. Ebenso wird das interne Konzept von ELAN 'Dienst und Accessor' [Fer03] verwendet, um eine eindeutige Aufgabenteilung zu erreichen, wie sie in Kapitel 2 eingeführt wurden.

ELAN unterstützt die Kommunikation von Anwendern, deren Lern- und Wissensprofile gut aufeinander passen. Hierzu werden innerhalb der *AwarenessView* dem Nutzer andere Personen im Umkreis angezeigt, die für ihn von Interesse sein könnten. In dieser Ansicht werden auch die dazugehörigen Dienste dargestellt. Abbildung 3.1 zeigt die *AwarenessView* mit dem angebotenen Dienst 'sharing'. Interessierte Nutzer können gezielt auf Dienste zugreifen und damit den Accessor auf ihrer Plattform starten. Folgende Anforderungen sind für den Dateifreigabedienst bzw. Accessor gefunden worden. Wir unterteilen in funktionale und nichtfunktionale Anforderungen.

- Die funktionalen Anforderungen sind:
  - Eine gezielte Bereitstellung von Dateien und Verzeichnissen,
  - das Entfernen von Dateien und Verzeichnissen aus der Bereitstellung,
  - die Übertragung einer Datei soll im Accessor veranlasst werden und,
  - mehrere Anwender sollen gleichzeitig Dateien empfangen können.

- Die nichtfunktionalen Anforderungen sind:
  - Die Datenübertragung soll zuverlässig sein,
  - voneinander getrennte Darstellungs- und Verwaltungslogik und,
  - laufende Übertragungen und bereitgestellte Dateien werden übersichtlich dargestellt.

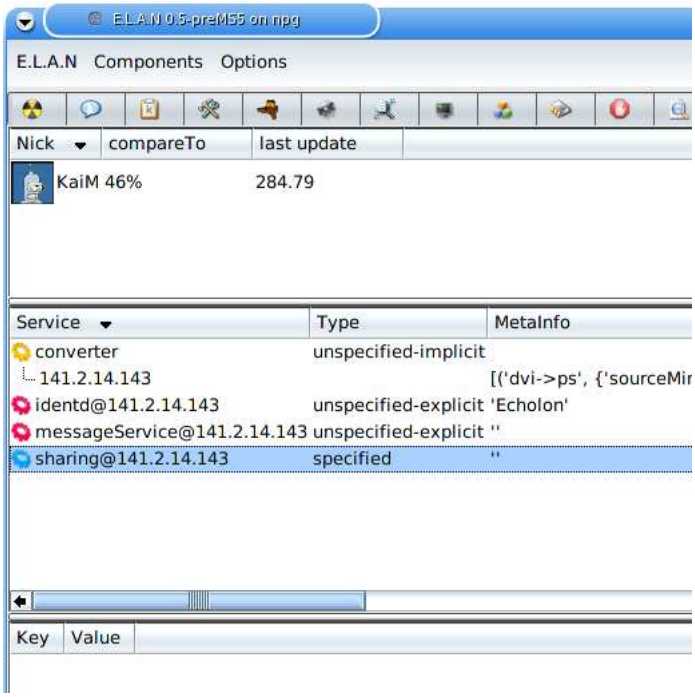

Abbildung 3.1: *AwarenessView* mit Dateifreigabedienst

3.2 Entwurf

Um die genannten Anforderungen zu realisieren, bietet sich eine Aufteilung in zwei wesentliche Teile an:

1. Die Ansicht und Verwaltung der bereitgestellten Dateien und,
2. die Übertragung der gewünschten Dateien inkl. Ansicht.

Die beiden genannten Punkte werden in einer Klassenhirachie entworfen. Dabei werden Gemeinsamkeiten von Dienst und Accessor herausgearbeitet und als gemeinsame Basisklassen realisiert. Daher werden im Folgenden jeweils die Gemeinsamkeiten besprochen, anschließend folgen die jeweiligen Spezialisierungen für Dienst und Accessor.

Abbildung 3.2 zeigt, die Klassenhierachie für den Dateifreigadienst als UML Diagramm. Abbildung 3.3 stellt die Klassenhierachie für den Dateifragebeaccessor dar. Qt-Klassen sind in beiden Diagrammen jeweils dunkler dargestellt. Unter Berücksichtigung beider

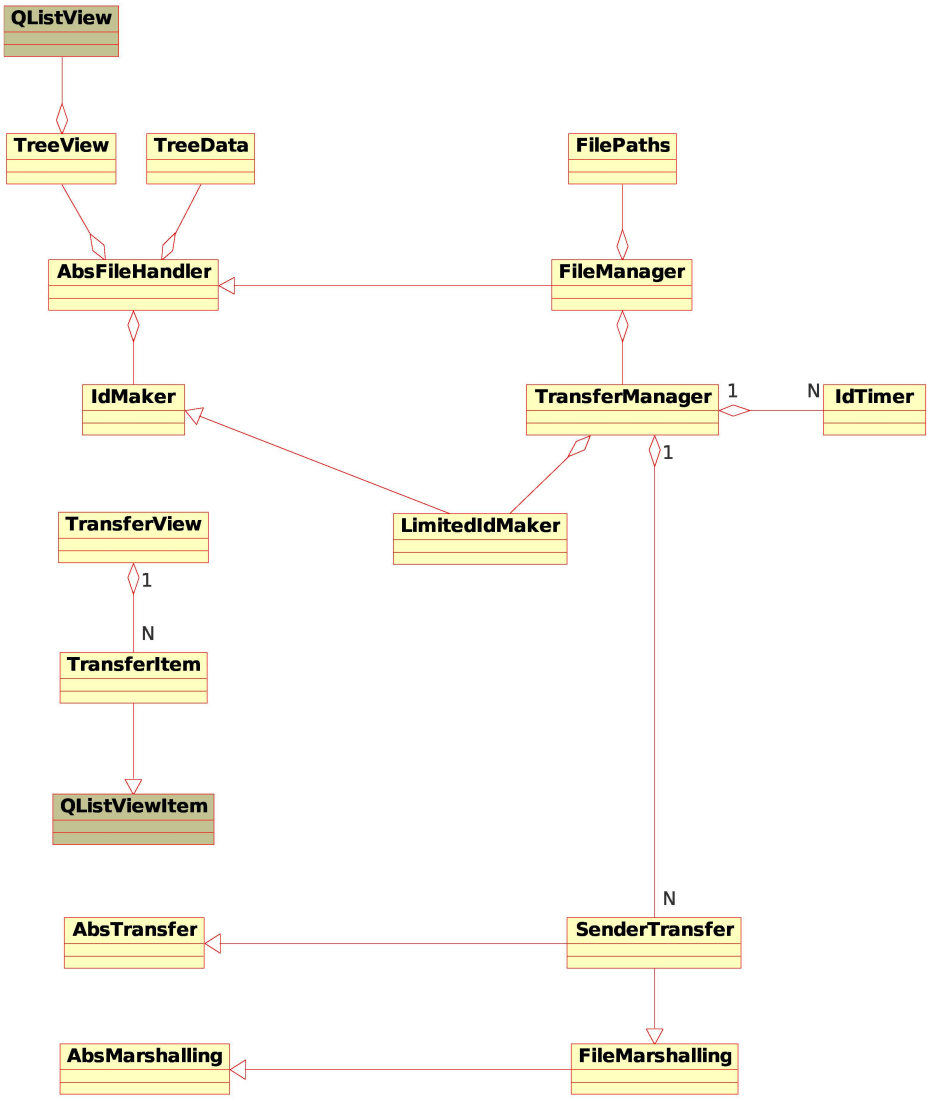

Abbildung 3.2: UML Klassendiagramm für den Dateifreigabedienst

Abbildungen kann entnommen werden, welche Klasse gemeinsam genutzt werden. Die im Entwurf vorgesehene Funktionalität wird in den folgenden Abschnitten erläutert.

3.2.1 Gemeinsame Funktionalität der Dateiansicht

Der Dienst muß dem Anwender die Verwaltung der bereitgestellten Dateien zu ermöglichen. Der Accessor muß die entfernt bereitgestellten Dateien anzeigen. Dies bedeutet, daß Dienst und Accessor als gemeinsame Grundlage die Ansicht besitzen. Die Gemeinsamkeiten von Dienst und Accessor können in einer abstrakten Klasse realisiert werden. Diese Klasse realisiert die Dateiansicht, kann aber nicht direkt instanziiert werden. Eine abstrakte Klasse muß durch Vererbung zu einer konkreten Klasse ausgeprägt werden. Somit verwenden Dienst und Accessor zwar unterschiedliche Klassen, welche aber auf einer ge-

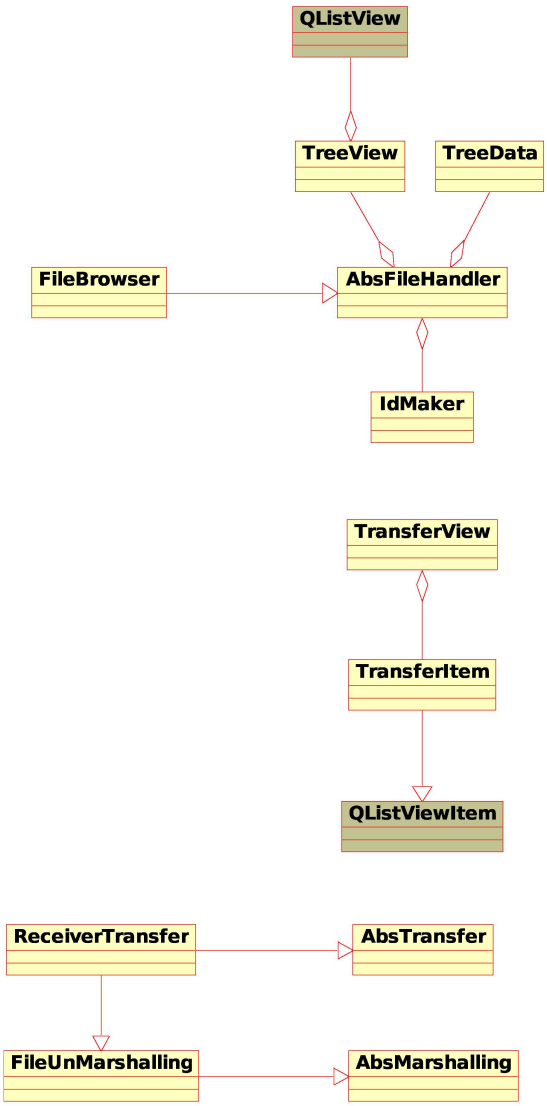

Abbildung 3.3: UML Klassendiagramm für den Dateifreigabeaccessor

meinsamen Funktionalität beruhen. Die abstrakte Klasse `AbsFileHandler` bildet die Basis für die Klasse `FileManager` beim Dienst und die Klasse `FileBrowser` beim Accessor.
Für die Ansicht der bereitgestellten Dateien wird eine Baumansicht gewählt. Dateibäume bieten eine übersichtliche Darstellung und kommen in vielen verbreiteten Programmen vor. Die Knoten in der Baumdarstellung sind entweder Dateien oder Verzeichnisse. Enthalten Verzeichnisse Dateien, werden diese unterhalb des Verzeichnisses etwas eingerückt und mit einer Linie verbunden dargestellt. Außerdem gibt es durch Plus- und Minussymbole vor Verzeichnissen die Möglichkeit, die Ansicht der enthaltenen Dateien ein- und auszuschalten. Die Trennung von Daten und Ansicht ist eine der gestellten Anforderungen, daher verwendet die Klasse `AbsFileHandler` die beiden Klassen `TreeData` und `TreeView` für diese Aufgabe.
Damit jede bereitgestellte Datei eindeutig identifiziert werden kann, benötigt diese ei-

ne eindeutige Kennung. Zu diesem Zweck wird die Klasse `IdMaker` innerhalb von `AbsFileHandler` verwendet.

TreeData

Aufgabe der Klasse `TreeData` ist es, die darzustellende Dateibaumstruktur in ein Format aus Standarddatentypen zu abstrahieren. Jeder Knoten des Dateibaum muß innerhalb der Datenstruktur die nötigen Informationen enthalten, um eine Darstellung daraus erzeugen zu können. Die verwendete Datenstruktur bildet die gemeinsame Grundlage für Dienst und Accessor. Auf Basis dieser Datenstruktur erfolgt der Austausch über die bereitgestellten Dateien. Es dürfen dabei keine Daten enthalten sein, die für den Accessor nicht von Belang sind. Aus Sicherheitsgründen sollten daher nur die relevanten Informationen zur Darstellung des Dateibaums gespeichert werden.
Zum Speichern der Knoten bietet sich eine assoziative Datenstruktur an, die mit einer eindeutigen Kennung versehen wird. Die Kennung wird mit Hilfe der Klasse `IdMaker` erzeugt. Es werden Methoden zum Hinzufügen und Entfernen von Knoten angeboten und durch `AbsFileHandler` verwendet.

TreeView

Die Klasse `TreeView` ist die Schnittstelle zur Ansicht und erzeugt aus den in `TreeData` hinterlegten Knoten den Dateibaum auf dem Bildschirm. Qt bietet die Klasse `QListView` an, mit welcher auf einfache Weise Baumstrukturen dargestellt werden können. Mit Hilfe der Qt-Klasse `QListViewItem` können dann Knoten erzeugt werden. Ebenfalls kann ein grafisches Symbol und ein Name problemlos den Knoten zugeordnet werden. Methoden zum Hinzufügen und Entfernen von Knoten werden angeboten.

IdMaker

Die Aufgabe der Klasse `IdMaker` ist die Verwaltung einer unendlichen Menge von Zahlen. Auf Anfrage muß die Klasse `IdMaker` eine Zahl zurückgeben, die bisher noch nicht vergeben wurde. Außerdem soll es möglich sein, nicht mehr benötigte Zahlen wieder zu löschen. Immer wenn ein neuer Knoten in der Ansicht hinzugefügt wird, ist eine eindeutige Kennung für die bereitgestellte Datei nötig. Diese wird von der Klasse `AbsFileHandler` bei `IdMaker` angefordert. Diese Kennung wird ebenfalls zur Speicherung des Knotens und in der Klasse `TreeView` verwendet.

3.2.2 Spezialisierungen der Dateiansicht für den Dienst

Der Dienst benötigt nicht nur eine Ansicht der bereitgestellten Dateien, er muß diese ebenfalls verwalten können. Die Verwaltung der bereitgestellten Dateien realisiert der Dienst durch einen Dateimanager. Dieser verwendet die gemeinsame Grundlage für die Dateiansicht der Klasse `AbsFileHandler` und erweitert diese. Dies ist nötig, weil dem Anwender des Dienstes zusätzliche Möglichkeiten gegenüber den Anwendern der Accessoren angeboten werden. Wie z.B. das Löschen der Dateien und Verzeichnisse, welche in die Bereitstellung aufgenommen wurden, Selektionsmöglichkeiten für die Einfügoptionen

der bereitgestellten Dateien im Dateibaum und die Option nutzerspezifische Einstellungen vornehmen zu können. Außerdem wird eine Verwaltung der Dateipfade für bereitgestellte Dateien benötigt.

Konfiguration

Der Konfigurationsdialog kann mit Hilfe von *Econf* (siehe Kapitel 2) realisiert werden. Damit der Anwender die gespeicherten Werte komfortabel ändern kann, wird ein Dialogfenster verwendet. Folgende Einstellungen sollen vorgenommen werden können:

- Festlegung eines Verzeichnisses, welches bei Start automatisch freigegeben wird,
- Festlegung der Sekunden bis zum Timeout bei der Dateiübertragung, sowie
- Festlegung der maximalen Anzahl der gleichzeitig stattfindenden Übertragungen.

FilePaths

Der Accessor benötigt im Gegensatz zum Dienst die Information über Pfade der bereitgestellten Dateien nicht. Der Dienst muß die Dateien zur Übertragung lesen können. Daher werden die Pfade inkl. Dateinamen separat in einer Instanz der Klasse `FilePaths` verwaltet. Dabei muß jede bereitgestellte Datei eindeutig identifizierbar sein. Dazu wird die zuvor in Abschnitt 3.2.1 genannte Kennung als Schlüssel verwendet.

FileManager

Um die erweiterten Möglichkeiten im Dateimanager für den Dienst zu realisieren, wird die Klasse `FileManager` als Spezialisierung der Klasse `AbsFileHandle` entworfen. Dabei erweitert die Klasse `FileManager` die Basisklasse um zusätzliche Methoden zum Entfernen von Dateien und Verzeichnissen sowie zum Bestimmen des Anhängpunktes im Dateibaum. Weiter verwendet der Dateimanager zur Verwaltung von Dateipfaden die Klasse `FilePaths`, mit deren Hilfe Dateipfade über ihre eindeutige Kennung erfragt werden können.

3.2.3 Spezialisierung der Dateiansicht für den Accessor

Der Accessor zeigt die entfernt bereitgestellten Dateien in einem Dateibaum an. Dazu verwendet der Accessor einen Dateianzeiger, der die Klasse `AbsFileHandle` als Basisklasse verwendet. Der Dateianzeiger erweitert die Klasse `AbsFileHandle` um die Möglichkeit, eine eintreffende Datenstruktur von `TreeData`, die einen Dateibaum widerspiegelt, in eine Baumansicht von `TreeView` zu konvertieren. Auch beim Accessor soll der Anwender Einstellungen vornehmen können.

Konfiguration

Der Konfigurationsdialog kann mit Hilfe von *Econf* realisiert werden. Folgende Einstellungen sollen vom Anwender des Accessor vorgenommen werden können:

- Festlegung des Verzeichnisses in dem empfangene Dateien gespeichert werden,
- quantitative Festlegung der Sekunden bis zum Eintreten eines Übertragungstimeout.

FileBrowser

Die Klasse `FileBrowser` bildet die Grundlage für den Dateianzeiger. Um die beschriebenen Erweiterungen für den Dateianzeiger zu erreichen, spezialisiert die Klasse `FileBrowser` die Klasse `AbsFlieHandle`. Eine zusätzliche Methode zum Erstellen neuer Dateibaum-Anzeigen wird dabei hinzugefügt. Diese erzeugt aus einer übergebenen `TreeData`-Datenstruktur einen Dateibaum mit Hilfe von `TreeView`.

3.2.4 Gemeinsamkeiten der Datenübertragung

Dienst und Accessor übertragen bereitgestellte Dateien blockweise unter Nutzung eines Übertragungsprotokolls, welches in Abschnitt 3.4 detailliert dargestellt wird. Damit der Dienst eine Datei blockweise lesen und der Accessor ein empfangene Datei blockweise zusammensetzen kann, wird ein Marshalling benötigt. Gemeinsamkeiten zwischen Accessor und Dienst können wieder in einer abstrakten Basisklasse zusammengefasst werden. Die Klasse `AbsMarshalling` wird dazu verwendet.
Dienst und Accessor sollen beide stattfindende Übertragungen übersichtlich darstellen können. Dazu wird eine Ansicht verwendet, die für jede Übertragung eine Zeile anzeigt. In dieser Zeile werden die einzelnen Informationen der Dateiübertragung dargestellt. Eine gemeinsame abstrakte Basisklasse ist auch hier zu verwenden - die Klasse `AbsTransfer`.

AbsMarshalling

Sie bildet die Grundlage für Marshalling und Unmarshalling. Benötigt werden Methoden zum Bestimmen der Blockanzahl, der Nummer des aktuellen Blocks und die Methode `next()` als leere Schnittstelle. Das Öffnen einer Datei zum Schreiben oder Lesen wird erst in den jeweiligen Spezialisierungen realisiert.

AbsTransfer

Jede Dateiübertragung wird durch ein eigenes Objekt repräsentiert. `AbsTransfer` bildet die gemeinsame Grundlage für Übertragungsobjekte beim Dienst und Accessoren. Attribute, die Informationen über eine Übertragung beinhalten, werden hier definiert.

TransferItem

Eine stattfindende Übertragung wird durch ein eigenes Darstellungsobjekt repräsentiert und erbt von der Qt-Klasse `QListViewItem`. Jedes Darstellungsobjekt basiert auf dem Marshalling und bietet dadurch die benötigten Informationen für die Anzeige. `TransferItem`-Instanzen werden von der Darstellungsklasse `TransferView` für die Ansicht verwendet. Zusätzlich wird ein spezieller Prozentbalken realisiert, der in einer festen Spalte die Methode `paint()` ersetzt. Durch das Marshalling sind die Anzahl der Blöcke sowie die aktuelle Blocknummer bekannt, der Prozentwert kann errechnet werden.

TransferView

Um eine Ansicht stattfindender Übertragungen zu realisieren, wird eine Darstellung pro Übertragung benötigt. Diese Ansicht kann zeilenweise organisiert werden und zeigt in verschiedenen Spalten weitere Informationen über die Übertragung. Die Klasse `TransferView` kümmert sich um die Darstellung der verschiedenen Zeilen. Dazu bietet sie Methoden zum Hinzufügen und Entfernen von Übertragungen an. Für jede Zeile wird eine eigene Ansichtsinstanz verwendet, die einen speziellen Prozentbalken realisiert - eine Instanz von `TransferItem`. Methoden zum Hinzufügen und Entfernen von Übertragungen sind vorhanden.
`TransferView` bietet zusätzlich eine Protokollansicht an, in der eintreffende Ereignisse als Textmeldung ausgegeben werden können.

3.2.5 Spezialisierung der Dateiübertragung für den Dienst

Im Dienst müssen die stattfindenden Dateiübertragungen verwaltet werden, denn mehrere Accessoren sollen gleichzeitig auf den Dienst zugreifen können. Die Verwaltung der Dateiübertragungen wird daher in der eigenen Klassse `TransferManager` verwaltet. Zum Erzeugen der Ansicht wird die Klasse `TransferView` verwendet, um die gewünschte Trennung von Daten und Ansicht zu erreichen. Der Übertragungsverwalter benötigt für seine Arbeit Klassen zwecks Begrenzung der gleichzeitigen Übertragungen, Marshalling und Timersteuerung der Übertragung.

Filemarshalling Damit eine bereitgestellte Datei in Blöcke zerlegt werden kann, muß diese geöffnet werden. `FileMarshalling` erbt von `AbsMarshalling` und öffnet einen übergebenen Dateipfad zum Lesen. Mit Hilfe der Datei- und Blockgröße läßt sich die Anzahl der Blöcke errechnen. Durch die Methode `next()` kann genau ein Block der Datei gelesen werden. Ist der gelesene Block der letzte, informiert die Methode `next()` darüber. Ein interner Zähler verweist auf den nächsten auszulesenden Datenblock der Datei.

LimitedIdMaker

Damit die Anzahl der gleichzeitig stattfindenden Dateiübertragungen begrenzt werden kann, wird eine Erzeugung von limitierten Kennummern durch `LimitedIdMaker` angeboten. Diese erzeugt Übertragungskennungen und kann durch das Limit dazu verwendet wer-

den, die Anzahl der stattfindenden Übertragungen zu begrenzen. Die Klasse `IdMaker` aus Abschnitt 3.2.1 wird durch Vererbung um ein Limit erweitert zur Klasse `LimitedIdMaker`.

IdTimer

Um eine Zeitsteuerung der Übertragungen zu ermöglichen, wird ein spezieller Timer benötigt. Diesem wird beim Start bekannt gemacht, von welcher Übertragung der Timer gestartet wurde. Läuft der Timer ab, kann mit Hilfe der Kennung die Übertragung identifiziert werden. Die Klasse `IdTimer` läßt sich als Spezialisierung der Klasse `QTimer` aufbauen. Dadurch kann das eintretende Ereignis an einen Slot gebunden werden.

SenderTransfer

Der Dienst verwaltet seine Übertragungen in Objekten der Klasse `SenderTransfer`. Diese Klasse erbt von `FileMarshalling` und `AbsTransfer`, und bildet dadurch die entscheidende Grundlage zur Dateiübertragung für den Dienst. Die Objekte der Klasse `SenderTransfer` werden sowohl von der Ansichtsklasse zur Darstellung, als auch vom Übertragungsverwalter für die Dateiübertragung verwendet.

TransferManager

Der Übertragungsverwalter kümmert sich um die Kommunikation mit dem Accessor. Er leitet Übertragungen ein, erzeugt dazu Instanzen von `SenderTransfer` und meldet diese bei der Ansicht an. Das Übertragungsprotokoll - siehe Abschnitt 3.4 - und die Einstellungen des Anwenders steuern den Übertragungsverwalter. Zur Limitierung von Übertragungen verwendet der Übertragungsverwalter die Klasse `LimitedIdMaker`. Die erzeugten Kennungen werden den einzelnen Übertragungen zugewiesen. Jeder Übertragung wird ein Timer zur Fehlerkontrolle zugewiesen.

3.2.6 Spezialisierung der Dateiübertragung für den Accessor

Auf einen Übertragungsverwalter kann beim Accessor verzichtet werden, denn er kann nur auf einen Dienst gleichzeitig zugreifen. Versucht der Anwender auf einen zweiten Dateifreigabedienst im Netz zuzugreifen, startet ein weiterer Accessor. Dieses Verhalten ist durch das ELAN-Rahmenwerk definiert.

FileUnMarshalling

Als Spezialisierung der Klasse `AbsMarshalling` kümmert sich `FileUnMarshalling` um das Zusammensetzen von Datenblöcken in eine Datei. Dazu wird eine Datei zum Schreiben geöffnet und eintreffende Blöcke mit der Methode `next()` an die Datei angehängt.

ReceiverTransfer

Die Klasse `ReceiverTransfer` hat die gleiche Aufgabe wie `SenderTransfer`. Sie soll alle Informationen, die zu einer Übertragung gehören, in einem Objekt vorhalten. Diese Objekte werden dann für Verwaltungs- und Anzeigelogik verwendet. Der entscheidende Unterschied zu `senderTransfer` besteht darin, daß Dateien empfangen werden, d.h. eintreffende Blöcke zusammengesetzt werden müssen. Daher bilden die Klassen `FileUnMarshalling` und `AbsTransfer` die Basis für `ReceiverTransfer`.

3.3 Implementierung

Der vorgestellte Entwurf von Dienst und Accessor der Deiteifreigabe wurde erfolgreich implementiert. In diesem Abschnitt soll die Implementierung kurz dargestellt werden.

3.3.1 Dienst

Der Dienst besteht aus Teilfenstern, die jeweils die Instanz einer Qt-Klasse zur Ansicht verwenden. Der Dateimanager und die Übertragungsansicht verwenden jeweils eine `QListView` und die Protokollansicht eine Instanz von `QTextEdit`. Das Layout der Fenster wurde mit dem Qt-Designer erzeugt und eine neue Klasse davon geerbt. Innerhalb der geerbten Klasse wurde die Dekoration für die Baumansicht angeschaltet, dadurch erscheinen Plus- und Minussymbole als auch Verbindungslinien vor Verzeichnissen. Zusätzlich wurde ein Slot implementiert, der die Methode `changeFolderIcon(item)` der Klasse `AbsFileHandler` ruft. Dieser Slot wurde mit den Signalen `expanded(QListViewItem*)` sowie `collapsed(QListViewItem*)` verbunden. Wird der Slot gerufen, wird ihm das auf- oder zugeklappte Element der Ansicht übergeben und an die Ansichtsinstanz weitergereicht. Diese erzeugt daraus die Ansicht mit geöffneter oder geschlossener Verzeichnisgrafik.
Es wird jeweils eine Instanz der Klassen `FileManager` und `SenderTransfer` erzeugt und diesen Referenzen auf die benötigten Fenster übergeben. Die im Entwurf beschriebenen Konfigurationsmöglichkeiten für den Dienst benötigen ein eigenes Dialogfenster. Das Rahmenwerk unterstüzt das Anzeigen eines Konfigurationsdialogs, dazu muß lediglich die Methode `configStructure` eine Referenz auf das Dialogfenster liefern.
Das Dialogfenster wurde mit dem Qt-Designer erzeugt. Die Klasse `sharingConfigDialog` erweitert den Dialog um Aktionen, die ausgeführt werden müssen, z.B. das Abspeichern der Werte in *Econf*, sobald der Dialog akzeptiert wurde. Abbildung 3.4 zeigt den erstellten Konfigurationsdialog. Ein Dateipfad kann angegeben werden, der automatisch beim Start des Dienstes freigegeben wird. Auf der nicht sichtbaren Reiterkarte des Dialogs können Einstellungen für die Übertragung vorgenommen werden.

Der Dateimanager

Die zentrale Aufgabe des Dateimanager ist die Verwaltung der bereitgestellten Dateien und der zugehörigen Ansicht. Er bietet nach außen Methoden zum Hinzufügen und Entfernen von Dateien bzw. Verzeichnissen an. Diese sind mit entsprechenden Ereignissen der

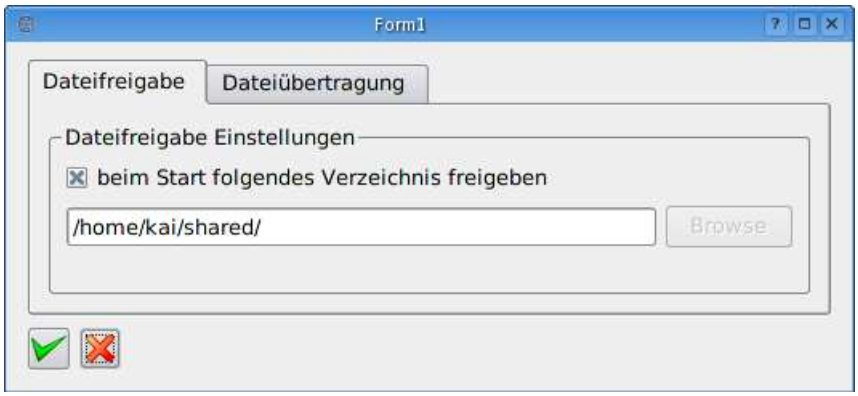

Abbildung 3.4: Dialog zum Konfigurieren des Dateifreigabedienstes

GUI verknüpft. Drückt der Anwender z.B. den Druckknopf 'Datei hinzufügen', wird die zugehörige Methode beim Dateimanager gerufen. Eine Sonderstellung nimmt das Hinzufügen und Entfernen von Verzeichnissen ein, dies geschieht rekursiv. Der Inhalt des Verzeichnises wird gelesen und abgearbeitet. Für jedes weitere Verzeichnis wird dann die Methode zum Hinzufügen von Verzeichnissen erneut gerufen. Die Rekursion bricht ab, wenn keine weiteren Verzeichnisse existieren und alle dort befindlichen Dateien hinzugefügt wurden. Es gibt weiter die Möglichkeit für den Anwender zu bestimmen, wo neue Dateien innerhalb der Baumstruktur bereitgestellt werden sollen. Um dies zu realisieren, wird beim Hinzufügen von Dateien und Verzeichnissen immer der Elternknoten, identifiziert durch seine Kennung, mitgegeben. Das Anhängen beginnt immer an der sogenannten `targetId`. Dies ist eine Variable, in welcher der aktuelle Einhängpunkt gespeichert ist. Wird ein Verzeichnis angehängt, dient dieses mit seiner Kennung als neuer Einhängpunkt für alle folgenden Dateien und Verzeichnisse.

Durch das Abspeichern des Einhängpunktes in `targetId` wird dem Anwender die Möglichkeit gegeben, den Einhängpunkt für neue Dateien selbst zu bestimmen. Drückt dieser die rechte Maustaste über einem Verzeichnis, kann der Menüpunkt *Freigaben hier einhängen* ausgewählt werden. Hierbei wird die Kennung des zu diesem Zeitpunkt aktiven Knotens in `targetId` gespeichert. Der angesprochene Kontextmenüpunkt ist auch auswählbar, wenn der Mauszeiger sich nicht über einem Verzeichnis befindet. Dann setzt die Methode den Einhängpunkt wieder auf die Wurzel des Baumes zurück.

Abbildung 3.5 zeigt verschiedene bereitgestellte Dateien. Direkt an der Wurzel des Baumes hängen dabei das Verzeichnis 'shared' sowie die Dateien 'firefox' und 'thunderbird'. Weiter sind am Verzeichnis 'shared' die beiden Dateien 'handbuch_webalizer.pdf' und 'testdaten' angehängt. Die von der `QListView` erzeugten Linien verdeutlichen den Eltern-Kind-Zusammenhang. Alle dargestellten Knoten sind `QListViewItems`.

Eine weitere wichtige Aufgabe des Dateimanagers ist der Zeitpunkt der letzten Änderung an der Freigabe. Dieser dient der Unterscheidung verschiedener Versionen der Dateiliste, denn nur wenn Dienst und Accessor die gleiche Version der Liste verwenden, wird einer Übertragung zugestimmt. Die private Methode `changed()` sorgt dafür, daß das interne Attribut `timestamp` aktualisiert wird. Sie wird von allen Methoden gerufen, die Änderungen durch den Anwender vornehmen. Den anderen Komponenten gewährt der Dateimanager über die Methode `lastChanged()` Zugriff auf das Attribut `timestamp`, damit diese den

aktuellen Stand erfragen können.

Der Dateimanager verwendet für die Ansicht der Baumstruktur das Widget `QListView` des GUI-Toolkits. Für jeden Knoten[1] des Baumes wird eine Instanz der Klasse `QListViewItem` erzeugt. Dabei muß dem Konstruktor der Elternknoten mitgeteilt werden. Elternknoten können alle anderen Knoten im Baum als auch die `QListView` selbst sein. Diese ist die Wurzel innerhalb der Baumdarstellung.

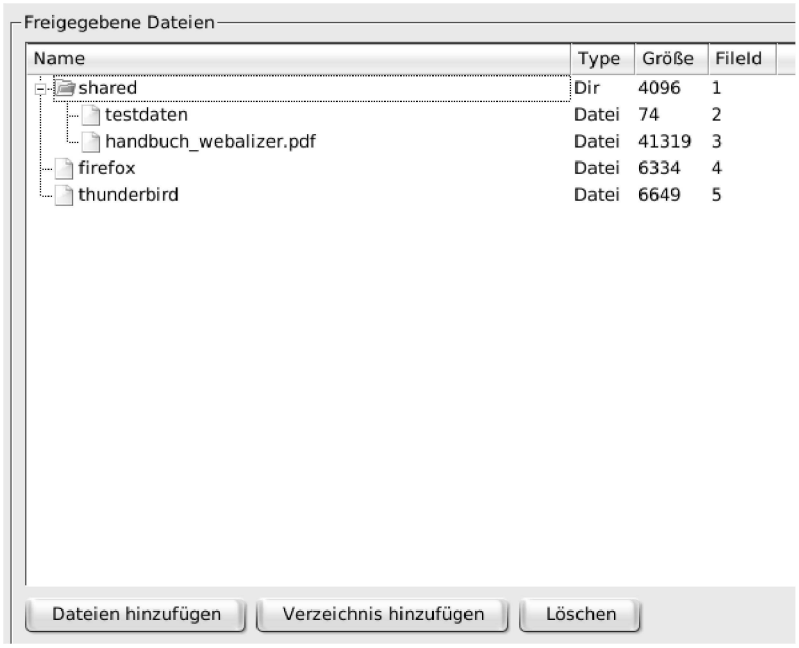

Abbildung 3.5: Dateimanager mit freigegebenen Dateien

Die Klasse Filemanager

Die abstrakte Klasse `AbsFileHandler` dient als Grundlage für die Klasse `Filemanager`. Instanzen der folgenden Klassen werden verwendet, um die gestellten Aufgaben zu erfüllen:

- `IdMaker` - erzeugt Kennungen für die Verwaltung der Dateien
- `TreeData` - verwaltet einen Dateibaum in einer Datenstruktur
- `TreeView` - zeigt einen Dateibaum an, der als `TreeData` gespeichert ist
- `FilePaths` - verwaltet die lokalen Pfade der Dateien

Die Klasse IdMaker

Zur Realisierung der gestellten Aufgabe verwendet die Klasse `IdMaker` einen internen Pool. In diesem werden alle bisher vergebenen Zahlen abgelegt. Wird nun über die Methode `next()` eine neue Zahl verlangt, wird intern solange 'hochgezählt' bis eine Zahl

[1] QListView unterscheidet nicht zwischen Blätter und Knoten in Bäumen. Daher werden wir im Kontext unserer Anwendung nur von Knoten sprechen.

gefunden wird, die nicht im Pool ist. Damit der Zahlenpool nicht bei jeder Anforderung durch `next()` vollständig durchsucht werden muß, wird intern immer die letzte bekannte Zahl in der Variable `lastKnown` gespeichert. Wird nun erneut nach einer Zahl gefragt, kann die Suche direkt bei dieser beginnen. Im besten Fall benötigen wir dann nur noch einen Vergleich, um eine freie Zahl zu finden. Problematischer ist das Laufzeitverhalten, wenn Zahlen wieder freigegeben werden, denn dabei spielt das gewünschte Verhalten des Zahlenpools eine Rolle. Am besten für das Laufzeitverhalten ist es, die Variable `lastKnown` durch die Zahlenfreigabe nicht zu verändern. Dabei ist allerdings die Ausnutzung der zur Verfügung stehenden Zahlen nicht optimal, es entstehen Löcher im Pool. Daher wird die `lastKnown` Variable auf die gelöschte Zahl gesetzt, wenn diese kleiner ist als die bisherige `lastKnown` Zahl. Dadurch kann es kurzfristig zu einem schlechteren Laufzeitverhalten kommen. Folgendes Szenario beschreibt den schlechtesten Fall:
Es wurden bereits n Zahlen zurückgeliefert, die kleinste Zahl wurde aus dem Pool gelöscht und eine neue Zahl angefordert. Dies war die kleinste Zahl im Pool, denn diese wurde zuvor gelöscht. Wenn nun eine weitere Zahl angefordert wird, zeigt `lastKnown` auf die kleinste bzw. erste Zahl im Pool und es sind n Zahlen bereits vergeben. Daher müssen nun n+1 Vergleiche ausgeführt werden, um die nächste freie Zahl zu finden.

Die Klasse TreeData

Jeder Knoten bekommt eine eindeutige Kennung zugeordnet, diese wird mit Hilfe der Klasse `IdMaker` erzeugt und dient als eindeutiger Schlüssel. Die Daten des Knoten werden in einer assoziativen Datenstruktur gespeichert. Es wird ein Dictionary verwendet, in welches für jeden Knoten ein Tupel abgelegt wird. Jeder Eintrag hat folgenden Aufbau:`[Kennung] =(Dateityp={Datei|Verzeichnis}, Dateiname, [Liste mit Kinder Kennungen], Kennung des Elternknoten, Größe in Byte)`

Der Eintrag 'Kennung des Elternknoten' dient dazu, den Elternknoten schnell zu finden. Die Liste der Kinder Kennungen beinhaltet, wie der Name schon sagt, alle Kennungen der Kinderknoten. Diese ist leer, falls der Knoten ein Verzeichnis ist. Abbildung 3.6 zeigt ein ausführliches Beispiel für die verwendete Datenstruktur.

Die Klasse TreeView

Die Klasse `TreeView` ist die Schnittstelle zur Ansicht. Dazu wird eine Instanz der Qt Klasse `QListView` verwendet, die dem Konstruktor übergeben wurde. Für jeden Knoten des Dateibaums wird eine Instanz der Klasse `QListViewItem` erzeugt. Diese Instanz muß eine Referenz auf den Elternknoten übergeben werden. Die restlichen Texte können einfach über die Methoden `setText(SpaltenNummer, Text)` gesetzt werden. Es ist möglich als Vaterknoten die `QListview` anzugeben, dann erscheint der neue Knoten als direkter Nachfolger der Wurzel. Beim Löschen eines Knotens muß darauf geachtet werden, daß dieser nicht ausgewählt ist. Genau dies ist aber der Fall, denn sonst wäre eine Zuordnung nicht möglich, welcher Knoten gelöscht werden soll. Daher wird vor dem eigentlichen Löschen in der `QListView` der vorherige Knoten ausgewählt. Mit der Methode `takeItem()` wird anschließend der Knoten aus der Ansicht entfernt. Die Methode erwartet als Parameter den Elternknoten des zu löschenden Knotens. Dieses kann innerhalb der `QListView` leicht mit der Methode `parent()` ermittelt werden.

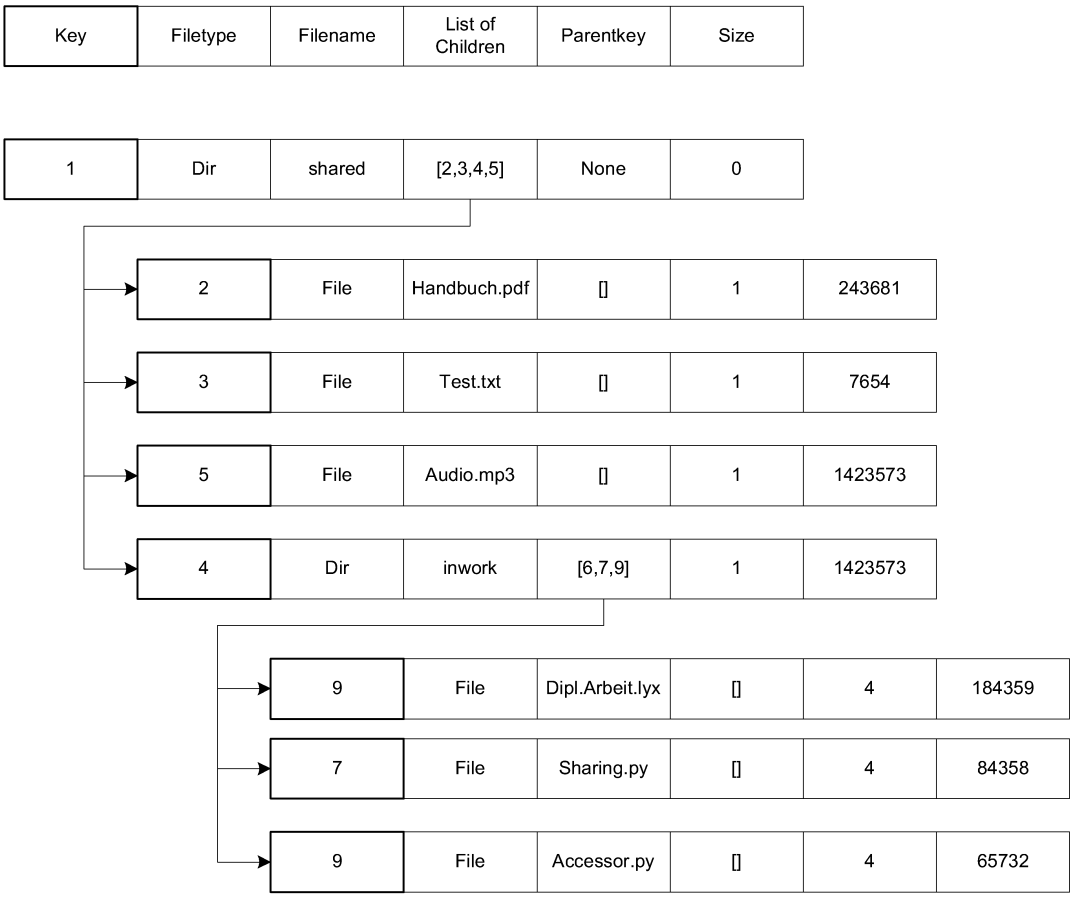

Abbildung 3.6: Daten innerhalb von `TreeData`

Die Klasse FilePaths

Dateinamen und Pfade werden in einer Instanz der Klasse `FilePaths` verwaltet. Diese wird nur vom Dienst innerhalb seiner Instanz von `Filemanager` verwendet. Die Speicherung erfolgt in einem assoziativen Dictionary mit der Dateikennung als eindeutiger Schlüssel. Als Schnittstelle nach außen werden von `FilePaths` Methoden zum Hinzufügen, Entfernen und Erfragen einer Kennung bei bekanntem Pfad angeboten.

Die Klasse Transfermanager

Die Aufgabe des Übertragungsverwalters besteht darin, die gesamte Kommunikation mit dem Accessor zu übernehmen. Er muß anfragenden Accessoren die aktuelle Liste der freigegebenen Dateien übermitteln und die Dateiübertragung in Blöcken arrangieren. Weiter soll er über Protokollereignisse informieren und alle in Bearbeitung befindlichen Dateiübertragungen übersichtlich darstellen.

Eine weitere wichtige Aufgabe, die im Übertragungsverwalter implementiert ist, stellt die Verwaltung der einzelnen Übertragungen dar. Dazu verwendet der Übertragungsverwalter die Klasse `LimitedIdMaker`. Diese ist eine Spezialisierung der vorher besprochenen

Klasse `IdMaker`. Mit Hilfe dieser erzeugt er eindeutige `TransferIds`. Ist die Anzahl maximal gleichzeitig stattfindender Übertragungen erreicht, bekommt der Übertragungsverwalter keine gültige Kennung zurück. Das Übertragungslimit kann im Konfigurationsdialog verändert werden.

Die zentrale Idee des Übertragungsverwalter besteht darin, alle Informationen für eine Dateiübertragung in einem Objekt zu speichern. Dieses Objekt bildet die gemeinsame Grundlage für die Darstellung sowie die Dateiübertragung.

Für die Dateiübertragung wurde ein Protokoll entworfen, das auf die besonderen Gegebenheiten des Ad-Hoc Netzes Rücksicht nimmt. Erneut wurde auf die Trennung der Ansichts- und Verwaltungslogik geachtet. Dies nahm direkten Einfluss auf den Klassenentwurf.

Der Übertragungsverwalter instanziiert folgende Klassen:

- `SenderTransfer` - als Spezialisierung von FileMarshalling
- `TransferView` - für die Ansicht
- `LimitedIdMaker` - zur Verwaltung der Übertragungen

Die Klassen SenderTransfer und FileMarshalling

Die Klasse `FileMarshalling` bildet die entscheidende Grundlage der Klasse `SenderTransfer`. Sie öffnet eine Datei und liefert diese in Blöcken fester Größe zurück. Alle Blöcke bekommen dabei intern eine Nummer, so daß eine Datei der Größe m anschließend aus $\lceil \frac{m}{Blockgröße} \rceil$ vielen Blöcken besteht. Diese Blöcke bilden die Grundlage für die Übertragung. Wird die Methode `next()` gerufen, liefert `FileMarshalling` die Daten des aktuellen Blocks zurück und erhöht den internen Zähler um eins. Bei erneutem Aufruf wird der nächste Block zurückgeliefert. `FileMarshalling` selbst basiert auf der abstrakten Klasse `AbsMarshalling`, welche mit ihren grundlegenden Methoden ebenfalls in `FileUnMarshalling` innerhalb des Accessor Verwendung findet. Die Klasse `SenderTransfer` erbt von `FileMarshalling` und speichert zusätzlich weitere Informationen über Empfänger und Versender ab. Diese Informationen können abgerufen und gespeichert werden - entsprechende Methoden sind vorhanden.

Die Klasse TransferView

Die Klasse `TransferView` erwartet eine `QListView`, um die stattfindenden Übertragungen darzustellen. Mit der Methode `newTransfer(SenderTransfer)` wird das übergebene Objekt bei der Ansicht bekannt gemacht. Es wird ein Objekt der Klasse `SenderTransfer` und zur Darstellung eines der Klasse `TransferItem` erzeugt.

Um dem Anwender weitere Informationen über das Übertragungsprotokoll mitzuteilen, erwartet die Klasse zusätzlich eine Instanz von `QTextEditor.` Dieses Widget ist auf 'Read Only' eingestellt, so daß Texte im Richtext Format ausgegeben, aber nicht editiert werden können. Damit zu jeder Zeit eine Nachricht im Protokollfenster ausgegeben werden kann, existiert die Methode `message(Nachricht)`. Diese gibt die übergebene Nachricht mit einem Zeitstempel versehen in einer eigenen Zeile aus.

Die Klasse TransferItem

Qt bietet einen eigenen Fortschrittsbalken an. Dieser ist allerdings ein eigenes Widget. Leider sind als Zellen einer `QListView` keine Widgets möglich. Daher mußte eine eigene Lösung realisiert werden. Die Klasse `TransferViewItem` erbt von `QListViewItem`, um die Darstellung des Prozentbalken innerhalb einer Spalte zu realisieren. Dafür wurde die Methode `paintCell()` durch eine neue überladen. Diese überprüft mit Hilfe der Spaltennummer, ob es sich um die gewünschte Spalte für den Prozentbalken handelt. Ist dies der Fall, errechnet sie aus den zur Verfügung stehenden Informationen `numberOfCurrentBlock` und `numberOfAllBlocks` des Objektes `SenderTransfer` den Prozentwert. Anschließend kann mit dem `QPainter` der Balken erzeugt werden. Mit Hilfe der aktuellen Spaltenbreite lassen sich die Koordinaten für das Zeichnen der Rechtecke leicht errechnen. Für alle anderen Spalten der Darstellung wird die Methode `paintCell` der ursprünglichen Klasse gerufen. Da alle weiteren Informationen, die dargestellt werden, Texte sind, wird hierfür einfach die vorhandene Methode `setText(spaltenNummer, text)` verwendet.

Die Klasse LimitedIdMaker

Die Klasse `LimitedIdMaker` ist eine Spezialisierung der Klasse `IdMaker` (siehe 3.3.1). Sie überlädt die Methoden zum Erzeugen und Freigeben von Zahlen, um ein verändertes Verhalten zu realisieren. Die zentrale Aufgabe ist weiterhin das Erzeugen von eindeutigen Kennungen. Allerdings soll eine veränderbare Schranke von erzeugten Kennungen nicht überschritten werden. Dieses Limit wird mit dem Konstruktor übergeben, kann aber nachträglich über die Methode `changeLimit(newLimit)` verändert werden. Der Übertragungsverwalter verwendet die Klasse `LimitedIdMaker`, um die Anzahl der gleichzeitig stattfindenden Übertragungen zu begrenzen.

Die Klasse IdTimer

Die Klasse `IdTimer(id)`, abgeleitet von `QObject`, stellt einen speziellen Timer zur Verfügung. Dem Konstruktor wird dabei eine Zahl `(id)` übergeben. Wird ein neuer `IdTimer` erzeugt, speichert das Objekt die Zahl intern ab. Über einen Aufruf der Methode `start(time)` wird der Timer gestartet. Nach der angegebenen Zeit in Sekunden löst der Timer ein Signal aus. Intern verwendet `IdTimer` einen `QTimer` und startet diesen. Das Signal `timeout()` wird mit dem Slot `slotTimeout()` verbunden, um gezielt reagieren zu können. Läuft der `QTimer` ab, löst er das Signal `timeout()` aus und der verbundene Slot von `IdTimer` wird gerufen. Dieser sendet mit der zu `QObject` gehörenden Methode `emit()` das eigene Signal `timeIsOver(Kennung)` aus. Ist ein Slot mit dem Signal der Klasse `IdTimer` verbunden, bekommt dieser die Kennung übergeben. Dies wird vom Übertragungsverwalter verwendet, um die Timeouts der einzelnen Datenübertragungen zu realisieren. Wird eine Übertragung begonnen, startet der Übertragungsverwalter einen `IdTimer` und übergibt die `TransferId`. Läuft alles nach Plan, wird der Timer regelmäßig zurückgesetzt (siehe 3.4). Das Signal `timeIsOver(id)` tritt daher niemals ein. Treten Probleme bei der Übertragung auf, ist es möglich, daß der Timer abläuft - das Signal tritt ein. Der dadurch gerufene Slot kann anschließend mit Hilfe der übergebenen Kennung die zugehörige Übertragung identifizieren.

3.3.2 Accessor

Die grafische Bedienoberfläche des Accessors besteht aus:

- dem Dateianzeiger - er verwendet eine `QListView`,
- der Übertragungsansicht - sie verwendet eine `QListView` und
- der Protokollansicht - sie verwendet ein `QTextEdit`.

Die Fenster wurden mit dem Qt-Designer entworfen und eine abgeleitete Klasse nimmt zusätzliche Einstellungen vor. Von der abgeleiteten Klasse wird eine Instanz erstellt, dadurch sind die Knoten des Widgets verfügbar. Nun werden noch Instanzen der Klassen `FileBrowser` und `ReceiverTransfer` erzeugt, welche im Konstruktor passende Widgets erwarten. Auch beim Accessor wurde ein Konfigurationsdialog entwickelt. Abbildung 3.7 zeigt den Konfigurationsdialog. Es ist ein Pfad zum Abspeichern empfangener Dateien sowie ein Timout für Übertragungen einstellbar. Von der durch den Qt-Designer erzeugten Klasse wird durch Vererbung die eigene Klasse `sharingAccessorConfigDialog` erzeugt. In dieser wird festgelegt, was passiert, wenn der Anwender den Dialog akzeptiert. In diesem Fall müssen die eingetragenen Werte abgespeichert werden. Da der Accessor seinen Dienst eindeutig kennt, muß er nur die stattfindenden Übertragungen verwalten. Dies macht er mit Hilfe der vom Dienst zugewiesenen `TransferId` in einem Directory. Da es keinen Übertragungsverwalter gibt, sind die Teile des Übertragungsprotokolls direkt im Accessor implementiert. Jeder Accessor muß nach seinem Start die Information erhalten, von welchem Dienst er gestartet wurde und an welcher Adresse sich der Dienst befindet. Außerdem benötigt der Dienst eine eindeutige Kennung, mit welcher er sich beim Dienst meldet. Die benötigten Informationen über den Dienst werden dem Accessor nach dem Start in der speziellen Lego-Nachricht `lego_StartServiceUsage` durch das Rahmenwerk mitgeteilt. Zur Erstellung einer eindeutigen Kennung bietet das Rahmenwerk die Methode `unique` an. Diese wird verwendet, um eine eindeutige Zahl zu erzeugen, die anschließend als `RequestId` verwendet wird.

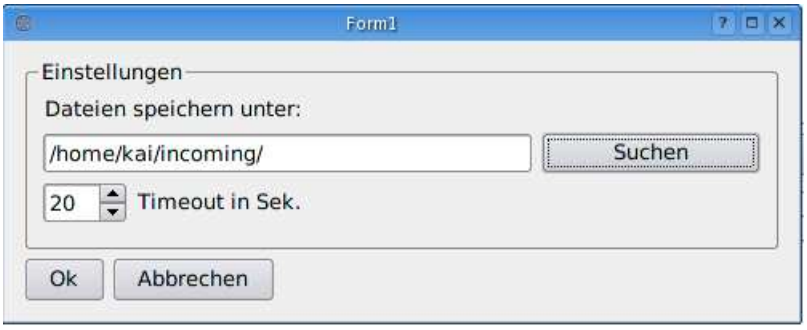

Abbildung 3.7: Dialog zum Konfigurieren des Dateifreigabeaccessors

Die Klasse FileBrowser

`FileBrowser` erbt von `AbstFileHandler` und erweitert diese um die Methode `newDataTree`. Diese erzeugt aus der Datenstruktur, einer Instanz von `TreeData`, die

zugehörige Ansicht der Klasse `TreeView`. Mit Hilfe dieser Methode kann jederzeit ein neuer Dateibaum angezeigt werden.

Die Klassen ReceiverTransfer und FileUnMarshalling

Der entscheidende Unterschied zu `senderTransfer` besteht darin, daß Dateien empfangen werden, d.h. eintreffende Blöcke zusammengesetzt werden müssen. Daher bildet die Klasse `FileUnMarshalling` die Basis für `ReceiverTransfer`. `FileUnMarshalling` erwartet im Konstruktor den Dateinamen und die Anzahl der Blöcke der zu empfangenden Datei. Ein interner Zähler wird auf 0 gesetzt, das ist der erste Block, und ein Filehandle erzeugt. Mit jedem Aufruf der Methode `next(blockdata, number)` wird ein Datenblock sowie die Blocknummer übergeben. Das Unmarshalling überprüft bei jedem Aufruf der Methode `next()`, ob der als Parameter übergebene Block der Erwartete ist. Trifft der korrekte Block ein, wird dieser mit Hilfe des Filehandler an die Datei angehängt und der interne Zähler erhöht. Anschließend wird überprüft, ob es sich um den letzten Block der Übertragung gehandelt hat. Sollte dies so sein, wird das Filehandle geschlossen und der Rückgabewert `lastBlock` geliefert. Die Klasse `ReceiverTransfer` hat die gleiche Aufgabe wie `SenderTransfer`. Sie soll alle Informationen, die zu einer Übertragung gehören, in einem Objekt vorhalten, daher erbt sie zusätzlich von `AbsTransfer`. Diese Objekte werden dann für Verwaltungs- und Anzeigelogik verwendet.

3.4 Das Übertragungsprotokoll

Im folgendem Abschnitt wird das implementierte Übertragungsprotokoll beschrieben, welches für den Dateifreigabedienst verwendet wird.

3.4.1 Motivation

Innerhalb des ELAN-Rahmenwerkes ist es möglich, beliebige Daten zwischen Stationen zu übertragen. Dabei muß allerdings eine Datenkonvertierung vorgenommen werden, denn die verwendete XMLRPC Schnittstelle ist nicht in der Lage, beliebige Datentypen zu übertragen [Mar03]. In Abhängigkeit von der Datenmenge steigt das Risiko, daß während einer Übertragung Probleme auftreten können. Verändert sich dabei z.B. die Route zwischen den teilnehmenden Stationen, bricht die Übertragung ab. Dieses Problem läßt sich umgehen, indem die Datenübertragungen in kleinere Blöcke aufgeteilt und jeweils einzeln verschickt werden. Zusätzlich können auftretende Probleme minimiert werden, weil im besten Fall nur einzelne Blöcke erneut beim Sender beantragt werden müssen. Sämtliche Nachrichten basieren auf der Ext In&Out Kommunikation [Fer03] des Rahmenwerks. Kommt eine Nachricht über die Ext-In Schnittstelle beim Dateifreigabedienst an, wird diese dispatched wie in Kapitel 2 beschrieben. Das Übertragungsprotokoll ist derart realisiert worden, daß auf der jeweiligen Empfänger- oder Senderseite nur die Methoden implementiert wurden, auf die reagiert werden muß. Möchte der Übertragungsverwalter bzw. der Accessor eine Nachricht verschicken, verwendet er die vom Rahmenwerk zur Verfügung gestellte Methode `post()`. Dieser muß er den Empfänger mitteilen (IP-Adresse sowie Komponentenname), sich als Absender identifizieren (Name und IP-Adresse) und die gewünschte Nachricht anhängen. Die Kommunikation zwischen Dienst und Accessor wird durch die

im Folgendem beschriebenen Nachrichtentypen abgewickelt. Dabei kümmert sie das ELAN-Rahmenwerk um die Zustellung der einzelnen Ext In&Out Nachrichten.

3.4.2 Nachrichtentypen

Das Übertragungsprotokoll umfasst folgende Nachrichtentypen:

- `RequestFilelist` - der Accessor beantragt die aktuelle Freigabeliste beim Dienst
- `Filelist` - eine aktuelle Freigabeliste trifft beim Accessor ein
- `RequestFile` - beim Dienst wird eine Dateiübertragung beantragt
- `NonAck` - der Dienst lehnt eine beantragte Dateiübertragung ab
- `AckReadyToSend` - der Dienst stimmt einer beantragten Dateiübertragung zu
- `RequestBlock` - der Accessor beantragt einen einzelnen Block einer Übertragung
- `Block` - ein Block einer laufenden Übertragung trifft beim Accessor ein
- `TransferComplete` - der Accessor teilt dem Dienst mit, daß eine Übertragung abgeschlossen wurde
- `ByeBye` - ein Accessor meldet sich beim Dienst ab
- `StopTransfer` - der Accessor beendet eine Übertragung
- `TransferError` - der Dienst beendet eine Übertragung

Es folgt eine detailierte Auflistung der einzelnen Nachrichtentypen. Zur besseren Übersicht ist am Anfang immer eine tabellarische Übersicht vorangestellt, in der Absender, Empfänger sowie die in der jeweiligen Nachricht enthaltenen Daten übersichtlich aufgeführt sind. Abbildung 3.8 zeigt den Ablauf einer erfolgreichen Dateiübertragung. Der Accessor erfragt die Dateiliste und bekommt diese. Anschließend fragt er eine Datei an, der Dienst gestattet dies. Anschließend beginnt die blockweise Übertragung. Am Ende der erfolgreichen Übertragung teilt der Accessor dem Dienst mit, daß die Übertragung abgeschlossen ist.

RequestFileList

Absender	Accessor
Empfänger	Dienst
Daten	`RequestId`, `RequestIP`, `AccessorName`, `OtherUser`

Der Absender möchte gerne vom Empfänger die aktuelle Version der Dateifreigabeliste haben. Die `RequestId` wird vom Accessor generiert. Sie ist eine eindeutige Nummer, die vom Rahmenwerk durch die Methode `unique()` erzeugt wird. Die `RequestId` dient der Identifizierung. `RequestIp`, `AccessorName` und `OtherUser` sind weitere Informationen, die der Empfänger benötigt, um dem Absender antworten zu können. Jeder anfragende Accessor wird intern beim Empfänger in einem Directory abgespeichert, die `RequestId` dient

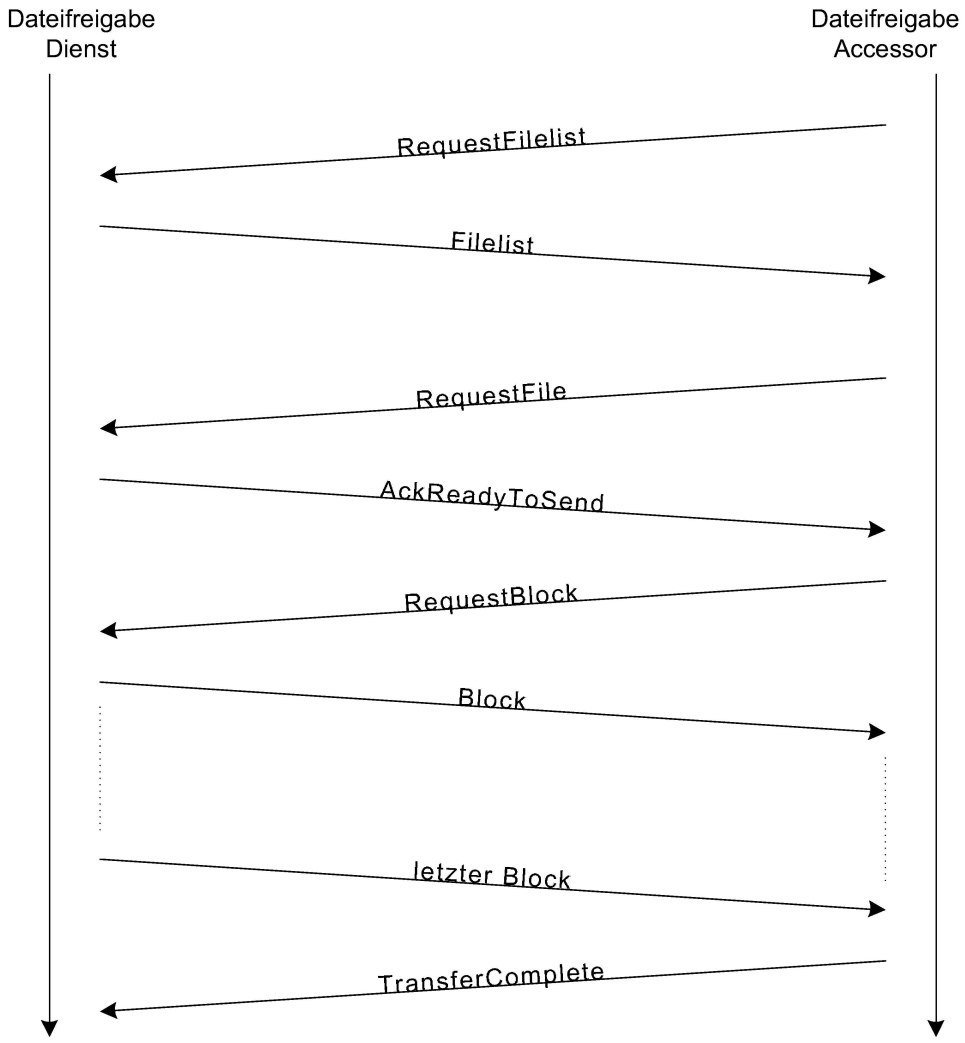

Abbildung 3.8: Bsp. für Datenübertragung

dabei als Schlüssel. Gespeichert werden `RequestIp`, `AccessorName`, `OtherUser` und der Zeitstempel der Dateiliste zum aktuellem Zeitpunkt (siehe 3.3.1). Trifft diese Nachricht beim Empfänger ein, überprüft dieser, ob er den Absender bereits kennt und dieser bereits die aktuelle Dateiliste bezogen hat. Sollte das der Fall sein, brauchen wir keine aktuelle Dateiliste zurückzuschicken und können das Transferaufkommen reduzieren. Der Absender erwartet auf eine `RequestFileList` keine Antwort. Trifft bei ihm eine Dateiliste ein, bereitet er diese auf und zeigt sie an.

Filelist

Absender	Dienst
Empfänger	Accessor
Daten	`RequestId`, `Data`

Der Empfänger erhält mit dieser Nachricht die aktuelle Dateifreigabeliste `Data` vom Absender. Entweder hat er die Dateiliste mit der Nachricht `RequestFilelist` beantragt oder erhalten, weil er beim Absender bekannt ist und seine Dateifreigabeliste nicht mehr aktuell ist. Letzteres tritt aber nur ein, wenn eine Dateiübertragung vom Absender zuvor abgelehnt wurde. Die `RequestId` wird mitgeschickt, damit der Empfänger überprüfen kann, ob die neue Dateifreigabeliste wirklich für ihn ist. ELAN stellt Nachrichten an alle Komponenten zu, diese müssen über den Dienstnamen entscheiden, ob die Nachricht für sie ist. Sollten weitere Accessoren innerhalb einer ELAN-Umgebung laufen, ist die Unterscheidung der Nachrichten nur über die `RequestId` möglich. Ist der Empfänger der Richtige, wird die Dateifreigabeliste ausgepackt und eine neue Ansicht durch den `FileBrowser` erzeugt.

RequestFile

Absender	Accessor
Empfänger	Dienst
Daten	`RequestId`, `FileId`, `Filename`

Der Absender beantragt beim Empfänger die Übertragung einer Datei. Zu diesem Zweck sendet er `RequestId`, die `FileId` und den Dateinamen (`FileName`) der gewünschten Datei mit. Der Empfänger prüft zuerst mit Hilfe der `RequestId`, ob der Absender die aktuelle Version der Dateifreigabeliste verwendet. Ist dies nicht der Fall, lehnt er die Übertragung mit der Nachricht `NoAck` ab und schickt dem Absender in einer zweiten Nachricht die aktuelle Dateiliste unaufgefordert zu. Der Absender bricht intern die Übertragung ab und zeigt die neue Dateiliste an. Der Anwender muß dann erneut die Dateiübertragung veranlassen. Hat der Absender die gleiche Dateifreigabeliste wie der Empfänger, prüft dieser, ob er eine `TransferId` vom `LimitedIdMaker` beziehen kann. Ist die Grenze an gleichzeitigen Übertragungen bereits erreicht, schlägt dies fehl und die Anfrage wird verworfen. Der Empfänger bricht in diesem Fall die Übertragung mit der Nachricht `NonAck` ab. Nur wenn die Dateifreigabelisten gleich sind und ein Übertragungsslot zur Verfügung steht, also eine `TransferID` zugewiesen wurde, stimmt der Empfänger der Dateiübertragung zu. Er sendet dem Accessor die Nachricht `AckReadyToSend` zurück, welche die gültige `TransferId` enthält. Mit dieser erzeugt der `Transfermanager` eine Instanz der Klasse `SenderTransfer` und meldet die neue Dateiübertragung bei der `TransferView` an. Weiter startet er einen `IdTimer`, um Übertragungsprobleme erkennen zu können. Der Absender hat nun bis zum

Ablaufen des Timer Zeit, den ersten Block der Datei zu beantragen. Läuft der Timer ab, kann die Zugehörigkeit über die im Signal enthaltene Kennung ermittelt werden. Der Empfänger sendet Nachricht `TransferError` an den Absender und bricht die Übertragung ab.

NoAck

Absender	Dienst
Empfänger	Accessor
Daten	`reason`

Der Empfänger hatte eine Datei zur Übertragung beantragt. Der Absender sendet die Nachricht, weil er der Übertragung nicht zustimmen kann. Dies kann passieren, wenn die Dateiliste des Empfängers nicht aktuell ist, oder wenn keine gültige `TransferId` bezogen werden konnte (siehe 3.4.2). Weiter erhält der Empfänger den Grund in der Variablen `reason` mitgeteilt.

AckReadyToSend

Absender	Dienst
Empfänger	Accessor
Daten	`transferId`, `numberOfBlocks`

Der Empfänger hat eine Datei zur Übertragung beantragt. Der Absender stimmt dieser zu und teilt dem Empfänger dies mit. Der Empfänger bekommt die Information, wie die gültige `TransferId` lautet und aus wievielen Blöcken (`numberOfBlocks`) die Datei besteht. Nun kann der Empfänger ein Objekt der Klasse `ReceiverTansfer` erzeugen und meldet die Instanz bei seiner `TransferView` an.

RequestBlock

Absender	Accessor
Empfänger	Dienst
Daten	`transferId`

Absender und Empfänger haben sich bereits darüber geeinigt, daß eine Dateiübertragung stattfindet. Der Absender muß nun Block für Block die Datei anfordern. Hierzu schickt er die Nachricht `RequestBlock`, um vom Empfänger den nächsten Block zu erhalten. Als Information hierzu genügt die `TransferId`, denn welcher Block als nächstes gesendet werden muß, ist im Objekt der Klasse `SenderTransfer` enthalten (siehe 3.4.2). Der Übertragungsverwalter kann durch die `TransferId` die Übertragung eindeutig identifizieren. Er muß jetzt nur noch die Methode `next()` des `SenderTransfer` Objekts rufen und erhält den gewünschten Datenblock zurück. Diesen verpackt er in eine `Block` Nachricht und schickt diese an den Absender zurück. Der zum `SenderTransfer` Objekt gehörende `IdTimer` wird zurückgesetzt, nun hat der Absender wieder die gleiche Zeit um den nächsten Block zu beantragen.

Block

Absender	Dienst
Empfänger	Accessor
Daten	`transferId`, `Blockdaten`, `numberOfBlock`

Der Absender schickt den aktuellen Block der Dateiübertragung an den Empfänger. Mit Hilfe der `Transferid` kann der Empfänger das passende `ReceiverTransfer`-Objekt identifizieren. Er gibt die Blockdaten und die Blocknmmer an die Methode `next(Blockdata,numberOfBlock)` des Objektes weiter. Diese überprüft, ob es sich um den richtigen Block handelt und hängt die Daten an die Datei an. Sollte der empfangene Block nicht der Letzte sein, beantragt der Empfänger den Nächsten. Ansonsten teilt der Empfänger dem Absender mit, daß die Übertragung erfolgreich beendet wurde. Dies geschieht durch Senden der Nachricht `TransferComplete`.

TransferComplete

Absender	Accessor
Empfänger	Dienst
Daten	`transferId`

Der Absender hat den letzten Block einer laufenden Übertragung erhalten. Er benachrichtigt den Empfänger darüber und schickt die `transferId` mit. Der Empfänger löscht die Übertragung aus seiner Liste der aktuellen Übertragungen. Die Objekte `SenderTransfer`, `IdTimer` sowie sämtliche Referenzen werden entfernt. Die `TransferId` wird freigegeben und dadurch das Limit der `LimitedIdMaker`-Instanz um eins erhöht. Damit wird die verwendete Kennung nicht nochmals vergeben, aber eine weitere kann angefordert werden.

ByeBye

Absender	Accessor
Empfänger	Dienst
Daten	`requestId`

Wird ein Accessor durch den Anwender beendet, schickt er diese Nachricht an den Empfänger. Dieser weiss durch die mitgeschickte `RequestId`, daß der entsprechende Absender beendet wurde und kann alle Informationen über diesen löschen.

StopTransfer

Absender	Accessor
Empfänger	Dienst
Daten	`transferId`

Diese Nachricht trifft beim Empfänger ein, wenn eine Übertragung vom Accessor abgebrochen wurde. Dies kann durch einen Übertragungsfehler oder einen abgelaufener Timer, eingetreten sein.

TransferError

Absender	Dienst
Empfänger	Accessor
Daten	`transferId`

Diese Nachricht trifft beim Dienst ein, wenn eine Übertragung vom Dienst abgebrochen wurde. Dies kann durch einen Übertragungsfehler oder einen abgelaufener Timer eingetreten sein.

3.4.3 Diskussion

Durch das implementierte Protokoll können Dienst und Accessor gezielt miteinander kommunizieren. Es ist möglich, daß mehrere Instanzen dieser innerhalb des Netzes existieren, ohne daß sie sich gegenseitig beeinflussen. Überflüssiges Nachrichtenaufkommen wie beim ERC, siehe Kapitel 2, ist durch das Protokoll nicht vorhanden. Allerdings bildet das Protokoll keine allgemeine Grundlage für Gruppenkommunikation. Es stellt nur eine 1:N Kommunikation für die implementierte Funktionalität zur Verfügung. Die Bindung zwischen Dienst und Accessor ist immer fix. Es sollte daher eine Lösung gefunden werden, die eine N:M Kommunikation ermöglicht, aber Nachrichten gezielt im Netz verschickt. Die Implementierung einer Zwischenschicht für Gruppenkommunikationskomponenten stellt einen interessanten Lösungsansatz bereit. Dabei übernimmt die Zwischenschicht die Verwaltung der Gruppe und kümmert sich um die Kommunikation der Anwendungen untereinander. Außerdem würden Protokolle zur Kommunikation der Komponenten untereinander durch eine Gruppenkommunikationsschicht deutlich geringer ausfallen.

Kapitel 4

Gruppenkommunikation und -verwaltung

In diesem Kapitel wird die Erweiterung des ELAN-Rahmenwerks um eine Gruppenkommunikationsschicht erläutert. Die Gruppenverwaltung sowie die Kommunikation soll vollständig vom Rahmenwerk realisiert werden. Dafür sollen die existierenden Möglichkeiten des Rahmenwerks genutzt und gegebenenfalls erweitert werden. Eine klare und eindeutige Schnittstelle soll zur Entwicklung von Gruppenkommunikationsdiensten bereitgestellt werden. Gruppenanwendungen sollen Nachrichten gezielt miteinander austauschen können. Zur Minimierung des Kommunikationsaufwands im Ad-Hoc Netz sollen überflüssige Nachrichten vermieden werden. Die Anforderungen für die Gruppenkommunikation lauten:

- Funktionale Anforderungen:
  - Die Verwaltung der Gruppe wird vollständig gekapselt,
  - die Gruppenkommunikation wird mit Hilfe der Verwaltung realisiert,
  - es darf keine Einschränkung der Gruppendienste in ihrer Kommunikation untereinander geben.
- Nichtfunktionale Anforderungen:
  - Eine möglichst einfache Anwendungsschnittstelle zur Entwicklung,
  - das ELAN-Rahmenwerk wird als Grundlage verwendet,
  - Gruppenkommunikationsdienste werden über die *AwarenessView* bekanntgemacht,
  - Änderungen am ELAN-Rahmenwerk sind möglichst gering zu halten, um unerwünschte Seiteneffekte zu minimieren.

4.1 Konzept I

Jeder Dienst verfügt innerhalb von ELAN über Metadaten, welche Anwendern weiterführende Informationen liefern. So kann ein Anwender leicht erkennen, weshalb ein Dienst für ihn von Interesse sein könnte. Metadaten sind innerhalb von ELAN nicht zur ausschließlichen Nutzung für die Anwender bestimmt. Dies schließt eine Nutzung für die Gruppenkommunikation nicht aus. Im weiteren Verlauf der Konzeptentwicklung stellte sich heraus, daß die Anzahl der Änderungen am Rahmenwerk zu groß gewesen wären. Mit Hilfe von prototypischen Implementierungen konnten die identifizierten Probleme bestätigt werden. Daher wurde die Entscheidung getroffen Konzept I zu überdenken. Das Konzept II wurde entwickelt und erfolgreich implementiert, dieses wird im nachfolgenden Abschnitt 4.2 besprochen.

4.1.1 Identifizierte Probleme

Bei einer prototypischen Implementierung traten Probleme auf, die zur Entscheidung führten, Konzept I zu verwerfen. Wurde eine Anwendung durch den Nutzer in ELAN gestartet, war es problemlos möglich den Gruppenverwaltungsdienst zu starten. Dies geschah mit Hilfe der Lego Nachricht `lego_StartService`. Allerdings meldete sich der Dienst nicht beim Anwenderprofil an. Er konnte daher auch nicht von interessierten Nutzern gesehen werden. Der Profilverwalter *WilliEd* reagierte offensichtlich nicht auf die verwendete Lego Nachricht. Als Lösung des Problems wurde eine zusätzliche Ext-In Nachricht im `WilliEd` implementiert. Diese Nachricht sendete die Anwendung, die den zentralen Gruppenverwalter starten wollte. Als Parameter diente der Name der Anwendung. Über die internen Methoden des `WilliEd` war es möglich, den universalen Gruppendienst zu starten und den Anwendungsnamen als Metadaten einzutragen. Der universale Gruppenverwalter tauchte nun innerhalb der *AwarenessView* mit den hinterlegten Metadaten auf, und interessierte Anwender konnten darauf zugreifen.
Das nächste Problem war, daß der Accessor beim Start durch das ELAN-Rahmenwerk keine Metadaten übergeben bekam. Über die Metadaten identifizierte der Accessor allerdings die Anwendung. Daher wurde versucht an die benötigte Information innerhalb des Gruppenverwaltungsdienstes zu gelangen. Wären die Metadaten dort abrufbar, könnten sie in einer gesonderten Nachricht an den Accessor geschickt werden. Innerhalb des Dienstes konnte ebenfalls keine Methode gefunden werden, um an die benötigten Metadaten zu gelangen. Diese liegen offensichtlich nur innerhalb des Anwenderprofils von ELAN vor.

Weitere Details, die zur Überarbeitung von Konzept I führten, waren:

- Große Veränderungen an bestehenden Komponenten können ungewollte Fehler und neues Verhalten im Rahmenwerk hervorrufen,
- es wird ein Kommunikationsprotokoll zwischen Anwendung und Dienst bzw. Accessor benötigt,
- zusätzlich wird ein Protokoll zwischen Dienst und Accessoren benötigt,

- es konnte keine Möglichkeit gefunden werden, einer Anwendung mitzuteilen, daß sie von einem Accessor gestartet wurde - ein weiterer Umbau des Rahmenwerks wäre notwendig.

Zwei Protokolle hätten über die Ext In&Out Schnittstelle kommuniziert. Dies entspricht allerdings nicht der zentralen Anforderung, unnötige Kommunikation zu vermeiden.

4.1.2 Gruppenverwaltung mit Dienst und Accessor

Die zentrale Idee von Konzept I war, die Gruppenverwaltung als Dienst und Accessor zu implementieren. Zu diesem Zweck würde ein universeller Gruppenverwaltungsdienst benötigt. Gruppenmitglieder würden einen einheitlichen Accessor verwenden. Zur Kommunikation zwischen Dienst und Accessor müsste ein Protokoll realisiert werden. Gruppenanwendungen sollten auf diese Art und Weise entweder mit dem Gruppendienst oder dem Accessor über die Ext In&Out Schnittstelle kommunizieren. Der Gruppendienst wäre hierbei der Gruppenverwalter und die Accessoren jeweils die Gruppenmitglieder.

4.1.3 Ablaufskizze

Folgender Ablauf für das Bereitstellen und Beitreten war vorgesehen. Ein Anwender startet eine Gruppenanwendung innerhalb des ELAN-Rahmenwerks. Diese Anwendung ist als Gruppenanwendung bekannt und startet den universellen Gruppenkommunikationsdienst. Anwendung und Dienst kommunizieren durch ein Protokoll miteinander. Dabei teilt die Anwendung dem Gruppenverwaltungsdienst ihren Namen mit, dieser wird in den Metadaten des Dienstes vermerkt. Der Gruppendienst wird anschließend an das ELAN-Anwenderprofil gebunden und im Netz bekanntgemacht. Interessierte Nutzer starten wie gewohnt den Accessor über die *AwarenessView*. Der universelle Accessor startet und bekommt beim Start die Lego Nachricht `lego_StartServiceUsage`. Mit Hilfe dieser Nachricht findet er den Gruppendienst und kann mit ihm über ein Protokoll kommunizieren. Der Accessor erhält auf diesem Weg die Metadaten, also den Anwendungsnamen und startet die entsprechende Komponente in ELAN. Abbildung 4.1 illustriert den beschriebenen Ablauf zur Verdeutlichung.

4.1.4 Diskussion

Der zentrale Vorteil des verworfenen Konzept I lag in der Entkoppelung von Anwendung und Gruppenverwaltung. Da die Kommunikation über Ext In&Out funktioniert, wäre es problemlos möglich gewesen einen Gruppenverwaltungsdienst innerhalb des Netzes zu verschieben. Die Anwendung hätte dann entweder mit einem lokal neu gestarteten Accessor kommuniziert oder weiterhin mit dem entfernten Gruppenverwalter. Auf Grund der Menge der identifizierten Probleme wurde Konzept II verworfen. Ein alternativer Entwurf wurde erarbeitet, der die gestellten Anforderungen erfüllt und dabei einen Großteil der beschriebenen Probleme nicht besitzt. Dieser Entwurf führte zu Konzept II, das im folgenden Abschnitt beschrieben wird.

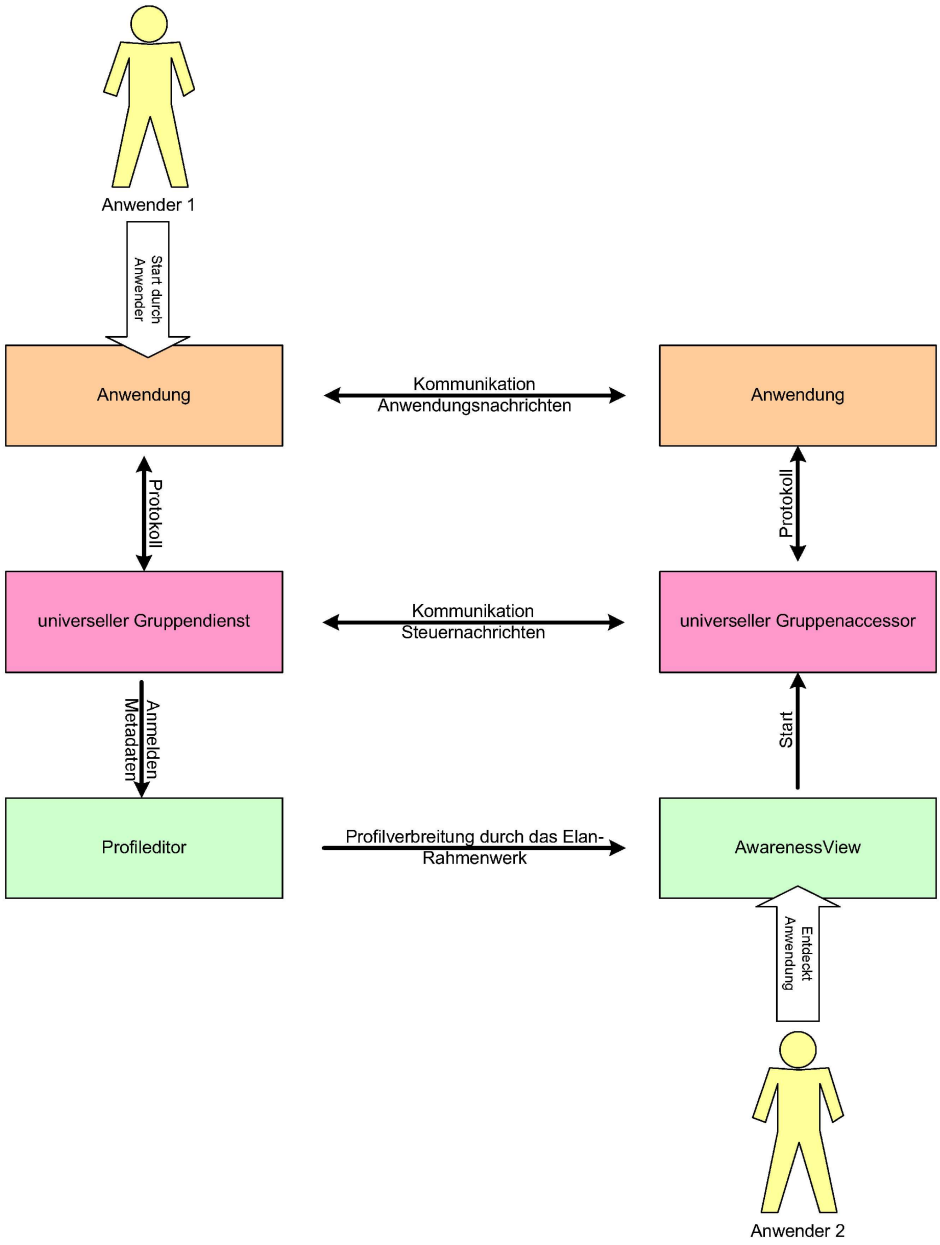

Abbildung 4.1: Startablauf und Kommunikation bei Konzept I

4.2 Konzept II

Jede Komponente innerhalb von ELAN basiert auf der Klasse `EComponent`. Diese Klasse bildet die einheitliche Schnittstelle zum Rahmenwerk. Dienste und Accessoren können auf dieser Anwendungsschicht aufsetzten. Dies wurde im Dateifreigabedienst in Kapitel 3 bereits gezeigt. Auf dieser Schicht wäre es möglich, eine weitere Schicht aufzusetzen. Diese neue Schicht würde dann die Grundlage für Gruppenanwendung und -dienste bilden. Dabei kann die neue Schicht so integriert werden, daß die bereits vorhandene Schicht,

welche die Grundlage für bestehende ELAN Komponenten bildet, problemlos weiterhin verwendet werden kann. Abbildung 4.2 verdeutlicht diesen Zusammenhang. Die Gruppenverwaltung würde so innerhalb der implementierten Anwendung stattfinden. Das Konzept wurde prototypisch implementiert.

4.2.1 Gruppenkommunikationsschicht für Anwendungen

Die neue Klasse `EGroupComponent` realisiert die Gruppenverwaltung und dient als neue Anwendungsschicht. Anwendungen erben von der Klasse und erreichen so die Schnittstelle zur Gruppenverwaltung. Dadurch ist keine Ext In&Out Kommunikation notwendig. Innerhalb von ELAN kann eine Anwendung nicht gleichzeitig Dienst und Accessor sein. Das Rahmenwerk unterscheidet diese beiden Komponenten allerdings nur am Namen. Damit eine einheitliche Anwendung realisiert wird, existiert zu jeder Gruppenanwendung zusätzlich ein entsprechender Gruppenaccessor. Dieser Accessor erbt die Gruppenanwendung und fügt dieser keine neuen Funktionen hinzu. Der Accessor passt nur die globalen Variablen an, welche zur Identifizierung innerhalb des Rahmenwerks verwendet werden. Dies ist nötig, damit der Accessor als solcher identifiziert wird und auch nicht als Dienst innerhalb des Profilverwalters auftaucht. Es entsteht die gleiche Anwendung als Accessor und das ELAN-Rahmenwerk wird nicht eingeschränkt.

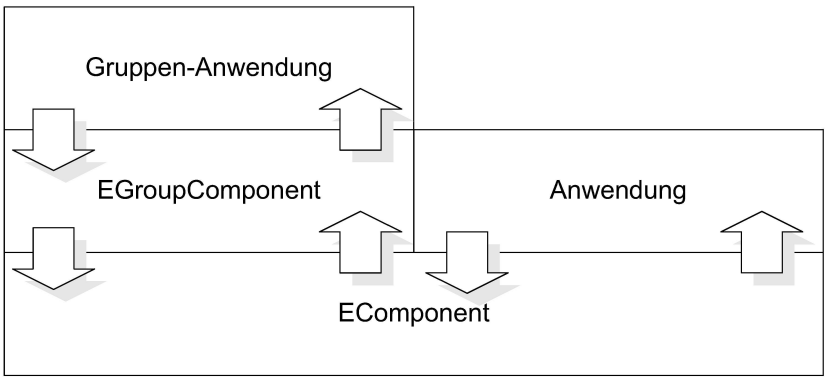

Abbildung 4.2: Gruppenanwendungen und Anwendungen beeinflussen sich nicht

4.2.2 Ablauf

Damit ein Anwender einen Gruppendienst zur Verfügung stellen kann, verwendet er den *WilliEd*2. Der *Wilied* stellt das aktuelle Lern- und Wissensprofil des Anwerders dar. Im *Wilied* startet der Anwender den gewünschten Gruppendienst. Dieser Dienst wird im Anwenderprofil als Angebot vermerkt. Das Anwenderprofil wird durch das ELAN-Rahmenwerk bekantgegeben und interessierte Anwender greifen auf den Gruppendienst zu. Das Rahmenwerk überträgt den Accessor über das Netz zum interessierten Anwender und die Anwendung startet bei diesem. Für das ELAN-Rahmenwerk verhalten sich Gruppenanwendungen identisch zu allen anderen Komponenten. Die Gruppenverwaltungsschichten der Anwendungen kommunizieren untereinander mit Hilfe des Rahmenwerks. Sie tauschen

Informationen bezüglich der Gruppe aus. Die Gruppenanwendungen kommunizieren über die Gruppenkommunikationsschicht miteinander und tauschen darüber anwendungsspezifische Informationen aus. Abbildung 4.3 verdeutlicht die einzelnen Startabläufe und stellt die Kommunikation zwischen den einzelnen Schichten dar.

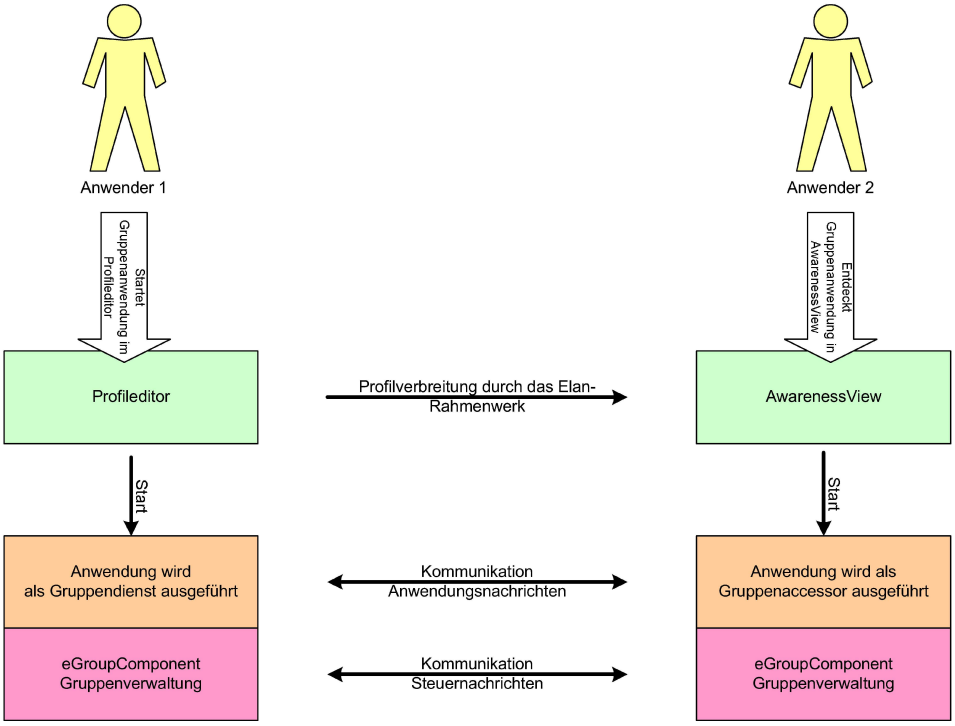

Abbildung 4.3: Abläufe bei Konzept II

4.2.3 Diskussion

Ein klarer Nachteil gegenüber Konzept I ist der Mehraufwand bei Neupositionierung eines Gruppendienstes innerhalb des ELAN-Rahmenwerks. Da Anwendung und Kommunikationsschicht eine gemeinsame Instanz sind, müsste die Anwendung vorübergehend gestoppt werden. Positionsänderungen von Diensten sind zum aktuellen Zeitpunkt innerhalb von ELAN schwierig. Ein Test mit den zur Verfügung stehenden Mitteln konnte nicht realisiert werden. Jedoch kann das Problem vernachlässigt werden, da die Rolle eines zentralen Gruppenverwalters in Ad-Hoc Netzen problematisch ist, siehe Abschnitt 4.5.3. Dieses Problem haben beide Konzepte gemein.

4.2.4 Vorteile

Gegenüber Konzept I wird kein Kommunikationsprotokoll auf Nachrichtenbasis zwischen Anwendung und Gruppenverwaltung benötigt. Dadurch wird das Nachrichtenaufkommen der Ext In&Out Schnittstelle erheblich reduziert.
Die Gruppenkommunikationsschicht setzt direkt auf der bestehenden Schnittstelle für ELAN-Komponenten auf. Dadurch müssen keine Veränderungen innerhalb des ELAN-Rahmenwerks vorgenommen werden. Aus diesem Grund sind durch die Gruppenkommunikationsschicht keine zusätzlichen Probleme für bestehende Komponenten zu erwarten.
Durch das verwendete Schichtenmodell wird eine einheitliche Schnittstelle zur Realisierung von Gruppenanwendungen zur Verfügung gestellt. Eine übersichtliche, klar definierte Schnittstelle erleichtert die Entwicklung von Gruppenanwendungen erheblich.
Außerdem wurden die Abläufe zum Starten und Stoppen der einzelnen Komponenten bereits erfolgreich bei der Implementierung des Dateifreigabedienstes verwendet - siehe Kapitel 3. Das 'Dienst und Accessor' Konzept von ELAN hat sich als zuverlässig erwiesen, dies läßt wenige Probleme für eine Implementierung von Konzept II erwarten.

4.3 Implementierung

Unter Berücksichtigung der verschiedenen Vor- und Nachteile sowie der gestellten Anforderungen wurde entschieden, Konzept II als Gruppenkommunikationsschicht zu implementieren. Dabei wurde die Klasse `EGroupComponent` realisiert. Diese bildet die Grundlage für alle Gruppenkommunikationsdienste und realisiert die nötigen Schnittstellen.

4.3.1 Verwaltung von Gruppen

Die Verwaltung der Gruppe findet zentral statt. Es gibt genau einen Gruppenvertreter, der als Verwalter fungiert. Der Gruppenvertreter wird als Dienst implementiert, alle weiteren Gruppenmitglieder sind Accessoren. Die Gruppenkommunikationsschicht verwaltet die Gruppe. Zu diesem Zweck unterscheiden wir zwischen Gruppenvertreter und Mitgliedern. Es wird ein zentraler Gruppenvertreter bestimmt, Mitglieder müssen sich bei diesem an- und abmelden. Der zentrale Gruppenvertreter ist derjenige, welcher die Gruppe initialisiert hat. Dabei ist das Wissen nützlich, daß ein Anwender gezielt einen Gruppendienst gestartet hat oder benutzen möchte. Die Gruppenkommunikationsschicht kann mit Hilfe des Anwendungsnamens erkennen, ob der Anwender einen Dienst gestartet hat oder einen Dienst benutzen möchte. Endet der Anwendungsname auf `Accessor`, möchte der Anwender einen Dienst benutzen, also einer Gruppe beitreten. Damit die Gruppenanwendung der Gruppe beitreten kann, muß diese mit dem Dienst in Kontakt treten. Dazu werden Name und IP-Adresse der Anwendung benötigt, welche als Gruppenverwalter und somit als Dienst auftritt. Das ELAN-Rahmenwerk schickt jedem Accessor diese Informationen in der gesonderten Nachricht `lego_StartServiceUsage` nach dem Start der Anwendung zu. Dienste bekommen beim Start durch das Rahmenwerk keine Nachricht durch *Lego* zugestellt. Um Dienste identifizieren zu können wird der Anwendungsname verwendet. Accessoren enden innerhalb von ELAN im Anwendungsnamen immer mit `Accessor`. Endet der Anwendungsname nicht mit `Accessor`, wurde die Anwendung als Dienst gestartet. Daher

initialisiert die Gruppenkommunikationsschicht sich als Gruppenvertreter - es entsteht also eine neue Gruppe. Andere Anwender können nun gezielt beitreten. Zur Verwaltung der Mitglieder verwendet die Gruppenkommunikationsschicht Steuernachrichten. Diese Nachrichten werden zwischen den einzelnen Gruppenmitgliedern über die Ext In&Out Schnittstelle verschickt. Dies ist eine Funktionalität der verwendeten Klasse `EComponent`, welche als Grundlage der Gruppenkommunikationsschicht dient. Gruppenanwendungen ist dies transparent. Für den Versand der Steuernachrichten ist es wichtig, die auf der Gruppenkommunikationsschicht laufende Anwendung zu kennen. Damit Nachrichten über die Ext In&Out Schnittstelle versendet werden können, wird jeweils ein Komponentenname als Empfänger und als Absender benötigt. Da die Gruppenkommunikationsschicht nicht als eigenständige Komponente im ELAN-Rahmenwerk auftritt, muß der Name der darauf laufenden Anwendung verwendet werden. Folgende Steuernachrichten werden von der Gruppenverwaltung verwendet:

- `joinGroup` - ein Accessor möchte der Gruppe beitreten
- `leaveGroup` - ein Accessor hat die Gruppe verlassen und meldet sich ab
- `closeGroup` - die Gruppe wurde geschlossen und die Accessoren informiert

joinGroup

Absender	noch unbekanntes Gruppenmitglied
Empfänger	Gruppenvertreter
Daten	`appName, appIp, Nickname`

Der Absender meldet sich beim Gruppenvertreter an. Dazu teilt er dem Empfänger den Namen der Anwendung, die IP Adresse und den Nickname seines Anwenders mit. Der empfangende Gruppenvertreter teilt dem neuem Mitglied eine eindeutige Kennung, mit Hilfe der Klasse `IdMaker` zu (siehe 3.3.1). Die erhaltenen Informationen werden mit der zugeteilten Kennung als Schlüssel in einem Dictionary gespeichert. Als Antwort schickt der Empfänger dem neuem Gruppenmitglied die Nachricht `joinAck(id)`.

joinAck

Absender	Gruppenvertreter
Empfänger	neues Gruppenmitglied
Daten	`id`

Der Empfänger wurde in die Gruppe aufgenommen und ihm vom Gruppenvertreter eine eindeutige Mitgliedsnummer, die `Id` zugewiesen. Der Empfänger merkt sich die Mitgliedsnummer und verwendet sie bei zukünftigen Nachrichten zur Identifizierung. Der Anwendungsname und IP-Adresse sind zur Identifizierung nicht ausreichend, da mehrere Instanzen der gleichen Anwendung nicht unterschieden werden können. Dies soll aber nicht einschränkt werden. Durch die eindeutige Mitgliedsnummer ist es möglich, verschiedene Instanzen der gleichen Anwendung zu nutzen. Der Anwender kann also in verschiedenen Gruppen aktiv sein.

leaveGroup

Absender	Gruppenmitglied
Empfänger	Gruppenvertreter
Daten	`id`

Der Absender teilt dem Gruppenvertreter mit, daß er aus der Gruppe austreten möchte - die Anwendung wurde geschlossen. Zur eindeutigen Identifizierung schickt er seine Mitgliedsnummer, die `Id`, mit.

closedGroup

Absender	Gruppenverwalter
Empfänger	alle Gruppenmitglieder
Daten	keine

Der Absender schickt diese Nachricht an alle Mitglieder der Gruppe. Er informiert darüber, daß die Gruppe geschlossen wird - der Anwender hat die Anwendung beendet. Die Empfänger der Nachricht können angemessen auf diese Nachricht reagieren.

4.3.2 Anwendungsschnittstelle

Gruppenanwendungen benutzen die Gruppenkommunikationsschicht mit Hilfe der Anwendungsschnittstelle. Um diese zu nutzen, importiert die Anwendung die Klasse `EGroupComponent` aus dem Rahmenwerk. Die Anwendung wird dann in einer eigenen Klasse implementiert, die von `EGroupComponent` erbt. Im Konstruktor der Anwendung muß dann der Konstruktor der Basisklasse `EGroupComponent` explizit gerufen werden. Anschließend stehen alle öffentlichen Methoden und Attribute der Elternklasse zur Verfügung und definieren in ihrer Gesamtheit die Anwendungsschnittstelle. Methoden und Attribute die nicht von der Anwendung verwendet werden dürfen, sind innerhalb der Klasse `EGroupComponent` als privat implementiert worden. Die Methoden der Anwendungsschnittstelle sind:

- `groupBootGUI` - die Anwendungs-GUI erzeugen
- `groupExtIn` - eine Gruppennachricht trifft ein
- `groupMembers` - die aktuelle Liste der Gruppenmitglieder trifft ein
- `groupPost` - eine Nachricht an die Gruppe senden

Bei den einzelnen Methoden ist zu unterscheiden, ob diese von der Gruppenkommunikationsschicht oder der Anwendung gerufen werden. Darüber kann bestimmt werden, in welche Richtung die Schnittstelle funktioniert. Ebenso bestimmt die Schnittstellenrichtung, durch welche Schicht der ausführbaren Programmcode für eine Methode erstellt wurde. Im folgendem Abschnitt werden die Methoden der Schnittstelle im Einzelnen besprochen. Dabei ist jeweils zur Identifizierung der Schnittstellenrichtung angegeben, wer diese Methode ruft und implementiert.

groupBootGUI

ausgeführt von:	Gruppenkommunikation
implementiert durch:	Anwendung
Parameter	`parent`

Die Methode `groupBootGUI` bildet die Schnittstelle zur grafischen Oberfläche. Falls eine GUI von der Anwendung gewünscht wird, kann die Anwendung mit Hilfe der Referenz auf das Elternfenster (`parent`) ein eigenes Widget erzeugen und gibt dieses anschließend mit `return` an die Gruppenkommunikationsschicht zurück. Die Gruppenkommunikation verwendet die Methode `bootGUI` der Klasse `EGroupComponent`, welche die Schnittstelle für Komponenten mit GUI ist. Die Gruppenkommunikationsschicht meldet dort einen `QWidgetStack` als Fenster an. Anschließend erzeugt sie ein eigenes Widget und legt dieses auf dem Widgetstapel ab. Das erzeugte Fenster verwendet die Gruppenkommunikationsschicht für eigene Ausgaben. Mit Hilfe des Widgetstapels kann die Anwendung in den Hintergrund geschaltet werden. Ein modaler Dialog würde das gesamte ELAN-Rahmenwerk blockieren und ist daher ungeeignet. Die Gruppenkommunikationsschicht ruft die Methode `groupBootGUI` und bekommt damit das in der Anwendung erzeugte Widget zurück. Dieses Widget legt die Gruppenkommunikationsschicht ebenfalls auf dem Widgetstapel ab. Wird eine Anwendung als Accessor gestartet, muß auf das Eintreffen der Gruppeninformationen gewartet werden. Solange wird durch die Gruppenkommunikationsschicht das eigene Widget auf dem Widgetstapel dargestellt. Die Klasse `QWidgetStack` bietet hierfür die Methode `raiseWidget(Id)` an. Die entsprechende Id des Widget wird beim Anmelden der einzelnen Widgets, durch die Methode `addWidget(Widget)` zurückgegeben. Hat die Gruppenkommunikation ihre Synchronisationsarbeiten abgeschlossen, wird durch den erneuten Methodenaufruf `raiseWidget(Id)` das Fenster der Anwendung auf dem Widgetstapel angezeigt.

groupExtIn

ausgeführt von:	Gruppenkommunikation
implementiert durch:	Anwendung
Parameter	`msg`

Damit Anwendungen auf eintreffende Nachrichten von den Gruppenmitgliedern reagieren können, existiert die Schnittstelle `groupExtIn`. Als Parameter `msg` wird die Nachricht übergeben. Die Auswertung der Nachricht ist Aufgabe der Anwendung. Intern realisiert die Gruppenkommunikationsschicht die Nachrichtenkommunikation über die Ext In&Out Schnittstelle der darunter liegenden Schicht. Trifft über die `ExtIn` Schnittstelle eine Nachricht ein, kann dies entweder eine Steuernachricht der Gruppenkommunikation oder eine Nachricht für die Anwendung sein. Da als Empfängername in beiden Fällen die Anwendung angegeben ist, wie unter 4.3.1 beschrieben, kann eine Unterscheidung nur durch den Nachrichtennamen erfolgen. Eintreffende Nachrichten werden dispatched und die zugehörige Methode gerufen. Anwendungsnachrichten sind innerhalb eines zusätzlichen Tupel eingebettet und tragen als ersten Parameter stets das Nachrichtenkommando `groupExtIn`. Daher werden diese Nachrichten immer an die gleichnamige Methode `groupExtIn` übergeben, welche von der Anwendung verarbeitet wird. Kann eine Nachricht nicht dispatched werden, wird diese ignoriert.

groupMembers

ausgeführt von:	Gruppenkommunikation
implementiert von:	Anwendung
Parameter:	`listOfMembers`

Treten Personen in die Gruppe ein oder aus, wird die Anwendung mit dieser Nachricht über die Veränderung der Gruppenzusammensetzung informiert. Möchte die Anwendung diese Information nutzen, kann sie die Methode `groupMembers` überladen. Von der Gruppenkommunikationsschicht wird der Parameter `listOfMembers` übergeben, dieser enthält eine Liste mit allen Namen der Gruppenmitglieder. Dabei können doppelte Namen vorkommen. Tritt ein neues Mitglied in eine Gruppe ein, schickt die Gruppenkommunikation die Steuernachricht `joinGroup` an den Gruppenvertreter. Dieser nimmt das neue Mitglied in die Mitgliederliste auf und schickt die Steuernachricht `joinAck(id)` zurück. Anschließend erzeugt der Gruppenvertreter eine aktuelle Liste aller Mitgliedernamen und schickt diese, mit Hilfe der `post()` Methode, an alle Gruppenmitglieder.

groupPost

ausgeführt von:	Anwendung
implementiert durch:	Gruppenkommunikation
Parameter:	`msg`

Ruft eine Anwendung die Methode `groupPost(msg)` der Gruppenkommunikation mit ihrer Nachricht `msg`, wird diese an alle Gruppenmitglieder gesendet. Die Nachricht trifft bei den Gruppenmitgliedern über die `groupExtIn` Schnittstelle ein und kann dort verarbeitet werden. Die Gruppenkommunikationsschicht einer Anwendung weiß, ob sie Gruppenvertreter oder nur einfaches Mitglied der Gruppe ist. Dies wird beim Start festgestellt und intern im Attribut `type` festgehalten. Trifft eine Anwendungsnachricht bei der Gruppenkommunikationsschicht ein und die Anwendung ist nur Gruppenmitglied, wird diese Nachricht über die `post` Methode an den Gruppenvertreter geschickt. Als Nachrichtenkommando wird dabei `groupPost` eingetragen. Beim Gruppenvertreter trifft über die `ExtIn` Schnittstelle die Nachricht ein und wird dispatched (siehe 4.3.2). Durch den Dispatcher wird die Methode `groupPost` der Gruppenkommunikationsschicht gerufen. Diesmal findet der Aufruf allerdings beim Gruppenvertreter statt. Der Gruppenvertreter schickt anschließend die Nachricht weiter an alle ihm bekannten Gruppenmitglieder.
Abbildung 4.4 illustriert eine mögliche Reihenfolge, in der die einzelnen Nachrichten an die Gruppenmitglieder verschickt werden. Die Nummern der einzelnen Nachrichten entsprechen dabei der Reihenfolge, in der diese verschickt werden. Diese Reihenfolge ergibt sich aus dem Zeitpunkt des Beitretens der einzelnen Gruppenmitglieder.

Abbildung 4.5 zeigt drei Instanzen von Gruppenchat. Ein Mitglied schickt 'Nachricht 1' an die Gruppe und diese trifft beim Verwalter ein. Der Verwalter schickt dann 'Nachricht 1' nacheinander an die beiden Mitglieder. Danach schickt das zweite Mitglied die 'Nachricht 2' an die Gruppen. Auch diese wird vom Verwalter verteilt. Die Reihenfolge bleibt identisch, da die Gruppe unverändert ist.

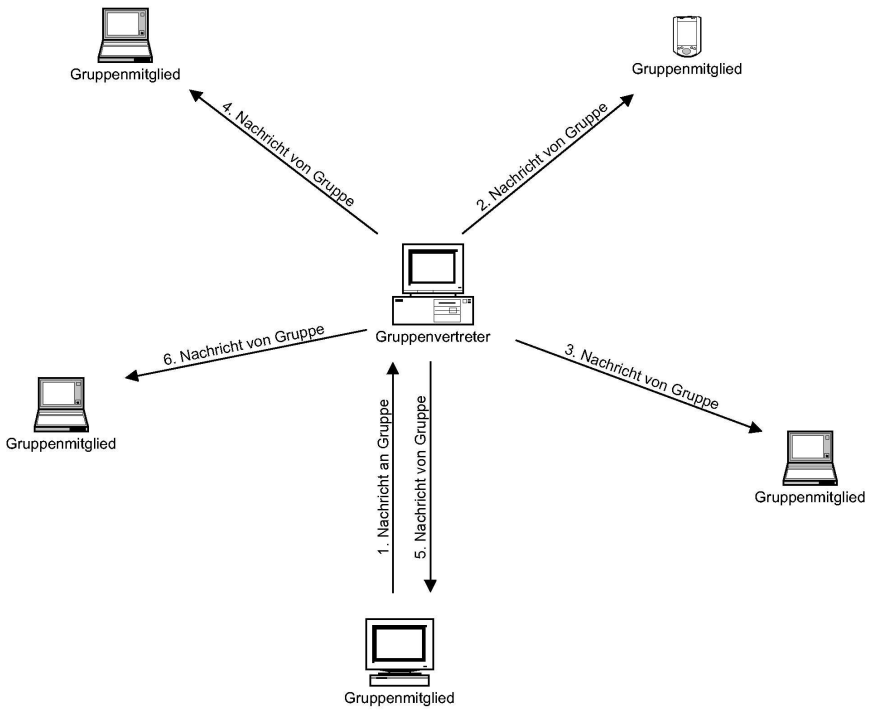

Abbildung 4.4: Bsp. für die Reihenfolge der Nachrichtenzustellungen

4.4 Beispiele für Gruppenanwendungen und deren Realisierung

Um die Funktion der Gruppenkommunikationsschicht zu demonstrieren, wurden zwei Anwendungen prototypisch implementiert. Diese nutzen die realisierte Anwendungsschnittstelle, um ihre Aufgaben zu erfüllen. Es wurde jeweils der Dienst als auch der geerbte Accessor, wie in Abschnitt 4.2.1 beschrieben, realisiert.

4.4.1 Gruppenchat

Der Gruppenchat stellt die einfachste denkbare Anwendung dar. Diese ist auf Basis der Gruppenkommunikationsschicht mit sehr geringem Aufwand realisierbar. Folgende Anforderungen werden an den `groupChat` gestellt:

- Alle Gruppenmitglieder sollen miteinander kommunizieren können,
- die Mitgliedernamen aller Gruppenmitglieder sollen für jeden zu sehen sein, und
- es gibt eine Historie, in der geschriebene Nachrichten inkl. Mitgliedername des Absenders zu sehen sind.

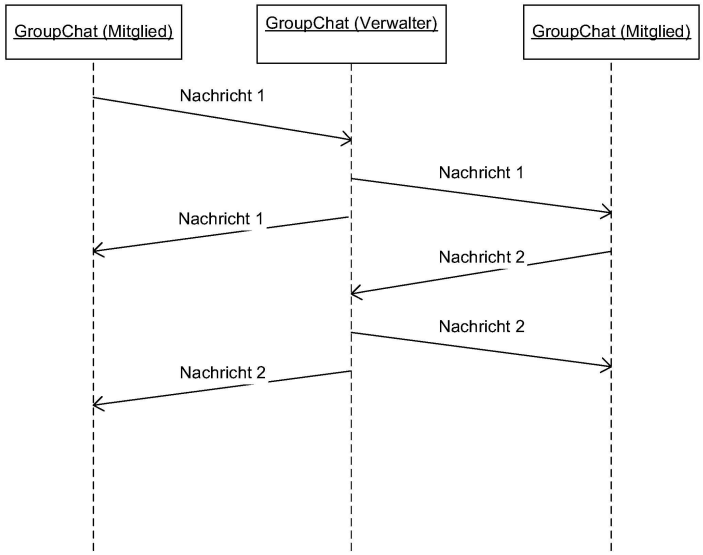

Abbildung 4.5: UML Sequenzdiagramm der Gruppenkommunikatiosnachrichten

Implementierung

Abbildung 4.6 zeigt den Python-Quellcode der Anwendung `groupChat` zum besseren Verständnis. Auf die `include` Anweisungen zur Verwendung existierender Klassen wurde dabei aus Platzgründen verzichtet. Trotzdem dokumentiert der Quellcode gut, daß mit geringem Aufwand ein einfacher Gruppendienst realisiert werden kann. Bei der Realisierung der Anwendung kommt eine einfache grafische Anwenderschnittstelle zum Einsatz.

Ein Fenster für die Mitgliederanzeigen wird benötigt, dafür wurde eine `QListView` verwendet - wie bereits bei der Dateifreigabe. Für die Anzeige der Historie wird ein `QTextEdit` gewählt, dabei wird die Eigenschaften 'nur lesen' bei diesem Fenster eingeschaltet. Dadurch kann Text nicht, wie in einem Editor, verändert werden. Außerdem wird das Anzeigeformat Richtext [Qt05] ausgewählt. Dadurch sind einfache Formatierungen der Textanzeige möglich. Außerdem wird eine Zeile zur Texteingabe benötigt, dafür eignet sich eine `QLineEdit` Instanz. Ein `QPushButton` wird hinzugefügt und mit dem Wort 'say' beschriftet. Zwei Qt-Signale sind für die Realisierung der Anwendung von Bedeutung. Zum Einen das Signal `returnPressed()` der Eingabezeile, sowie `clicked()` des Druckknopfs. In beiden Fällen ist der Grund für das Auslösen des Signals identisch. Der Anwender hat einen Text eingegeben und möchte diesen an die Gruppe schicken. Daher werden diese beiden Signale mit dem einzigen existierenden Slot innerhalb der Anwendung, der Methode `slotSay` verbunden. Wird eines der beiden Signale ausgelöst, wird der Slot gerufen und ausgeführt. Innerhalb des Slots wird der Text aus der Eingabezeile ausgelesen und diese anschließend gelöscht und der Mitgliedername des Anwenders wird ausgelesen. Mitgliedsname und Text werden dann zusammen in ein Tupel gepackt und mit Hilfe der Methode `groupPost(msg)` als Nachricht an die Gruppe geschickt.
Die Methode `groupExtIn(msg)` wird vom Gruppenchat überladen, dadurch kann sie auf Nachrichten reagieren, die an die Gruppen geschickt werden. Trifft eine solche Nachricht

```
class groupChat( EGroupComponent ):
    u"""groupChat 4 Elan"""
    name = "groupChat"
    spec = 0.9
    service = "specified"
    version = __version__
    author = __author__
    purpose = __doc__
    gui = "1"

    def __init__( self ):
        EGroupComponent.__init__( self )

    def groupBootGUI( self, parent ):
        self.widget = groupChatMainWindow( parent )
        qt.QObject.connect( self.widget.pbSay, qt.SIGNAL('
            clicked()'), self.slotSay )
        qt.QObject.connect( self.widget.leSay, qt.SIGNAL('
            returnPressed()'), self.slotSay )
        return self.widget

    def slotSay( self ):
        msg = str( self.widget.leSay.text() )
        self.widget.leSay.clear()
        self.groupPost( (self.myNickname, msg) )

    def groupExtIn( self, parm):
        history = self.widget.teHistory.text()
        nick, msg = parm
        msg = '<b>%s</b> %s <br>'% (nick, msg)
        self.widget.teHistory.setText( history.append( msg ) )

    def groupMembers( self, memberList ):
        itemIt = qt.QListViewItemIterator( self.widget.lwUsers )
        while itemIt.current():
            self.widget.lwUsers.takeItem( itemIt.current() )
        for name in memberList:
            newItem = qt.QListViewItem( self.widget.lwUsers,
                None )
            newItem.setText(0, str(name) )
```

Abbildung 4.6: Python-Quellcode der Gruppenanwendung Gruppenchat - groupChat.py

ein, muß sie von einer Gruppenchatanwendung stammen. Denn nur gleichartige Anwendungen organisieren sich als Gruppe, dafür sorgt die Gruppenkommunikationsschicht. Da die Anwendung nur an einem Punkt im Programmablauf Nachrichten an die Gruppe verschickt, nämlich in der Methode `slotSay(msg)`, muß die Nachricht von dort kommen. Dies bedeutet, daß die Methode `groupExtIn(parm)` im Gruppenchat immer dann von der Gruppenkommunikationsschicht gerufen wird, wenn ein Gruppenmitglied aktiv am Chat teilnimmt - also eine Nachricht schickt. Der Gruppenchat erhält in der Methode `groupExtIn(parm)` die Tupel, welche Mitgliedsname und Text enthalten. Diese packt die Anwendung wieder aus, zerlegt sie also in Mitgliedsname und Text. Diese Information wird aufbereitet, der Mitgliedername vor den Text geschrieben, mittels Fettschreibung hervorgehoben und an die vorhandene Chathistorie angehängt.
Der Gruppenchat überlädt die Methode `groupMembers(listOfMembers)`, um die Mitgliederliste auszuwerten. Diese Liste trifft unregelmäßig von der Gruppenkommunikationsschicht ein und die Methode wird gerufen. Zur Aufbereitung erzeugt der Gruppenchat für jedes Mitglied ein `QListViewItem`, schreibt den Nickname hinein und hängt es bei der `QListView` an. Die Mitgliedernamen tauchen untereinander in der verwendeten `QListView` auf.

4.4.2 Gemeinsame Zeichenfläche

Um zu zeigen, daß die Gruppenkommunikationsschicht eine probate Abstraktion zur Entwicklung von Gruppenanwendungen darstellt, wurde eine weitere Anwendung implementiert, dei gemeinsame Zeichenfläche, auf der alle Gruppenmitglieder mit festgelegten Werkzeugen arbeiten können. Die Anforderungen an die Zeichenfläche sind wie folgt:

- Die Anzahl der Werkzeuge ist beschränkt und allen Teilnehmern stehen die gleichen Werkzeuge zur Verfügung,
- alle Gruppenmitglieder können auf der Fläche malen - es findet keine Erlaubnissteuerung statt,
- alle Gruppenmitglieder werden durch ihren Mitgliedsnamen anderen Teilnehmern bekanntgemacht,
- Aktionen, die vor dem Beitreten in die Gruppe stattfanden, werden nicht nachträglich kommuniziert.

Um die Auswahl zu beschränken, wurden die Werkzeuge *Text*, *Rechteck* und *Ellipse* ausgewählt. Zusätzlich soll die Möglichkeit bestehen eine Füllfarbe zu wählen, falls das jeweilige Werkzeug dies unterstützt. Ein Farbauswahldialog ist dafür zu verwenden.

Der Malbereich

Für die Darstellung von zweidimensionalen grafischen Objekten ist die Qt-Klasse `QCanvas` optimiert. Existiert ein solches Canvas, kann mit Hilfe von zusätzlich spezialisierten Klassen auf dem Canvas gezeichnet werden. Diese Klassen bieten verschiedenartige Elemente an und werden gemeinsam von der Klasse `QCanvasItem` abgeleitet. Für die realisierten

Abbildung 4.7: Werkzeugleiste

Werkzeuge sind die Qt-Klasse `QCanvasText`, `QCanvasRectangle` und `QCanvasEllipse` verwendet worden. Es existieren weitaus mehr spezialisierte Klassen für grafische Objekte, deren vollständige Auflistung der Qt-Dokumentation unter [Qt05] entnommen werden kann.
Qt trennt die Darstellungs- und Verwaltungslogik auf. `QCanvas` leistet die Verwaltung des grafischen Bereichs und `QCanvasView` ist für die Ansicht verantwortlich. Daher ist es nötig, dem Konstruktor von `QCanvasView` eine Referenz auf das verwendete `QCanvas` mitzuteilen. Diese Referenz kann auch nachträglich mitgeteilt werden, aber solange sie nicht bekannt ist, kann die Ansicht nichts darstellen. `QCanvasView` ist ein Widget, welches es ermöglicht, diese innerhalb von den üblichen Layoutelementen von Qt zu verwenden. Die Canvasansicht reagiert zusätzlich auf Ereignisse der Maus. Daher wurde durch Vererbung die eigene Ansichtklasse `whiteCanvasView` abgeleitet. Durch Überladen entsprechender Methoden wird innerhalb dieser Klasse auf eintretende Mausereignisse reagiert. Folgende Tabelle bildet kurz den Zusammenhang zwischen Mausereignis und dadurch gerufener Methoden ab:

Ereignis	Methode
Maustaste wurde gedrückt	`contentsMousePressEvent(QMousEvent)`
Maus wird mit gedrückter Taste bewegt	`contentsMouseMoveEvent(QMousEvent)`
Maustaste wurde losgelassen	`contentsMouseReleaseEvent(QMousEvent)`

Alle Ereignisse übergeben der jeweiligen gerufenen Methode als Parameter eine Instanz der Klasse `QMouseEvent`. `QMouseEvent` beinhaltet alle Informationen über die Maus zum Zeitpunkt des Ereignisses. Diese Informationen werden ausgewertet und wichtige Informationen für die Zeichenfläche gewonnen. Die Position des Mauszeigers ist für die Anwendung von Interesse. Dazu hält das Eventobjekt die Methoden `x()` und `y()` bereit. Die zurückgelieferte X- bzw. Y-Position bezieht sich dabei auf die linke, obere Ecke des Canvas. Dort beginnt der Wertebereich mit der Koordinate (0/0).

Die Werkzeugleiste

Die Werkzeugleiste kann einfach mit dem Qt-Designer realisiert werden. Abbildung 4.7 stellt die realisierte Werkzeugleiste dar. Dazu wurden drei Druckknöpfe erzeugt, die auf Toggle eingestellt sind. Dies bedeutet, daß der Knopf exklusiv ein- oder ausgeschaltet ist. Anschließend wurden die Druckknöpfe in einer so genannten Druckknopfgruppe arrangiert. Diese Druckknopfgruppe bestimmt das Verhalten aller in ihr befindlichen Druckknöpfe. Die Druckknopfgruppe wurde auf exklusiv gesetzt. Dadurch wird erreicht, daß nur einer der Druckknöpfe innerhalb der Gruppe gedrückt sein kann. Wird ein anderer Knopf gewählt, wird dadurch der vorherige Druckknopf losgelassen. Dies ist sinnvoll, da die Auswahl auf ein Werkzeug beschränkt sein soll.
In einer zusätzlichen Anordnungsgruppe wird eine Auswahlbox mit dem Text 'füllen', sowie

ein weiterer Druckknopf 'Farbe' hinzugefügt. Diese Elemente sind später für die Auswahl der Füllfarbe bestimmt. Bereits innerhalb des Qt-Designer lassen sich einige Slots sinnvoll mit Signalen verbinden. Eine erste Funktionalität wird dadurch vorgegeben:

- Das Text-Werkzeug wird immer mit einer Farbe gefüllt, dies legt das verwendete `QCanvasText` Element fest. Das Signal `toggled(bool)` des Druckknopfs für das Textwerkzeug wurde mit den beiden Slots `setDisabled(bool)` und `setChecked(bool)` der Farbauswahlbox verbunden. Dadurch kann das Füllen mit Farbe nicht abgeschaltet werden.
- Das Ellipse-Werkzeug wird ebenfalls immer mit einer Farbe gefüllt. Ein `QPen` wird vom benutztem `QCanvasEllipse` Element nicht verwendet. Dies bedeutet, daß kein Rand gezeichnet wird. Daher wurde das Signal `toggled(bool)` von diesem Druckknopf mit den beiden Slots `setDisabled(bool)` und `setChecked(bool)` der Farbauswahlbox verbunden. Es wird so das gleiche Verhalten wie beim Textwerkzeug erreicht - das Füllen mit Farbe ist immer aktiviert.
- Außerdem wir das Signal `toggled(bool)` der Farbauswahlbox mit dem Slot `setEnabled(bool)` des Druckknopf für die Farbauswahl verbunden. So kann nur eine Farbe ausgewählt werden, wenn die Farbauswahlbox vorher aktiviert wurde, sonst ist der Druckknopf nicht nutzbar.

Das Layout der einzelnen Elemente

Leider sind nicht alle verwendeten Elemente der Zeichenflächenanwendung im Qt-Designer realisierbar. Der QT-Designer bietet nur eine eingeschränkte Auswahl der tatsächlich existierenden Elemente der Qt-Klassenbiliothek an. Daher mußte ein Layout der einzelnen Elemente teilweise per Hand erzeugt werden. Dies ist für den Malbereich, eine `QCanvasView`, der Fall. Eine ausführlichere Beschreibung über `QCanvasView` wird im Abschnitt 4.4.2 gegeben.
Für das Gesamtlayout wurde ein `QGridLayout` gewählt. Dieses sorgt dafür, daß alle weiteren Layouts sich an einem Gitter ausrichten. Dadurch kann das Widget auf Größenänderungen reagieren. Innerhalb des Gitters wird ein `QVBoxLayout` verwendet, welches alle angemeldeten Widgets übereinander organisiert. Die Reihenfolge der Anmeldung bestimmt dabei die Position der einzelnen Widgets. Eine Instanz der Werkzeugleiste wird erzeugt und angemeldet. Dadurch erscheint die Werkzeugleiste oberhalb der restlichen Fenster. Zusätzlich wird `QHBoxLayout` instanziert beim Layout angemeldet, dadurch erscheint es unter der Werkzeugleiste. Dieses neue Layout arrangiert Widgets nebeneinander. Bei diesem Layout werden anschließend erzeugte Instanzen der Klassen `QCanvasView` und `QListView` angemeldet. Es entstehen zwei nebeneinander liegende Widgets, die sich unterhalb der Werkzeugleiste befinden. Abbildung 4.8 verdeutlicht das realisierte Layout nochmals.

Die Mitgliederansicht

Wie bereits beim Gruppenchat werden in dieser Ansicht die Namen aller Gruppenmitglieder dargestellt. Das Verhalten und die Implementierung sind analog zum Gruppenchat. Die Methode `groupMembers(listofMembers)` nimmt die aktuellen Mitgliederliste entgegen und erzeugt für jedes Mitglied eine `QListViewItem`. Anschließend wird der Name mit

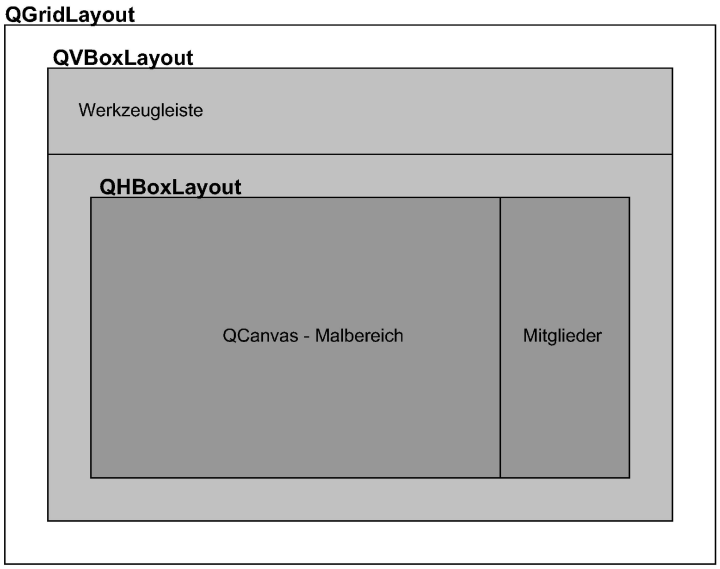

Abbildung 4.8: Layout der Zeichenflächenanwendung

Hilfe der Methode `setText(text)` in das Element geschrieben. Jedes `QlistViewItem` wird der `QListView`-Ansicht bekannt gemacht, die Elemente erscheinen untereinander.

Die Dialogfenster

Ein Standarddialog aus Qt erlaubt dem Anwender, eine Farbe zu wählen. Dabei hat der Anwender die Möglichkeit, bevorzugte Farben abzuspeichern. Dadurch kann er diese bei erneuter Farbwahl leichter wiederfinden. Der beschriebene Dialog wird von der Klasse `QColorDialog` zur Verfügung gestellt. Zur Verwendung von `QColorDialog` wurde ein Slot innerhalb der Fensterklasse der Anwendung realisiert. Zusätzlich wird die aktuell ausgewählte Farbe in der Variablen `currentColor` gespeichert. Der Slot wurde mit dem Signal `clicked()` des Druckknopfs zur Farbwahl verbunden. Innerhalb des Slot wird die statische Methode `getColor` der Klasse `QColorDialog` verwendet. Statische Methoden benötigen zum Rufen kein vorher erzeugtes Objekt der Klasse. Der Methode `getColor` wird als Parameter die zuletzt ausgewählte Farbe, ihr Elternfenster sowie ein Name für den Dialog übergeben. Durch diesen Aufruf erscheint das Dialogfenster und blockiert die Anwendung, bis der Anwender eine Farbe ausgewählt hat. Die ausgewählte Farbe wird als Rückgabewert geliefert und als Pinselfarbe gesetzt. Zusätzlich wird noch ein kleines Rechteck in der ausgewählten Farbe erzeugt und auf den Druckknopf gelegt. Dadurch sieht der Anwender immer, welche Füllfarbe gerade ausgewählt ist.

Der zweite verwendete Dialog wurde mit dem Qt-Designer erzeugt. Der Dialog hat die Aufgabe, einen Text vom Anwender entgegenzunehmen. Zu diesem Zweck wurde eine Texteingabezeile (`QLineEdit`) und die beiden Druckknöpfe 'Ok' und 'Abbrechen' hinzugefügt. Wird in Qt ein Dialog akzeptiert, ruft dieser die Methode `accept()`. Wird ein Dialog abgelehnt, ruft er `reject()`. Daher wurden die Signale `returnPressed()` der Eingabezeile

und `clicked()` des Druckknopfes 'Ok' mit dem Slot `accept()` verbunden. Außerdem das Signal `clicked()` des Druckknopfes 'Abbrechen' mit dem Slot `reject()`. Um den im Qt-Designer erzeugten Dialog nutzen zu können, wurde die neue Klasse `WhiteTextInput` erstellt. Diese Klasse erbt den Dialog und überlädt die Methoden `accept()` sowie `reject()`. Auf diese Art wird innerhalb von `accept()` der Text aus der Eingabezeile ausgelesen und im Attribut `text` gespeichert. Da das Attribut `text` nicht privat ist, kann es von außen jederzeit ausgelesen werden. Zur Realisierung des Textwerkzeugs wird eine Instanz der Klasse `WhiteTextInput` als Eingabedialog verwendet.

Darstellung der einzelnen Werkzeuge auf dem Malbereich

Um die Auswahl eines Werkzeugs zu bemerken, werden die einzelnen `toggled(bool)` Signale der Werkzeugleistenknöpfe mit entsprechenden Slots verbunden. Tritt das entsprechende Signal auf, wird in der Variable `type` vermerkt, welches Werkzeug ausgewählt ist. Die folgenden Beschreibungen der einzelnen Werkzeuge gehen davon aus, daß der Anwender zuvor das entsprechende Werkzeug in der Werkzeugleiste ausgewählt hat. Weiter werden die Abläufe beschrieben, wie auf der eigentlichen Malfläche die Darstellung erzeugt wird.

Das Textwerkzeug Sobald der Anwender die linke Maustaste auf dem Malbereich drückt, wird der Texteingabedialog der Klasse `WhiteTextInput` angezeigt. Eine Erläuterung zu diesem Dialog steht unter 4.4.2. Wurde der Dialog akzeptiert, wird der eingegebene Text ausgelesen. Anschließend wird ein `QCanvasText` Objekt erzeugt, welches als Parameter den Text und das zu verwendende Canvas übergeben bekommt - beides ist bekannt. Mit Hilfe des Mausereignisses, beschrieben unter 4.4.2, wird die Position auf dem Malbereich bestimmt. Das `QCanvasText` Objekt kann anschließend mit der Methode `move(x,y)` an die aktuelle Position gesetzt werden. Außerdem wird die Farbe des Textes durch die Methode `color(color)` festgelegt, diese ist in der Variablen `currentColor` gespeichert. Dadurch erscheint der Text an der Stelle auf dem Malbereich, wo die Maustaste gedrückt wurde. Verschiebungen der Maus bei gedrückter Taste müssen beim Text nicht berücksichtigt werden.

Das Rechteck Zur Darstellung des Rechtecks müssen zwei Mausereignisse berücksichtigt werden. Der Anwender soll das Rechteck auf dem Malbereich aufziehen können. Daher speichert das Ereignis Drücken der linken Maustaste auf dem Malbereich die Startposition in entsprechenden Variablen ab. Bereits zu diesem Zeitpunkt wird eine Instanz der Klasse `QCanvasRectangle` erzeugt. Dabei werden als Parameter die Startkoordinaten und die aktuelle Breite und Höhe, sowie das Canvas übergeben. Die Breite und Höhe sind zum Startzeitpunkt beide Null. Verschiebt der Anwender bei gedrückter Taste die Maus, tritt das zweite Ereignis ein. Die daraus resultierende neue Mausposition sorgt für eine Neuberechnung der Höhe und Breite des `QCanvasRectangle` Objekts. Dabei werden die Absolutbeträge berechnet, denn der Anwender kann die Position der Maus auch dahingehend verändern, daß die einfache Differenz negativ ist. Durch die Methode `setSize(width,height)` wird das `QCanvasRectangle` Objekt entsprechend angepasst und dargestellt. `QCanvasRectangle` Objekte benutzen einen `QPen` und einen `QBrush`. Der

`QPen` bestimmt das Aussehen der Rahmenlinie in Farbe und Form. Die Rahmenlinie wurde bei der Anwendung unveränderlich auf eine durchgehende, ein Punkt starke, schwarze Linie eingestellt. Der `QBrush` bestimmt das Füllmuster und die Füllfarbe. Die Füllmusterauswahl beschränkt sich auf völlig oder gar nicht ausgefüllt. Die Auswahlbox 'füllen' genügt dafür als einfache Auswahlmöglichkeit. Das `toggled(bool)` Signal der Auswahlbox wird an einen sehr simplen Slot gebunden. Dieser Slot hinterlegt je nach Wert das entsprechende Füllmuster in `QBrush`. Der Programmcode für den Slot ist in Abbildung 4.9 abgebildet.

```
def slotToggleFill( self, bool=0):
    self.brush.setStyle( [QBrush.NoBrush, QBrush.
        SolidPattern] [bool] )
```

Abbildung 4.9: slotToggleFill.py

Die Ellipse Die Darstellung der Ellipse ist relativ ähnlich wie die des Rechtecks. Drückt der Anwender die linke Maustaste, wird die Klasse `QCanvasEllipse` mit den Parametern Breite, Höhe und dem Canvas instanziert. Die Mausposition wird mit Hilfe des eingetretenen Mausereignisses bestimmt. Wird die Maus bei gedrückter Taste verschoben, wird die Breite und Höhe berechnet. Dies erfolgt analog zum Rechteck. Auch hier kann mit der Methode `setSize(width,height)`, wie beim Rechteck, die neue Breite und Höhe festgelegt werden. Die Ellipse kennt keine obere, linke Ecke. Die verwendete Koordinate beschreibt den Mittelpunkt. Daher muß mit der Methode `move(x,y)` die Ellipse an die richtige Position gebracht werden. Die neue Position des Mittelpunkts der Ellipse lässt sich wie folgt berechnen: $x_{Position} = x_{StartPkt} + Breite/2$ und $y_{Position} = y_{StartPkt} + Höhe/2$ `QCanvasEllipse` verwendet keinen `QPen`, es wird kein Rand dargestellt. Der verwendete `QBrush` verhält sich analog zum Rechteck.

paintEvent

Alle bisher beschriebenen Aktionen sind im zentralen Hauptfenster der Anwendung, also in der dort verwendeten Klasse `whiteCanvasView`, implementiert. Im Falle der Zeichenfläche kann viel Arbeit bereits innerhalb des Widget erledigt werden. Eine Anwendung besteht aber nicht nur aus ihrem Fenster. Die Kommunikation mit der Gruppe wird in der Anwendungsklasse `whiteboard.py` realisiert. Zur Realisierung sind für die Anwendungsklasse Informationen wichtig, die bisher nur von der Fensterklasse verarbeitet werden, z.B. wann der Anwender ein Werkzeug verwendet, welches Werkzeug dies war oder welcher Brush verwendet wurde. Damit die Anwendungsklasse über diese für sie wichtigen Ereignisse informiert wird, wurden eigene Signale innerhalb der Fensterklasse eingefügt. Der Malbereich sendet das Signal `paintEvent(canvasItem)`, sobald die Maustaste losgelassen wird. Denn zu diesem Zeitpunkt ist innerhalb des Canvaselement alles gespeichert, was für die Anwendungsklasse wichtig ist. Die Hauptklasse der Anwendung verbindet das Signal `paintEvent(CanvasItem)` mit ihrem Slot `slotPaintEvent`. Innerhalb des Slot wird das übergebene `CanvasItem` analysiert und in einem Tupel gespeichert. Dieser Tupel repräsentiert das verwendete Werkzeug mit all seinen Attributen, d.h. ein Marshalling für Canvaselemente.

externalPaint

Die Kommunikation der Gruppe erfolgt über die Gruppenkommunikationsschicht. Eine gemeinsame Zeichenfläche teilt allen anderen Teilnehmern mit, wenn lokal ein Werkzeug verwendet wurde. Diese Information erhält die Anwendung durch das Signal `paintEvent`. Das Canvaselement wurde in einen Tupel verpackt, kann also in dieser Form an die Gruppe geschickt werden. Vor das Tupel wird für den Dispatcher ein Nachrichtenkommando geschrieben. Dazu wird ein weiterer Tupel erzeugt, das Kommando `externalPaint` als String in das erste und der Datentupel in das zweite Attribut gespeichert. Dieser neue Tupel wird anschließend mit der Methode `groupPost(msg)` als Nachricht `msg` verschickt. Trifft bei einem Gruppenmitglied über die `groupExtIn` Schnittstelle eine Nachricht ein, wird sie dispatched - siehe Kapitel 2. Das dispatchen sorgt dafür, daß bei der Zeichenfläche die `externalPaint(Tupel)` Methode ausgeführt wird, wenn ein anderer Teilnehmer etwas gezeichnet hat. Diese Information trifft in Form einer Gruppennachricht ein. Das Rufen der Methode `externalPaint(Tupel)` sorgt dafür, daß aus dem übergebenen Tupel wieder ein entsprechendes Canvaselement erzeugt wird. Mit Hilfe der Canvasansicht wird dieses anschließend dargestellt.

4.5 Diskussion und Evaluation

Die implementierte Lösung demonstriert die Möglichkeit, wie Gruppenkommunikation in einem Ad-Hoc Netz realisiert werden kann. Einzelne Probleme, die berücksichtigt werden sollten, werden im Folgendem kurz umrissen.

4.5.1 Zentraler Gruppenvertreter

In der implementierten Fassung der Gruppenkommunikationsschicht wird ein zentraler Gruppenvertreter verwendet. Dies ist bei Ad-Hoc Netzen nicht vorteilhaft. Durch die Beschaffenheit von Ad-Hoc Netzen kann es vorkommen, daß einzelne Stationen im Netz dauerhaft oder auch temporär nicht erreichbar sind. Ist dies der zentrale Gruppenvertreter, kann die gesamte Gruppe nicht mehr miteinander kommunizieren.
Eine alternative Lösung wäre, die Verwaltung der Gruppe auf verschiedene Gruppenvertreter zu verteilen, z.B. eine Aufteilung der Gruppe in Subgruppen. Für jede Subgruppe existiert dann genau ein Gruppenvertreter, der mit anderen Gruppenvertretern kommuniziert. Wenn nach einem festgesetzten Verfahren solche Vertreter ausgewählt würden, könnte eine vollständige Dezentralisierung erreicht werden. Treten dann Fehler in der Kommunikation des Ad-Hoc Netzes auf, zerfallen die Gruppen in kleinere autonome Gruppen. Es kann nicht mehr die ganze Gruppe miteinander kommunizieren, aber ein Teil und die Kommunikation bleibt auf diese Art erhalten. Auch Verfahren, in denen keine Gruppenvertreter eingesetzt werden, sind denkbar. Jedes Gruppenmitglied kommuniziert dann mit einer ausgewählten Menge von anderen Mitgliedern. Wird die Menge dieser Mitglieder geschickt gewählt, kann die gesamte Gruppe miteinander kommunizieren. Dabei kann es allerdings zu stärkeren Verzögerungen kommen, weil Nachrichten evtl. über viele verschiedene Knoten im Netz übertragen werden.

4.5.2 Nachrichtenaufkommen

Zur Realisierung von Gruppenkommunikationsdiensten muß innerhalb eines Ad-Hoc Netzes eine Multicastfunktionalität realisiert werden. Diese Leistung erbringt die Gruppenkommunikationsschicht, allerdings mit einem deutlichen Nachteil. Es entsteht ein erhöhtes Nachrichtenaufkommen auf dem Netz. In normalen Netzen werden Multicastnachrichten nur einmal verschickt, erst die Router auf der Übertragungsstrecke trennen Multicastnachrichten auf, um Teilnehmer in verschiedenen Subnetzen zu erreichen. Dies wäre wünschenswert für die Kommunikation innerhalb von Ad-Hoc Netzen, müsste dann aber auf Routingebene implementiert sein. Verschiedene Vorschläge für Multicastalgorithmen in Ad-hoc Netzen werden in [Per00, Che01] genauer erläutert. Optimierungen für die erstellte Lösung sind denkbar, erfordern allerdings von den einzelnen Gruppenmitgliedern mehr Wissen über die Gruppe. Dies kann aber innerhalb der Gruppenkommunikationsschicht implementiert werden, so daß die Anwendung davon nichts oder nur sehr wenig mitbekommt.

4.5.3 Position von Gruppenvertretern

Die Position des Gruppenvertreters ist von zentraler Bedeutung. Befindet dieser sich auf einem 'schwachen' Gerät oder ist abzusehen, daß die verwendete Station nicht mehr lange im Ad-Hoc Netz zur Verfügung steht, sollte der Gruppenvertreter umziehen. Eine clevere Wahl, welche Station zum Gruppenvertreter wird, sollte daher von Anfang an verwendet werden. In der derzeitigen Implementierung vom ELAN-Rahmenwerk erscheint dies allerdings nur bedingt möglich. Zwar sind Nachrichten vorgesehen, mit denen die Middleware über bestimmte Ereignisse informiert, z.B. der Akku der Station ist bald leer. Zusätzlich sind auch Mechanismen zur Migration einer Komponente vorgesehen, dies wurde aber nicht während der Implementierung getestet - siehe Kapitel 6.2. Das zentrale Problem ist, daß es keine Möglichkeit gibt, den Status einer Komponente zu verändern. Anwender bekommen derzeit nur die Existenz von Diensten innerhalb der *AwarenessView* angezeigt, somit sind Dienste immer der zentrale Zugangspunkt zu Gruppenanwendungen. Derzeit kann aus einem Dienst kein Accessor und umgekehrt gemacht werden. Wäre dies möglich, könnte mit einfachen Steuernachrichten die Gruppenkommunikationsschicht nötige Verwaltungsinformationen umziehen und anschließend gezielt den Status der Anwendung verändern.

Kapitel 5

Optimierung der Gruppenverwaltung und -kommunikation

In diesem Kapitel werden mögliche Optimierungen der Gruppenverwaltung betrachtet. Dabei werden zwei Verfahren betrachtet, die eine dezentrale Organisation der Gruppe ermöglichen. Diese Ansätze werden auf Anwendungsebene realisiert, im Gegensatz zu fest verdrahtete Netzen in denen Multicast meist auf Schicht 3 oder auch 2 realisiert wird. Dabei entsteht ein Multicastbaum vom Sender zu den Empfängern, der den einzelnen Stationen bekannt ist. An diesem Baum entlang werden Multicastachrichten an Gruppenmitglieder gesendet. In Ad-Hoc Netzen ist dieser Ansatz auf Grund der sich schnell verändernden Netztopologie problematisch. Diese Änderungen erfordern ständige Anpassungen der Multicastbäume und erscheinen daher als ungeeignet. Auf Grund dessen entstand die Idee, die Gruppenverwaltung innerhalb der Anwendungsschicht zu realisieren und bildet die Grundlage der in Kapitel 4 realisierten Gruppenkommunikationsschicht. Diese verwendet zur einfacheren, prototypischen Realisierung einen zentralen Gruppenverwalter. Dieser Ansatz hat eine entscheidendes Problem. Tritt ein Problem beim Gruppenverwalter auf, ist die Kommunikation der gesamten Gruppe gestört. Die essentielle Komponente der Gruppenkommunikation ist die Gruppenverwaltung. Um eine ausfallsichere Lösung zu finden, sollte die Verwaltung in einem Ad-Hoc Netz dezentralisiert gelöst werden. Auf diese Art kann eine hohe Zuverlässigkeit für die Gruppenkommunikation erreicht werden. Eine mögliche Lösung muß dabei flexibel auf die Veränderungen der Gruppe reagieren und dabei das Netz nicht mit Verwaltungsnachrichten überfluten.

5.1 Narada-Protokoll

Das Narada-Protokoll [CRZ00] wurde für den Einsatz in Ad-Hoc Netzen entworfen. Narada verwendet ein Overlay-Netz um Gruppenkommunikation zu realisieren, welches auf Anwendungsebene funktioniert. Ein Overlay-Netz ist genau das, was der Name vermuten läßt. Es liegt über einem bestehenden Netz und verbindet die einzelnen Knoten untereinander, ohne sich um die darunter liegende Netzstruktur zu kümmern. Dabei werden die

einzelnen Knoten des Overlay-Netzes als Punkt-zu-Punkt Verbindungen realisiert. Diese Verbindungen nutzen die zu Grunde liegende Transportschicht, welche für eine zuverlässige Datenübertragung sorgt. Dadurch werden Veränderungen innerhalb der Netztopologie "kompensiert und schlagen sich auf der Ebene des Overlays lediglich durch veränderte Verzögerungen nieder" [BKW02]. Dieses Konzept kommt der Struktur von Ad-Hoc Netzen entgegen. Ein Overlay-Netz spiegelt eine ausgewählte Gruppe von Netzknoten wieder, bildet also ein logische Gruppe. Diese Tatsache nutzt das Narada-Protokoll geschickt. Ist das Overlay erst bei allen Knoten bekannt, kennen sie die Gruppe und können daraus einen Multicastbaum errechnen. Das Protokoll verwendet dazu einen Algorithmus zum Berechnen von einem minimal spannenden Baum [CLR90] auf dem Overlay. Dabei ist der Sender die Wurzel und die Empfänger sind die Blätter. Ist der Multicastbaum errechnet, kann der Knoten mit diesem bestimmen, wie er Nachrichten senden muß, um alle Gruppenteilnehmer zu erreichen.

5.1.1 Gruppenverwaltung

Damit das Narada-Protokoll die Gruppe verwalten kann, verwendet es Verwaltungsnachrichten. Diese versenden die einzelnen Knoten in regelmäßigen Zeitintervallen an ihre Nachbarn, ein genauer Wert ist nicht festgelegt. Dabei tauschen sie Informationen über die ihnen bekannten Knoten in Form von Listen aus. Zusätzlich wird eine Sequenznummer zu den Einträgen der Liste hinzugefügt. Mit Hilfe der Sequenznummer kann jeder Knoten, der eine Liste empfängt, das Alter des Eintrags ermitteln. Durch dieses einfache Verfahren werden nacheinander sämtliche Gruppenmitglieder miteinander bekannt gemacht. Jeder Knoten kennt nach einer gewissen Zeit sämtliche Gruppenmitglieder und mit welchen anderen Gruppenmitgliedern diese verbunden sind - das Overlay. Ein Vorteil des regelmäßigen Nachrichtenaustausches ist, daß Veränderungen innerhalb der Netztopologie leicht erkannt werden. Informationen darüber, daß ein Knoten nicht mehr existiert oder eine Strecke gestört ist, verbreiten sich von Knoten zu Knoten weiter.

Einer Gruppe beitreten

Damit ein neues Mitglied einer Gruppe beitreten kann, muß es lediglich von einem Mitglied eine Mitgliederliste erhalten. Wie dies geschehen soll, ist im Narada-Protokoll nicht explizit beschrieben. Ein möglicher Ansatz könnte das Mithören und Auswerten von Nachrichten sein. Dies wäre in einem Ad-Hoc Netz mit geringem Aufwand realisierbar und wird in einem Entwurf im Absschnitt 5.3.1 beschrieben. Sobald ein neuer Teilnehmer einen Nachbar identifiziert hat, tritt er mit diesem in Kontakt. Um sich innerhalb der Gruppe bekanntzumachen, sendet er seine noch leere Liste an den Nachbarn. Dieser nimmt den neuen Knoten in seine Liste auf und schickt diese an seine Nachbarn - darunter ist auch der neue Knoten. Alle weiteren Mitglieder erfahren durch den regelmäßigen Informationsaustausch von der Veränderung und nehmen dadurch das neue Mitglied in ihre Listen auf. (vergleiche [CRZ00])

Verlassen einer Gruppe

Zum Verlassen der Gruppe meldet sich ein Gruppenmitglied bei seinen Nachbarn ab. Diese vermerken die Information in seiner Liste. Durch den regelmäßigen Informationsaustausch verbreitet sich die Information innerhalb der Gruppe. Alle anderen Gruppenmitglieder entfernen auf diese Art das ehemalige Mitglied aus ihrer Liste.
Analog wird verfahren, wenn ein Problem beim Versenden von Nachrichten auftritt. Wird dieses vom versendenden Knoten bemerkt, vermerkt er dies innerhalb seiner Liste. Die Information verbreitet sich innerhalb der Gruppe und die einzelnen Mitglieder können ihre Strategie anpassen. (vergleiche [CRZ00])

5.1.2 Abschlußdisskussion

Das Narada-Protokoll bietet einen interessanten Ansatz zur Verwaltung der Gruppenmitglieder. Allerdings ist dies im ELAN-Rahmenwerk nicht ohne komplexere Erweiterungen realisierbar. Das zentrale Problem liegt darin, daß ein Netzknoten innerhalb von ELAN nicht bestimmen kann, welche Knoten Nachbarn sind. Dazu würde eine Metrik (siehe 6.2) benötigt, wie bereits in [Fer03] angeregt.
Ein Nachteil ist, daß die einzeln gekapselten Aufgaben des Schichtenmodells aufgebrochen werden. Das Overlay-Netz realisiert auf Anwendungsebene ein über dem realen Netz liegendes virtuelles Netz. Dies hat zur Folge, daß Aufgaben tieferliegender Schichten für das virtuelle Netz erneut gelöst werden müssen. Diese Aufgaben erfordern zusätzliche Rechenleistung innerhalb der Endgeräte. Aber gerade dies kann bei sehr kompakten Endgeräten problematisch sein. Hohe Prozessorlast verringert die Akkulaufzeiten erheblich und ein totaler Ausfall der Station könnte zu Lasten des gesamten Ad-Hoc Netzes gehen - Partitionierungen können entstehen.
Die Anzahl der Teilnehmer ist problematisch, denn die "Netzbelastung durch Kontrollverkehr steigt quadratisch, zur Anzahl der Teilnehmer an" [BKW02]. Um dem entgegen zu wirken, kann das Zeitintervall für den regelmäßigen Nachrichtenaustausch angepasst werden. Eine Veränderung des Zeitintervalls hat dabei zur Folge, daß das Overlay-Netz träger auf Veränderungen reagiert. Die gute Anpassung des Netzes ist aber einer der zentralen Vorzüge.
Nicht jeder Knoten im Netz hat die gleichen Informationen zum selben Zeitpunkt. Dies kann dazu führen, daß aus 'Unwissenheit' Nachrichten verschickt werden, die nicht mehr zugestellt werden können. Der Empfänger ist evtl. über eine neue Route erreichbar oder aber aus der Gruppe ausgetreten. Es entstehen daher weitere Overhead-Nachrichten.
Damit Versandprobleme erkannt werden können, müssten Bestätigungsnachrichten realisiert werden. Erst dann kann ein Knoten erkennen, daß seine verschickte Nachricht nicht angekommen ist oder eine Strecke gestört ist. Quittungen bedeuten aber ein zusätzliches Nachrichtenaufkommen - zusätzlicher Overhead.
Auf Grund der beschriebenen Probleme erscheint es fraglich, ob Overlay-Netze die gewünschte Leistung erbringen können. Der Ansatz, Multicast auf Ebene der Transportschicht zu realisieren, würde vor allem die Anwendungen entlasten selber Routingarbeit leisten zu müssen. Das Narada-Protokoll nutzt außerdem das gemeinsame Medium von drahtlosen Ad-Hoc Netzen nicht aus. Diese Tatsache ist auch in [BKW02] als eine mögliche Optimierung angeregt worden.

5.2 Gnutella-Protokoll

Gnutella wurde ursprünglich zum Tauschen von Dateien im Internet entwickelt. Für das Protokoll existieren verschiedenste Implementierungen, die meist als Client bezeichnet werden. Dabei ist diese Bezeichnung nicht vollständig richtig. Jedes auf dem Gnutella-Protokoll basierende Programm verhält sich gleichzeitig als Server und Client, im Kontext von Gnutella wird deshalb von Servlets gesprochen. Gnutella verwendet eine dezentrale Struktur. Weil Filesharingprogramme oftmals zum Austausch urheberrechtlich geschützter Daten verwendet werden, scheint der dezentrale Ansatz für die Anwender eine risikofreiere Nutzung zu erlauben. Auch läßt sich das Netz nicht durch die Abschaltung zentraler Server leicht stören. Ob dies die Intention zur Entwicklung von Gnutella war, kann nicht überprüft werden. Als Möglichkeit ist es allerdings nicht auszuschließen. Die erste Implementierung des Gnutella-Protokolls wurde von der Firma Nullsoft vorgestell, die eine Tochtergesellschaft der AOL Time Warner ist. Im Interesse der Muttergesellschaft wurde das angebotene kostenlose Produkt aus dem Profil der Firma Nullsoft entfernt. Das Gnutella-Protokoll verwendet einen Ansatz zur Gruppenverwaltung, der für Ad-Hoc Netze adaptiert werden kann.

5.2.1 Gruppenverwaltung des Gnutella-Protokolls

Gnutella verwendet Fluten, um andere Teilnehmer zu finden. Fluten gilt als suboptimal, denn Nachrichten verbreiten sich mehrfach im Netz. Dieser Overhead kann allerdings deutlich reduziert werden. Einen Ansatz dazu wird in [NTCS99] erläutert. Das Gnutella-Protokoll hinterlegt in jedem Knoten Routinginformationen. Diese Routinginformationen spiegelt den für diesen Knoten erreichbaren Teil der Gruppe wieder. Die tatsächliche Anzahl aller Teilnehmer ist keinem einzelnen Knoten im Netz bekannt. Mit sogenannten Crawlern wurde versucht, die Gesamtheit eines durch das Gnutella-Protokoll erzeugten Netzes zu visualisieren.

Um andere Knoten im Netz zu finden, verwendet das Gnutella-Protokoll eine eigene `Ping`-Nachricht. Ein Knoten erzeugt eine eindeutige Nachrichten-Id, setzt einen TTL-Zähler[1] und sendet anschließend die `Ping`-Nachricht an alle ihm bekannten Nachbarmitglieder.
Trifft bei einem Knoten eine `Ping`-Nachricht ein, sind Nachrichten-Id und TTL-Zähler von Bedeutung. Ist der Wert des TTL-Zähler Null, ist die Nachricht zu alt und wird verworfen. Falls der TTL-Wert größer als Null ist, wird die Nachricht verarbeitet. Dazu speichert der Knoten die Nachrichten-Id in einer internen Liste ab. Außerdem vermerkt der Knoten, woher er diese Nachricht erhalten hat. Anschließend schickt er dem Absender der Nachricht eine `Pong`-Nachricht zurück, dekrementiert den TTL-Zähler und leitet die `Ping`-Nachricht an alle ihm bekannten Mitglieder weiter. Dabei wird der ihm bekannte Absender der `Ping`-Nachricht ausgenommen.
Wichtig ist, daß eine `Ping`-Nachricht bereits früher, über einen anderen Nachbarn bei einem Knoten eingetroffen sein kann. Daher wird vor dem Weiterschicken der `Ping`-Nachricht in der internen Liste geschaut, ob für die Nachrichten-Id bereits ein Eintrag existiert. Ist dies der Fall, ist die Nachricht zuvor bereits auf einem kürzeren Weg eingetroffen und muß nicht weiter verschickt werden. Die Nachricht wird verworfen. `Pong`-Nachrichten[2]

[1] Abkürzung für 'Time to live'
[2] Im Gnutella-Protokoll wird diese Nachricht auch als `Ping-Response` bezeichznet.

werden nur durch eine eintreffende `Ping`-Nachricht versendet. Die aus der `Ping`-Nachricht bekannte Nachrichten-Id wird ebenfalls für die `Pong`-Nachricht verwendet. Innerhalb der `Pong`-Nachricht verschickt der Absender seine IP-Adresse sowie Informationen über die freigegebenen Daten[3]. Trifft eine `Pong`-Nachricht bei einem Knoten ein, sieht dieser in seiner internen Liste nach, um mit Hilfe der Nachrichten-Id zu ermitteln, wohin er die `Pong`-Nachricht weiterleiten muß. Ist der Knoten der Absender der `Ping`-Nachricht, wertet er die enthaltenen Informationen aus. Ein neues Gruppenmitglied, das über andere Knoten erreichbar ist, wurde so dem Absender der `Ping`-Nachricht bekanntgemacht. Auf eine `Ping` Nachricht erhält der Absender mehrere `Pong`-Nachrichten als Antwort.
Weitere Nachrichten des Gnutella-Protokolls sind nicht von Interesse, da diese den Dateiaustausch zwischen zwei Mitgliedern der Gruppe als Punkt-zu-Punkt Verbindung initialisieren bzw. abwickeln. Uns interessiert am Gnutella-Protokoll vorwiegend die dezentrale Verwaltung von Gruppen. Zum Verlassen der Gruppe ist im beschriebenen Gnutella-Protokoll Version 0.4 keine Nachricht vorgesehen. Eine detailierte Beschreibung des Gnutella-Protokolls kann unter [rfc05] gefunden werden.

5.2.2 Ablaufbeispiel

Abbildung 5.1 illustriert beispielhaft den Ablauf, wenn ein Knoten die erreichbaren anderen Teilnehmer mit Hilfe von `Ping`- und `Pong`-Nachrichten ermitteln möchte. Die dünnen, geraden Linien stellen dabei eine nachbarschaftliche Verbindung dar. Die gebogenen, dickeren Linien symbolisieren eine Nachricht, die über die Verbindung geschickt wird. Die einzelnen Schritte visualisieren das Netz zu unterschiedlichen, nacheinander eintretenden Zeitpunkten.

Schritt 1: Der initiierende Rechner versendet eine `Ping`-Nachricht, um Informationen über die aktuelle Zusammensetzung der Gruppe zu erhalten.

Schritt 2: Der Initiatorrechner erhält die erste Information über die Gruppe durch die `Pong`-Nachricht seines direkten Nachbarn. Der Nachbar verschickt die `Ping`-Nachricht weiter an alle seine Nachbarn.

Schritt 3: Es ist zu sehen, daß die `Pong`-Nachrichten auf dem Rückweg durch das Netz sind und weitere `Ping`-Nachrichten verschickt werden. Der Rückweg kann durch die hinterlegten Routinginformationen in den einzelnen Knoten gefunden werden.

Schritt 4: Ein entscheidender Moment im Gnutella-Protokoll ist dargestellt. Es werden zwei `Pong`-Nachrichten nacheinander an den urspünglichen Initiatorrechner gesendet. Dargestellt durch das 2x vor der Nachricht. Diese werden nicht in einer gemeinsamen Nachricht verschickt, sondern einzeln. Durch diese werden dem Teilnehmer die Nachbarn seines Nachbarn bekannt gemacht. Außerdem werden zwei weitere `Ping`-Nachrichten an Knoten verschickt, die bereits früher `Ping`-Nachrichten erhalten hatten.

[3] Für Filesharing von Bedeutung

Schritt 5: Es werden keine weiteren `Pong`-Nachrichten durch die Empfängerknoten der `Ping`-Nachrichten zurückgesendet. Außerdem treffen erneut zwei `Pong`-Nachrichten beim ursprünglichen Absender ein. Dem Initiatirrechner ist somit die vollständige Gruppe bekannt.

5.2.3 Bootstrap-Problem

Dezentrale Netze zeichnen sich dadurch aus, daß meistens kein eindeutiger Ansprechpartner existiert. Daher muß ein neuer Teilnehmer, der beitreten möchte, mindestens einen Nachbarn kennen. Ein möglicher Lösungsansatz sind zentrale Server, die Listen mit bekannten Teilnehmern vorhalten und bei gezielter Anfrage dem interessierten Teilnehmer eine oder mehrere Zugangsadressen mitteilen. Eine zentralisierte Lösung ist allerdings nicht unbedingt vorteilhaft. Dadurch wird die dezentralisierte Netzstruktur aufgebrochen und ein Teil der Vorteile abgeschwächt. Z.B. kann das Netz leichter 'verwundet' werden, wenn der zentrale Zugangsserver ausfällt.
Alternativ kann der neue Teilnehmer `Ping`-Nachrichten an zufällig generierte Adressen schicken. Es ist dann nur eine Frage der Zeit, ob und wann er eine `Pong`-Nachricht erhält, also dem Netz beitritt. Dieser Ansatz ist allerdings unter Umständen für den Anwender der zeitaufwendigste. Mit geschicktem Raten kann die Zeit sicherlich verkürzt, aber nicht vollständig abgeschafft werden.
Ein Zwischenspeichern der in vorherigen Sitzungen verwendeten Adressen kann evtl. Abhilfe schaffen. Haben frühere Teilnehmer eine feste Netzadresse, ist die Wahrscheinlichkeit größer, an dieser Adresse wieder einen Gnutella-Teilnehmer zu finden. Von dieser These ausgehend könnte eine Cacheliste, mit klassifizierten Adressen, ebenfalls eine akzeptable Lösung darstellen. Aber auch in diesem Fall kann die Wartezeit nur verringert, aber nicht vollständig abgeschafft werden.
Allgemein bleibt festzustellen, daß der Eintritt in eine Gnutella-Gruppe eine Frage der Wartezeit ist, denn ohne mindestens einen Nachbarn kann ein neuer Teilnehmer nicht beitreten. Das Auffinden eines solchen Nachbarn benötigt Zeit. Zentralisierte Adressenserver, zufälliges Raten und Caching von guten zuvor bekannten Adressen können die Wartezeit verkürzen aber nicht vollständig verhindern. Auch im IRC[4] werden Adressen von Gnutella-Teilnehmer bekannt gemacht. Diese Information muß der Anwender dann allerdings zum Eintreten in eine Gruppe manuell eingeben.

5.2.4 Verbesserungsvorschläge

Das Gnutella-Protokoll bietet Möglichkeiten zur Optimierung. Die als stabil gekennzeichnete Version 0.4 bildet derzeit die gemeinsame Grundlage verschiedener Dateiaustauschprogramme. Dabei verwenden diese teilweise die Version 0.4 des Protokolls, aber auch diverse Verbesserungen wurden vorgeschlagen, diskutiert und in diversen Tauschprogrammen wie z.B. in Limewire [lim05] und Bearshare [bea05] realisiert.

[4] Internet Relay Chat [irc05]

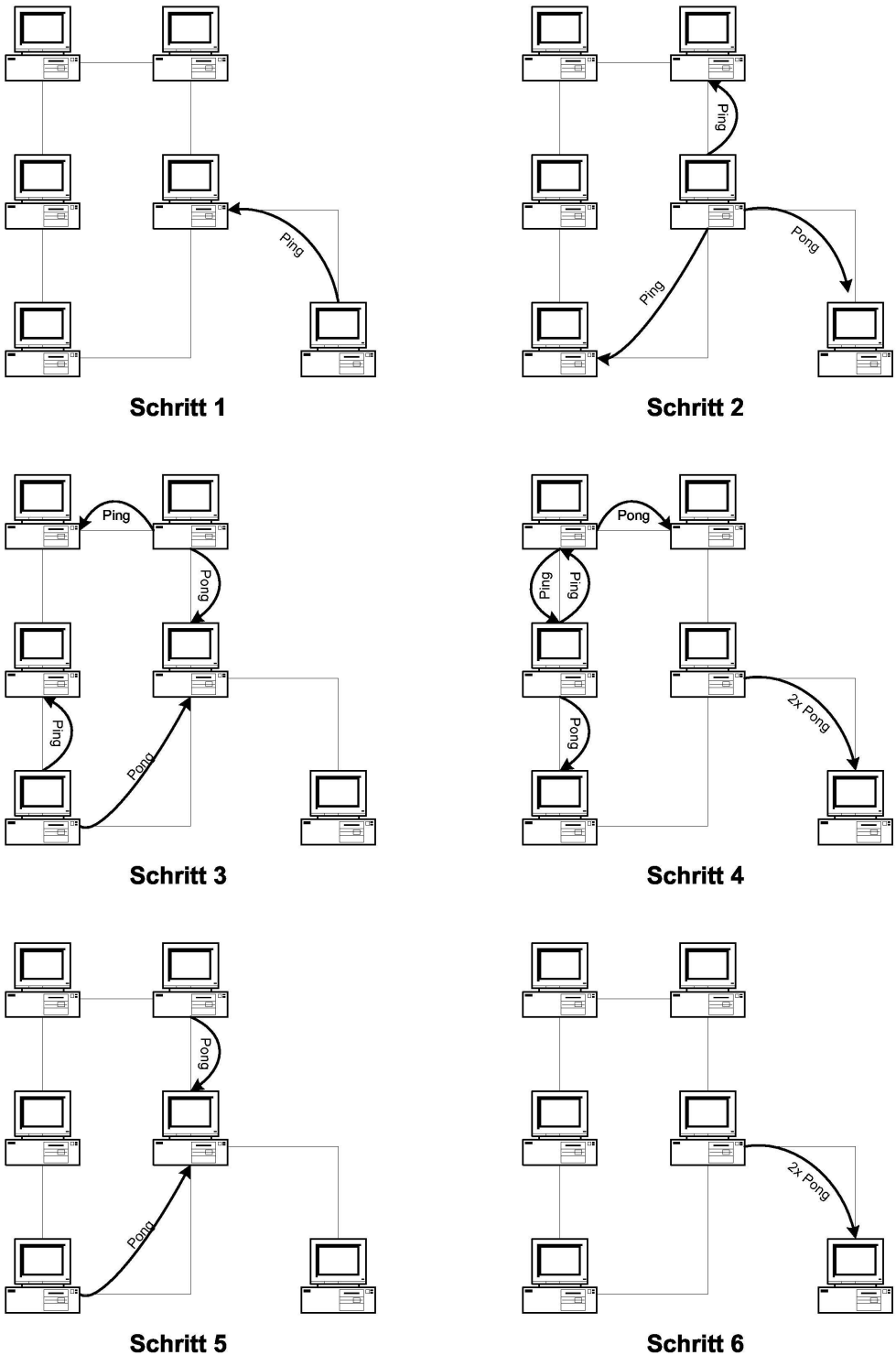

Abbildung 5.1: Ablauf der Ping- und Pong-Nachrichten beim Gnutella-Protokoll

Pong-Nachrichten Reduktion

Ein wesentlicher Teil des Nachrichtenaufkommens wird durch die zurück versendeten `Pong`-Nachrichten verursacht. Sammelt ein Knoten `Pong`-Nachrichten ein und verschickt sie in

einer einzigen Nachricht, kann das Nachrichtenaufkommen deutlich reduziert werden. Anhand der Nachrichten-Id ist das Sammeln problemlos möglich. Ein einfacher Zeitintervall wäre an dieser Stelle einsetzbar, verändert aber die Skalierbarkeit des Netzes. Durch einen clever gewählten Zeitintervall kann dieser Nachteil und das Nachrichtenaufkommen verringert werden.
Jeder Knoten erhält maximal soviele `Pong`-Nachrichten von seinen direkten Nachbarn, wie er `Ping`-Nachrichten verschickt hat. Unter Berücksichtigung des TTL-Zählers kann folgendes angenommen werden. Ist der TTL-Wert hoch, werden nahezu gleichviele `Pong`-Nachrichten von den direkten Nachbarn eintreffen, wie `Ping`-Nachrichten verschickt wurden. Ist der TTL-Wert niedrig, steigt die Wahrscheinlichkeit, daß ein Nachbarknoten im Netz schon bekannt ist. Daher sinkt die Anzahl der zu erwartenden `Pong`-Nachrichten von direkten Nachbarn. Mit Hilfe eines vom TTL-Wert abhängigen Schwellwerts wird jeweils bestimmt, auf wieviele `Pong`-Nachrichten der direkte Nachbar warten soll. Wird der Schwellwert überschritten, werden alle `Pong`-Nachrichten in einer Nachricht gesammelt weitergeleitet. Treffen nachträglich weitere `Pong`-Nachrichten der direkten Nachbarn ein, werden diese erneut gesammelt. Eine Anpassung des Schwellwerts sollte dann allerdings vorgenommen werden. Zusätzlich zum Schwellwert sollte ein Timeout-Zähler vorgesehen werden, damit bei nicht erreichtem Schwellwert die `Pong`-Nachrichten nicht auf Dauer im Knoten zum Versand gelagert werden. Geschachtelte `Pong`-Nachrichten können von einem Knoten genauso behandelt werden wie normale `Pong`-Nachrichten. Mit Hilfe eines Schwellwerts, der vom TTL-Wert abhängt, ist gegenüber der reinen Zeitsteuerung eine Verbesserung zu erwarten.

5.2.5 Limewire und Bearshare Verbesserungen

Die folgenden Verbesserungen des Gnutella-Protokolls wurden durch die Entwickler von Limewire und Bearshare angeregt und implementiert.

Ultrapeers

Die Idee der Ultrapeers ist eine Klassifizierung der Knoten innerhalb des Gnutella-Netzes. Dabei geschieht die Klassifizierung der einzelnen Knoten nach ihrer Bandbreite. Ein sehr guter Knoten, der viel Bandbreite besitzt, kann als Proxy für viele Clients verwendet werden. Dieser Proxy-Knoten wird dann als Zugangspunkt im Netz angesehen und entlastet somit eine schmalbandige Verbindung, wie z.B. eine 56k Modem Leitung. Dabei werden `Ping`- und `Pong`-Nachrichten nicht direkt vom schmalbandigen Knoten versendet bzw. empfangen, da dies nur am Ultrapeer geschieht. Ein weiterer Vorteil des Ultrapeers Ansatzes ist, daß das Gnutella-Netz dadurch besser skaliert, da durch den Proxy-Knoten als Zugangspunkt für schwächere Knoten Routinginformationen zwischen weniger Knoten, mit durchschnittlich schnellerer Anbindung, ausgetauscht werden. (vergleiche [Ber05])

Pong-Caching

Limewire und Bearshare verwenden Pong-Caching zur Reduktion des Nachrichtenaufkommens. Die Idee des eingesetzten Verfahrens besteht darin, eintreffende `Pong`-Nachrichten in einem Cache zu speichern. Trifft innerhalb einer Zeitspanne erneut eine `Ping`-Nachricht

ein, kann der Cache zurückgesendet werden.
Wenn jeder Knoten Pong-Caching verwendet, könnte das Nachrichtenaufkommen deutlich reduziert werden. Um den Cache innerhalb eines Knotens zu aktualisieren, wird an alle Nachbarn nach einem festen Zeitintervall eine erneute `Ping`-Nachricht versendet. Dies erscheint etwas paradox, allerdings wenn alle Knoten im Netz Pong-Caching verwenden, führt eine `Ping`-Nachricht nur dazu, daß der Pong-Cache des Nachbarknoten zurückgegeben wird. Somit tauschen benachbarte Knoten nur noch Cache-Informationen miteinander aus. Diese Informationen spiegeln das Wissen des Knoten über die Gruppe wieder. Die entstehende Kommunikation ist daher ähnlich dem Narada-Protokoll - siehe Abschnitt 5.1.1.

Abbildung 5.2 verdeutlicht den Ablauf (vergleiche [Roh05]): Im ersten Teil der Abbildung möchte ein neuer Teilnehmer B dem Netz beitreten, er sendet dazu eine `Ping`-Nachricht an A. Daraufhin schickt A eine `Pong`-Nachricht zurück und leitet die `Ping`-Nachricht von B weiter an alle seine Nachbarn.
Nach einer Weile treffen die `Pong`-Nachrichten für B bei A ein. A gibt diese Nachrichten alle weiter an B.
Im zweiten Teil der Abbildung hat B das Netz verlassen - warum ist nicht von Bedeutung. Ein neuer Teilnehmer C schickt eine `Ping`-Nachricht an A. A 'fälscht' nun die für C zu erwartenden `Pong`-Nachrichten mit Hilfe der Nachrichten-Id von C. A schickt an C die `Pong`-Nachrichten zurück, hat aber zuvor keine `Ping`-Nachrichten an seine Nachbarn verschickt. C empfängt die `Pong`-Nachrichten und kennt dadurch die Gruppe.
Der Nachrichtenaustausch mit der Nachbarschaft kann aber auch ein Problem hervorrufen. Der schlechteste Fall tritt dann ein, wenn alle Knoten im Netz sich gleichzeitig entschließen ihren Cache zu erneuern. In diesem Fall treffen alle `Ping`-Nachrichten auf einen nicht mehr gültigen Cache. In diesem Fall müssen die Knoten warten, bis alle `Pong`-Nachrichten zum Veranlasser geroutet wurden. `Ping`- und `Pong`-Nachrichten sollten auf Grund ihrer Wichtigkeit für das Netz gegenüber anderen Gnutella-Nachrichten bevorzugt werden. Eine Zeitspanne von 3 Sekunden für das Veralten der Cacheinformationen sollte nicht unterschritten werden. Kürzere Intervalle können zum beschriebenen schlechtesten Fall führen. (vergleiche [Roh05])

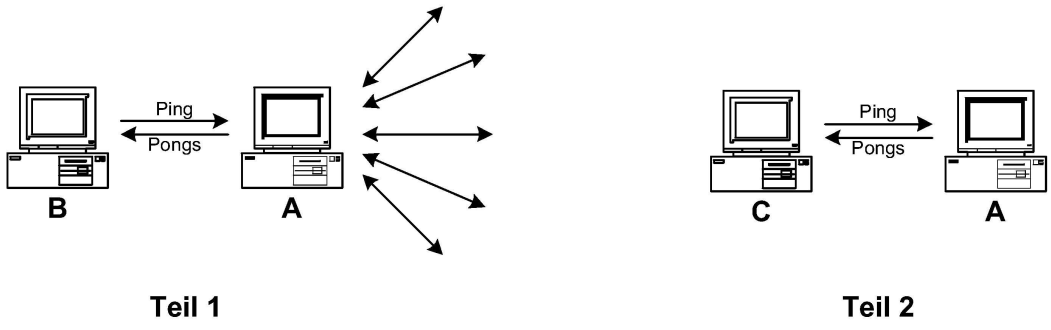

Abbildung 5.2: Ablauf beim Pong-Caching von Limewire und Bearshare

Ping-Multiplexing

Zusätzliche Schwierigkeiten treten ein, wenn zwei `Ping`-Nachrichten kurz hintereinander bei einem Knoten eintreffen. Abbildung 5.3 verdeutlicht das Problem. Es sind drei Knoten A, B und C in einer Reihe miteinander verbunden. A schickt eine `Ping`-Nachricht an B. B leitet die Nachricht weiter an C. Noch bevor B eine Antwort von C auf seine `Ping`-Nachricht erhält, verbindet sich ein neuer Knoten D mit B und schickt eine `Ping`-Nachricht. Zu diesem Zeitpunkt ist der Cache von B noch leer. Damit B auf die `Ping`-Nachricht reagieren kann, müßte der Knoten die Nachricht weiter an C leiten.

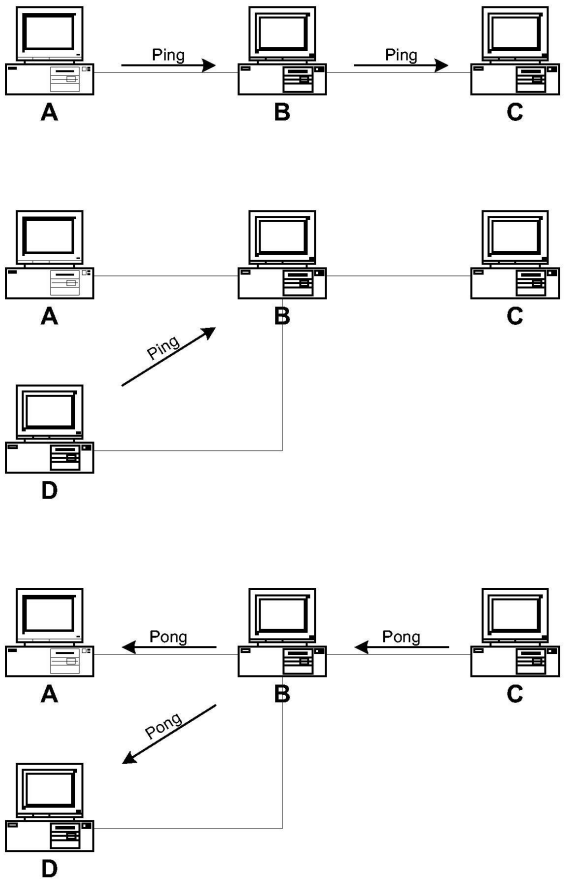

Abbildung 5.3: Ablauf beim Ping-Multiplexing von Limewire und Bearshare

Mit Hilfe von Ping-Multiplexing kann die zweite Ausbreitung der `Ping`-Nachrichten an die Nachbarn von B vermieden werden. Mehrere eingehende `Ping`-Nachrichten werden dazu in nur eine weitergeleitete `Ping`-Nachricht verschachtelt - gemultiplext. Entsprechend werden eintreffende `Pong`-Nachrichten demultiplexed. Aus einer eintreffenden `Pong`-Nachricht werden so mehrere gemacht. Dies ist auch die Lösung für die oben beschriebene Situation. B sendet keine erneute `Ping`-Nachricht an seine Nachbarn. Trifft die `Pong`-Nachricht auf die ursprüngliche `Ping`-Nachricht von A nach B ein, wird diese ebenfalls an D versendet. (vergleiche [Roh05].
Die Informationen in der `Pong`-Nachricht, die an D geschickt wird, sollten dabei um die Existenz von A erweitert werden. Analog weiss A nichts von D, denn dieser Knoten war

nicht vorhanden, als B seine Nachbarn suchte. Das Anpassen der `Pong`-Nachrichten kann allerdings vernachlässigt werden. Durch das regelmäßige Erneuern der Caches in A,B und C wird dieser Unterschied im Netz zügig behoben.

5.2.6 Abschlußbetrachtung

Das Gnutella-Protokoll erzeugt ein sehr hohes Aufkommen an Verwaltungsnachrichten in Form von `Ping` und `Pong`. Analysen der Version 0.4 des Gnutella-Protokolls haben gezeigt, daß über 50 Prozent der transportierten Nachrichten aus `Ping` und `Pong` bestehen [Mun05]. Dies ist vor allem für Knotenpunkte mit geringer Bandbreite problematisch. Die beschriebenen und implementierten Verbesserungen am Gnutella-Protokoll können vor allem das Nachrichtenaufkommen deutlich verringern. Dies ist von zentraler Bedeutung, denn es wäre nicht tragbar, daß jeder Netzteilnehmer in regelmäßigen Abständen das Netz mit `Ping`- und `Pong`-Nachrichten überflutet.
Auch die Klassifizierung der Knoten erscheint äußerst geschickt, denn durch diesen simplen Mechanismus wird ein einfaches Bandbreitenmanagement erreicht.
Ein scheinbarer Nachteil des Gnutella-Protokolls ist es, daß die einzelnen Gruppenmitglieder nur einen Teil der tatsächlichen Mitglieder kennen. Die Menge der bekannten Gruppenmitglieder ist dabei durch den TTL-Wert begrenzt. Ein TTL-Wert von 7 hat allerdings bereits Teilnehmerzahlen von über 10.000 Nutzern bei Crawler Visualisierungen ergeben [SGG02]. Sicherlich ist dies für den weltweiten Austausch von Dateien keine große Anzahl, aber auch für diese Problem existieren Verbesserungsvorschläge. Bei einer mögliche Adaption auf Ad-Hoc Netze kann der scheinbare Nachteil evtl. zum Vorteil werden, denn die räumliche Ausdehnung des Netzes ist begrenzt. Ein sinnvoller TTL-Wert sollte in einer exemplarischen Implementierung ermittelt werden können - auch eine Möglichkeit zum Einstellen wäre denkbar.

5.3 Gnutella Ad-Hoc

Die beiden vorgestellten Protokolle Narada und Gnutella stellen einen interessanten Lösungsansatz zur dezentralen Verwaltung von Gruppen zur Verfügung. Narada wurde zur Nutzung in Ad-Hoc Netzen entwickelt und eine Implementierung bzw. Tests sind derzeit in der Entwicklung [BKW02]. Gnutella hingegen wurde speziell für den Festnetzeinsatz entwickelt. Dabei stand die dezentrale Selbstverwaltung im Vordergrund. Durch die eingeflossenen Verbesserungen stellt es sich als interessantes Protokoll dar. Tests mit Ad-Hoc Netzen sind nicht bekannt.

5.3.1 Gemeinsames Medium

In Ad-hoc Netzen werden versendete Nachrichten von allen im Empfangsbereich befindlichen Stationen empfangen und die Stationen teil sich das Medium Luft gemeinsam. Gezieltes Senden von Nachrichten erfolgt derart, daß alle Nachrichten die empfangen werden nur verarbeitet werden, wenn sie für den Empfänger bestimmt sind. Andere eintreffende Nachrichten, die für einen anderen Empfänger bestimmt sind, werden verworfen. Datenübertragungen über die Funkschnittstelle haben also die Eigenschaft, innerhalb der

Sendereichweite ihre Nachrichten zu fluten. Abbildung 5.4 zeigt ein Overlay-Netz (durchgezogene Linien) zwischen den Knoten A und C. Dabei erfolgte die Kommunikation bisher immer über den Knoten B. A, B und C befinden sich allerdings in gegenseitiger Reichweite. Sendet A eine Nachricht an B, empfängt C diese ebenso. Da sie allerdings scheinbar für B bestimmt ist, wird sie von C verworfen. Erst wenn B die Nachricht weiterleitet, empfängt C diese. A hört diese Nachricht wieder mit und verwirft sie. Es entsteht ein erheblicher Overhead bei der illustrierten Situation. Dies muß beim Narada-Protokoll die Routingschicht erkennen und gegebenenfalls die Verbindung ändern. Könnte A direkt Nachrichten an C senden (gestrichelte Linie), wäre B weiterhin in der Lage, die Nachrichten mitzuhören. Abbildung 5.4 enhält zusätzlich die Stationen X, die ohne ihren Empfangsbereich darge-

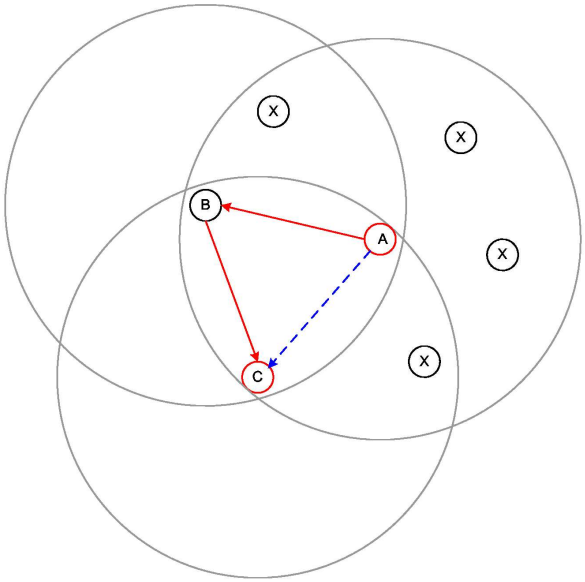

Abbildung 5.4: Bsp. für gemeinsames Medimu

stellt sind. Diese befinden sich alle innerhalb des Sendebereichs von A, so daß sie unnötig die von A an C verschickten Nachrichten empfangen. In WLAN-Netzen mit Infrastruktur, also einem dezidiertem Zugangspunkt, kann Mithören zum Ausspionieren von Informationen benutzt werden. Diese Tatsache erregt immer wieder die Aufmerksamkeit der Medien, vor allem weil als Verschlüsselungsverfahren eingesetzte Methoden mitlerweile als unsicher gelten [Sch05].

5.3.2 Entwurf

Das Gnutella-Protokoll verwendet kein Overlay-Netz und muß daher auch nicht die nötige Routingarbeit auf Anwendungsebene realisieren. Overlay-Netze weichen die typischen Schichtenmodelle auf. Bei Gnutella kommen Punkt-zu-Punkt Verbindungen erst beim Dateitransfer zustande, dies ist für uns allerdings nicht von Interesse. Kann das vorgestellte Gnutella-Protkoll für Ad-Hoc Netz verwendet werden?
Die Nachbarschaftsbeziehung zwischen einzelnen Knoten im Gnutella-Netz ist ähnlich zur Nachbarschaft, die durch das geteilte Medium entsteht. Zu diesem Zweck könnte innerhalb

der Transportschicht im ELAN-Rahmenwerk eine spezielle Nachrichten-Id eingeführt werden. Diese Nachrichten-Id informiert darüber, daß die empfangene Nachricht eine Gruppennachricht ist und wird innerhalb der Schichten weiter nach oben gereicht. Dabei dient die Nachrichten-Id als Auswahlkriterium - somit bleibt gezielte Kommunikation weiterhin möglich.
Die `Ping`- und `Pong`-Nachrichten könnten von einer Guppenverwaltungsschicht verarbeitet werden, wie in Kapitel 4 beschrieben und die Grundlage für Gruppenanwendungen bilden. Die Kapselung der Kommunikation in einer Gruppenkommunikationsschicht als Grundlage für Gruppenanwendungen sollte beibehalten werden. Dies erleichtert die Entwicklung der Anwendungen erheblich.
Die im Gnutella-Protokoll verwendete Klassifizierung von Knoten wäre zusätzlich gut innerhalb eines Ad-Hoc Netzes nutzbar, denn für die unterschiedlichen Leistungsklassen der Geräte und eintretenden Situationen im Ad-Hoc Netz wäre eine Klassifizierung hilfereich. So könnten PDAs mit geringerer Rechenleistung und Laptops mit nahezu aufgebrauchtem Akku schlechter eingestuft werden als PCs mit permanentem Stromanschluß. Außerdem könnten Verbindungsqualitäten für die Klassifizierung verwendet werden. Ein Gerät, daß sich an der Grenze des Empfangsbereichs befindet, könnte schlechter eingestuft werden, um gute Skalierung für die Gruppenkommunikation zu ermöglichen.
Als problematisch könnte sich Kommunikation erweisen, wenn Gruppenmitglieder nicht über den Nachrichtenversand im gemeinsamen Empfangsbereich erreicht werden können. Zwar erweitert sich der Bereich mit jeder durch unterschiedliche Knoten ausgesendeten `Ping`-Nachricht, aber eine Partitionierung in einzelne Gruppen kann nicht ausgeschlossen werden. Dieses Problem löst das Narada-Protokoll etwas geschickter. Durch die Punkt-zu-Punkt Kommunikation tritt eine Partitionierung möglicherweise später ein. Dieses Problem erscheint aber durch eine Erweiterung am Gnutella-Protokoll lösbar. Denn alle Knoten empfangen die Gruppennachrichten. Hört ein Knoten immer wieder `Ping`-Nachrichten, aber keine `Pong`-Nachrichten mit, kann er davon ausgehen, daß der Gruppenknoten keine Mitglieder in der Nähe des Mithörers finden konnte. Die verwendete Nachrichten-Id des Gnutella-Protokolls hilft ihm dabei. Er könnte dann die `Ping`-Nachricht ebenfalls aussenden, um den Broadcastbereich zu vergrößern. Findet sich auf diese Art und Weise ein anderer Gruppenteilnehmer, bleibt der Knoten als Vermittlungsknoten auf Transportebene erhalten. Ein Knoten kann aber auch in regelmäßigen Abständen `Ping`-Nachrichten mithören, und die `Pong`-Antworten verpassen, weil der Absender dieser Nachricht nicht im Bereich des Vermittlungsknoten liegt. Trotzdem leitet der Knoten die `Ping`-Nachricht weiter und bietet sich als Vermittlungsknoten an. Tritt die `Ping`-Nachricht nun bei einem Gruppenmitglied ein, kann dieses mit Hilfe der Nachrichten-Id entscheiden, ob eine `Pong`-Nachricht zurück geschickt wird. Ist dies der Fall wurde ein besserer Weg zu oder ein neuer Gruppenteilnehmer gefunden. Das Vermittlungskonten-Verhalten kann zufallsgesteuert sein, so daß nicht alle erreichbaren Knoten ständig `Ping`- bzw. `Pong`-Nachrichten reproduzieren. Abbildung 5.5 stellt dar, daß der Knoten A den Knoten B nicht erreichen kann. Sendet A nun eine `Ping`-Nachricht aus, kommt keine `Pong`-Nachricht zurück. C hört dies mit und entscheidet - wann bzw. wie soll nicht weiter festgelegt werden - die `Ping`-Nachricht zu replizieren. Diese Nachricht erreicht B und B antwortet mit einer `Pong`-Nachricht. C empfängt ebenfalls die `Pong`-Nachricht und wiederholt diese ebenfalls. Somit kann A die `Pong`-Nachricht erhalten und hat ein Gruppenmitglied gefunden. Der Knoten C verhält sich als Gruppenmitglied, obwohl bei ihm keine Gruppenanwendung ausgeführt wird. Er übernimmt die Rolle eines Vermittlerknotens. Das gewünschte Verhalten kann innerhalb der Transport- bzw. Routingschicht implementiert werden.

Eine entscheidende Frage ist, wie die Gruppenkommunikation realisiert werden soll. Das Gnutella-Protokoll bietet einen guten Ansatz, um die Verwaltung der Gruppe geschickt zu arrangieren. Mit den 'hinterlegten' Routinginformationen kann allerdings der Versand von Gruppennachrichten nicht realisiert werden. Ein Vorschlag könnte sein, daß jeder Knoten, der eine Nachricht an die Gruppe schicken möchte, diese an alle seine bekannten Nachbarn verschickt - wie `Ping`-Nachrichten. Auf eine Nachrichten-Id kann dabei verzichtet werden, es muß kein Rückweg für diese Nachricht hinterlegt werden. Nachrichten werden von Knoten an ihre Nachbarn weitergeleitet, wenn sie zur Gruppe gehören, also auch von den beschriebenen Vermittlungsknoten. Eine Verbreitungsbeschränkung im Netz durch einen TTL-Zähler sollte bei diesem Verfahren angedacht werden.

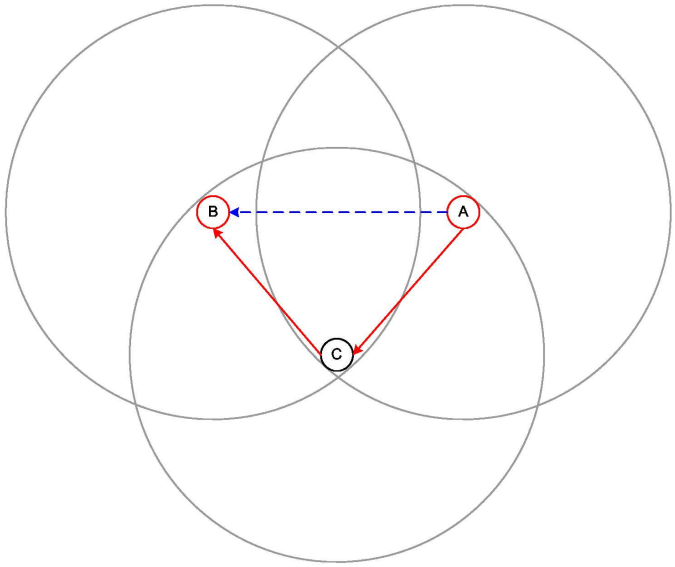

Abbildung 5.5: Bsp. für Gnutella Ping im Ad-Hoc Netz zwischen Knoten A und B. C vermittelt den Ping weiter an B.

5.3.3 Diskussion

Das Gnutella-Protokoll erscheint innerhalb des ELAN-Rahmenwerks als umsetzbar. Es bietet eine dezentralisierte Gruppenverwaltung, benötigt geringen Rechenaufwand innerhalb der einzelnen Knoten, und eine Aufgabenteilung in verschiedenen Schichten erscheint realisierbar. Der vorgestellte Entwurf bietet mit Sicherheit noch Möglichkeiten für Verbesserungen. Allerdings nutzt der Entwurf die Eigenart des gemeinsamen Mediums aus und verlagert nicht aufwendige Rechenarbeiten in die Anwendungsebene.
Punkt-zu-Punkt Verbindungen können weiterhin über die bestehende Transportschicht realisiert werden. Eine Gruppenanwendung, die beide Verbindungsarten nutzt, könnte ein erweiterter Dateifreigabedienst werden. Dienst und Accessor könnten derart verändert werden, daß die Bekanntmachung der Freigabeliste an die gesamte Gruppe über Gruppenkommunikation erfolgt. Datenübertragungen würden dann weiterhin als Punkt-zu-Punkt Verbindung zwischen Anbieter und Nutzer realisiert. Dabei kann das implementierte Übertragungsprotokoll aus Kapitel 3 weiter auf Anwendungsebene verwendet werden. Durch die

Gruppenkommunikationsschicht könnte der Dateifreigabedienst zusätzliche Möglichkeiten bekommen. Dateifreigabelisten wären so allen Gruppenmitgliedern gleichzeitig bekannt, Übertragungen würden unabhängig ablaufen und der Zugriff auf mehrere Dienste und somit einer Vielzahl von Dateien wäre möglich.

Kapitel 6

Zusammenfassung und Ausblick

Als Abschluß dieser Arbeit wird eine Zusammenfassung über die entworfenen und realisierten Konzepte gegeben. Weiter wird ein Ausblick formuliert, der mögliche Erweiterungen, Veränderungen und Anregungen des ELAN-Rahmenwerk motiviert.

6.1 Zusammenfassung

In der vorliegenden Arbeit wurden innerhalb des ELAN-Rahmenwerkes verschiedene Komponenten entworfen und realisiert, die Gruppenkommunikation verwenden. Dabei wurden verschiedene Ansätze der Gruppenkommunikation beleuchtet. Der ERC aus Kapitel 2 verwendet eine ungeordnete Gruppenkommunikation ohne gezielte Verwaltung der Gruppe. Dabei entstehen Nachrichten im Netz, die vermieden werden können. Diesen Overhead galt es in der nachfolgenden Anwendung durch ein gezieltes Kommunikationsprotokoll zu reduzieren. Zusätzlich löst der Dateifreigabedienst in Kapitel 3 Kommunikationsprobleme, die bei der Übertragung großer Dateimengen auftreten können. Dazu verwendet die Dateifreigabe ein Marshalling, das Dateien in Blöcke zerlegt und sicher über das Ad-hoc Netz transportiert. Damit eine sichere Übertragung möglich ist, werden Quittungsnachrichten und ein definiertes Fehlerverhalten benötigt. Daher wurde ein entsprechendes Übertragungsprotokoll entwickelt und implementiert. Als entscheidender Nachteil stellte sich heraus, daß die Dateifreigabe nur eine 1:N Kommunikation ermöglicht.
Damit eine N:M Kommunikation möglich ist, wurde die so genannte Gruppenkommunikationsschicht in Kapitel 4 entwickelt und implementiert. Diese bildet eine gemeinsame Grundlage für Gruppenkommunikationsdienste. Dabei ist die Frage der Verwaltung der Gruppe ein wichtiges Problem in Ad-hoc Netzen. Innerhalb der Gruppenkommunikationsschicht kommt eine zentralisierte Lösung der Verwaltung zum Einsatz. Mit Hilfe dieser kann gezeigt werden, daß Gruppenkommunikation in Ad-hoc Netzen realisierbar ist. Um zu zeigen, daß die Gruppenkommunikationsschicht eine zufriedenstellende und stabile Lösung anbietet, wurden zwei Gruppenanwendungen realisiert. Die Anwendung *groupChat* ist ein Chat-Programm. Die beim ERC erkannten Probleme treten beim *groupChat* nicht auf, denn durch Verwendung der Gruppenkommunikationsschicht entsteht kein überflüssiges Nachrichtenaufkommen.
Das *whiteboard* bietet einen gemeinsamen Zeichenbereich mit einer festen Anzahl von Werkzeugen an. Gruppenmitglieder der *whiteBoard*-Anwendung können gemeinsam auf

der Zeichenfläche die Werkzeuge kollaborativ anwenden. Mit Hilfe dieser Anwendung kann die Leistungsfähigkeit der Gruppenkommunikationsschicht und des Ad-hoc Netzes gut demonstriert werden.
Abschließend wurden in Kapitel 5 zwei alternative Methoden der Gruppenverwaltung beschrieben. Darunter das für Ad-hoc Netze entworfene Narada-Protokoll und das durch Dateiaustausch im Internet bekannt gewordene Gnutella-Protokoll. Beide Protokolle bieten alternative Verfahren zur dezentralisierten Gruppenverwaltung an. Durch eine denzentralisierte Verwaltung wird vor allem die Ausfallsicherheit erhöht, allerdings entstehen auch Nachteile. Ein erhöhtes Nachrichtenaufkommen und deutlich ansteigende Prozessorlast stehen einer verbesserten Stabilität gegenüber. Derzeit stattfindende Tests mit dem Narada-Protokoll werden die Leistungsfähigkeit dokumentieren. Für das Gnutella-Protokoll sind bisher keine Tests in Ad-hoc Netzen bekannt. Eine mögliche Adaption für Ad-hoc Netze und das ELAN-Rahmenwark wurde als Abschluß skizziert.

6.2 Ausblick

In diesem Abschnitt werden verschiedene Ideen kurz dargestellt, die für evtl. zukünftige Arbeiten als Anstoß dienen könnten. Die skizzierten Ideen sind im Laufe der Erstellung dieser Arbeit entstanden, spielten aber kein wesentliche Rolle bei der Entwicklung der Gruppenkommunikation in ELAN.

Gnutella-Protokoll für Ad-hoc Netze untersuchen Der vorgestellte Entwurf des Gnutella-Protokolls für Ad-hoc Netze erscheint interessant, dabei ist die Frage der Gruppenkommunikation allerdings unklar. Evtl. könnten die in den Knoten befindlichen Routinginformationen für eine gezielte Gruppenkommunikation Verwendung finden. Denn diese Informationen spiegeln den Teil eines Netzes wieder, der bei festgesetztem TTL-Wert für den Knoten erreichbar ist. Dabei müsste untersucht werden, inwiefern das Gnutella-Protokoll erweitert werden kann, um effektive Gruppenkommunikation zu ermöglichen. Eine Implementierung innerhalb von ELAN wäre denkbar. Anpassungen an der Routingschicht wären nötig, könnten aber zuerst in einem veränderten *Lanrouter* verwirklicht werden.

Ausfallsicherung von Endgeräten Die Optimierung der Kommunikation in Ad-hoc Netzen spielt eine wesentliche Rolle. Dies geschieht oftmals mit erheblichem Mehraufwand an Prozessorlast. Allerdings fordert Prozessorlast einen wesentlichen Teil der Energiereserve mobiler Endgeräte. Der gestiegene Energieverbrauch kann zu einem Stationsausfall führen und bedeuten, daß das Ad-hoc Netz sich neu konfigurieren muß. Dabei können schlechtestenfalls auch ungewollte Partitionen entstehen. Dieses Risiko könnte erheblich verringert werden, wenn Endgeräte klassifiziert würden. Mit Hilfe einer Geräteklassifizierung können dann verschiedene Algorithmen verwendete werden, um die Laufzeit einer Station zu erhöhen.
Eine Klassifizierung könnte in ELAN leicht realisiert werden. Die Middleware bietet bereits Nachrichten an, die über verschiedene Ereignisse informieren. Verschiedene Faktoren könnten auf den Status eines Endgerätes Einfluß nehmen - welche gilt zu untersuchen. Die resultierende Information daraus könnte die Middleware veranlassen, einfachere oder

schwerere Algorithmen zu verwenden. Zusätzlich sollte die Middleware die Anwendungsschicht informieren, damit diese evtl. nötige Verhaltensanpassungen ebenfalls vornehmen kann.

Geplante Ad-hoc Netze Um die spontane Bildung von Netzen zu unterstützen, könnten feste Stationen angeboten werden. Diese fixen Netzknoten können in einer Entfernung voneinander positioniert sein, so daß kein Fixknoten innerhalb der Reichweite eines Anderen ist. Zusätzlich wäre es möglich, auf Fixknoten exklusive, lokale Gruppendienste anzubieten. Z.B. könnte ein Lernraum einen Fixknoten anbieten, der bei ihm angemeldete Lerngruppen kennt. Dadurch könnten andere Nutzer zu der Gruppe hinzustoßen, oder erfahren, daß der Lernraum belegt ist.
Eine deutliche Verbesserung der Ad-hoc Netzproblematik könnte durch Fixknoten ebenso erreicht werden. Dazu wäre es nötig, daß eine geringe Anzahl von Endgeräten ausreicht, um zwischen Fixpunkten zu vermitteln. Ebenso könnten klassische Store-and-Forward Dienste mit Hilfe von Fixknoten angeboten werden.
Ein wesentlicher Vorteil von Ad-hoc Netzen würde dadurch allerdings abgeschwächt. Fixknoten bedeuten Investitionen in Infrastruktur, was gerade bei Ad-hoc Netzen nicht nötig ist. Dabei kann diese Investition gering gehalten werden, wenn Fixknoten an Stellen installiert werden, wo bereits lokale Endgeräte vorhanden sind und angebotene Dienste sinnvoll erscheinen. Eine Kombination aus Fixknoten und Store-and-Forward Dienst könnte z.B. in einer Bibliothek realisiert werden. Der Ausleihcomputer bietet zusätzlich über eine WLAN Karte einen Fixknoten an und betreibt dabei einen Dienst, bei dem Bücher bestellt werden können. Trifft ein vorbestelltes Buch wieder ein und der Besteller geht mit seinem Endgerät im Treppenhaus an der Bibliothek vorbei, erreicht ihn diese Information. Evtl. erhält er auch die Information im weiter entfernten Hörsaal, allerdings ist dies vom aktuellen Ad-hoc Netz abhängig.
Da diese Arbeit sich vorwiegend mit den Aufgaben der Anwendungsschicht beschäftigt hat, kann keine Aussage getroffen werden ob, ELAN um Fixknoten problemlos erweiterbar ist. Eine zusätzliche Klassifizierung für lokal gebundene Dienste wäre mit geringem Aufwand erreichbar.

Änderungen, Anpassungen und Erweiterungen im Elan-Rahmenwerk ELAN ist durch verschiedenen Entwickler entstanden. Die Quellcodebasis ist sehr übersichtlich und gut dokumentiert. Wenn ELAN über das Prototypstadium hinaus wachsen soll, muß die Oberfläche vereinfacht werden. Für Anwender erscheint es verwirrend, ob es sich bei laufenden Programmen um feste Komponenten, einen gestarteten Dienst oder einen Accessor handelt. Eine einfache Trennung der verschiedenen Programmarten in der Reiterleiste würde dabei bereits Übersicht schaffen.
Die Debugging-Komponenten sind für Endanwender verwirrend und könnten für diese Anwendergruppe entfernt werden. Für Entwickler sollten diese hilfreichen Tools selbstverständlich weiterhin optional zur Verfügung stehen. Eine Trennung von Anwender- und Entwicklerversion wäre zu erwägen.
Eine Anpassung an Python 2.4 ist ebenfalls nötig. Im Verlauf dieser Arbeit erschien Python in der Version 2.4 und Tests ergaben, daß ELAN auf dieser Version nicht ausgeführt werden kann. Dies liegt wahrscheinlich an einem Modul, das eine priorisierte Warteschlange bereitstellt. In ELAN wurden Attribute dieses Moduls verwendet, die nicht als privat markiert waren. Das Modul wurde allerdings in Python 2.4 erneuert, so daß die Attribute

nicht mehr vorhanden sind. Versuche, das Problem zu beheben, wurden zu Gunsten anderer Arbeiten eingestellt. Derzeit wird weiterhin Python 2.3 für ELAN verwendet.
Während der Entwicklung dieser Arbeit konnte nicht abschließend getestet werden, ob die Migration von Komponenten zuverlässig funktioniert. Migration bildet eine wichtige Grundlage für Dienste und kann bei optimaler Anwendung die Erreichbarkeit eines Dienstes erhöhen. Daher sollte die Migration von Komponenten innerhalb von ELAN überprüft und gegebenenfalls implementiert und untersucht werden.
Eine Metrik könnte als ein Zahlenwert in ELAN die Verbindungsgüte zwischen Knoten widerspiegeln. Dabei könnten verschiedene Faktoren in die Metrik einfließen. Entfernung der Stationen, vor Kurzem aufgetretene Fehler bei Übertragungen oder Art des Endgerätes sind ein paar dieser Möglichkeiten. Eine ausführliche Betrachtung über Einflüsse auf das ELAN-Rahmenwerk, die durch eine Metrik erreicht werden können, könnte durchgeführt werden.

Anhang A

Anpassungen im Rahmenwerk

Im den folgenden Abschnitten werden die Erweiterungen des ELAN-Rahmenwerks vorgestellt, die notwendig sind, um die in den Kapiteln 2,3 und 4 vorgestellten Komponenten, Dienste und Accessoren zu implementieren.

A.1 '<all>'-Adressierung

Das Rahmenwerk ließ in der Methode `post()` nur einen Absender und einen Empfänger zu. Daher wurde eine zusätzliche Adressierungsmöglichkeit implementiert, um eine Nachricht an mehrere Empfänger gleichzeitig senden zu können. Awareness-Profile werden durch die Middleware im Netz verteilt und mit Profilen anderer Nutzer verglichen. Alle eingetroffenen Profile werden innerhalb der Middleware im Directory `wilidipManager` gespeichert. Durch die bereitgestellte Methode `getStatus()` kann erfragt werden, ob ein Profil als interessant eingestuft wurde. Der folgende Python-Quellcode verdeutlicht die durchgeführte Änderung in der Datei `xmlrpccomunicator.py` innerhalb der Middleware:

```
def EXTOUT(self, args, sender):
  # TargetIP = <all> - sendet an alle 'interessierten' die Nachricht
  sourceService, targetService, targetIP, targetcmd, targetparms = args
  if targetIP == "<all>":
    for targetIP in self.wilidipManager.wilidips:
      if self.wilidipManager.wilidips[targetIP].getStatus() == "interesting":
        self.EXTOUT_send (sourceService, targetService, targetIP, targetcmd, targetparms)
  else:
    self.EXTOUT_send (sourceService, targetService, targetIP, targetcmd, targetparms)
```

A.2 Nachrichtenversand beschleunigt

Das Rahmenwerk benötigte zu Beginn dieser Arbeit für den einfachen Versand von einer Nachricht ca. 2 Sekunden. Diese Zeitspanne erschien auf Grund der verwendeten Kommunikation durch den *Lanrouter* als sehr lang. Der *Lanrouter* simuliert ein Ad-hoc Netz auf Basis eines fest verdrahteten Netzwerks. Nach dem Durchsuchen der Kommunikationsabläufe konnte festgestellt werden, daß innerhalb der Middleware an verschiedenen Stellen im Quellcode `time.sleep(n)` Methoden gerufen wurden. Dabei steht n für die Anzahl an

Sekunden, die gewartet wird. Auffällig waren diese Befehle innerhalb der Routenberechnung, sobald eine Nachricht verschickt wird. Durch Entfernen des `time.sleep(5)` Aufrufs konnte die Kommunikation erheblich beschleunigt werden. Der nachfolgende Python-Quellcode zeigt den auskommentierten Befehl zum Warten in der Datei `middleware.py`:

```
def lego_routeRequest(self, args, sender):
  ip = args[0]
  self.debugOut( "> MW: Requesting route to '%s'" % ip)
  self.com('routeRequest', ip)
  # *** FIXME? ***
  #time.sleep(0.3)
```

A.3 Dienste mit GUI

Das Rahmenwerk besaß keine Möglichkeit, daß Dienste eine eigene GUI starten konnten. Dazu wurde der mehrstufige Startvorgang innerhalb von ELAN angepasst. Dadurch werden Dienste wie Komponenten eingestuft und beim Start des Rahmenwerks erkannt und bereits registriert. Ob ein Dienst mit GUI gestartet wird, ist nun von der Variablen `preload` innerhalb der Konfigurationsdatei abhängig. Dienste sollten immer einen `preload` von 0 haben, sonst werden diese von ELAN wie eine normale Komponente sofort gestartet. Dabei werden diese aber nicht beim Profil bekannt gemacht.

Anhang B

Ausgewählter Quellcode

B.1 EGroupComponent

Die Basisklasse der Gruppenanwendung mit der realisierten Kommunikation.

```
#-----------------------------------------------------------------------#
import time
#-----------------------------------------------------------------------#
from groupWidget import *
from elan.common import netutils
from elan.framework import framework, qt
from elan.framework.component import EComponent
from elan.framework.classes.Ids import IdMaker
#-----------------------------------------------------------------------#

class EGroupComponent( EComponent ):
    '''Groupcomponent base class'''
    name = "UntitledComponent"
    version = "1.0"
    spec = "0.2"
    author = "Kai-Marcel Teuber <teuber@tm.informatik.uni-frankfurt.de>"
    purpose = __doc__
    gui = None

    def __init__( self ):
        u'''baseclass 4 groupcommunication'''
        EComponent.__init__(self)
        self.application = str(self.classname)
        self.myIp = netutils.getIP()
        self.members = {}
        self.newId = IdMaker()
        self.memberId = 0
        self.config = framework().config()
        self.myNickname = self.config.get( 'component.config', 'nickname',\
         netutils.getUserAtHost() )

# -------------------------------------------------------------------------
# overload the following methods

    def groupBootGUI( self, parent=0 ):
        u''' overload the method to boot a gui in your application'''
        return None

    def groupExtIn( self, msg ):
        u'''to receive a msg from the group overload this one'''
        pass
```

```
    def groupMembers( self, listOfMembers ):
        u'''overload this to get a list of all known members'''
        pass

# ---------------------------------------------------------------------------
# internel methods do not overload them!

    def bootGUI( self, parent ):
        u'''use bootGUI from eComponente'''
        self.widgetStack = QWidgetStack( parent )
        self.groupWidget = groupWidget( parent )
        self.appWidget = self.groupBootGUI( self.widgetStack )
        self.awId = self.widgetStack.addWidget( self.appWidget )
        self.gwId = self.widgetStack.addWidget( self.groupWidget )
        self.widgetStack.raiseWidget( self.gwId )

        self.__groupMsg( 'Groupapplication found...' )
        if self.application[-8:] == 'Accessor':
            self.__groupMsg( 'started as groupaccessor ... waiting' )
        else:
            self.__groupMsg( 'started as groupservice ... waiting' )
            self.type = 'service'
            self.serviceName = self.classname
            self.serviceIp = '127.0.0.1'
            self.__appStart()
        return self.widgetStack

    def __groupMsg( self, msg ):
        self.widgetStack.raiseWidget( self.gwId )
        msg += '\n' #zeilenumbruch anhängen
        msg = str( self.groupWidget.te.text() ) + msg
        self.groupWidget.te.setText( msg )

    def __appStart( self ):
        self.widgetStack.raiseWidget( self.awId )
        self.post('EXTOUT', self.application, self.serviceName, self.serviceIp,\
         'joinGroup', (self.application, self.myIp, self.myNickname) )

    def shutdown( self, reason ):
        U'''eComponent method - we do the groupcomponent shutdown here'''
        if self.type == 'accessor':
            self.post('EXTOUT', self.application, self.serviceName, self.serviceIp,\
             'leaveGroup', ( self.memberId ) )
        else:
            print 'Oh oh, damit wird die ganze Gruppe geschlossen!!'

    def lego_StartServiceUsage( self, params, sender ):
        serviceIp, serviceName = params
        if not serviceName == self.application:
            return
        serviceName= serviceName[:-8] # ACCESSOR abschneiden!
        self.type = 'accessor'
        self.serviceName = serviceName
        self.serviceIp = serviceIp
        self.__appStart()

#Dispatcher
#-------------------------------------------------------------------------#
    def lego_EXTIN ( self, parms, sender ):
        envelope, cmdtuple =parms[0], parms[1]
        sourceService, sourceIP = envelope[0]
        targetService, targetIP = envelope[1]
        cmd, args = cmdtuple
        if not targetService == self.application:
            return
        dispatched = getattr( self, cmd, None)
        if dispatched:
```

```
            dispatched( args )
        else:
            print "%s: Nachricht '%s' kann nicht dispatched werden!" % \
             (self.application, cmd)

# Gruppenkommunikation
#---------------------------------------------------------------------------#

    def groupPost( self, msg ):
        u'''send msg to all members'''
        if (self.type == 'accessor'):
            self.post('EXTOUT', self.application, self.serviceName, self.serviceIp,\
             'groupPost', msg)
        elif (self.type == 'service'):
            for k in self.members:
                appName, appIp, nickName = self.members[k]
                self.post('EXTOUT', self.serviceName, appName, appIp, 'groupExtIn',\
                 msg )

    def __listMembers( self ):
        memberList = [self.members[i][2] for i in self.members]
        for k in self.members:
            appName, appIp, nickName = self.members[k]
            self.post('EXTOUT', self.serviceName, appName, appIp, 'groupMembers',\
             memberList  )

# Nachrichten für den Gruppernvertreter (Service)
#---------------------------------------------------------------------------#

    def joinGroup( self, parm ):
        u'''service msg someone want's to join the group'''
        appName, appIp, nickName = parm
        self.memberId = self.newId.next()
        self.members[self.memberId] = (appName, appIp, nickName)
        self.post('EXTOUT', self.serviceName, appName, appIp, 'joinAck', self.memberId)
        self.__listMembers()

    def leaveGroup( self, parm ):
        u'''service msg someone want's to leave the group'''
        id = parm
        nickName = self.members[id][2]
        del self.members[id]
        self.newId.freeId(id)
        self.__listMembers()

# Nachrichten für Gruppenmitglieder
#---------------------------------------------------------------------------#

    def joinAck( self, id):
        self.memberId = id

    def closeGroup( self ):
        u'''group became closed - sorry'''
        self.__groupMsg( 'Die Gruppe wurde geschlossen, beende Anwendung' )
```

B.2 WhiteBoardView

Das Hauptfenster von Shared-Whiteboard in dem alle Werkzeuge dargestellt werden.

```
class WhiteCanvasView( QCanvasView ):
    '''a canvas to show the whiteboard'''
    def __init__(self,c=None ,parent=None, name=None ,f = 0):
        QCanvasView.__init__( self, c, parent, name, f )
```

```
        self.parent = parent
        self.canvas = c
        self.type = 'Rechteck'
        self.zPos = 1
        self.color = QColor(255,32,128)
        self.brush = QBrush( self.color )
        self.brush.setStyle( QBrush.NoBrush )

    def setColorPixmap( self, Color ):
        self.emit( PYSIGNAL( "colorChanged( QColor )" ), ( Color, ) )

    def contentsMousePressEvent( self,e ):
            U'''mouse pressed - do something'''
        self.startPos = ( int(e.x()), int(e.y()) )
        self.stopPos = ( int(e.x()), int(e.y()) )
        if self.type == 'Rechteck':
            self.canvasItem = QCanvasRectangle( self.startPos[0], self.startPos[1],\
             0, 0, self.canvas )
            self.canvasItem.setBrush( self.brush )
            self.canvasItem.setPen( QPen(QColor(0,0,0) ))
        elif self.type == 'Kreis':
            self.canvasItem = QCanvasEllipse( 0,0, self.canvas )
            self.canvasItem.move( self.startPos[0], self.startPos[1] )
            self.canvasItem.setBrush( self.brush )
            self.canvasItem.setPen( QPen(QColor(0,0,0) ))
        elif self.type == 'Text':
            U'''show text-inputdialog'''
            dialog = WhiteTextInput( None, 'Bitte Text eingebe', True)
            if dialog.exec_loop() == QDialog.Accepted :
                self.canvasItem = QCanvasText( dialog.text, self.canvas)
                self.canvasItem.move( self.startPos[0], self.startPos[1] )
                self.canvasItem.setColor( self.color )
        self.canvasItem.setZ( self.zPos )
        self.canvasItem.show()
        self.canvas.update()
        if self.type == 'Text':
            self.contentsMouseReleaseEvent( e )

    def contentsMouseMoveEvent( self,e ):
        U'''mous still pressed'''
        self.stopPos = ( int(e.x()), int(e.y()) )
        width = self.stopPos[0] - self.startPos[0]
        height = self.stopPos[1] - self.startPos[1]
        self.canvasItem.setSize( abs(width), abs(height) )
        if self.type == 'Rechteck':
            self.canvasItem.setSize( width, height )
        elif self.type == 'Kreis':
            self.canvasItem.move( self.startPos[0]+width/2, self.startPos[1]+height/2)
        self.canvasItem.show()
        self.canvas.update()

    def contentsMouseReleaseEvent( self, e ):
        U'''mouse released'''
        self.emit( PYSIGNAL( "paintEvent( canvasItem)" ), ( self.canvasItem, ) )
        self.canvasItem.setCanvas( None )
        del self.canvasItem
        self.canvas.update()

# slots connected with the GUI
#---------------------------------------------------------------------------#
    def slotNewZPosition( self, newZ ):
        self.zPos = newZ + 1

    def slotTextSelected( self, bool=0 ):
        if bool:
            self.type = 'Text'

    def slotCircleSelected( self, bool=0 ):
```

```
        if bool:
            self.type = 'Kreis'

    def slotBoxSelected( self, bool=0 ):
        if bool:
            self.type = 'Rechteck'

    def slotColorDialog( self ):
        currentColor = self.brush.color()
        newColor = QColorDialog.getColor(currentColor, self, "ColorDialog")
        self.brush.setColor( newColor )
        self.color = newColor
        self.setColorPixmap( newColor )

    def slotToggleFill( self, bool=0):
        self.brush.setStyle( [QBrush.NoBrush, QBrush.SolidPattern] [bool] )
        self.setColorPixmap( self.brush.color() )
```

Literaturverzeichnis

[bea05] *Bearshare the power to share.* `http://www.bearshare.com/`. Version: Mai 2005

[Ber05] Berkes, Jem E.: *Decentralized Peer to Peer Network Architecture Gnutella and Freenet.* `http://www.sysdesign.ca/jem_berkes/academic.html`. Version: Mai 2005

[BKW02] Blödt, Steffen ; Kuhm, Eckehardt ; Wehrle, Klaus: Peer-to-Peer-basierte Gruppenkommunikation in mobilen Ad-hoc-Netzwerken. 2002. – Forschungsbericht. DFG-Schwerpunktprogramm SPP1140 "Basissoftware für selbstorganisierende Infrastrukturen für vernetzte mobile Systeme"

[BS04] Blanchette, Jasmin ; Summerfield, Mark: *C++ GUI-Programmierung mit Qt3.* Addison Wesley, 2004

[Che01] Cheng, Edward: *ON-Demand Multicast Routing in Mobile Ad Hoc Networks.* `http://citeseer.ist.psu.edu/article/on-01demand.html`. Version: 2001

[CLR90] *Kapitel* 24. In: Cormen, Thomas H. ; Leiserson, Charles E. ; Riverst, Ronald L.: *Introduction to Algorithms.* The MIT Press, 1990, S. 498–513

[CRZ00] Chu, Yang-Hua ; Rao, Sanjay G. ; Zhang, Hui: A case for end system multicast. (2000), 1-12. `http://citeseer.ist.psu.edu/chu01case.html`

[Fer03] Fertig, Christian: *Entwicklung von Komponenten für eine Verteilungsplattform zur Unterstützung kollaborativer Anwendungen.* Frankfurt am Main, Johann Wolfgang von Goethe-Universität, Diplomarbeit, November 2003

[irc05] *Internet Relay Chat - Wikipedia.* `http://de.wikipedia.org/wiki/Internet_Relay_Chat`. Version: Mai 2005

[Lau02] Lauer, Michael: *Python und GUI-Toolkits.* Mitp, 2002

[lim05] *Limewire Developer Resources.* `http://www.limewire.com/developer/`. Version: Mai 2005

[Man05] Mansmann, Urs: Boom Phase - Voice over Ip auf dem Siegeszug. In: *CT Magazin für computer technik* 12 (2005), S. 96–101

[Mar03] Martelli, A.: *Python in a Nutshell.* O'Reilly, 2003. – 654 S

[Mun05] Munz, Florian: *Analyse des Gnutella-Protokolls und dessen Verbesserungen*. Version: März 2005. `http://agva.informatik.uni-kl.de/pgc/ausarbeitungen/munz.pdf`. – Ausarbeitung

[NTCS99] Ni, Sze-Yao ; Tseng, Yu-Cheee ; Chen, Yuh-Shyan ; Sheu, Jang-Ping: The Broadcast Storm Problem in a Mobile Ad Hoc Network. In: *ACM Press* Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking (1999), S. 151–162

[onl05] online heise: *Qt künftig auch für Windows unter GPL*. `http://www.heise.de/newsticker/meldung/56129`. Version: Febuar 2005

[Per00] Perkins, Charles E.: *Ad Hoc Networking*. Addison Wesley, 2000

[PyQ05] *PyQt: A set of Python bindings for the Qt toolkit*. `http://www.riverbankcomputing.co.uk/pyqt/`. Version: Mai 2005

[Pyt05] *Python Programming Language*. `http://www.python.org/`. Version: Mai 2005

[Qt05] *TROLLTECH Documentation: Qt C++ GUI Application Development Toolkit and Qtopia - Embedded Linux Graphical Application Environment*. `http://doc.trolltech.com/`. Version: Mai 2005

[qt305] *Trolltech - Open Source Downloads*. `http://www.trolltech.com/download/opensource.html`. Version: Mai 2005

[rfc05] *Gnutella Stable 0.4*. `http://rfc-gnutella.sourceforge.net/developer/stable/`. Version: Mai 2005

[Roh05] Rohrs, Christopher: *Ping Pong Scheme*. `http://www.limewire.com/developer/pingpong.html`. Version: Mai 2005

[Sch05] Schmidt, Michael: Der WEP-Wall bricht. In: *CT Magazin für computer technik* 10 (2005), S. 182

[SGG02] Saroiu, S. ; Gummadi, P. ; Gribble, S.: *A Measurement Study of Peer-to-Peer File Sharing Systems*. `citeseer.ist.psu.edu/saroiu02measurement.html`. Version: 2002

[sip05] *Riverbank : SIP : Overview*. `http://www.riverbankcomputing.co.uk/sip/index.php`. Version: Mai 2005

[Wei03] Weigend, Michael: *Python ge-packt*. 1. Mitp, 2003 (ge-packt). – 526 S

[Wor03] Wordaz, Thorsten: *Entwicklung von editierbaren, korrelierbaren Interessensprofilen und deren Integration in ein Rahmenwerk für Kollaborative Anwendungen*. Frankfurt am Main, Johann Wolfgang von Goethe-Universität, Diplomarbeit, November 2003