Universität des Saarlandes

Sprachsteuerung im Aufzug

Projektseminar Computerlinguistik
Fortgeschrittenenpraktikum Informatik
Hauptseminar Phonetik

Abschlußbericht

Eric Auer, Claudia Crispi, Christian Dressler, Cordula Klein,
Kerstin Klöckner, Norbert Pfleger, Diana Raileanu
{eric,crispi,dressler,cklein,kekl,pfleger,raileanu}@coli.uni-sb.de

11. März 2002

Inhaltsverzeichnis

Vorwort

Das Projektseminar "Sprachsteuerung im Aufzug" wurde im Sommersemester 2000 an der Universität des Saarlandes in Saarbrücken durchgeführt. Dabei entwickelte sich eine fruchtbare interdisziplinäre Zusammenarbeit zwischen den Fachrichtungen Computerlinguistik, Phonetik und Informatik.

Die Veranstaltung wurde von Prof. Manfred Pinkal (Computerlinguistik), Prof. Bill Barry (Phonetik) und Dr. Joachim Niehren (Informatik) geleitet, und von vielen weiteren Mitarbeitern und Diplomanden der beteiligten Fachrichtungen dauerhaft mitgetragen. Dazu gehören Bistra Andreeva, Dipl. Ing. Attilio Erriquez, Malte Gabsdil, Dr. Jacques Koreman, und Dr. CJ Rupp.

Die Realisisung des Projektes gelang letztendlich aber nur mit Hilfe von 8 engagierten Studenten: Eric Auer, Claudia Crispi, Christian Dressler, Cordula Klein, Kerstin Klöckner, Norbert Pfleger, Diana Raileanu, und Linards Ticmanis. Der vorliegende Projektbericht wurde von 7 der beteiligten Studenten geschrieben. Jeder Student verfaßte ein Kapitel, an dessen Durchführung er maßgeblich beteiligt war.

Name	**Bericht-Kapitel**
Claudia Crispi	2. Dialogdesign
Kerstin Klöckner & Claudia Crispi	3. Wizard of Oz Experiment
Norbert Pfleger	4. Architektur, Modellierung & Implementierung des Dialogsystems
Diana Raileanu & Cordula Klein	5. Spracherkennung im Fahrstuhl mit HTK
Christian Dressler	6. Steuerung
Eric Auer	7. Soundkarten

Tabelle 0.1.: Zuordnung von Studenten zu Bericht-Kapiteln

Bedanken möchten wir uns an dieser Stelle bei folgenden Menschen und Firmen, die uns bei der Umsetzung unseres Projektes zur Seite standen: Herr Weiland und Herr Tom Scharf vom "Liftdienst Weiland", Herr Dr. Robert Neuss von der Firma "Usability Engineers", Herr Klaus Albert von der Firma "Newlift GmbH", Herr Michael Mungai von "Musik-Service Mungai", Herr Henning Müller von "Henning's Musik-Shop" sowie die Herren Bogdan Jokisz und Simon Hirth von der Systemgruppe der Fachrichtung Computerlinguistik. Besonderer Dank geht auch an Cordula Klein für die Aufnahme der Fahrstuhläußerungen.

1. Einleitung

Ziel des Projektseminars “Sprachsteuerung im Aufzug” im Sommersemester 2000 war die Entwicklung eines Dialogsystems, das die natürlich-sprachliche Bedienung von Aufzügen ermöglicht. Die Idee zu einem solchen System entstand im Zuge der Renovierungsarbeiten am Institut für Computerlinguistik und Phonetik, während derer ein neuer Fahrstuhl in das Gebäude eingebaut wurde. Die Firma “Weiland Liftdienst”, die für den Einbau des Fahrstuhls verantwortlich war, war gerne zu einer Zusammenarbeit bereit und sorgte für die nötigen Installationen im Fahrstuhlschacht.

Die Funktionalität des beschriebenen Sprachsteuerungssystems wurde im Hinblick auf seine Akzeptanz durch den Benutzer entwickelt. Dazu wurde in einem ersten Schritt ein virtuelles System entworfen und dann durch Simulation in einem sogenannten “Wizard of Oz-Experiment” an Benutzern getestet. Der finale Entwurf ergab sich dann aus den Erfahrungen des Experimentes.

Spracherkenner und Dialogsystem sind die wesentlichen Komponenten des realisierten Sprachsteuerungsystems für den Aufzug. Im Verlauf des Seminars wurde von den Studenten ein phonembasierter Spracherkenner mit Hilfe des “Hidden Markov Toolkit” (HTK) entwickelt. Alternativ wurde auch ein kommerzielles Spracherkennungssytem der Firma Lernout & Hauspie eingesetzt. Das implementierte Dialogmodul koordiniert die verschiedenen Komponenten und die möglichen Abläufe von Steuerungsdialogen mit einem geeigneten endlichen Automaten.

Das Projektseminar war bereits zu Beginn interdiziplinär angelegt. Es hat einen funktionierenden Prototypen hervorgebracht, der beispielhaft zeigt, wie sich Forschungsergebnisse aus den Gebieten Computerlinguistik, Phonetik und Informatik in praktisch relevante sprachtechnologische Systeme umsetzen lassen. So könnte das existierende System zum Beispiel als behindertengerechter Fahrstuhl genützt werden.

Der vorliegende Bericht ist folgenermaßen aufgebaut: Kapitel 2 beschäftigt sich mit den Fragen des Dialogszenarios und dem daraus resultierenden Design eines Dialogautomaten, Kapitel 3 beschreibt die Vorbereitung und Durchführung des Wizard of Oz Experiments. In Kapitel 4 wird die Architektur, Modellierung und Implementierung des Dialogsystems beschrieben, Kapitel 5 berichtet über die Spracherkennung mit HTK. Kapitel 6 beschreibt die Anpassung der Steuerung und Kapitel 7 berichtet über den Einbau der Soundkarte.

2. Dialogdesign

Die Entwicklung von Systemen, die einen Dialog mit einem Menschen führen können, ist schon lange eine Herausforderung für die Wissenschaft. In den letzten zehn Jahren haben sich wesentliche Verbesserungen in der Sprachtechnologie ergeben, die die Implementierung eines natürlichsprachlichen Dialogsystems ermöglichen.

2.1. Szenario

Der zu entwickelnde Prototyp des sprachgesteuerten Aufzugs sollte die folgenden Anforderungen erfüllen:

- Der Spracherkenner mußte sprecherunabhängig sein, d.h. es wurde ein Spracherkenner benötigt, der in der Lage ist Spracheingaben unabhängig von Aussprache, Akzent und Tonlage zu verstehen.
- Das System sollte so implementiert sein, dass es jeder ohne große Vorkenntnisse benutzen kann.
- Der Dialog mit dem Aufzug sollte natürlich und flexibel sein und daher möglichst in Echtzeit ablaufen.

Daraus resultierend ergaben sich eine Menge Fragen für die Lösungen gefunden werden mußten:

- Soll man den Fahrstuhl durch Außenmikrofone ansprechen können, d.h. soll es möglich sein den Fahrstuhl über das Außenmikrofon zu rufen bzw. gleich das Fahrtziel anzugeben?
- Soll die Dialoginitiative vom Benutzer oder vom Aufzug ausgehen?
- Soll der Dialog nur für einen oder auch für mehrere Benutzer konstruiert sein?
- Wie lange soll der Spracherkenner nach einer vom System gestellten Frage aufmerksam sein und eine Benutzereingabe erwarten?
- Wie soll sich der Aufzug verhalten, wenn die Eingabe nicht verstanden wird oder ambig ist?

- Welches Vokabular soll verwendet werden?
- Wie sollen die Zieleingaben der Benutzer bestätigt werden?
- Was geschieht, wenn der Benutzer ein bereits gegebenes Kommando korrigieren will, bzw. wenn der Aufzug das Kommando falsch verstanden hat?
- Soll der Fahrstuhl mehrsprachig sein? Dies wäre natürlich sinnvoll, da die Computerlinguistik und auch die Phonetik von internationalen Gästen und Studenten besucht wird.

Um diese Fragen zu klären, mußte zunächst festgestellt werden, welche Informationen zur Verfügung standen und wie diese verwendet werden konnten. Ein wichtiger Informationsträger war die Steuerung, sie liefert Informationen über den aktuellen Aufenthaltsort des Aufzugs, bzw. über seine Fahrtrichtung. Die Signale über den Zustand der Außen– und Innentür (geöffnet oder geschlossen) standen zwar zur Verfügung, konnten aber nicht verwendet werden, da die Signalübertragung zu lange dauerte. Die im Aufzug eingebaute Lichtschranke war nicht geeignet um festzustellen ob bzw. wieviele Personen im Aufzug sind, und eine Ermittlung über das Gewicht war nicht möglich.

Aus diesen Informationen, den Fragestellungen und den vorher gestellten Anforderungen entstand schließlich das folgende Szenario:

- Keine Sprachsteuerung über die Außenmikrofone, da dies wegen der ständigen Geräuschkulisse im Gebäude sehr problematisch ist und die Synchronisation mehrerer paralleler Spracherkenner voraussetzt. Der Spracherkenner würde nicht unterscheiden können, ob jemand einen Fahrbefehl abgegeben hatte, oder ob sich einfach nur Leute miteinander unterhalten. Deshalb soll zunächst das Rufen des Aufzugs von außen nur durch Tastendruck erfolgen.
- Da es nicht möglich ist festzustellen, ob sich jemand im Fahrstuhl befindet oder nicht, soll der Benutzer die Aufmerksamkeit des Spracherkenners erlangen, indem er ihn in der Kabine mit einem Schlüsselwort anspricht. Damit ist der Beginn des Dialogs klar definiert, und es werden Mißverständnisse vermieden, d.h. sollte der Spracherkenner eine nicht an den Aufzug gerichtete Äußerung als Fahrkommando verstehen, so wird dieses, ohne das vorherige Nennen eines Schlüsselworts, nicht ausgeführt. Als Schlüsselwörter wurden **Aufzug** und **Fahrstuhl** festgelegt. Nachdem eins dieser Wörter erkannt wurde, erfolgt die Frage des Aufzugs: *"In welchen Stock möchten Sie bitte?"*, und der Spracherkenner erwartet eine Zieleingabe.
- Der Dialog wurde für einen Benutzer konstruiert. Dies hat den Vorteil, dass der Spracherkenner nach dem Erkennen einer Zielangabe diese sofort bestätigen, und die Fahrt beginnen kann. Bei mehreren Benutzern wäre ein Zeitfenster von Nöten, um verschiedene Eingaben abzuwarten. Da es dem Aufzug aber nicht möglich ist festzustellen, ob sich mehrere Benutzer darin befinden, wäre das Zeitfenster bei nur einem Benuzter zu groß, bzw. es würden unnatürlich lange Pausen entstehen.

- Damit war auch die Frage beantwortet, wie lange der Spracherkenner eine Eingabe erwarten sollte. Es mußte ein Zeitfenster festgelegt werden, das groß genug war, um dem Benutzer eine Eingabe zu ermöglichen, aber auch klein genug, um einen flüssigen Dialog zu gewährleisten. Über die nötige Größe dieses Zeitfensters sollte das Wizard of Oz Experiment (Kapitel 3) Aufschluß geben.

- Die Erkennung der Zieleingabe erfolgt durch Keywordspotting. Als Keywords wurden *Keller, Untergeschoss, Erdgeschoss, erster Stock, zweiter Stock, dritter Stock, vierter Stock, fünfter Stock, Professor Pinkal, Professor Barry, Professor Uszkoreit, Systemgruppe, Konferenzraum* und *Phonetik* festgelegt.

- Falls die Zieleingabe des Benutzers nicht verstanden wird, soll es eine Rückfrage des Aufzugs geben. Wenn als Ziel Phonetik angegeben wird, fragt der Aufzug: *"Möchten Sie in den vierten oder fünften Stock?"*, da sich die Phonetik über diese beiden Etagen erstreckt.

- Nach der Zieleingabe erfolgt durch den Aufzug eine gesprochene Bestätigung des Ziels: *"Ich fahre Sie in den ... Stock."*.

- Wurde das Ziel falsch verstanden, ist es dem Benutzer möglich die Zielangabe zu korrigieren. Die gleiche Möglichkeit besteht auch für den Fall, dass der Benutzer versehentlich ein falsches Ziel angegeben hat.

- Das Szenario wurde auf die deutsche Sprache beschränkt. Mehrsprachigkeit sollte erst hinzukommen, wenn der deutsche Dialog funktionierte.

2.2. Dialogmodell

Auf der Basis dieses Szenarios wurde ein Dialogmodell (Abbildung 2.1) entwickelt, dessen Ablauf wie folgt aussieht:

Spricht der Benutzer den Fahrstuhl in der Kabine durch Verwendung des Schlüsselworts an, erfolgt eine Rückfrage des Aufzugs nach dem Ziel, und der Aufzug erwartet eine Eingabe. Erfolgt keine Zielangabe innerhalb des Zeitfensters, kehrt der Dialog zum Ausgangszustand zurück.
Wird eine Zielangabe gemacht, oder wurde vom Benutzer ein direktes Fahrkommando, wie z.B.: *"Fahrstuhl, dritter Stock!"*, eingegeben, dann erfolgt eine Bestätigung des Ziels, wenn die Eingabe erkannt wurde. Danach wird ein Zeitfenster gesetzt, damit es dem Benutzer möglich ist, das Ziel zu ändern, bzw. ein falsch verstandenes Ziel zu korrigieren. Erfolgt keine Korrektur, beginnt die Fahrt.
Wenn das Ziel nicht verstanden wird, erfolgt eine Rückfrage des Aufzugs, wird das Ziel dann immer noch nicht verstanden, fordert der Aufzug den Benutzer auf, die Tasten zu

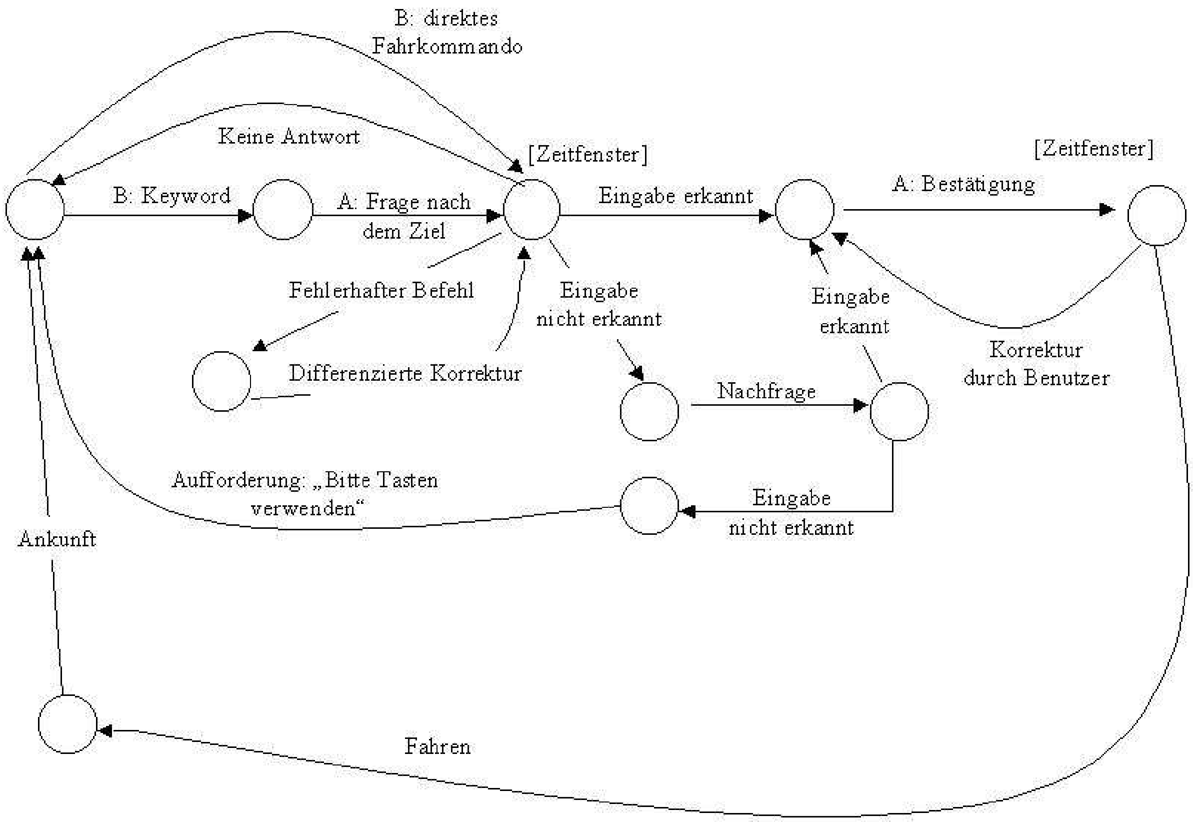

Abbildung 2.1.: Dialogmodell

verwenden. Sobald das gewünschten Stockwerk erreicht ist und der Dialog nicht abgebrochen wurde, erfolgt die Ansage des Fahrstuhls: *“Sie sind jetzt im . . . Stock.”*.

Dieses Dialogmodell diente als Grundlage für das Wizard of Oz Experiment (siehe Kapitel 3). Die Implementierung des Dialogs wird ausführlich in Kapitel 4 beschrieben.

3. Wizard of Oz Experiment

In diesem Kapitel wird ein Wizard of Oz Experiment im Zusammenhang mit der Entwicklung des sprachgesteuerten Fahrstuhls beschrieben.
Abschnitt 3.1 beschreibt, was ein Wizard of Oz Experiment ist, sowie Gründe für seine Durchführung. Abschnitt 3.2 behandelt eine konkrete Anwendung eines Wizard of Oz Experiments und gliedert sich in seine Planung (3.2.1), Entwicklung der Aufgabenstellung (3.2.2), Entwicklung des Fragebogens (3.2.3) und die Durchführung des Experiments (3.2.4). Abschnitt 3.3 beschreibt die Auswertung der gewonnen Daten und unterteilt sich in die Auswertung der Fragebögen (3.3.1) und die Auswertung der Sprachdaten (3.3.2). Das Kaptel schließt mit der Konklusion (Abschnitt 3.4).

3.1. Was ist ein Wizard of Oz Experiment?

Die Entwicklung eines benutzerfreundlichen Dialogsystems erfordert umfangreiche Experimente mit realen Benutzern. Ein simulierter Mensch–Mensch Dialog eignet sich bei unserem sprachgesteuerten Fahrstuhl zum einen prinzipiell nicht, da ein Mensch–Mensch Dialog in einem Fahrstuhl nicht vorkommt (es sei denn, es gibt einen Liftboy). Zum anderen lässt sich ein Mensch–Maschine Dialog schwer durch einen Mensch–Mensch Dialog simulieren — es sei denn, zwei Personen sitzen sich gegenüber und einer "spielt" den Fahrstuhl und der andere den Benutzer. Ein weiterer Grund, warum sich ein Mensch–Mensch Dialog dafür nicht eignet, ist der, dass Menschen dazu tendieren, bei einem Dialog mit einer Maschine anders zu reden, als wenn sie mit einer anderen Person kommunizieren.[1]

Ein Wizard of Oz (kurz: WOZ) Experiment ist eine Studie, bei der den Versuchspersonen gesagt wird, daß sie mit einem natürlichsprachlichen System interagieren, obwohl sie das in Wirklichkeit nicht tun. Stattdessen wird die Interaktion durch einen menschlichen Operator, den Wizard, vermittelt.

Prinzipiell lassen sich zwei Arten von WOZ Experimenten unterscheiden: eine mit frei-

[1] Bei linguistischen Untersuchungen hat man herausgefunden, dass Sprecher sich den vermeintlichen Eigenschaften des Gesprächspartners anpassen. So unterscheidet sich Sprache, die an Kinder gerichtet ist (Kindersprache) von Sprache gerichtet an Ausländer (z.B. kurze Sätze oder Erhöhen der Lautstärke) oder an Erwachsene (Korrekturen oder unvollständige/ungrammatische Sätze). Es gibt gute Gründe anzunehmen, dass solche Anpassungen auch beim Dialog mit einer Maschine gemacht werden.

en und die andere mit vorgegebenen Antwortmöglichkeiten. Bei der Variante mit freien Antwortmöglichkeiten hat der Wizard freie Hand bei der Wahl seiner Antworten. Im Gegensatz dazu muß sich der Wizard bei der Variante mit den vorgegebenen Antworten an diese halten. Diese Art ist dann sinnvoll, wenn im Verlauf der Systemplanung schon mehr oder weniger feste Vorstellungen existieren, die nur auf ihre Tauglichkeit bzw. Benutzerfreundlichkeit hin überprüft und gezielt Daten gesammelt werden sollen. Demgegenüber bevorzugt man ein WOZ Experiment mit freien Antwortmöglichkeiten, wenn man sich noch keine oder wenig Gedanken über die potentiellen Wirkung des Systems gemacht hat und möglichst viele verschiedene Daten sammeln möchte.

Ziel eines Wizard of Oz Experiments ist die Sammlung von Mensch–Maschine Daten. Solche Experimente sind deshalb vor der kompletten Implementierung des Systems sinnvoll, da man so, die beim Experiment gewonnenen Daten auswerten und nutzen kann, um ein System zu erstellen, ohne zuvor schon ein komplettes System implementiert zu haben, das danach wieder re–implementiert oder sogar neu geplant werden muss. Die WOZ Technik eignet sich deshalb für ein solches Experiment, weil man so Sprachdaten, die für die effektive Entwicklung eines Systems wichtig sind, sammeln und Hypothesen testen kann. Dazu gehört der Dialogverlauf, z.B. gibt es unnatürliche Situationen, sogenannte "dead ends", bei denen unklar ist, wer sprechen soll: der Fahrstuhl oder der Benutzer. Außerdem dienen die gesammelten Sprachdaten auch dem Trainieren des Spracherkenners.

Im nächsten Abschnitt wird beschrieben, wie diese Experimentaltechnik auf unser Fahrstuhlexperiment angewandt wurde.

3.2. Unser Experiment

Um zu testen, ob das von uns entwickelte Dialogmodell (siehe Kapitel 2 und Kapitel 4) von den Benutzern eines sprachgesteuerten Fahrstuhls akzeptiert wird, haben wir ein Wizard of Oz Experiment am "Tag der offen Tür" an der Universität des Saarlandes durchgeführt, da an diesem Tag mit einer hohen Besucherzahl gerechnet werden kann und somit genügend (potentielle) Versuchspersonen zur Verfügung stehen.
Bei unserem Experiment handelt es sich um ein WOZ Experiment mit vorgegebenen Möglichkeiten. Wir verwendeten ein fixes Modell (siehe Abbildung 2.1) und vorgefertigte Aufnahmen, die bei Bedarf abgespielt wurden.

3.2.1. Planung des Experiments

Bei der Vorbereitung des WOZ Experiments musste zunächst überlegt werden, was wir überhaupt mit dieser Studie erreichen wollten. Zum einen wollten wir verschiedene Unklarheiten beseitigen: es sollte die Frage geklärt werden, ob Benutzer anders auf eine aufgenommene Frauenstimme oder auf eine synthetisierte Computerstimme ansprechen würden. Welche Fahrkommandos würden am ehesten verwendet und akzeptiert? Wie reagiert der

Benutzer, wenn der Aufzug die Eingabe falsch versteht? Wie groß mußten die diversen Zeitfenster im Dialogdesign sein?
Der Dialog sollte schließlich so sein, dass es dem Benutzer angenehm war, mit dem Aufzug zu kommunizieren, ohne sich überfordert oder verunsichert zu fühlen. Also mußte dieses Experiment ganz genau geplant werden.

Es wurde ein 2x2x3 Design mit folgenden Variablen entworfen: 1. *Modus* mit den Werten "Mensch" und "Maschine" (das heisst, erfolgt die Sprachausgabe des Fahrstuhls mit einer aufgezeichneten Frauenstimme oder mit einer synthetisierten Computerstimme), 2. *ques* — kurz für "question" — mit den Werten "spezifisch" und "unspezifisch" (betreffend die Fragen des Fahrstuhls) und 3. *Dialogverlauf* mit den Werten "richtig"/"falsch" bzw. "gar nicht verstanden". Unter einer spezifischen Frage wird verstanden, dass der Aufzug seine Frage detailliert formuliert, wie z.B. *"In welche Etage möchten Sie bitte?"*, während die unspezifische Frage nur aus *"Wohin möchten Sie bitte?"* besteht. Diese Unterscheidung in der Fragestellung sollte zeigen, ob der Benutzer bei der spezifischen Fragestellung in seiner Eingabe manipuliert wird und folglich eine Etagen–Angabe macht, oder sich gleich verhält wie bei einer unspezifischen Frage. Mit der unspezifischen Fragestellung sollte herausgefunden werden, was dem Benutzer natürlicher erscheint und deshalb auch am häufigsten Verwendung findet (Stockwerksangabe oder Zieleingaben wie z.B.: *"Ich möchte zu Prof. Pinkal"*.).

Die verschiedenen Variablen des Versuch–Designs sollten bei der Durchführung des Experiments systematisch durchvariiert werden. Außerdem wurde festgelegt, dass jeder Benutzer bei der dritten Fahrt (es waren drei Fahrten pro Person vorgesehen (siehe Aufgabenstellung Abschnitt 3.2.2)) falsch verstanden wurde, d.h. der Aufzug würde um eine erneute Angabe des Ziels bitten.

3.2.2. Entwicklung der Aufgabenstellungen

Am Ende des Experiments sollte für jede Etage ungefähr die gleiche Anzahl an Sprachaufnahmen vorliegen. Deshalb wurde eine konkrete Aufgabenstellung für die Versuchspersonen entwickelt, die einen kontrollierter Ablauf des Experiments gewährleisten sollte. Dabei mußte unbedingt darauf geachtet werde, dass die Versuchsperson den Aufzug "unbelastet" benutzte, d.h. sie sollte bei den Eingaben ihre Worte frei und unbeeinflußt wählen. Dies war deshalb so wichtig, weil die Sprachaufnahmen Aufschluß darüber geben sollten, welche Wörter für die verschiedenen Etagen als Zieleingaben verwendet werden. Da diese Aufnahmen auch von der Phonetikgruppe benötigt wurden, um für den Spracherkenner Lautmodelle zu trainieren, mußte die Aufgabenstellung so konzipiert sein, dass alle Etagen abwechselnd angefahren (bzw. angesprochen) wurden. Außerdem durften die Fahrten nicht zu lange dauern, denn das hätte größere Wartezeiten für die nächsten Versuchspersonen zur Folge gehabt. Zum anderen sollte der Benutzer mehrmals die Möglichkeit haben Spracheingaben zu machen, um sich so einen Eindruck der Leistungsfähigkeit des Systems zu verschaffen und sich eine Meinung darüber zu bilden.

Aus den vorgenannten Gründen kamen mündlich gestellte Aufgaben nicht in Frage, weshalb die Aufgabenstellung in schriftlicher Form formuliert wurde. Zunächst wurde ein Instruktionsblatt (Anhang A) entwickelt, worin die Verwendung der Schlüsselwörter "Aufzug" und "Fahrstuhl" erklärt wurde. Diese Instruktionen sollten den Versuchspersonen vor Beginn der Fahrt von einem Helfer vorgelesen werden und gewährleisten, dass alle Benutzer die gleichen Instruktionen erhielten.

Die schriftliche Aufgabenstellung war eine heikle Angelegenheit. Die Aufgabe mußte klar definiert sein, sollte leicht verständlich sein und durfte den Benutzer nicht in der Wahl seiner Worte beeinflussen. So wurde ein graphisches Modell für die Aufgabenstellung entwickelt, das einen Plan der fünf Etagen des Gebäudes enthielt. In jeder Etage war die Etagen–Nummer und der Name einer Person oder eines Raumes angegeben (siehe Anhang B), um dem Benutzer die Möglichkeit zu geben, bei der Zieleingabe zwischen der Angabe der Etagen–Nummer und dem Personennamen bzw. der Raumbezeichnung zu wählen. Daneben war eine Aufgabenstellung formuliert, die eine fiktive Situation darstellte: *"Stellen Sie sich vor, Sie haben einen Brief für Professor XY. Sie fahren zu seinem Büro."*. Dies sollte die Versuchsperson veranlassen, sich den angegeben Raum selbst auf dem eingezeichneten Plan herauszusuchen und dann dem Aufzug ein entsprechendes Sprachkommando zu geben um in die genannte Etage zu gelangen. Am Ziel angekommen fand die Person an der Tür des betreffenden Raumes eine weitere Aufgabenstellung vor. Dadurch mußte jede Versuchsperson mehrmals den Aufzug benutzen, blockierte ihn aber nicht ständig, für den Fall, dass es einen großen Besucherandrang geben sollte.

Es gab zwei verschiedene Aufgabenstellungen, die immer abwechselnd an die Versuchspersonen ausgegeben wurden. Die erste Aufgabenstellung deckte den Keller und die Etagen zwei und vier ab, während die zweite Aufgabenstellung die Etagen eins, drei und fünf abdeckte. Dies gewährleistete, dass alle Etagen abwechselnd angefahren wurden und von jeder Etage genügend Sprachaufnahmen enstanden.

3.2.3. Entwicklung des Fragebogens

Natürlich wurden noch mehr Informationen benötigt als durch die Sprachaufnahmen zu erhalten waren. Deshalb wurde ein Fragebogen (siehe Anhang C) erstellt, der Aufschluß über die Akzeptanz des *sprachgesteuerten Fahrstuhls* beim Benutzer geben sollte. Die Personen wurden darin zur allgemeinen Bedienung des Aufzugs, zu persönlichen Eindrücken und nach Verbesserungsvorschlägen befragt. Es wurden sowohl "geschlossene" als auch "offene" Fragestellungen dazu verwendet.

3.2.4. Durchführung

Am Tag der offenen Tür sollte der Versuch um zehn Uhr beginnen. Um die Besucher der Universität vom geplanten Experiment in Kenntniss zu setzen, waren Flugblätter auf dem Campus verteilt worden. Da nicht vorauszusehen war, wieviele Personen Interesse an einer

Testfahrt haben würden, wurde eine Terminliste vorbereitet, in die sich die Interessenten eintragen konnten. Entsprechend der eingetragenen Reihenfolge durften sie dann den Aufzug testen.

Bevor die Versuchsperson den Fahrstuhl bestieg, las ihr ein Helfer das Instruktionsblatt vor. Anschließend erhielt sie die Aufgabenstellung und einen Briefumschlag, der gemäß der Aufgabenstellung abzugeben war. Dies sollte die auszuführende Aufgabe realistischer erscheinen lassen. Nachdem die Versuchsperson ihre Fahrt beendet hatte und wieder im Erdgeschoß angelangt war, durfte sie den Briefumschlag öffnen und wurde gebeten, den darin enthaltenen Fragebogen auszufüllen. Anschließend wurde sie darüber aufgeklärt, dass es sich bei diesem Experiment um ein Wizard of Oz Experiment gehandelt hatte.

3.3. Auswertung

Anhand der beim Wizard of OZ Experiment gewonnen Daten — den von den Versuchspersonen nach dem Experiment ausgefüllten *Fragebögen* einerseits und den *Sprachdaten* (die Aufzeichnung der Versuchspersonen beim Dialog mit dem Fahrstuhl) andererseits — erhofften wir uns eine Verbesserung des Prototyps des sprachgesteuerten Aufzugs.

Leider lief bei der Realisierung des Experiments nicht alles so reibungslos ab, wie es vorgesehen war. Es gab verschiedene Leute, denen wir den Ablauf doch nochmals erklären mussten, oder solche, die sich nicht an die Aufgabenstellung hielten. Hinzu kam ein starker Besucherandrang und kleine technische Probleme. Dies alles führte dazu, dass die Variablen nicht wie geplant varriiert werden konnten und der Aufzug meistens im Modus "Mensch–spezifisch" gefahren wurde. Deswegen wurden die Auswertungen zusammengefaßt.

3.3.1. Auswertung der Fragebögen

Die Auswertung der Fragebögen dient dazu, Anregungen und Kritik der Versuchspersonen bezüglich des Systems *sprachgesteuerter Aufzug* konstruktiv zu nutzen, um das System zu erweitern und zu verbessern. Insgesamt hatten 51 Personen Fragebögen ausgefüllt, deren Auswertung in Tabelle 3.1 enthalten ist.

3.3.2. Auswertung der Sprachdaten

Die gesammelten Sprachdaten wurden so aufbereitet, dass jeder Fahrt im Fahrstuhl eine Menge von Wave–Dateien entsprechen, die fortlaufend durchnumeriert sind. Insgesamt wurden ca. 200 Dialoge aufgenommen. Mit Hilfe dieser Sprachdaten soll ein angemessenes Sprachmodell (siehe Kapitel 5) erstellt werden. Dieses dient als Grunglage für den Spracherkenner, indem häufig vorkommende Phrasen in das Modell aufgenommmen werden. Dazu

Nr	Frage	Antwort	Anzahl
1	bevorzugtes Kommando?	Fahrstuhl	19
		Aufzug	18
		Computer	5
		Lift	2
		alle vier	5
		persönlicher Name	2
2	Höflichkeit?	unhöflich	1
		normal	50
		zu höflich	0
	Ausführlichkeit?	zu knapp	2
		angemessen	49
		zu ausführlich	0
3	Stimme des Aufzugs?	angenehm	48
		unangenehm	3
		verständlich	49
		unverständlich	2
4	Vergleich zu anderen sprachgesteuerten Systemen?	sehr gut	3
		gut	47
		befriedigend	1
		ausreichend	0
		mangelhaft	0
		nicht zufriedenstellend	0
5	Was fehlt?	Aufzug soll “Bitte” sagen	50
		Wegbeschreibung zum Büro einer Person	7
		Mehrsprachigkeit	2
		Mikrofon in Kinderhöhe	1
6	Würden Sie den Aufzug wieder benutzen?	ja	51
		nein	0

Tabelle 3.1.: Auswertung Fragebogen

wurden die Sprachdaten mit einem von uns eigens dafür entwickelten Schema (siehe Anhang D) transliteriert.

Das Transliterations–Schema besteht hauptsächlich aus tags, die ein wenig an HTML–tags erinnern (da sie in "<>" eingeschlossen sind). Jede Äußerung des Fahrstuhls (diese entsprechen den Übergängen im Dialogmodell) ist mit einem tag gekennzeichnet, der die Äußerung unverwechselbar identifiziert und der jeweils mit dem Buchstaben "L" beginnt. Die Dialogzustände, in denen sich der Fahrstuhl befinden kann, sind durch einen tag gekennzeichnet, der mit einem "Z" beginnt. Die Kodierung der Zustandsänderungen sollte dazu dienen, zustandsabhängige Sprachmodelle zu generieren. Besonders wichtig sind dabei die tags, welche kennzeichen, ob der Fahrstuhl (<f>) oder der Benutzer (<b>) spricht, denn daraus lassen sich alle Benutzeräußerungen und das dabei verwendete Vokabular ermitteln. Außerdem haben wir gekennzeichnet, ob es sich um Überlappungen (<u>), Interjektionen (<i>) oder um Hintergrundgeräusche (<noise>) handelt. Die Liste aller verwendeter tags inklusive ihrer Bedeutung findet sich in Anhang D.

Eine Transliterationsdatei besteht aus einem **header** (Kopf der Datei) und den einzelnen **turns**. Als turns werden die jeweiligen Sprecherbeiträge bezeichnet, was in unserem Fall die Äußerungen des Fahrstuhls und der Versuchspersonen sind. Der header beinhaltet 1. den *Namen* des Tranliterierers/der Transliteriererin, 2. das *Datum* der Transliteration, 3. die *Experimentnummer*, die identisch mit dem Namen der Wave–Datei ist, 4. die Variablen des Designs ohne Dialogverlauf (zur Erinnerung: *Modus* (Werte: Mensch oder Maschine) und *"ques"* (Werte: spezifisch oder unspezifisch)), 5. die *Versuchspersonen–Nummer* und 6. eventuelle *Anmerkungen* des Transliterierers/der Transliteriererin. Der Rest der Datei besteht, wie oben schon erwähnt, aus den einzelnen turns. Jeder turn beginnt mit dem Datum, an dem der Dialog aufgezeichnt wurde und der genauen Uhrzeit. Danach werden die Zustandsänderungen kodiert, sowie die Sprecherbeiträge transliteriert. Abbildung 3.1 verdeutlicht, wie ein aufgenommer Dialog aussieht, nachdem er transliteriert wurde.

Man kann dem header entnehmen, dass die Transliteration von Malte Gabsdil (mg) am 10.07.2000 erstellt wurde. Da es sich um einen fiktiven Dialog handelt, trägt das Experiment keine Nummer, stattdessen ersetzen drei "x" die das Experiment kennzeichnenden Zahlen. Der Modus der Äusserungen des Fahrstuhls wurde mit der menschlichen Stimme ausgegeben (mode: Mensch), und die Art der Fragen ist spezifisch. Die Fahrt wurde von der Versuchsperson Nummer x gemacht. Anmerkungen gibt es keine. Danach folgt die Transliteration des Dialogs zwischen Benutzer und Fahrstuhl, der folgendermaßen ablief:

Der Benutzer beginnt den Dialog mit dem Fahrstuhl ordnungsgemäß, in dem er das Kommando *"Fahrstuhl"* sagt. Dabei geht der Dialog von Zustand Z1 mit dem Übergang L1 in den Zustand Z2 über.
In Z2 angekommen fragt der Fahrstuhl den Benutzer in welchen Stock dieser möchte (dies entspricht dem Übergang L2); gleichzeitig gibt der Benutzer sein Fahrtziel (*"Dritter Stock"*) an. Diese sich überlappenden Äußerungen sind durch die tags <u> (Beginn der Überlap-

```
<h>
bearbeitet von: mg
datum: 10.7.00
experiment: experimentxxx
mode: Mensch
ques: spezifisch
vp: x
Anmerkungen
</h>

2000-06-03__hh-mm-ss: <Z1,L1,Z2> <b> Fahrstuhl.</b>

2000-06-03__hh-mm-ss: <Z2,L2,Z3> <f> In <u> welchen Stock </u>
                      <u> <b> dritte Etage </b> </u> möchten
                      Sie bitte? </f>

2000-06-03__hh-mm-ss: <Z3,L3_n,Z7> <f> Ich habe Sie nicht <u>
                      verstanden </u> </f> <Z7,L3_f,Z9> <u>
                      <b> Dritter Stock </u> habe ich
                      gesagt. </b>

2000-06-03__hh-mm-ss: <Z9,L4_4,Z11> <f> Ich fahre Sie in den
                      vierten Stock. </f>

2000-06-03__hh-mm-ss: <Z11,L7,Z12> <b> <kr> Halt. <i> Aehm </i>
                      in den dritten. </kr> </b>

2000-06-03__hh-mm-ss: <Z12,L8_4,Z13> <f> Sie sind jetzt im
                      vierten Stock. </f>

2000-06-03__hh-mm-ss: <noise> <Z13,L9,Z1>
```

Abbildung 3.1.: Beispieldialog

pung) und </u> (Ende der Überlappung) markiert. Der Dialog befindet sich nun im Zustand Z3. Im folgenden wird auf die Zustandsänderungen verzichtet.
Daran schließt sich die Frage des Fahrstuhls an: *"Ich habe Sie nicht verstanden."*. Auch hier beginnt der Benutzer parallel zum letzten Wort der Äusserung des Fahrstuhls seine Rede und wiederholt sein Fahrziel (*"Dritter Stock habe ich gesagt."*).
Der Spracherkenner hat das Ziel falsch verstanden und sagt: *"Ich fahre Sie in den vierten Stock."*.
Nun nimmt der Benutzer eine Korrektur (gekennzeichnet durch den tag (<kr>) vor, in dem er *"Halt"* sagt, eine Interjektion (<i> Aehm </i>) macht und sein gewünschtes Fahrtziel wiederholt (*"In den dritten"*).
Allerdings kam diese Korrektur zu spät, das verstandene Ziel (vierter Stock) wurde schon an die Steuerung weitergeleitet und der Benutzer wird in den vierten Stock gefahren. Dies erkennt man an der letzten Äusserung des Fahrstuhls, die lautet: *"Sie sind jetzt im vierten Stock."*.
Danach wurden noch Hintergrundgeräusche aufgezeichent (<noise>) und der Fahrstuhl geht in den Ausgangszustand Z1 zurück.

Auf diese Weise wurden alle Sprachdaten transliteriert und aus diesen Daten wurde mit Hilfe eines Perl–Skripts ein Vokabular erstellt. Wie bereits erwähnt, dienten die Benutzeräußerungen dazu, ein Sprachmodell zu erstellen. Näheres dazu findet sich in Kapitel 5.

3.4. Konklusion

Im Großen und Ganzen haben wir viel durch das Wizard of Oz Experiment gelernt. Zum Beispiel haben wir ein Vokabular für Korrekturen erhalten, welches in einer spätere Version des Aufzugdialogs implementiert wird. Leider konnte das Experiment nicht richtig ausgewertet werden. Dies lag zum größten Teil daran, dass die Variablen beim Experiment nicht konsequent durchvariiert wurden; somit konnten wir nicht mit einhundertprozentiger Sicherheit herausfinden, ob die Benutzer eine aufgenommene menschliche Stimme oder eine synthetisierte bevorzugen, bzw. ob sie spezifische oder unspezifische Fragen des Fahrstuhls präferieren.

Was sollte das nächste Mal besser gemacht werden? Zum einen sollten die Variablen konsequent variiert werden, damit relevante Daten zum Auswerten zur Verfügung stehen. Zum anderen hätte vor dem eigentlichen WOZ Experiment ein Probelauf mit wenigen Versuchspersonen gemacht werden sollen, damit mögliche Fehler bzw. Probleme hätten beseitigt werden können. Außerdem hätten wir vor Beginn des Experiments genauer spezifizieren sollen, was wir damit herausfinden wollen.

4. Architektur, Modellierung & Implementierung des Dialogsystems

Im Folgenden werden die wesentlichen Konzepte und Modelle des Sprachgesteuerten Fahrstuhls dargestellt. Aus den Anforderungen an ein solches System lassen sich recht einfach die wesentlichen Komponenten des Systems ableiten. Allgemein betrachtet muß ein Dialogsystem in der Lage sein die Informationen zu sammeln, die benötigt werden, um eine konkrete Aufgabe zu lösen (zum Beispiel Ausgabe von Informationen oder dergleichen). Ein solches System muß also gesprochene Anweisungen verstehen und entsprechend reagieren. Die Reaktionen des Systems können auf der einen Seite aus sprachlichen Äußerungen, das System führt also einen Dialog mit dem Benutzer und auf der anderen Seite aus Datenbankanfragen oder Befehlen an externe Geräte bestehen.

Die Anforderungen an das Dialogmodul eines sprachgesteuerten Aufzuges sehen ähnlich aus, so sollte das System den Benutzer bei der Eingabe des Fahrtziels unterstützen (Dialog), weiterhin sollte das System die Zieleingabe des Benutzers an die Fahrstuhllogik weiterreichen. Die Frage ist jedoch, welche Äußerungen eines Benutzer muß der Dialog verarbeiten können und wie sollten die Reaktionen des Systems aussehen. Aus diesem Grund haben wir das in Kapitel 3 beschrieben Wizard–of–Oz Experiment durchgeführt. Aus den hieraus gewonnen Erfahrungen ist ein recht konkretes Modell des Dialog entstanden, welches die Grundlage für die Implementierung diese sprachverstehenden Systems bildete. Auch spielten die Erfahrungen aus der Entwicklung des Programms, welches dem Wizard–of–Oz Experiment zugrunde liegt, eine große Rolle bei der Entwicklung dieses Systems. So hat sich zum Beispiel an dem Treiber, der die Fahrstuhllogik anspricht und bereits im Wizard-of-Oz Experiment verwendet wurde, nichts mehr geändert (zum Thema Ansteuerung der Fahrstuhllogik siehe Kapitel 6).

4.1. Architektur

Auch die grobe Architektur wurde aus Wizard–of–Oz System übernommen. Wobei jedoch an Stelle des Menschen nun ein Spracherkenner getreten ist und die Entscheidung, wie es im Dialog weitergeht treffen jetzt Selektionfunktionen (siehe Abschnitt 4.2.1). Die grobe

Architektur des Systems besteht aus drei Kernmodulen (siehe Abbildung 4.1). Das zentrale Modul stellt der Dialog dar, hier laufen sämtliche Ein– und Ausgabe–Kanäle zusammen. Der Spracherkenner ermöglicht die Kommunikation des Benutzer mit dem Dialog–System und die Fahrstuhlsteuerung stellt das dritte Modul dar. Eigentlich sollte man von vier Kernmodulen ausgehen, da der Spracherkenner nur die unidirektionale Kommunikation zwischen Benutzer und Dialog ermöglicht. Das vierte Modul wäre somit die Sprachausgabe, aber in dem vorliegenden System ist die Sprachausgabe lediglich durch das Abspielen einzelner WAVE–Files realisiert und es erscheint etwas hochgegriffen, dies als eigenständiges Modul zu bezeichnen. Bei Systemen, die über eine größere Variabilität der Sprachausgabe verfügen sollen, ist es jedoch sinnvoll von einem eigenständigen Modul für die Sprachausgabe zu sprechen.

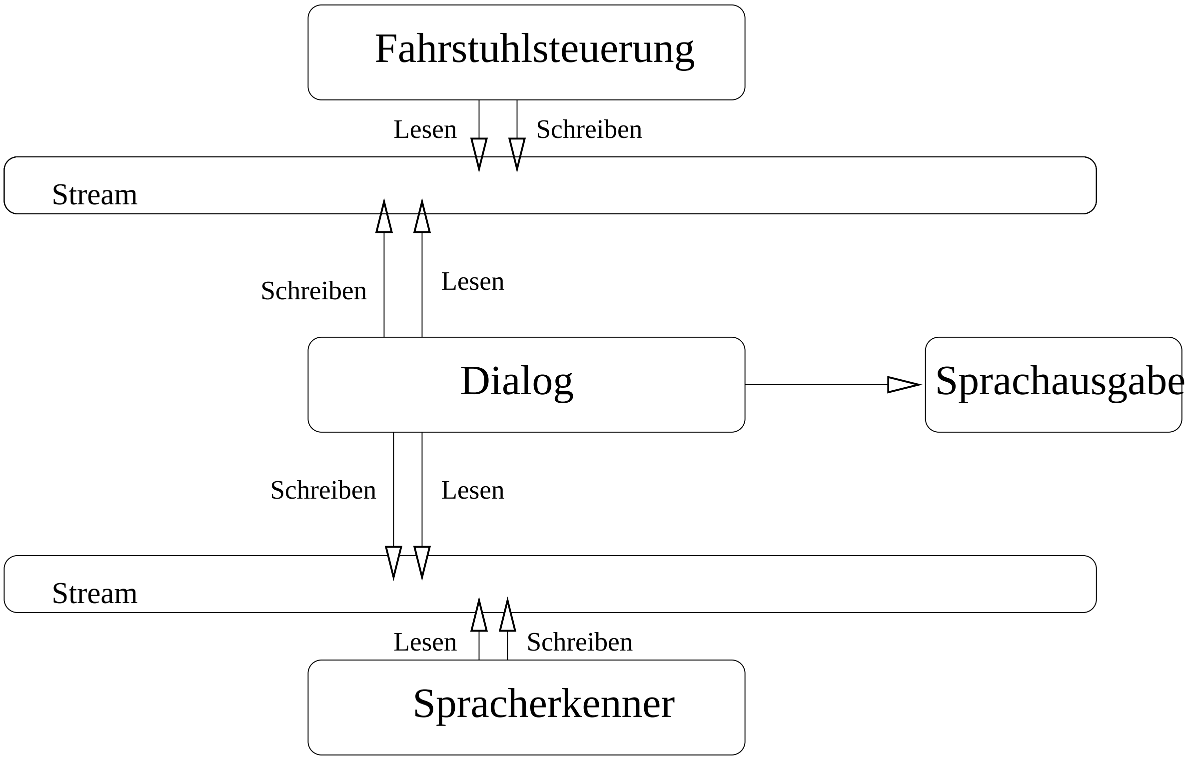

Abbildung 4.1.: Überblick über die Kern–Module

4.1.1. Anforderungen an das Dialogmodul

Wie bereits erwähnt ist das Dialogmodul das zentrale Modul in diesem Design. Denn hier laufen nicht nur alle Ein– beziehungsweise Ausgabe–Kanäle zusammen, sondern sämtliche Reaktionen des System, ob sprachlicher oder mechanischer Hinsicht, werden von diesem Modul veranlaßt. Das Dialogmodul muß also, im Hinblick auf den Dialogverlauf, ein Regelwerk enthalten, daß auf der einen Seite den Benutzer bei der Bedienung des Systems unterstützt (Frage – Antwort) und auf der anderen Seite aus den Anfragen des Benutzers die entsprechenden Fahrtbefehle für die Fahrstuhlsteuerung ableiten kann.

Im Hinblick auf die Steuerung des Dialogverlaufs muß das Dialogmodul eine größt mögliche Flexibilität aufweisen. So muß das System in der Lage sein auf verschiedenartige natürlichsprachliche Anfragen des Benutzer zu reagieren und im Falle mehrdeutiger Befehle, den Benutzer zu einem eindeutigen Befehl führen. In dem vorliegenden System ist der grobe Dialogverlauf in der folgenden Form realisiert (siehe auch Kapitel 2). Der Benutzer muß zunächst durch die Schlüsselwörter "Fahrstuhl" oder "Aufzug" den Dialog anstoßen und hat danach die Möglichkeit entweder einen direkten Fahrbefehl zu tätigen (zum Beispiel "Fahrstuhl, dritter Stock") oder er kann die Nachfrage des Systems ("Wohin möchten Sie bitte") abwarten. Die Struktur der Fahrbefehle ist so flexibel wie möglich gehalten, so kann der Benutzer direkt Etagen ("Ich möchte in den dritten Stock"), Personennamen (" zu Professor Pinkal, bitte") oder Gruppen beziehungsweise Einrichtungen ("bitte zur Systemgruppe") benennen. Im Falle einer ambigen Anfrage "Ich möchte zur Phonetik" (der Fachbereich Phonetik befindet sich auf zwei Etagen), frägt das System nach, in welche der beiden Etagen der Benutzer möchte (" Möchten Sie in den vierten oder fünften Stock").

4.1.2. Kommunikation der Module

Ein zentraler Punkt bei der Entwicklung eines solchen System ist die Kommunikation der einzelnen Module. So sollte das Dialogmodul ständig die Möglichkeit haben Anfragen an das Spracherkennermodul zu schicken, um zu erfahren, ob der Benutzer etwas gesagt hat. Um keine Äußerungen des Benutzer zu verlieren, sollte der Spracherkenner ständig aktiv sein, also paralell, als eigenständiger Prozeß laufen. Weiterhin sollte der Dialog die Möglichkeit haben ständig Informationen über den Zustand des Fahrstuhls (in welchem Stock er sich befindet, usw.) abzufragen. Hieraus ergibt sich bereits eine starke Anforderung an das Systemdesign. Die Module müssen prinzipiell unabhängig von einander laufen. Diese Nebenläufigkeit erschwert natürlich die Kommunikation der einzelnen Module. Eine Möglichkeit miteinander zu kommunizieren, sind *Streams*. *Streams* sind im Grunde genommen Datenströme, die einen definierten Anfang, aber kein Ende besitzen und bieten den Vorteil, daß der schreibende Prozeß immer am Ende des *Streams* etwas hinzufügen kann und daß der lesende Prozeß trotzdem noch ältere Information vom *Stream* lesen kann (näheres zu Implementierung der *Stream*–Kommunikation, siehe Abschnitt 4.2.2 Streamkommunikation). Ein Problem bei der Kommunikation via *Streams* ist die Ungewißheit, zu welchem Zeitpunkt eine Information auf einen *Stream* geschrieben wurde, diese Problematik wird im Abschnitt Zeitverwaltung näher betrachtet.

4.2. Modellierung

Im folgenden werden die wesentlichen Konzepte, die in die Implementierung dieses Systems eingeflossen sind, dargestellt. Im Mittelpunkt stehen hierbei der Dialog, die Anbindung des Spracherkenners und die Zeitverwaltung.

4.2.1. Dialog

Die Anforderungen an den Dialog wurden ja bereits in Kapitel 2 beschrieben, wie lassen sich nun diese Anforderungen im Hinblick auf Effiziens und Wartung beziehungsweise Erweiterung modellieren? Wir haben uns dafür entschieden den Dialog in Form eines endlichen Automaten zu realisieren. Allerdings wurde das Konzept des endlichen Automaten um einen globalen Speicher, der in jedem Zustand des Automaten zugänglich ist, und um die Möglichkeit Aktionen, beim Übergang von einem Zustand in einen anderen auszuführen, erweitert. Diese Aktionen sind Funktionen, die zum Beispiel eine Sprachausgabe oder das Absetzen eines Fahrtbefehls ermöglichen.

Der Dialog besteht aus einer endlichen Menge von Zuständen, diese Zustandsmenge ist eine Liste von Datenstrukturen, die ihrerseits wiederum Datenstrukturen enthalten. So besteht ein Zustand aus einem Feld (*record*), das auf der einen Seite die möglichen Übergänge in einen Folgezustand (auch in Form eines Feldes) und auf der anderen Seite eine Selektionsfunktion enthält. Diese Selektionsfunktion wählt aus der Menge der möglichen Übergänge denjenigen aus, der am ehesten dem akuellen Situation entspricht. In einem Zustand des Dialogs, in dem vom Benutzer eine Äußerung erwartet wird, stellt die Selektionsfunktion zunächst eine Anfrage an das Spracherkennermodul und wählt dann abhängig vom Ergebnis dieser Anfrage den passenden Übergang in einen Folgezustand aus.

Die Spezifikation der Übergänge erfolgt in Form von Funktionen, wobei jede Übergangsfunktion als Argument ein Atom, das dem Folgezustand entspricht, erwartet. Der Rückgabewert dieser Funktionen besteht wiederum aus einem Feld, das neben dem Folgezustand und einem Text, der den Übergang beschreibt, eine anonyme Prozedur enthält, in deren Rumpf der Aufruf der einzelnen Aktionen kodiert ist. Durch diesen rekursive Aufbau der zum Automaten gehörigen Datenstrukturen und der daraus resultierenden kompakten Darstellung wird sowohl die Entwicklung beziehungsweise Wartung als auch die Verarbeitung des Automaten vereinfacht.

Wie bereits am Anfang dieses Abschnittes erwähnt, verfügt dieser Automat über einen zusätzlichen Speicher. Dieser Speicher dient in erster Linie dazu die Komplexität des Automaten zu verringern, da eventuelle Argumente nicht in Form zusätzlicher Zustände kodiert werden müssen. Das Konzept des zusätzlichen Speichers wird unter anderem für die Verwaltung der zeitlichen Abfolge benötigt, so muß die Anfrage der Selektionsfunktion an den Spracherkenner einen Zeitpunkt enthalten, ab wann eine relevante Äußerung des Benutzer zu erwarten ist. Auch wird ein eventuell erkanntes Ziel in diesen Speicher geschrieben, um dieses Ziel für nachfolgende Zustände zu erhalten.

Um diesen Automaten verarbeiten zu können wird ein Interpretierer benögtigt. Hierbei handelt es sich im wesentlichen um eine rekursive Funktion, die ausgehend von einem Zustand mittels der Selektionsfunktion den Folgezustand berechnet und die zu dem berechneten Übergang gehörige Aktion ausführt. Der Endzustand, der eigentlich nur aus einem leeren Übergang zum Startzustand besteht, muß gesondert gekennzeichnet sein, da der Interpretierer diesen Zustand nutzt, um systemweit den Speicher zu bereinigen. Diese Prozesse der Speicherplatzverwaltung werden im Abschnitt 4.3.2 näher erläutert.

4.2.2. Schnittstellen

Streamkommunikation

Wie bereits erwähnt kommuniziert das Dialogmodul mit Spracherkenner und der Fahrstuhlsteuerung über *Streams*. Abstrakt gesehen produzieren oder konsumieren *Streams* sequentiell Datenstrukturen, die durch Ein– oder Ausgabe–Operatoren gelesen beziehungsweise geschrieben werden.

Sowohl beim Spracherkenner als auch bei der Steuerung handelt es sich um externe Prozesse, die in einer eigenen Shell laufen. In dem Vorliegenden System werden für jedes dieser Module zwei *Streams* verwaltet. Der erste *Stream* ist die direkte Verbindung mit dem Modul über *std–in* beziehungsweise *std–out*, alle Daten liegen auf diesem *Stream* in ihrer ursprünglich Form vor (*Strings*). Diese rohen Daten werden in Datenstrukturen konvertiert und auf einen zweiten *Stream* geschrieben.

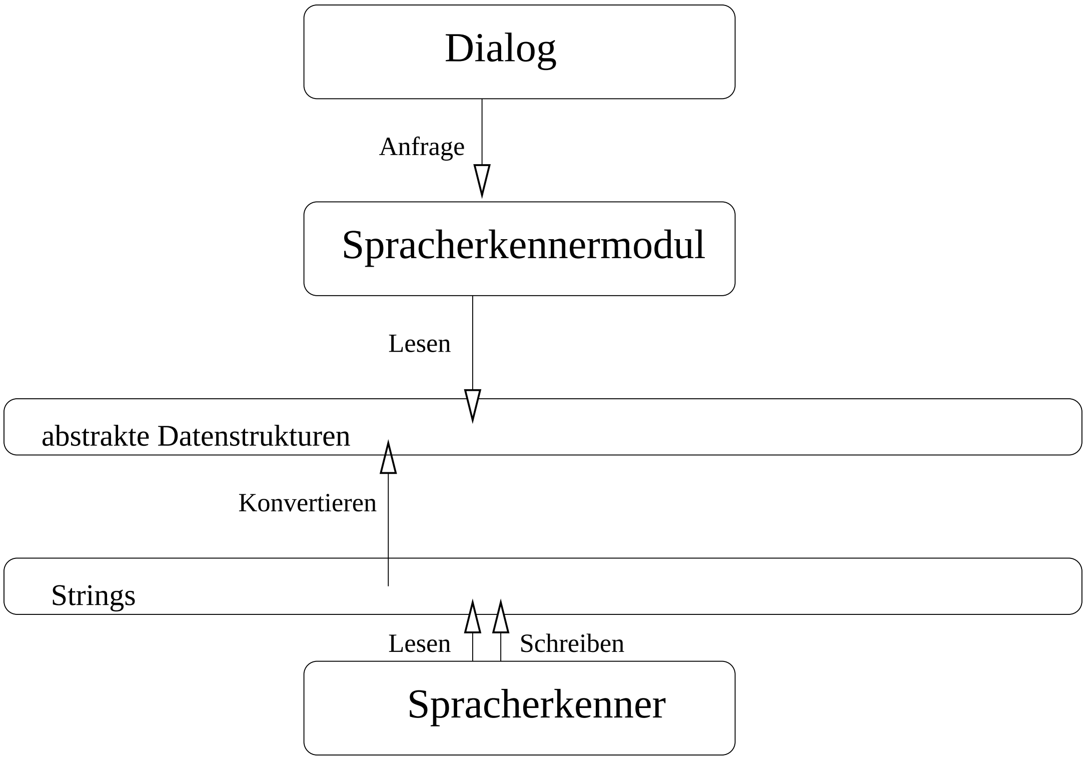

Abbildung 4.2.: Streamkommunikation am Beispiel des Spracherkenners

Im Falle des Spracherkenners (siehe auch Abbildung 4.2) besteht der erste *Stream* aus Paketen in *String*–Form, die Äußerungen darstellen. Diese Äußerungen werden analysiert und auf dem *Stream*, der die komplexe Datenstrukturen enthält, liegen die Wörter dieser Äußerungen in Form einer abstrakten Datenstruktur paketweise an. Diese Wörter verfügen neben den Informationen, um welches Wort es sich handelt und dessen Wahrscheinlichkeit, auch über ein Zeitintervall in dem das Wort aufgetreten ist. Der Vorteil dieses zweiten *Streams* liegt darin, daß Anfragen an den Spracherkenner wesentlich einfacher zu verarbeiten sind. So besteht auch die Möglichkeit mehrmals die gleichen Informationen des *Streams* zu lesen. Auch läßt sich auf diese Art und Weise recht komfortabel ein *Timeout*–Mechanismus realisieren. So kann nachträglich eine Datenstruktur *Timeout* an eine beliebige Stelle des *Streams* eingefügt werden, die es ermöglicht, die Suche auf dem *Stream* nach einer gewissen Zeit abzubrechen.

Anbindung des Spracherkenners

In der Vorliegenden Version des Systems kommt als Spracherkenner ein Programm der Firma Lernout&Hauspie zum Einsatz. Die Ergebnisse des Erkennungsprozesses werden paketweise auf einen *Stream* geschrieben. Es gibt zwei unterschiedliche Arten von Paketen, die jeweils durch die Schlüsselwörter *begin* und *end* umramt sind:

- *Control Statechange* der Form: control(topic:statechange old:... new:... time:...) wobei ein Zustandswechsel des Spracherkenners den Beginn einer Äußerung signalisiert, was zur Folge hat, daß der Dialog auf jeden Fall noch auf das Ergebnis dieser Äußerung wartet

- *rec(version:lh tot:... start:... end:...)*:
  wobei *tot* die Wahrscheinlichkeit und *start* bzw. *end* die Start– und End–Zeit der ganzen Äußerung repräsentieren. Ein solches Paket entspricht dem Ende eines Erkennungsprozesses und enthält zusätzlich die eventuell erkannten Wörter dieser Äußerung, zum Beispiel:

```
begin
rec(version:lh tot:66 start:678594.143 end:678595.653
Fahrstuhl()
in()
den()
dritten()
Stock()
end()
```

Eine wichtige Entscheidung beim Design eines solchen Systems ist die Art und Weise wie die Spracherkennung ablaufen soll. Soll die Spracherkennung immer aktiv sein (*Continious–spech–mode*), d.h. immmer “zuhören”, oder soll sie nur dann aktiv sein, wenn ein externer

Prozeß dem Erkenner ein Signal geschickt hat (*Open–Microphone*).

In dem vorliegenden System haben wir uns für die erste Variante, der konstanten Erkennung entschieden. Allerdings mit der Einschränkung, daß dem Erkenner ein *break* Signal gechickt werden kann, das den Erkenner sofort in einen *IDLE*–Modus (Ruhezustand) versetzt. Dieses *break*–Komando wird dann benötigt, wenn das System selbst eine Sprachausgabe tätigt, um zu verhindern, daß der Erkenner diese Äußerungen verarbeitet und auf den *Stream* schreibt ("Selbstaufnahme").

Auf Seite des Spracherkennermoduls werden die relevanten Pakete, die auf diesem *Stream* liegen, herausgefiltert, konvertiert und auf einen weiteren *Stream* geschrieben. Dieser zweite *Stream* dient dann als Grundlage für Funktionen, die innerhalb von Zeitintervallen nach bestimmen Wörtern suchen. So wartet die Funktion *WaitKeyword* ab einem gewissen Zeitpunkt solange, bis der Spracherkenner eins der Schlüsselwörter erkannt hat, die als Parameter mitgegeben werden. Dazu wird das erste Paket auf dem zweiten *Stream* untersucht, ob dessen Startzeit kleiner als die, durch den Funktionsaufruf übergebene Startzeit der Suchanfrage ist. Ist dies der Fall, wird das nächste Paket auf dem *Stream* untersucht, andernfalls wird getestet, ob eins der Schüsselwörter in diesem Paket enthalten ist. Die Funktion *WaitKeyword* wartet solange, bis eins der Schlüsselwörter auf dem *Stream* anliegt. Die zweite Funktion, die als Schnittstelle zum Spracherkenner dient, ist die Funktion *Nextutterace*. Diese Funktion wartet nur eine bestimmte Zeit, ob ein Element einer Menge von Wörtern auf dem zweiten *Stream* anliegt. Taucht in dieser Zeitspanne das Wort nicht auf, so gibt die Funktion *nil* zurück. So wird der Dialog abgebrochen, wenn zum Beipsiel auf die Frage des Systems "Wohin möchten Sie bitte?" nach einer gewissen Zeitspanne keine Reaktion gekommen ist.

Anbindung der Fahrstuhlsteuerung

Die Schnittstelle zur Fahrstuhlsteuerung besteht im wesentlichen aus einem Treiber, der die Komunikation mit der Fahrstuhlsteuerung über die Serielle Schnittstelle verwaltet (zur Beschreibung dieses Treibers siehe Kapitel 6). Dieser Treiber ermöglicht auf der einen Seite Fahrbefehle (in der Form *Ziel(0)*) abzusetzen und auf der anderen Seite kann der augenblickliche Zustand des Fahrstuhl abgefragt werden (in der Form *info*).

Auf der Dialog–Seite sind die Anforderungen an die Fahrstuhlsteuerung recht einfach, Fahrbefehle müssen abgesetzt werden und Anfragen über den Zustand des Fahrstuhl, zum Beispiel in welchem Stock er sich befindet, müssen bearbeitet werden. Dies läßt sich mit der Funktionalität des Treibers einfach umsetzten. Ein Problem bereiten jedoch die Latenzzeiten des Kommunikationsprotokolls. So kann es durchaus zwischen 500 – 750 msec dauern, bis eine Anfrage über den Zustand des Fahrstuhls beantwortet wird. Da die Anfragen sequentiell abgearbeitet werden. hat dies zur Folge, daß in dieser Zeit keine Fahrbefehle abgesetzt werden können. Um jetzt eine Warteschlage für die Befehle und Anfragen zu vermeiden, wurde das folgende Schema für dieses Modul implementiert. Das Fahrstuhlsteuerungsmodul frägt zunächst einmal immer den Zustand des Fahrstuhls ab und beantwortet Anfragen des Dialogs immer mit der zuletzt eingegangen Information. Wenn der Dialog kurz vor dem Absetzen eines Fahrbefehls steht, wird dies dem Modul Fahrstuhlsteuerung mitgeteilt und die Steuerung unterbricht die ständige Abfrage des Fahrstuhlzustandes. Nachdem der Fahrbefehl abgesetzt wurde, beginnt die Steuerung wieder mit kontinuierlichen Zustandsabfrage. Dadurch wird sichergestellt, daß kein Fahrbefehl auf eine noch laufende Anfrage warten muß.

4.2.3. Zeitverwaltung

Die Zeitverwaltung (*Timing*) ist in einem System wie diesem, aufgrund der Echtzeitbedingung und durch die Notwendigkeit der Synchronisation verschiedener Prozesse, eine heikle Angelegenheit. Die Synchronisation wird durch die Nebenläufigkeit des Spracherkenners und der Fahrstuhlsteuerung zusätzlich erschwert. Wie bereits im vorherigen Abschnitt beschrieben, kommuniziert das Dialog–Modul mit dem Spracherkenner über zwei Funktionen (*waitKeyword* und *Nextutterance*) , die ständig Zugriff auf alle Pakete haben, die der Spracherkenner geliefert hat. Bei dieser Art der Informationsverwaltung stellt sich jedoch die Frage, welches Paket überhaupt zeitlich gesehen das richtige Paket für die aktuelle Anfrage ist.

Wie beschrieben verfügt jedes Paket über eine Start– und End–Zeit. Um das richtige Paket auswählen zu können, muß das Dialog–Modul über die Information verfügen, ab welchem Zeitpunkt vom Benutzer eine Eingabe erwartet wird beziehungsweise erwartet werden könnte. Aus diesem Grund gibt es neben dem Speicher für ein eventuelles Ziel auch einen Speicher für den Zeitpunkt, ab dem relevante Eingaben beginnen könnten. Der Inhalt der Speicherzelle wird dann bei einem Aufruf der Funktionen *Nextutterance oder* waitKeyword als Argument übergeben.

Wichtig ist jetzt noch, wann dieser Zeitspeicher gesetzt wird. Wenn das System zum Beispiel selbst eine Sprachausgabe getätigt hatte, wäre es am sinnvollsten direkt nach dem Ende der Äußerung diesen Speicher zu setzen, um eine eventuelle Äußerung des Benutzer nicht zu verpassen.

Dies wirft jedoch das Problem der "Selbstaufnahme" auf. Wenn das System selbst eine Sprachausgabe tätigt, ist es sehr wahrscheinlich, daß der Spracherkenner diese wieder aufnimmt und als Eingabe dem System zur Verfügung stellt. Allerdings benötigt der Spracherkenner nach dem Ende der Äußerung des Systems noch ein gewisse Zeit diese zu Verarbeiten. Während dessen ist der Erkenner allerdings inaktiv, d.h. weitere Sprachsignale können nicht verarbeitet werden und sind verloren. Würde der Benutzer direkt auf die Äußerung des Systems etwas sagen, zum Beispiel gegen eine Zielbestätigung protestieren, würde das System von diesem Protest nichts mitbekommen.

Um dieses Phänomen der Selbstaufnahme zu vermeiden, müssen der Spracherkenner und der Dialog zeitlich syncronisiert sein, d.h. es darf immer nur ein Prozeß den Audiokanal nutzen. Wie bereits im vorherigen Abschnitt beschrieben, ist dies in diesem System durch ein *Break*–Kommando realisiert, der Spracherkenner hat den Vorrang und prinzipiell immer Zugriff auf den Audiokanal, wenn jedoch von Seiten des Dialogs eine Sprachausgabe geplant ist, schickt dieser dem Spracherkenner ein *Break*–Kommando. Solange diese Kommando nicht widerrufen wurde, ist der Spracherkenner inaktiv. Nach der Sprachausgabe, widerruft der Dialog das *Break*–Kommando, setzt den Zeitspeicher und der Spracherkenner wird wieder aktiv.

Ein weiterer Punkt in dem das Timing eine Rolle spielt, ist die Synchronisation des Dialoges mit der Fahrstuhlsteuerung (zur Implementierung des Treibers zur Fahrstuhlsteuerung siehe Kapitel 6). Solange kein Fahrbefehl abgesetzt werden soll, frägt der Dialog in kurzen Abständen den Zustand des Fahrstuhls ab. Da das verwendete Protokoll relativ lange Latenzzeiten aufweist, würden Probleme entstehen, wenn der Dialog ein Fahrtziel absetzen möchte, aber noch auf ein Ergebnis des Kommandos vorher gewartet werden muß. Zur Vermeidung von längeren Wartezeiten, wird deshalb bereits kurz bevor der Dialog das Fahrtkommando absetzt, der Prozeß der Zustandsabfrage angehalten, und erst danach wieder aktiviert.

4.3. Implementierung

Das vorliegende System wurde in der Programmiersprache *Oz* des Mozart Systems (siehe www.mozart-oz.org) entwickelt. Die Vorteile dieser Programmiersprache liegen im wesentlichen in der Platformunabhängigkeit, der Netzwerkfähigkeit und der sehr einfachen Handhabung von nebenläufigen Prozessen. Unter Netzwerkfähigkeit ist die Möglichkeit der direkten Kommunikation via Datenstrukturen ziwschen mehreren Prozessen, die auf verschiedenen Rechnern laufen, über Internet beziehungsweise Intranet. Weitere wichtige Eigenschaften dieser Sprache sind auch die direkte Unterstützung rekursiver Datenstrukturen, die effiziente Verarbeitung rekursiver Funktionen und natürlich die automatische Speicherbereinigung (*Garbage collection*). Zu erwähnen sind auch noch die Constraint-Programmierfähigkeiten des Mozart Systems, die allerdings für dieses Projekt nicht ausgenutzt wurden. Die eben angeführten Eigenschaften bieten auch andere Hochsprachen, allerdings meist nur eine Teilmenge dieser Eigenschaften. Weiterhin kam auch die Sprache *C* zum Einsatz, da die API (Funktionsbibliothek) des Spracherkenners in *C* vorlag.

4.3.1. Modularisierung

Um die Entwicklung und Wartung zu erleichtern wurde das komplette System weitestgehend modularisiert. Im Anhang E befinden sich die einzelnen Dateien, an dieser Stelle werden lediglich die wesentlichen Module namentlich aufgelistet.

- *start.oz*: Initialisierung des Systems

- *lift.oz*: Interpretierer für den Dialog
- *dialog.oz*: Spezifikation des Automaten
- *recognizer.oz*: Verwaltung des *high–level* Daten–*Stremas*
- *recognizer–interface.oz*: Verwaltung des Spracherkenner–*Streams* (auf *String*–Ebene)
- *lift–control.oz*: Verwaltung des *high–level* Daten–*Streams*
- *lift–control–interface*: Verwaltung des Fahrstuhlsteuerungs–*Streams* (auf *String*–Ebene)
- *parameter.oz*: Verwaltung aller für das System relevanter Parameter
- und weiteren "Hilfsprogrammen": *stack.oz, ticket.oz, memo.oz, time.oz, ...*

4.3.2. Verwaltung des Arbeitsspeichers

Eigentlich fallen bei einem solchen System keine größere Strukturen an, die ein ausgefeiltes Speichermanagement notwendig machen, zumal die Hochsprache *Oz* bereits über eine eingebaute Speicherbereinigung (*Garbagecollector*) verfügt. Diese ermöglicht, während der Laufzeit des Systems, Speicherbereiche, die nicht referenziert werden, wieder freizugeben. Der Einsatz von *Streams* erfordert jedoch gewisse Speicherverwaltungsroutinen, da *Streams* zwar über einen eindeutigen Anfang, aber nie ein Ende verfügen. Dies hat für die Speicherverwaltung zur Folge, daß diese Datenstruktur immer größer wird und Speicherbereiche vom Anfang des *Streams* nicht frei gegeben werden können, solange noch Referenzen auf diese Teile existieren. Um dieses Problem zu umgehen, muß zu einem geeigneten Zeitpunkt der Lesekopf auf dem *Stream* nach vorne bewegt werden. Alle Informationen, die sich dann vor dem Lesekopf befinden, werden dann vom *Garbagecollector* frei gegeben.

4.3.3. Platformunabhängigkeit

Die Programmiersprache *Oz* wurde, wie bereits erwähnt, auch wegen ihrer Platformunabhängigkeit ausgewählt. Diese Entscheidung hat sich als eine sehr glückliche erwiesen. Ursprüngliche wurde dieses Dialogsystem unter *Linux* entwickelt. Damals wurde auch noch der in Kapitel 5 beshriebene Spracherkenner HTK der Firma Entropic verwendet. Während der Entwicklung der Schnittstellen zum Spracherkenner traten jedoch einige Probleme auf, so war es nur mit großem Aufwand möglich zum Startzeitpunkt der Äußerung (Echtzeitverhalten) eine Meldung des Spracherkenners zu erhalten. Auch bereitete die lange Verarbeitungszeit des Erkenners größere Probleme, die allerdings auf noch nicht richtig ausgereizte Einstellungen in der Konfigurationsdatei des Erkenners zurück zu führen sind. Aus diesem Grunde entschlossen wir uns zweigleisig zu fahren und bis die Probleme mit HTK beseitigt sind, einen anderen Erkenner einzusetzen. Augenblicklich kommt in unserem System ein Spracherkenner der Firma Lernout & Hauspie zum Einsatz, der allerdings nur unter *Windows* lauffähig ist.

Bei der Portierung des System nach *Windows* traten zwei Problemfelder hervor. So stellte sich heraus, daß Pipes, das heißt birektionale *Streams*, unter *Windows* nicht die gleiche Funktionalität haben wie unter *Linux/Unix*. Aus diesem Grund mußte die Kommunikation zwischen dem Spracherkenner und dem Spracherkennerinterface über eine *Socket*–Verbindung realisiert werden. Da der Treiber für die Fahrstuhlsteuerung bisher nur unter *Linux* zur Verfügung steht, läuft die Kommunikation in diesem Bereich auch über eine Netzverbindung. Ein weiteres Problemfeld ist der Einsatz von Peripherieprogrammen, wie zum Beispiel ein Tool zur Soundausgabe. Die meisten Programme unter *Windows* kennen keine Kommandozeilen–Argumente, sondern können nur über den Einsatz von Maus und Tastatur gesteuert werden, was die Integration in ein solches System erschwert beziehungsweise fast unmöglich macht.

4.3.4. Parameter des Systems

Um das System so weit wie möglich flexibel zu halten, können in der Datei *parameter.oz* verschiedene Parameter gesetzt werden. Einige dieser Parameter beziehen sich auf die Selektion betriebssystemabhängiger Funktionen andere spezifizieren Verzeichnispfade. Zusätzlich gibt es noch einen Parameter *TimeOut* der direkt Einfluß auf das Dialogverhalten hat, da an dieser Stelle definiert wird, wie lange auf eine Äußerung des Benutzers gewartet wird. Die folgende Aufstellung enthält die wichtigsten Systemparameter, die vor jedem Systemstart überprüft werden sollten:

- *Control* Optionen: *fake / real / none*
  Zu Testzwecken man auf die Fahrstuhlsteuerung verzichten bzw. diese vortäuschen

- *ControlInterface* Optionen: *internet / local*
  Setzt die Verbindungsart mit der Fahrstuhlsteuerung

- *Recognizer* Optionen: *HTK / LH / fake*
  Selektiert den Spracherkenner bzw. zu Testzwecken kann man auf den Spracherkenner verzichten bzw. diesen vortäuschen

- *ControlLocation* Optionen: *coli / ps*
  Dieser Parameter ist wichtig für bei der Internetverbindung mit der Steuerung

- *Audio* Optionen: *windows / linux* Selektiert das Tool und das zugrundliegende Betriebsystem für die Audiowiedergabe

- *TimeOut* Optionen: Zeitangabe in Millisekunden
  Spezifiziert, wie lange auf eine Äußerung des Benutzer gewartet werden soll

Die restlichen Parameter in dieser Datei müssen normalerweise nicht verändert werden, da sie sich im wesentlichen auf Pfade und interne Vorgänge beziehen. Am Ende dieser Datei kann für Testzwecke und zur Fehlersuche die Ausgabe des Systems modifierziert werden. So kann gezielt die Ausgabe einzelner Module verfolgt werden. Im normalen Betrieb ist diese Option deaktiviert.

4.3.5. Versionskontrolle & MakeFile

Versionskontrolle

Bei der Entwicklung größerer Systeme bereitet die Versionskontrolle meist Probleme, vor allem, wenn mehrere Leute gleichzeitig an gleichen beziehungsweise eng miteinander zusammenhängender Code-Fragmente, arbeiten. Durch unerwartete Seiteneffekte oder einfach Inkompalibitäten entsteht aus einem lauffähigen System ein nicht mehr zu überblickendes Chaos. Aus diesem Grund wurden sämtliche Komponenten des Systems einem *CVS*–Verzeichnis verwaltet.
CVS ist ein Unix-Tool zur Versionskontrolle von Dateien beziehungsweise ganzen Verzeichnisbäumen. Jeder Entwickler *checkt* sich seine eigene Version des Systems aus, das heißt er erstellt eine lokale Kopie des Systems, und arbeitet an dieser Kopie weiter. Bevor Änderungen in das globale System kopiert werden, kann man seine lokale Kopie *updaten*, das heißt eventuelle änderungen von anderen Personen aus dem globalen Verzeichnisbaum kopieren und dann die Lauffähigkeit des Systems testen. Darüber hinaus bietet *CVS* die Möglichkeit Änderungen an den Dateien rückgängig zu machen und ist der Lage jeden Zustand der Dateien wiederherzustellen.

MakeFile

Wie bereits in Abschnitt 4.3.1 beschrieben, besteht dieses System aus verschiedenen Modulen. Um das System zu starten müssen zunächst alle Module kompiliert und verschiedene andere Dinge vorbereitet werden. Diese Initialisierung des Systems wird durch ein *MakeFile* vorgenommen, dem verschiedene Parameter übergeben werden können. So wird durch den Befehl *make lift* das komplette System initialisiert und bei Bedarf einzelne Module neu kompiliert. Weiterhin besteht die Möglichkeit durch den Befehl *make debug* das System im Testmodus zu starten, der erweiterte Ausgaben über den Verbrauch von Systemressourcen liefert.

5. Spracherkennung im Fahrstuhl mit HTK

Obwohl in der automatischen Spracherkennung mittlerweile hohe Erkennungsraten erzielt werden, stellt ein in einen Fahrstuhl eingebauter Spracherkenner besondere Anforderungen und erwartet vom Entwickler bestimmte Entscheidungen, um die optimale Erkennungsrate zu erzielen.

Voraussetzung: Sprecherunabhängigkeit
Da ein Fahrstuhl eine potentiell infinite Anzahl an Benutzern hat, die das Gebäude besuchen, muß das verwendete System, anders als z.B. ein Diktiersystem für den PC oder ein Spracherkenner im Auto, sprecherunabhängig sein. Das heißt, daß sprecherspezifische Unterschiede wie Sprechgeschwindigkeit, Sprechtonhöhe (Grundfrequenz) und dialektale Herkunft keine Unterschiede in der Erkennung bewirken dürfen.

Folge: eingeschränktes Lexikon
Eine optimale Erkennung trotz dieser Variabilität innerhalb der Gruppe der Benutzer kann normalerweise nur bei einem eingeschränkten Lexikon gewährleistet werden. Diese Einschränkung stellt in unserem Fall kein Problem dar, da hauptsächlich Stockwerksangaben, Lehrstuhlbezeichnungen und Namen von Mitarbeitern erkannt werden müssen.

Ziel: continuous speech recognition
Da die Variabilität der Benutzeräußerungen in einem Fahrstuhl sehr beschränkt ist, würde ein *keyword spotting* ausreichen. Hierbei wird aus einem kompletten Satz, wie z.B. "Ich möchte in den dritten Stock" nur das Schlüsselwort "dritten" erkannt. Eine andere Möglichkeit ist *continuous speech recognition*, wobei alle Wörter einer Äußerung erkannt werden. Wir haben uns für eine Mischung aus beidem entschieden, d.h., daß mehr als nur die Schlüsselwörter erkannt werden, jedoch nicht alle. Das erlaubt uns, eine komplexere Grammatik zu erstellen, die die möglichen Benutzeräußerungen modelliert.

Weg: Kontextunabhängige Lautmodelle trainieren
Der erste Schritt in einem Spracherkennungsprozeß ist das Training von entweder Wortmodellen, kontextabhängigen oder kontextunabhängigen Lautmodellen. Für Wortmodelle wird ein Korpus benötigt, welches genug Realisierungen der zu trainierenden Wörter enthält. Ein solches Korpus zu erstellen ist sehr zeitintensiv, da die Daten erhoben und von

Hand segmentiert werden müssen. Da uns die entsprechenden Ressourcen nicht zur Verfügung standen, wollten wir ein bereits segmentiertes Korpus benutzen. Keines dieser Korpora hat genug Wiederholungen der für uns relevanten Schlüsselwörter, weshalb ein Training von Wortmodellen für uns nicht in Frage kam. Von den Möglichkeiten der Lautmodelle haben wir uns für letztere entschieden, da die Anzahl einer Laute in einem bestimmten Kontext (kontextabhängig) selbstverständlich geringer ist als die Anzahl über alle Kontexte hinweg (kontextunabhängig).

Frage: kommerzieller Spracherkenner oder ein eigens trainierter?
Die Entscheidung für einen kommerziellen Spracherkenner (z.B. ASR 1600 von Lernout & Hauspie) hätte einige Vorteile geboten. Dieses System ist beispielsweise im ersten Jahr der Benutzung kostenlos und von anderen Anwendern bereits getestet. Desweiteren sind die Modelle trainiert und weitgehend optimiert und ein Lexikon ist bereits vorhanden. Da dieser Erkenner in vielen verschiedenen Umgebungen einsetzbar sein muß, ist er außerdem so trainiert worden, daß er sehr robust gegenüber verschiedenen akustischen Hintergründen, d.h. Geräusch im Sprachsignal, ist.

Der größte Nachteil dieser Lösung ist, daß die Lautmodelle nicht speziell auf den Einsatz im Fahrstuhl trainiert wurden. Die Robustheit des Spracherkenners kann zwar ein großer Vorteil sein, allerdings ist die individuelle Anpassung der bereits vortrainierten Modelle auf unsere spezielle Umgebung im Fahrstuhl nur sehr eingeschränkt möglich. Als weitere Nachteile sind zu nennen, daß die Software zu diesem Erkenner kein fertiges Paket ist, sondern noch weiter angepaßt werden muß, dieser Erkenner nur unter Windows läuft und, sollte er kommerziell genutzt werden, sehr hohe Kosten entstünden.
Demgegenüber stand die Entscheidung für einen selbsttrainierten Spracherkenner mit dem Hidden Markov Tool Kit (HTK) der Firma Entropic. Die Nachteile dieser Variante ergeben sich zum Großteil aus den Vorteilen des kommerziellen Erkenners. So ist ein für einen speziellen akustischen Hintergrund trainierter Erkenner nicht so robust gegenüber anderen Hintergrundgeräuschen, es sei denn, er wird mit einer extrem großen Datenmenge trainiert, von der nicht angenommen wurde, daß sie durch ein einfaches Experiment, bei dem im Fahrstuhl Aufnahmen der Benutzer gemacht werden, erzielt werden kann. Dieses ist außerdem ein Zeitaufwand, der bei einem kommerziellen Spracherkenner nicht aufgebracht werden muß. Desweiteren waren die Kosten für eine kommerzielle Nutzung von HTK unbekannt.

Dennoch hat dieses System sehr große Vorteile. Abgesehen davon, daß es unter Unix läuft (wie der übrige Teil der für unseren Fahrstuhl entwickelten und eingesetzten Software auch) und wesentlich billiger ist, als der Erkenner von Lernout & Hauspie, hat sich HTK in der Forschung am Institut für Phonetik als flexibel und schnell erwiesen. Das System bietet einen direkten Zugriff auf akustische Parameter und die Möglichkeit eigene Lautmodelle zu trainieren, um den Erkenner individuell an unseren Fahrstuhl anzupassen. Da die Hintergrundgeräusche des Fahrstuhls (z.B. Echo) in der Datenbank, die zum Trainieren benutzt wurde, nicht vorhanden sind, kann eine weitere Verbesserung des Erkenners durch das Abspielen der Datenbank im Fahrstuhl erreicht werden. Das daraus resultierende neue Sprachmaterial kann für das Nachtrainieren der Lautmodelle verwendet werden.

Damit wird sowohl das Problem der fehlenden Robustheit des Erkenners als auch das der aufwendigen Segmentierung eigener Aufnahmen behoben.

5.1. Vorverarbeitung

Sprache ist ein Signaltyp, der sich durch hohe Variabilität und durch eine Entwicklung über die Zeit auszeichnet. Um ein solches Signal adäquat widerzuspiegeln, bietet sich eine stochastisch-probabilistische Hidden Markov Modellierung besonders an (siehe Abbildung 5.1).
McInnes und Mervyn (1988)[1] definieren Hidden Markov Modelle (HMM) wie folgt: "The

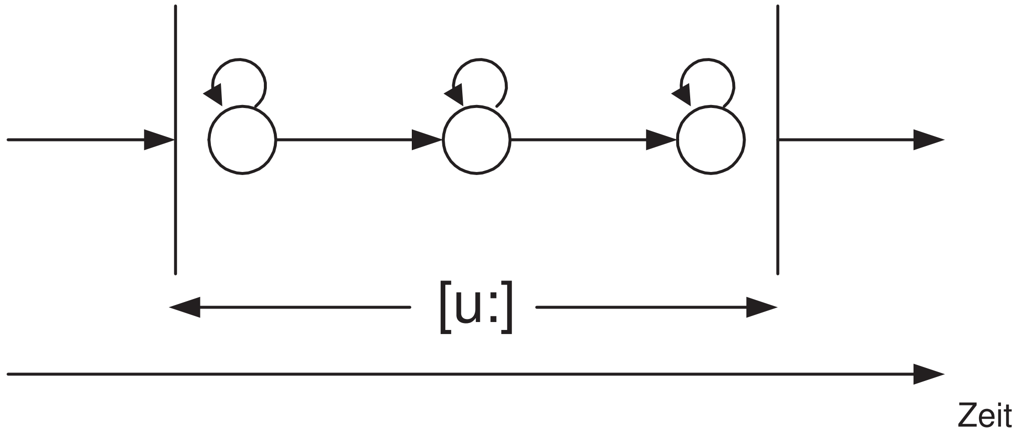

Abbildung 5.1.: HMM für den Laut [u:]

essential idea of the HMM approach is that each word in the vocabulary is represented by a set of states, including an initial and a final state, with probabilities of transitions from state to state, and for each state a probability distribution for the emission of a vector of acoustic parameters". In unserem Fall werden nicht Wörter, sondern Laute modelliert, d.h. die drei Kreise in Abbildung 5.1 spiegeln die Zustände wider, in denen sich unser Modell zu einem bestimmten Zeitpunkt t jeweils befindet. Es handelt sich um ein *left-to-right model*, dass zum Zeitpunkt $t+1$ entweder in den nächsten Zustand übergeht (transition probability) oder im momentanen Zustand (self-loop-probability) bleibt. D.h., das Modell kann nur von einem Zustand in den nächsten übergehen, nicht jedoch in einen vorhergehenden, und es können keine Zustände übersprungen werden. In jedem Zustand wird nun ein Output (*observation*) kreiert, solange, bis das Modell seinen Endzustand T erreicht hat. Das Output wird mit mehreren Wahrscheinlichkeitsverteilungen (mixtures) modelliert, um alle

[1]F. McInnes und A. Mervyn. Automatic speech recognition using word reference patterns. In Mervyn A. Jack und John Laver. *Edits 4: Aspects of speech technology.* Edinburgh: Edinburgh University Press. 1988

möglichen Variationen wie z.B. Kontext– und Sprecherabhängigkeit abzudecken. Spezifisch für einen Markov Prozess ist, daß Zustand *t+1* probabilistisch bestimmt wird und nur von seinem direkt vorhergehenden Zustand t abhängig ist. Im Folgenden wird der Prozess der Spracherkennung beschrieben, von der Bereitstellung des Trainingsmaterials bis hin zum eigentlichen Erkennungsprozess. Zunächst wird kurz die Datenbank vorgestellt, die die Basis unseres Trainingsmaterials bildet und deren Anpassung an HTK, sowohl die Anpassung der Signaldateien als auch die der Labeldateien. Weiter werden die zwei Schritte des Trainings (Initialisierung und Re–estimation) beschrieben, deren Ergebnis die Hidden Markov Modelle sind. Im vorletzten Abschnitt wird kurz auf die Struktur von Lexikon und Grammatik eingegangen, die anschließend zusammen mit den HMMs für den Testprozess, d.h. die Erkennung unbekannter Äusserungen, benutzt werden.

5.1.1. Trainingsmaterial

Das ursprüngliche Trainingsmaterial, das Kiel Corpus, ist eine wachsende gesprochene Sprachdatensammlung der deutschen Lese- und Spontansprache, welche seit 1990 am Institut für Phonetik und Digitale Sprachverarbeitung (ipds Kiel) aufgezeichnet und segmentell etikettiert wird. Derzeit umfasst das Kiel Corpus über vier Stunden etikettierter Lesesprache sowie knapp vier Stunden etikettierter Spontansprache. Der gelesene Teil umfasst von 26 weiblichen und 27 männlichen Sprechern gesprochene deutsche Prosa. Die drei spontansprachlichen Teile enthalten spontane Dialoge und Terminabsprachen von 26, 16 bzw. 10 Sprechern. Das Korpus lässt sich in drei Arten von Dateien untergliedern: Lexika (*.lsv), Signaldateien (spontane Sprache → *.r16, *.l16; gelesene Sprache → *.16) und Labeldateien (*.s1h). Letztere sind mit den Signaldateien zeitlich synchronisiert und enthalten deren Verschriftung, in Form von orthographischer und kanonischer Transkription und segmentellen Labels.

5.1.2. Signalverarbeitung

Da HTK nicht mit dem Sprachsignal direkt, sondern mit aus ihm abgeleiteten Parametern arbeitet, und auch die im Kiel Korpus verwendeten Label zu komplex für unsere Zwecke sind, wurden im Vorfeld sowohl die Signaldateien als auch die Labeldateien des Kiel Korpus für HTK angepasst. Um die spektralen akustischen Parameter abzuleiten, benutzt man *samples*.

Bei der Umsetzung eines kontinuierlichen analogen Signals in ein, aus diskreten Werten bestehendes, digitales Signal wird die Spannung, die vom Mikrofon, Kassettenrecorder oder DAT–Recorder kommt, in kleinen zeitlichen Abständen gemessen. Diese Messergebnisse, die abgespeichert werden, nennt man Samples. Beim Kiel Korpus geschieht das 16000 mal pro Sekunde (sampling rate = Abtastrate = 16000 Hz). Ein Sample repräsentiert also 1/16000 Sekunde des Zeitsignals, d.h. 1/16 Millisekunden = 625 100–Nanosekunden–Einheiten[2].

[2]Das ist die Einheit, die HTK intern verwendet.

Das Skript *HCopy* (siehe Anhang F) leitet aus den Signaldateien spektrale akustische Parameter ab (in unserem Fall zwölf MFCC (*Mel Frequency Cepstral Coefficients*), Energie und Delta–Parameter, d.h. die Differenz zwischen den gleichen Parametern mehrerer Vektoren als gewichtete Mittelwerte), die als Merkmalsbündel (*parameter vectors*) in .mfc–Dateien geschrieben werden. Diese Dateien werden anschließend als Trainingsmaterial für den Spracherkenner verwendet.

5.1.3. Verarbeitung der Labeldateien für die Lauterkennung

Die Umwandlung der Labeldateien wurde vorgenommen, da die im Kiel Korpus verwendete Annotation detaillierter und komplexer ist als für unsere Zwecke notwendig. Beispielsweise werden nicht nur zusätzliche Informationen zu Betonung und Melodie (*Prosodie*) gegeben, sondern auch auf lautlicher Ebene Unterscheidungen getroffen, die in unserem Fall nicht relevant sind. Ein Beispiel dafür findet sich in Abbildung 5.2. Hier wird bei nasalem Vorkontext das /d/[3] in "dreizehn" als [n] realisiert, welches die gleiche Artikulationsstelle hat. Das wird im Kiel Korpus[4] mit "d-n" kenntlich gemacht. Da die akustische Struktur des entsprechenden Signalteils einem /n/ entspricht, ist für uns das Label "n" ausreichend.

Für die Umwandlung der Labeldateien wurden zwei Perl–Skripte angewendet, die sich im Anhang G befinden. *lab_one_mit_grenzen.pl* wandelt eine einzige Datei im Kiel Korpus–Format in eine Datei im HTK–Format um, während *lab_all_mit_grenzen.pl* das gleiche für eine Liste von Dateien macht. Das Ergebnis des zweiten Skripts ist eine *.mlf* Datei (**M**aster **L**abel **F**ile), die alle HTK–Label–Dateien enthält, die im weiteren Trainingprozeß benutzt werden. Die Syntax sieht folgendermaßen aus:

lab_one_mit_grenzen.pl	*Kiel–Korpus–Label–Datei*	*HTK–Label–Datei*
lab_all_mit_grenzen.pl	*Kiel–Korpus–Label–Dateien–Liste*	*MLF–Datei*

Die Abbildung 5.2 zeigt eine Beispieldatei des Kiel Korpus. Sie ist wie folgt strukturiert:

1. Dateiname
2. ortographische Repräsentation, endet mit 'oend'
3. kanonische Repräsentation, endet mit 'kend'
4. Transkription, die durch die Plazierung der Labels des Segmentierers erstellt wurde
5. Zeitangaben und Labels/Ettikettierung

HTK braucht nur den fünften Teil, allerdings in einem anderen Format. Die entsprechende HTK Version von der Datei im 5.2 zeigt die Abbildung 5.3. Wir werden zuerst das Kiel

[3] Ein Laut in // stellt das Phonem dar, während [] die phonetische Realisierung eines Lautes symbolisiert.

[4] Für die Bedeutung der restlichen Labels, die im Kiel Korpus benutzt wurden, siehe K. J. Kohler. *Arbeitsberichte (AIPUK) Nr. 29.* Kiel: Institut für Phonetik und digitale Sprachverarbeitung Universität Kiel. 1995

```
g317a005.s1h
SVA005: <#Klicken> gut , um <:<#Klopfen> dreizehn Uhr:> <#Klicken> .
oend
  :k  g 'u: t  ,  Q U m+  :k d r 'aI t s e: n  :k Q 'u: 6  .  :k
kend
  :k  c:  &%1^  g -h 'u: t -h  ,  &0  Q- U m+  &0.  &1)
  :k- d-n r 'aI t s e:-@ n  &1.  &2^  :k- Q- -q 'u:6  .  &2;  &PGn  :k
hend
      86 #:k          0.0053125
      86 #c:          0.0053125
      86 #&%1^        0.0053125
      86 ##g          0.0053125
     313 $-h          0.0195000
     818 $'u:         0.0510625
    1562 $t           0.0975625
    2132 $-h          0.1331875
    2409 #,           0.1505000
    2409 #&0          0.1505000
    2409 ##Q-         0.1505000
    2409 $U           0.1505000
    3125 $m+          0.1952500
    3877 #&0.         0.2422500
    3877 #&1)         0.2422500
    3877 ##:k-        0.2422500
    3877 $d-n         0.2422500
    4492 $r           0.2806875
    5673 $'aI         0.3545000
    6813 $t           0.4257500
    7689 $s           0.4805000
    8633 $e:-@        0.5395000
    8917 $n           0.5572500
    9702 #&1.         0.6063125
    9702 #&2^         0.6063125
    9702 ##:k-        0.6063125
    9702 $Q-          0.6063125
    9702 $-q          0.6063125
    9702 $'u:6        0.6063125
   12448 #.           0.7779375
   12448 #&2;         0.7779375
   12448 #&PGn        0.7779375
   12448 #:k          0.7779375
```

Abbildung 5.2.: Kiel Korpus Datei

```
"*/g317a005.*.lab"
58750    200625         Klicken
58750    58750   ##
58750    200625         g
200625   516250         gh
516250   981250         u
981250   1337500        t
1337500         1510625         th
1510625         1510625         ##
1510625         1958125         U
1958125         2428125         m
2428125         2428125         ##
2428125         2812500         n
2812500         3550625         r
3550625         4263125         aI
4263125         4810625         t
4810625         5400625         s
5400625         5578125         @
5578125         6068750         n
6068750         6068750         ##
6068750         7785000         u6
```

Abbildung 5.3.: HTK Datei – Lautmodelle

Korpus–Format und das HTK–Format detailiert beschreiben, und dann die Umwandlung des Ersten in das Zweite. Das Kiel Korpus–Format werden wir anhand vom Beispiel (5.1.a) erklären. Jede Zeile enthält drei Spalten. In der ersten steht die Anzahl der *Samples*, die vom Aufnahmeanfang bis zu diesem Zeitpunkt im Sprachsignal gezählt wurden (in unserem Beispiel waren es **86**). Das Label selbst (**##g**) befindet sich in der zweiten Spalte. Schließlich registriert die dritte Spalte die Zeit in Sekunden (**0.0053125**) zwischen Aufnahmeanfang und dem jeweiligen Teil des Sprachsignals.

(5.1)				
		Samples	**Label**	**Dauer in ms**
	a)	86	##g	0.0053125
		313	$–h	0.0195000
		Anfang	**Ende**	**Label**
	b)	58750	200625	g
		200625	516250	gh

Das Ende des Sprachsignalteils, der mit dem Label gekennzeichnet wurde, kann man berechnen, wenn man die Dauer des entsprechenden Sprachsignalteils kennt. Für prosodische Ettiketierung und ähnliches ist die Dauer gleich null. Die Dauer ergibt sich aus dem absoluten Wert der Differenz dieses Sample–Wertes und dem nächst folgenden, der nicht gleich dem aktuellen Wert ist. In Abbildung 5.2 ist die Dauer von "**##g**" $227 = |86 - 313|$. Wir werden diese zwei Werte *Anfang_Samples_Kiel_Korpus* (hier 86) und *Ende_Samples_Kiel_Korpus* (hier 313) nennen.
Beispiel (5.1.b) zeigt das HTK–Format. Die ersten zwei Spalten spezifizieren die Zeitangaben für den Anfang und das Ende des Labels in 100–Nanosekunden–Einheiten. Das Label selbst steht in der dritten Spalte.

Die Umwandlung des Kiel Korpus–Formats in das HTK–Format geschieht in zwei Schritten. Zuerst werden die neuen Zeitangaben aus den alten abgeleitet, und dann werden die Labels angepasst. Für die HTK–Zeitangaben eines Labels werden die Anfang–Samples– und Ende–Samples–Werte benutzt, die dem Label im Kiel Korpus entsprechen. Da bei der Ableitung der spektralen Parameter ein Fenster von 16 ms benutzt wurde, in dem das Signal in der Mitte des Fensters am stärksten zu den Parameterwerten beigetragen hat (Hamming–Window), soll der Outputvektor von spektralen Parametern für die Mitte des Fensters, statt, wie üblicherweise in HTK, am Anfang ausgegeben werden. Um die Labeldatei zeitlich mit der Signaldatei zu koordinieren, wird eine Korrektur von 8 Samples (halbe Fensterbreite) vorgenommen. Die Formeln sind dann:

$$Label\_Anfang\_HTK = (Anfang\_Samples\_Kiel\_Korpus + 8) * 625$$
$$Label\_Ende\_HTK = (Ende\_Samples\_Kiel\_Korpus + 8) * 625$$

Die neu berechneten HTK–Zeitwerte für unser Beispiel sieht man in (5.1.b).

HTK verlangt nicht nur andere Zeitwerte, sondern auch andere Labels als das Kiel Korpus. Im Folgenden werden alle Änderungen inklusive Beispiele aufgelistet, die mittels Perl–Skripten durchgeführt wurden.

- Die Labels, die nur Geräusche repräsentieren (ohne Überlappung mit Sprachmaterial), werden durch die Geräusche selbst ersetzt.
  :k $\longrightarrow$ *Klicken* *s:*$\longrightarrow$ *Schmatzen*

- Geräusche, die sich mit Sprache oder mit vorherigen Geräuschen überlappen, z.B. *Klopfen* werden gelöscht (*##:k-* in Abbildung 5.2).

- Geräusche, die am Ende einer Aufnahme vorkommen, werden gelöscht (das letzte *Klicken* in Abbildung 5.2) .

- ## markiert Wortgrenzen, also werden Labels, die ## enthalten, in zwei Labels zerlegt. Die Dauer des Signalteils mit dem Label ## ist gleich null.

$$58750 \quad 200625 \quad \#\#g \quad \longrightarrow \quad \begin{array}{lll} 58750 & 58750 & \#\# \\ 58750 & 200625 & g \end{array}$$

- Wenn ein Label für einen Plosiv (p, t, k, b, d, g) von einem *-h* Label gefolgt wird (Lösungsphase des Plosivs), wird *-h* mit *Plosivh* ersetzt.

58750	200625	*g*	$\longrightarrow$	58750	200625	*g*
200625	516250	*-h*		200625	516250	*gh*

- weitere Änderungen, die genau in dieser Reihenfolge durchgeführt werden müssen:

```
(#|$)&...                                ⟶  nichts
a                                        ⟶  A
(a|e|i|o|u|y|2):                         ⟶  a|e|i|o|u|y|2
E:                                       ⟶  EE
#c:                                      ⟶  nichts
#(,|.|;|?|-MA|_)                         ⟶  nichts
(,|.|;|?|-MA|_)                          ⟶  nichts
$(-MA|=/+-|/-|/+|=/+|=/-|;|_|-~|=)       ⟶  nichts
+$                                       ⟶  nichts
$#                                       ⟶  $
$before–after                            ⟶  $after
#before–after                            ⟶  #after
(#|$)(″|′|”)label                        ⟶  label
(:k|l:|q:|r:|s:|w:|g:|q|v:|z:|n:)        ⟶  nichts
```

5.1.4. Verarbeitung der Labeldateien für die Worterkennung

Nach der Anpassung des Kiel Korpus, haben wir das für den Trainingsprozeß nötige Korpus bekommen. Dieses Korpus war allerdings dazu geeignet, HMMs für Lautmodelle zu trainieren. Da eine Überlegung war, HMMs für Wortmodelle zu trainieren, haben wir dafür das Kiel Korpus ein zweites mal angepasst. Die notwendigen Perl–Skripte befinden sich im Anhang H.

Anpassung des Kiel Korpus Variantenlexikons

Zuerst haben wir mittels des Variantenlexikons *kielcd4.lsv* ein Ausprachelexikon erstellt. kielcd4.lsv ist ein Variantenlexikon des bisher gesammelten spontansprachlichen Materials des Kiel Korpus.

(5.2)	a)	"ahnlich	Q'E:nlIC	Q-:q'E:nlIC	2	1	G124A002	8
	b)	"ahnlich		EEnlIC				

Jeder der Lexikoneinträge hat folgende Struktur: (siehe Beispiel (5.2.a)):

1. ortographische Repräsentation des lexikalischen Eintrags

2. kanonische Repräsentation des lexikalischen Eintrags

3. Transkription der Aussprachevariante
4. Auftretenshäufigkeit des lexikalischen Eintrags im Korpus
5. Auftretenshäufigkeit der Aussprachevariante
6. Platzierung des Eintrags im Dialog (Dialog– und Textreferenz)
7. Platzierung des Eintrags im Text

Ein Aussprachelexikon wurde erstellt, indem die ortographische Form (erste Spalte) und die Aussprachevariante (dritte Spalte) behalten wurden. Letztere wurde ähnlich wie oben beschrieben an das HTK Format angepasst. Wichtig ist, daß jede Variante nur einmal im ganzen Lexikon vorkommen darf. Wenn zwei Wörter dieselbe Variante haben, wird das Wort ins Aussprachelexikon aufgenommen, das alphabetisch zuerst aufgetreten ist. (5.2.b) zeigt den Eintrag von (5.2.a) nach der Anwendung unseres Skriptes. Die entprechende Syntax ist:

erzeugt_aussprache_lexikon.pl   kielcd4.lsv   aussprache_lexikon.txt

Anpassung der Labeldateien

Der zweite Schritt unseres Ansatzes war die Herstellung einer .mlf Datei für das Training von Wortmodellen. *create_master_file_lexicon.pl* nimmt als Input die .mlf Datei, die für Lautmodelle erzeugt wurde. Alle Labels, die zum selben Wort (Labels zwischen zwei Wortgrenzen) gehören, werden aneinander gereiht. So entsteht eine Aussprachevariante, die mit der ortographischen Form vom Aussprachelexikon ersetzt wird. Wortanfang ist der Anfang vom ersten und Wortende das Ende vom letzten Label. Geräusche werden als einzelne Wörter betrachtet. Ein Beispiel für das Output des Skriptes sieht man in Abbildung 5.4. Das Input dafür war die Datei in Abbildung 5.3. Die Syntax für diese Herstellung ist:

create_master_file_lexicon.pl   phondat.mlf   lex.mlf

```
"*/g317a005.*.lab"
58750   200625  Klicken
58750   1510625 gut
1510625 2428125 um
2428125 6068750 dreizehn
6068750 7785000 Uhr
.
```

Abbildung 5.4.: HTK Datei – Wortmodelle

5.2. Trainingsprozess

Das Ziel dieser Phase ist die eigentliche Erstellung der Lautmodelle. Das Training besteht aus drei Schritten — Festlegung der HMM–Konfiguration, Initialisierung und Re–estimation, die mittels mehrere Skripte (HCopy, HInit, HRest) durchgeführt werden (siehe Anhang F).

Festlegung der HMM–Konfiguration

Vor dem eigentlichen Training müssen einige Parameter gesetzt werden, die die HMMs vorläufig bestimmen. Wie schon in Abschnitt 5.1 erwähnt, legen wir ein left–to–right Modell zugrunde. Um eine Lautsequenz gut zu beschreiben, sollten desweiteren verschiedene HMMs verschiedene Anzahlen von Zuständen haben. Wir haben uns für fünf Zustände bei Diphtongen und drei Zustände bei allen anderen Lauten entschieden. Für jeden Zustand müssen beide Übergangswahrscheinlichkeiten (Zustand verlassen, im Zustand bleiben) zusammen 1 ergeben. Wir haben uns für eine Einteilung von je 0.5 entschieden. Um die gesamte Variabilität jedes Lautes möglichst gut zu beschreiben, wurden für jeden Zustand 16 mixtures festgelegt. Zusätzlich werden ein Anfangs– und ein Endzustand definiert, die jedoch nicht zur Modellierung des Lautes beitragen.

Initialisierung

Bei der Initialisierung werden, mit Hilfe des HTK–Programms HInit, aus jeder Signaldatei die Sequenzen an Sprachmaterial genommen, die einem bestimmten Label entsprechen. Das Label ist immer das gleiche (phonemische Label), während die entsprechenden Signalteile abhängig vom Kontext, in dem ein Laut gesprochen wurde, variieren. Beispielsweise wird [u] in Wörtern wie “Hut” und “Tuch” unterschiedlich ausgesprochen, aber mit dem gleichen Label erfasst (siehe Anhang I). Alle Signalsequenzen, die einem Label entsprechen, werden für die Erstellung eines HMMs benutzt. In einem HMM mit n Zuständen und einer Dauer von d in ms, beschreibt jeder Zustand genau d/n ms (*uniform segmentation*). D.h., bei einem [u] mit einer Dauer von 60 ms, entspricht jeder der drei Zustände (Transition vom vorherigen Laut zu [u], das [u] selbst, Transition von [u] zum nachfolgenden Laut) genau 20 ms (siehe Abbildung 5.5).

Re–estimation

Diese äquidistante Einteilung einer Lautsequenz wird der Variabilität eines Sprachlautes nicht gerecht. Die Transition von einem [d] zu einem [i:] verläuft beispielweise schneller als die von einem [d] zu einem [a:]. Im ersten Fall ist die Zunge bei beiden Lauten an benachbarten Stellen am Gaumen, während im zweiten Fall die beiden Laute an sehr unterschiedlichen Positionen im Mundraum artikuliert werden. Auch ist die Realisierung eines Lautes von Sprecher, Artikulationsgenauigkeit oder Kontext abhängig. Um sich also der artikulatorischen Realität anzunähern und, um gleichzeitig die Erkennungsrate zu verbessern, wird als zweiter Schritt die Re–estimation mit Hilfe des HTK–Programms HRest

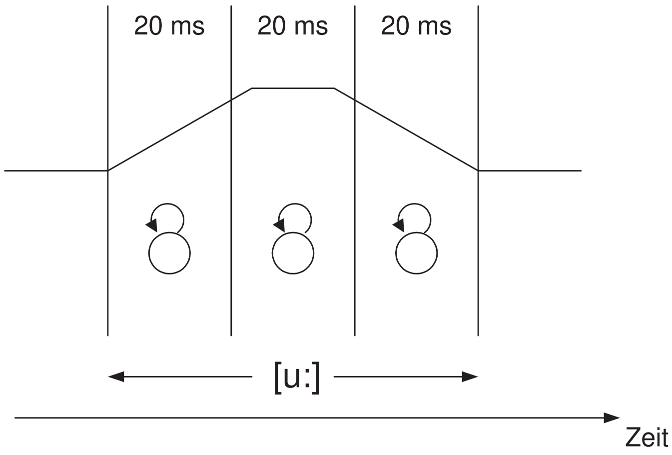

Abbildung 5.5.: Training: Initialisierung

durchgeführt. Hierbei wird die Wahrscheinlichkeit eines Zustandes ermittelt, und die Zustände werden dann gewichtet (siehe Abbildung 5.6). Re–estimation ermittelt mit Hilfe eines *vorwärts-rückwärts Algorithmus* (forward/backward oder Baum–Welsh algorithm), wie hoch die Wahrscheinlichkeit für einen bestimmten Zustand ist. Hierbei wird ein Zeitfenster durch das zu evaluierende HMM "geschoben" und für jeden zeitlichen Abschnitt anhand der ermittelten Parameterwerte bestimmt, in welchem der drei bzw. fünf Zustände sich das HMM am ehesten befindet. Eine solche Schätzung kann mehrfach wiederholt werden bis die Zustände als sehr wahrscheinlich und das HMM als trainiert gelten. Wenn der

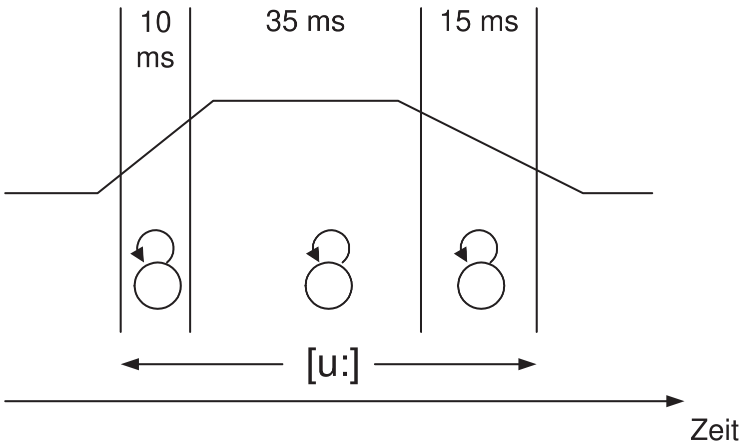

Abbildung 5.6.: Training: Re–estimation

Unterschied zwischen dem Input und Output einer Wiederholung unter einen bestimmten Schwellenwert fällt (Konvergenzbedingung), wird der Prozeß beendet. Damit das System nicht unendlich weiter läuft (für den Fall daß die Konvergenzbedingung nicht erfüllt wird)

wird eine maximale Anzahl der Wiederholungen festgelegt. In unserem Fall ist sie 20. Wenn sie erreicht wird, ist die Re–estimation abgeschlossen. Zu erwähnen ist noch, daß HInit sowohl die gelesenen als auch die spontansprachlichen Sprachdaten des Kiel Korpus verwendet, da soviel Material wie möglich benutzt werden sollte. Die gelesene Sprache dient zur groben Schätzung der Lautmodelle, mit den spontansprachlichen Daten sollen die Modelle in Richtung der späteren Benutzeräußerungen angepasst werden. HRest dagegen verwendet nur den spontansprachlichen Teil, da hier viele Reduktionsprozesse auftreten, genau wie später in den Benutzeräusserungen im Fahrstuhl.

Der Trainingsprozeß liefert also die trainierten HMMs, die zusammen mit einem Lexikon und einer Grammatik die Grundlage des Erkennungsprozesses bilden.

5.3. Lexikon, Grammatik

Ausser der Lautmodelle benötigt der Testprozess ein Lexikon und eine Grammatik. Das Lexikon (siehe Anhang J) wurde manuell erstellt. Ein Lexikonausschnitt wird in Abbildung 5.7 dargestellt. !PHONE-SET bezeichnet die Menge aller trainierten Lautmodelle, inklu-

```
!PHONE-SET               hmm_p hmm_t hmm_k hmm_b hmm_d hmm_g hmm_Q hmm_f ...
 ...  hmm_U hmm_U6 hmm_aI hmm_aU hmm_OY hmm_@ hmm_6 hmm_Pause  hmm_Atmung
 hmm_Schmatzen  hmm_Raeuspern hmm_Schlucken hmm_Lachen hmm_Klicken
 hmm_Klopfen  hmm_Rascheln hmm_Geraeusch

!SIL            []       hmm_Pause
!ANY            []       hmm_p
...
fahrstuhl       [F]      hmm_f hmm_a6 hmm_S hmm_t hmm_u hmm_l
fahrstuhl       [F]      hmm_f hmm_a6 hmm_C hmm_t hmm_u hmm_l
...
#"Erst(er)"
1               [1]      hmm_Q hmm_E6 hmm_s hmm_t hmm_6
...
#"Bitte"
bitte           []       hmm_b hmm_I hmm_t hmm_@
```

Abbildung 5.7.: Lexikonausschnitt

sive verschiedene Geräusche. Für den Rest der Einträge gilt, daß die erste Spalte das zu erkennende Wort und die dritte Spalte seine entsprechende Lautsequenz enthält, während die zweite Spalte das Output spezifiziert, welches im Fall der Erkennung an das Dialogmodul weitergegeben wird. Jedes Wort wird mit mehreren Aussprachevarianten aufgelistet.

Wörter, die im Dialog eine Funktion erfüllen (Schlüsselwörter), bilden den grössten Teil der Lexikoneinträge. Wörter wie "bitte" werden aufgrund ihrer Häufigkeit als Zielwörter erkannt, werden jedoch zur Zeit noch nicht im Dialog berücksichtigt (die zweite Spalte bleibt leer). !ANY und !SIL sind das garbage-model des Spracherkenners, d.h. wenn keine Wörter erkannt werden können, wird irgendetwas erkannt. Auf Basis des eigenen Sprachgefühls wurden mögliche Äußerungen der Fahrstuhlbenutzer aufgelistet. Daraus wurde eine Grammatik extrahiert, die im Anhang K definiert ist.

6. Steuerung

Im folgenden Abschnitt soll ein Überblick über die Aufzugsteuerung sowie dem dazugehörigen Treiber, welcher die Kommunikation zwischen Dialogsystem und Steuerung realisiert, dargestellt werden. Als Treiber ist ein Programm zu verstehen, welches in direkter Kommunikation zu einer entsprechenden Hardwareeinheit steht und anderen Programmen eine abstrakte Kommunikationsschnittstelle zu dieser Hardware zur Verfügung stellt.
Die hier beschriebene Lösung versteht sich im augenblicklichen Entwicklungsstand als eine Spezialloesung, die, im Gegensatz zu dem Dialogsystem, nicht modular aufgebaut ist und sich nicht direkt auf Aufzugsteuerungen anderer Hersteller übertragen lässt.
Im Rahmen dieses Projektes beschränkt sich die Aufgabe des Treibers auf die Möglichkeit den jeweiligen Status des Aufzugs an das Dialogsystem weiterzuleiten, sowie einen Tastendruck auf dem Bedienfeld des Fahrkorbes zu simulieren. Weitere Funktionen können bei Bedarf implementiert werden.

6.1. Aufzugsteuerung und Anschluß

Bei der verwendeten Aufzugsteuerung handelt es sich um das von der Firma *NewLift* hergestelle modell „*KST*". Dass dieses Modell bereits eine Schnittstelle zum Anschluß an einen PC, bzw. ein Modem verfügt, war zum einen sehr hilfreich, da somit keine eigene Schittstelle entwickelt werden musste. Zum anderen mussten aber damit auch einige Einschränkungen in Kauf genommen werden. Genannt seinen hier Limitierungen der maximalen Kabellaenge zwischen Rechner und Steuerung, sowie Probleme, die das Design des Protokolls mitbrachte (siehe Abschnitt „Übertragungsprotokoll").
Die Verbindung zwischen Steuerung und Rechner wird über eine serielle Schnittstelle (RS232) hergestellt. Diese Schnittestelle kann in verschiedenen Modi betrieben werden, wird jedoch von Seiten der KST bereits fest in einen bestimmten Modus geschaltet, bei dem nur die folgenden, minimal notwendigen Leitungen belegt sind:

- *TX*: ausgehendes Signal
- *RX*: eingehendes Signal
- *GROUND*: Signalmasse

Damit ergibt sich trivialerweise für eine Verbindung zwischen Steuerung und Rechner folgende Pin-Belegung der D-Sub-Stecker wie sie in Abbildung 6.1 dargestellt wird.

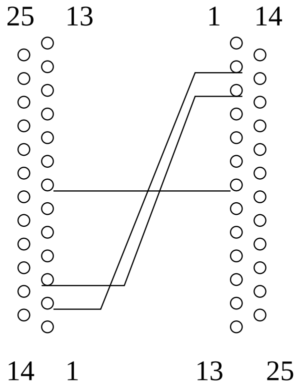

Abbildung 6.1.: DSub-25 auf DSub-25

Um nun das oben angesprochene Problem der auf 15m limitierten Kabellänge zu bewältigen und die benötigten 70m zu ueberwinden, wurde die vorhandene Netzwerkverkabelung genutzt. Dabei handelt es sich um hochwertige, abgeschirmte Twisted-Pair-Kabel, die jeweis zwei verdrillte Leitungen zum Senden und Empfangen von Daten bereitstellen. Der Vorteil der verdrillten Drähte der Kabel, eine Reduzierung der entstehenden Störungeni, kann jedoch nicht genutzt werden. Hierfür müssten die Signale differenziell übertragen werden, die serielle Schnittstelle übertraegt ihre Signale jedoch als absolute Werte. Dennoch reicht Messungen zufolge die zusätzliche elektrische äußere Abschirmung der Kabel aus, um eine störungsfreie Datenübertragung zu gewährleisten.
Abbildung 6.2 zeigt nun die erweiterte Darstellung der Pin-Belegung der D-Sub-Stecker sowie die Belegung der für die Twisted-Pair-Leitungen verwendeten RJ-45-stecker.

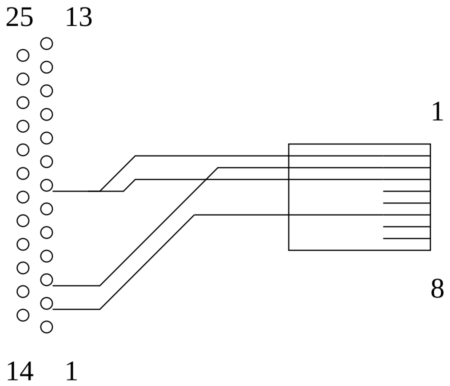

Abbildung 6.2.: DSub-25 auf RJ-45

6.2. Übertragungsprotokolle

Dieser Abschnitt soll einen knappen Überblick über die beiden verwendeten Protokollen vermitteln, die beide unter dem Namen *NLF401* (NewLiftFormat 401) zusammengefasst sind. Da die Firma NewLift selbst eine umfassende Dokumentation zu diesen Protokollen zur Verfügung stellt, soll hier nur insoweit auf wichtige Details eingegangen werden, wie

sie für die Implementierung notwendig sind.
Bei den beiden Protokollen handelt es sich ein Transportprotokoll und um ein sogenanntes Application-Protokoll. Als Problem stellte sich heraus, dass diese beiden Protokolle an einigen Stellen miteinander verflochten sind, was eine sinnvolle modulare Implementierung leider nicht möglich gemacht hat.

6.2.1. Transportprotokoll

Bei dem Transportprotokoll handelt es sich um ein einfaches sogenanntes *Bouncing-Protokoll*, daß jeweils ein Packet mit der Groeße von einem Byte versenden kann. Als Bouncing-Protokoll bezeichnet man ein Protokoll, bei dem die beiden Kommunikationsendpunkte jeweils abwechselnd als Sender, bzw. Empfänger aggieren. Ein Endpunkt erwartet also nach jedem gesendeten Datenpacket eine Antwort des anderen Endpunktes, bevor das nächste Packet gesendet wird. Diese Eigenart wird u.a. dazu benutzt, um den Ausfall eines Endpunktes zu entdecken.
Zusätzlich verfügt das hier verwendete Transportprotokoll übeer eine Fehlererkennung und auch „-korrektur", die durch das zweimalige Senden und Vergleichen jedes Datenpacketes erreicht wird. Hierfür muß allerdings eine Einbuße der Übertragungsgeschwindigkeit um Faktor zwei in Kauf genommen werden. Dies kann jedoch u.U. bei häufigen und zeitkritischen Anfragen an die Steuerung zu Problemen führen.
Um die Funktionsweise des Transportprotokolls zu verdeutlichen, wird das Versenden des Bytes A anhand der Abbildung 6.4 demonstriert. Abbildung 6.3 zeigt den Aufbau eines Datenpacketes. Zu erkennen ist, daß die obere Hälfte (Bits 4 bis 7) dazu dienen, um Statusinformationen auf Protokollebene zu übertragen. Die unteren vier Bits dienen als Container für die zu übertragenden Nutzdaten.

Fail	Empt	Nib2	Rept	D3	D2	D1	D0

Fail: ... Fehler im letzten Packet
Empt: ... Container ist leer (ist also nicht Null)
Nilb2: ... sendet Bits 0 bis 3
Rept: ... wiederholtes Senden

Abbildung 6.3.: DSub-25 auf RJ-45

Das Transportprotokoll ist also nur zum Transportieren beliebiger Datenpackete gedacht und enthält keinerlei Kommandos (z.B. Fahrbefehle), die von der Steuerung interpretiert werden müssen.

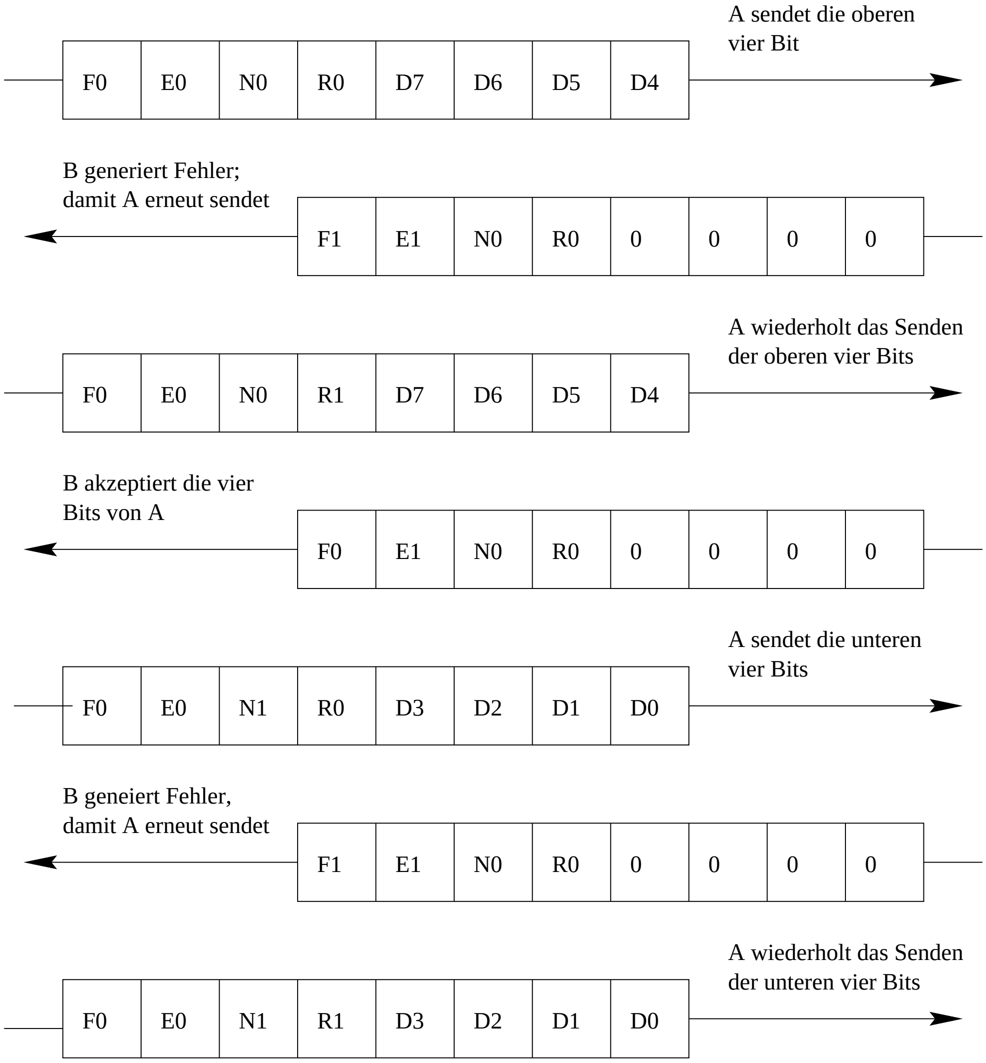

Abbildung 6.4.: DSub-25 auf RJ-45

6.2.2. Application-Protokoll

Das Application-Protokoll enthält nun die eigentlichen Kommandos an die Steuerung. Hierzu wird wiederum zwischen drei verschiedenen Protokollen unterschieden:

- *Normal 401 Message*
- *Free-Format 401 Message*
- *Extended Free-Format 401 Message*

Wie schon an den Bezeichnungen zu erkennen, untescheiden sich diese Protokolle vor allem in der Größe der ihrer Datenpackete. Weiter soll auf sie allerdings nicht eingegangen werden, da nur wenige Informationen über die möglichen Befehle zur Verfügung stehen und die Protokolle somit auch nicht modular implementiert wurden.

6.3. Implementierung

Primäres Ziel der Implementierung des Treibers war ein möglichst Platformunabhängiger und einfacher *C*-Code, um das Programm möglichst einfach auf alle von *OZ* unterstützten Systeme zu portieren. Es wurden daher nur *POSIX*-kompatible Systemfunktionen und -bibliotheken benutzt, da diese von den meisten *UN*X*-Deviraten, sowie Windows-NT unterstützt werden.
Der Treiber kann wie folgt in zwei logische, parallel-laufende Unterprozesse (sogenannte *Threads*) unterteilt werden:

- *Interface-Thread*
- *Communication-Thread*

6.3.1. Interface-Thread

Der Interface-Thread, der aus dem eigentlichen ursprünglichen Hauptprozess hervor geht, ist nach der Initialisierung der seriellen Schnittstelle für die Kommunikation mit dem Benutzer oder einem weiteren Programm zuständig.

6.3.2. Communication-Thread

Über den Communication-Thread läuft parallel zum Interface-Thread die Kommunikation zwischen Treiber und Steuerung. Da die Kommunikation laut der Spezifikation des Transportprotokolls ununterbrochen aufrecht erhalten werden muss, wurde die entsprechende Funktion in einen eigenen Thread ausgelagert, um ein Blockieren durch die Ein-/Ausgabe-Funktionen des Interfaces zu vermeiden.

6.3.3. Thread-Kommunikation

Die Kommunikation zwischen dem Communication- und dem Interface-Thread wird durch zwei Warteschlangen sowie sogenannte Semaphoren realisiert. Die Warteschlagen stellen hierbei einen Puffer zwischen den beiden Threads dar und transportieren zum einen ausgehende Daten vom Interface- zum Communication-Thread (*nlf_ queue.tx*) und zum anderen eingehende Daten in die umgekehrte Richtung (*nlf_ queue.rx*). Die Semaphoren (Statusvariablen, bzw. oft auch als „Ampeln" bezeichnet) werden dazu benutzt, um den Status der Warteschlagen anzuzeigen und das Absenden von Daten im Communication-Thread, bzw. das Ausgeben von Daten im Interface-Thread zu steuern;

6.4. Steuerkommandos

6.5. Interface

Das Interface des Treibers ist durch eine interaktive Kommandozeile realisiert. In seiner bisherigen Implementierung werden die beiden folgenden Kommandos unterstützt:

- *info()*
  gibt die aktuelle Position, Fahrtrichtung und den Status der Innen- und Außentür zurück.

- *ziel(n)*
  setzt ein Fahrkommando zum n-ten Stockwerk.

Beispiel:

```
[user /] # nlf401
info()
info(kabinentuer:auf drehtuer:zu pos:1 ziel:255 richtung:keine)
ziel(5)
ok()
info()
info(kabinentuer:zu drehtuer:zu pos:5 ziel:5 richtung:keine)
```

7. Soundkarten

7.1. Aufgaben des Teilprojektes

Da ursprünglich für den sprechenden Fahrstuhl eine Sprechstelle pro Stockwerk vorgesehen war und nicht nur eine in der Kabine, befasse ich mich mit der Auswahl von Mehrkanal-„Soundkarten" und ihrer Einbindung in das Gesamtsystem der Fahrstuhlsteuerung. Den Begriff „Soundkarte" verwende ich dabei als Oberbegriff für eine Vorrichtung, um Sprachsignale oder sonstige Töne mit einem PC zu verarbeiten. Es muss sich dabei nicht um ein einzelnes Gerät handeln, es ist beispielsweise auch denkbar, jede Sprechstelle mit einem eigenen Wandler auszustatten, der die Sprachsignale vor Ort verarbeitet und an ein Computernetzwerk (zum Beispiel Ethernet) angeschlossen ist.

In den folgenden Abschnitten meine ich mit Soundkarte nicht mehr den Oberbegriff, sondern eine Baugruppe, die in einen PC eingebaut wird, um ihn in die Lage zu versetzen, Töne zu verarbeiten. Meistens hat eine Soundkarte direkt Anschlüsse für Lautsprecher / Mikrofon etc., manchmal befinden sich diese aber auch in einer speziellen Box, die an die Soundkarte angeschlossen wird. So werden die Anschlüsse besser erreichbar und man kann den Einfluss von Störsignalen reduzieren, die im Innern des PC vorhanden sind.

Um ein „Tonverarbeitungsgerät" in Verbindung mit einem PC einsetzen zu können, braucht man immer auch einen passenden Treiber , der eine geräteunabhängige Schnittstelle zu den auf dem PC laufenden Programmen zur Verfügung stellt. Das geschieht zum Beispiel über eine simulierte Datei, aus der die empfangenen Töne wie aus einer Audiodatei gelesen oder zu sendende Töne geschrieben werden können, oder über einen festen Satz von Funktionen wie „spiele diesen Datensatz als Ton ab", die dann von den anderen Programmen aufgerufen werden.

Für unsere Anwendung benötigen wir insgesamt 7 Sprechstellen (1 pro Stockwerk und 1 in der Kabine) für die volle Ausbaustufe. Da sich bald zeigte, dass eine Sprechstelle in jedem Stockwerk das ganze Projekt sehr komplex machen würde, wurden nur wenige praktische Experimente in dieser Richtung durchgeführt. Ich beschreibe hier also Möglichkeiten, die erst in einer weiteren Ausbaustufe auch wirklich zum Einsatz kommen werden.

Die Sprachsignale müssen letztlich von Software zur Spracherkennung analysiert werden, also ist es wichtig, auf die Schnittstelle zur verwendeten Software zu achten. Zuerst wurde das Paket HTK (Hidden Markov Toolkit) von Entropic unter Linux (x86) eingesetzt, später dann der Spracherkenner ASR 1600 M und das Toolkit dazu von Lernout & Haupsie unter Windows NT. HTK gibt es auch für Windows, der L & H Erkenner läuft auch mit Windows 95/98. Je nach verwendetem Spracherkenner und Betriebssystem muss jeweils ein passender Treiber vorhanden sein.

7.1.1. Grundsätzliche Probleme

Um die Komplexität des Projektes überschaubar zu halten, ist der Fahrstuhl bisher (Dezember 2000) noch auf 1 Sprechstelle in der Kabine beschränkt. Dadurch kommt im Programm weniger Nebenläufigkeit vor, ausserdem ist es technisch schwierig, alle Stockwerke so zu verkabeln, dass eine gute Tonqualität erreicht wird – die Akustik im Flur ist ohnehin für Spracherkennung nicht besonders gut.

Ein weiteres Problem ist die Aufmerksamkeitssteuerung: Wenn auch ausserhalb des Fahrstuhls die Steuerung nur durch Sprache möglich sein soll (ohne Rufknopf), muss ein Spracherkenner ständig bereit sein, trotz wechselnder Umgebungsgeräusche sofort beim Erkennen eines Schlüsselwortes („Fahrstuhl", „Aufzug", ...) anzusprechen und im Optimalfall gleich den Rest der Äusserung zu analysieren („Fahrstuhl, ich möchte in den 3. Stock"). Im Zweifelsfall müsste man für jede Sprechstelle ständig einen Spracherkenner laufen lassen. Dabei besteht die Gefahr, so viel Speicher oder Rechenleistung zu verbrauchen, dass die Verarbeitung auf mehrere Computer verteilt werden muss.

Das Problem lässt sich entschärfen, indem man wie bei einer Sprechanlage das jeweilige Mikrofon nur einschaltet, solange der Rufknopf gedrückt ist. Damit laufen die Spracherkenner nur nach Bedarf und müssen nicht dauernd Nebengeräusche analysieren, während ohnehin kein Dialog stattfindet.

Eine elegantere Lösung arbeitet mit einem Schlüsselwort, zum Beispiel „Fahrstuhl". Nur dieses Wort startet einen Dialog. Schlüsselwort und Rufknopf lassen sich kombinieren: Wird das Wort nicht genannt, kommt einfach der Aufzug (Rufknopf im alten Sinn), sonst wird ein Dialog begonnen (Rufknopf wie bei einer Sprechanlage). Ein Spracherkenner muss bei Verwendung eines Schlüsselwortes nicht versuchen, alle in der Nähe des Mikrofons gesprochenen Worte zu erkennen und entscheiden, ob die Worte an den Fahrstuhl gerichtet waren. Das Schlüsselwort kann auch leichter erkannt werden, da nur eine Ja/Nein-Entscheidung gebraucht wird und nicht mehrere vielleicht ähnliche Worte aus einem grossen Wortschatz in Betracht gezogen werden müssen. Man kann weiterhin vom Benutzer fordern, nachdem er das Schlüsselwort gesagt hat, erst auf eine Bestätigung zu warten:

Fahrstuhl
Wohin möchten Sie bitte?
Ich möchte in den 3. Stock

Wenn mehr Resourcen (Speicher, Rechenleistung) verbraucht werden als ein einzelner Rechner anbietet, aus anderem Grund nicht genug Spracherkenner auf einem einzelnen Rechner laufen, oder einfach die Soundkarten oder sonstigen Geräte auf mehr als einen Rechner verteilt sind, müssen Sprachdaten in einem Rechnernetzwerk übertragen werden. Dabei kann es sich um unverarbeitete Mikrofonsignale oder schon um Ergebnisse der Spracherkennung handeln. Entsprechende Treiber müssen die Spracherkenner dann mit dem Netzwerk verbinden, da die Spracherkenner selten bereits mit einer Option „Mikrofon via Netzwerk abfragen" oder „Ergebnisse ans Netzwerk schicken" ausgestattet sind.

Ich befasse mich in Abschnitt 7.4.3 mit dem Problem, HTK unter Linux via Netzwerk Mikrofondaten empfangen zu lassen. Da sowohl bei HTK als auch beim ASR 1600 eine Schnittstelle für eigene Treiber vorgesehen ist, ist es zumindest prinzipiell auch unter Windows denkbar, Mikrofondaten im Netzwerk zu verschicken. Unter Linux ist allerdings der Sound-Treiber sehr gut dokumentiert. Auch sein Programmtext wird mitgeliefert, es ist also sogar denkbar, den Treiber direkt in zwei Teile aufzuspalten, die per Netzwerk miteinander kommunizieren – die Spracherkenner müssen dann gar nicht angepasst werden, obwohl die Soundkarte sich nicht im selben Rechner befindet. Diesen Ansatz habe ich nicht weiter verfolgt, da er umfangreiche Veränderungen am Soundtreiber erfordert.

7.2. Lösungsansätze

7.2.1. Verarbeitung analog und zentral

Dieser Lösungsansatz geht davon aus, dass die Mikrofone mit normaler Tontechnik bedient werden und die Umwandlung in digitale Computerdaten erst direkt in dem Rechner erfolgt, in dem die Spracherkennungssoftware läuft (oder direkt an diesen angeschlossener Hardware). Ausreichend Rechenleistung in diesem einen Rechner wird vorausgesetzt.

Für 6 Stockwerke plus die Kabine sind 7 Eingänge und 7 Ausgänge nötig, damit in keinem Fall die Resourcen ausgehen können. Eine typische Wahl wäre damit eine Soundkarte mit 8 Eingängen und 8 Ausgängen, wie sie in semiprofessionellen Tonstudios und für Harddisk-Recording (ein PC ersetzt ein mehrspuriges Tonband im Studio) verwendet wird.

Soweit genügend Steckplätze zur Verfügung stehen, lassen sich zumindest bei modernen PCI-Soundkarten auch einfach mehrere einfache Karten gleichzeitig in einen PC einbauen und verwenden.

Im Abschnitt 7.3 „Verfügbare Soundkarten” werden verschiedene Produkte verglichen und besprochen, zunächst führe ich aber noch einige grundsätzlich andere Ansätze auf.

7.2.2. Die Sprachdaten kommen digital an einem Rechner an

Für weitere Skalierbarkeit auf mehr Sprechstellen ist es denkbar, mehrere Rechner mit Soundkarten zu versehen und die aufgenommene Sprache per Netzwerk weiterzureichen. Umgekehrt kann die Verwendung eines Computernetzes auch ermöglichen, den Resourcenverbrauch für die Spracherkennung auf mehrere Rechner zu verteilen.

Eine eher dezentrale Lösung sieht vor, direkt bei den Mikrofonen das Tonsignal zu digitalisieren und über Datenkabel zum Auswerterechner zu transportieren. Das verkompliziert die einzelnen Sprechstellen, vereinfacht aber je nach Konstruktion der Sprechstellen die Hardware am Auswerterechner und ist weniger störanfällig als ein analoges Signal. Gerade in der Nähe des Fahrstuhles und von Computern treten verstärkt elektromagnetische Störungen auf.

Die Schwierigkeit bei den „digitalen Sprechstellen” liegt in der Umwandlung zwischen analoger Sprache und digitalen Daten im Netz.

Wegen der geringeren Anforderungen an die Tonqualität (der Mensch versteht leichter den Lift als umgekehrt) sollte man überlegen, zumindest die Sprachausgabe weiterhin analog im Rechner (oder den Rechnern) zu erzeugen, was die Sprechstellen vereinfacht und das Netz entlastet.

Eine interessante Möglichkeit für die Digitalisierung der Mikrofonsignale ist die Verwendung von sogenannten GSM-Encodern (in Form kleiner Spezialcomputer, solche kann man fertig kaufen)

, die die aufgenommene Sprache mit einer auf Sprache optimierten Methode komprimieren und dann über seriellen Schnittstellen übertragen. Dabei fallen weniger als 14000 bit/s an, was eine robuste Signalübertragung ohne besondere Hardware erlaubt. Eine Erklärung des GSM Algorithmus findet sich unter `http://kbs.cs.tu-berlin.de/%7Ejutta/toast.html`

Die GSM-Kompression wird auch für Handies verwendet, die Sprachqualität ist nach der Kompression also nur noch so gut wie wenn der Fahrstuhl mit den Sprechstellen per Handy sprechen würde.

Wenn die Spracherkennung auf einem einzelnen PC laufen soll, benötigt dieser dann ausser mehrerer serieller Schnittstellen keine besondere Hardware. Entsprechende Schnittstellenkarten sind verfügbar und ausgereift, früher wurden sie gerne bei Netzwerkanbietern verwendet, um mehrere Modems an einen Rechner anzuschliessen. Treiber sind für Linux, Windows, . . . vorhanden.

Die grössere Schwierigkeit liegt bei der Spracherkennungssoftware, die oft GSM (noch dazu von einer seriellen Schnittstelle) nicht als Eingabeformat vorsieht. Gerade HTK ist hier aber erweiterbar: Man kann mit (ggf. selber zu schreibenden) Zusatzmodulen Audiodaten aus einem Datenstrom einlesen, den man wiederum durch Entkomprimierung der GSM-Daten (Rückumwandlung GSM in normale Audiodaten) zum Beispiel durch das (kostenlose) Unix-Tool sox gewinnt. Sox ist ein Programm, welches verschiedene Audio-Datenformate ineinander konvertiert, zum Beispiel GSM, WAV und Rohdaten zum Austausch mit Treibern über /dev/dsp oder /dev/audio.

Für höhere Sprachqualität sollte weniger stark oder gar nicht komprimiert werden. Dazu ist allerdings spezielle Hardware nötig, zum Beispiel der Einsatz digitaler Signalprozessoren (DSP) die das Signal digitalisieren, komprimieren und dann in einem geeigneten Protokoll digital weiterleiten.

Unkomprimierte digitale Daten bekommt man bereits als paralleles oder serielles Signal aus relativ preiswerten ADC (Analog Digital Konverter) Bausteinen, das hohe Datenaufkommen und das Fehlen eines „intelligenten" Protokolls fordern aber gut abgeschirmte Kabel und eine spezielle Verarbeitung im Zielrechner.

Bei Verwendung eines echtzeitfähigen Treibers (oft reicht ein Linux-Kerneltreiber, oder man verwendet DOS oder ein spezielles Betriebssystem für Echtzeitverarbeitung – Windows ist weniger geeignet) kann dazu eine einfache PIO (Parallel Input Output) Steckkarte genügen, wobei der PC die seriellen Signale aller Sprechstellen unter relativ hohem Rechenaufwand wieder zusammensetzt. Eine sinnvollere Methode ist die Verwendung mehrerer serieller Schnittstellen im PC, die Geschwindigkeit und Datenformat bewältigen (115200 kbit/s reichen nur für etwa 8 kHz Samplingrate , also Amplitudenmessungen pro Sekunde, bei einer Genauigkeit von 12 Bit, was gerade noch etwa Telefonqualität ist).

Noch höhere Datenraten lassen sich mit speziellen seriellen Schnittstellenbausteinen oder einer DSP Steckkarten (diesmal zum Dekodieren der seriellen Daten) am empfangenden PC erreichen. Den DSP zu programmieren ist aber wegen der Forderung nach paralleler Verarbeitung in Echtzeit eine schwierige Aufgabe, die Spezialwissen erfordert.

Für die digitale Übertragung in hoher Qualität ist es also ab einem bestimmten Punkt sinnvoller, die Sprechstellen selber mit Computern oder DSPs auszustatten. Diese könnten dann z.B. über normales Ethernet kommunizieren, gegebenenfalls sogar über ein einzelnes BNC-Kabel (nur bis 10 Mbit/s). Damit muss man nicht mehr getrennte Kabel für jede Sprechstelle verlegen, sofern sowohl die Aufnahme als auch die Wiedergabe digitalisiert werden. Gerade bei BNC muss allerdings die Netzwerkkapazität beachtet werden:

Das Datenaufkommen im Netzwerk ist bei heute verbreitetem Fast Ethernet (100 Mbit/s) kein Problem: Bei 44100 Amplitudenmessungen mit 16 Bit Genauigkeit fallen 0.7 Mbit/s pro Kanal an. Für 7 Kanäle ergibt das 5 Mbit/s, wenn man also auch noch die Dinge die der Lift sagt per Netz übertragen will, droht bei 10 Mbit/s Ethernet ein „Datenstau".

Diese Betrachtungen sollten aber als „worst case" angesehen werden, da sie von Mono CD-Qualität ausgehen und vom vollständig gleichzeitigen Betrieb aller Sprechstellen. Zumindest für die Äusserungen des Fahrstuhls sollte sogar bei 8 Bit Genauigkeit und einer Samplingrate (Amplitudenangaben pro Sekunde) von 22050 Hz noch eine angenehme Tonqualität erreicht werden. Für die Spracherkennung ist jedoch 16 Bit Genauigkeit wünschenswert, da das Signal/Rausch-Verhältnis (SNR) verbessert wird und noch Reserven vorhanden sind um schlecht ausgesteuerte Signale zu verarbeiten. Die Lautstärke kann dann noch per Software korrigiert werden. Ein Spracherkenner kann bei entsprechendem Design sogar mit 8000 Hz Samplingrate und 8 Bit Genauigkeit arbeiten, da dies aber nur noch mässige Telefonqualität ist, sind 22050 Hz und 16 Bit (Qualität etwa wie Musik-Kassette oder UKW-Radio) für zuverlässige Erkennung sinnvoller.

7.2.3. Verwendung mehrerer Rechner: Echte Skalierbarkeit

Wie im letzten Abschnitt beschrieben, ist es möglich und für grössere Anlagen sinnvoll, die Sprachverarbeitung auf mehrere Rechner zu verteilen.

Dabei kann man entweder die Sprachsignale direkt an mehreren Stellen empfangen – jeder Rechner hat eine eigene Soundkarte oder sonstige Empfangsmöglichkeit für genau so viele Kanäle wie er analysiert – oder die Sprachsignale digital von zentraler Stelle aus verteilen, wobei dann aber ggf. die Kapazität der Netzwerkverbindung entsprechend vorhanden sein muss.

Bei der digitalen Verteilung kann man die Sprachdaten vor dem Verteilen komprimieren. Dazu bieten sich z.B. der GSM-Algorithmus aus dem Handy-Bereich oder MP3 (mpeg audio layer 3) an, welches versucht, für Menschen unhörbare Klanganteile wegzulassen und das deshalb gerne für die Kompression von Musik ohne hörbaren Qualitätsverlust verwendet wird.

Dabei ist aber mit deutlichem Verbrauch an Rechenleistung zu rechnen, da GSM eigentlich besonders für spezielle Signalprozessoren mit extra auf GSM abgestimmten Recheneinheiten geeignet ist und bei MP3 die hörbare Qualität und nicht der Verbrauch an

Rechenleistung vorrangiges Entwicklungsziel war.

Sobald der Resourcenverbrauch für die Kompression in der Grössenordnung dessen liegt, was die Spracherkennung an Rechnerkapazität benötigt, wird die Kompression jedoch nutzlos. Wie oben gesagt, ist das Datenaufkommen ohnehin zumindest bei Verwendung von Fast Ethernet selbst für dutzende von Tonkanälen auch ohne Kompression noch akzeptabel.

Für besonders grosse Anlagen ist es trotzdem sinnvoller, jedem Rechner eine eigene Soundkarte einzubauen bzw. bei digitalen Sprechstellen jeweils wenige Sprechstellen direkt mit einem Rechner zu verbinden statt über ein gemeinsames Netz. Über das gemeinsame Netz laufen dann nur noch Daten der Art von „An Sprechstelle 42 wurde das Wort FÜNF mit einer Wahrscheinlichkeit von 80 Prozent gehört” – das bedeutet minimale Anforderungen an das gemeinsame Netz selbst bei riesigen Systemen.

7.2.4. Mehrfachnutzung von Resourcen: scheinbare Skalierbarkeit

Ein alternativer Ansatz besteht darin, weniger Kanäle als Sprechstellen komplett auszubauen. Zum Beispiel würde man überall Mikrofone installieren, die Soundkarte am Rechner hätte aber zu wenig Anschlüsse. Da es sehr unwahrscheinlich ist, dass an vielen Sprechstellen absolut gleichzeitig gesprochen wird, lässt sich eine einfachere Soundkarte ohne im Normalfall feststellbare Beeinträchtigungen verwenden.

Bei diesem Ansatz benötigt man als Zusatzhardware einen elektronischen Umschalter, der die vorhandenen Soundkartenanschlüsse computergesteuert den jeweils aktiven Sprechstellen zuteilt. So ein Gerät lässt sich einfach und preiswert konstruieren und beispielsweise über die Druckerschnittstelle ansteuern.

Welche Anforderungen an so eine Hardware gestellt werden und wie sie aufgebaut und angesteuert werden kann, wird unter 7.3.4 recht ausführlich beschrieben!

Ich schätze für unsere Anwendung genügen 2-4 Eingänge und 2-4 Ausgänge: Eine typische PC-Soundkarte, zum Beispiel ein SoundBlaster hat 2 Eingänge und 2 bis 4 Ausgänge die völlig unabhängig verwendet werden können. Weitere Anschlüsse sind oft über einen Mixer verbunden und können nicht eigenständig eingesetzt werden.

Bei Verwendung einer einzelnen Standardsoundkarte müsste der Fahrstuhl also erst dann Benutzer warten lassen, wenn er bereits an 4 Stellen absolut gleichzeitig spricht. Angesichts der Kürze typischer Fahrstuhläusserungen wie „Ich fahre Sie jetzt ins Untergeschoss” ist dieser Fall extrem unwahrscheinlich.

Das grössere Problem beim Einsatz einer sehr einfachen Soundkarte ist die Spracherkennung: Das Mikrofon in der Kabine sollte zumindest bei stillstehendem Aufzug ständig aktiv sein, wodurch für alle Sprechstellen auf den Stockwerken zeitweise nur noch ein Kanal übrig bleibt.

Falls jedoch mindestens 4 Eingänge und 4 Ausgänge vorhanden sind, ist durch Mehrfachnutzung die Versorgung von etwa 8 oder 12 Sprechstellen resourcensparend möglich.

7.2.5. Den Aufwand verteilen: Hardware direkt in den Sprechstellen

In diesem Abschnitt beschreibe ich verschiedene Wege, die Verarbeitung dezentral vorzunehmen. Die Verwendung von Ruf- oder Sprechknöpfen und Hardware zum Erkennen wenigstens einiger Stichwörter direkt an den Sprechstellen reduziert sogar bei ansonsten zentraler Verarbeitung der Sprachdaten die Anforderungen an den Zentralrechner deutlich!

Ein einfacher, aber uneleganter Ausweg ist die Verwendung von Rufknöpfen – Der Fahrstuhl öffnet nur für die Mikrofone einen Kanal, an denen ein Benutzer den Sprechknopf gedrückt hält. Es ist dann sogar möglich, falls kein Kanal mehr frei ist, den Benutzer um Geduld zu bitten (per Sprachausgabe oder indem eine „Sprechen" Lampe erst aufleuchtet, wenn nach wenigen Sekunden wieder ein Kanal freigeworden ist). Alternativ kann der Fahrstuhl wie bei einem klassischen Aufzug einfach zu dem Stockwerk kommen, in dem der Knopf gedrückt wurde, sofern keine Kapazitäten für einen gesprochenen Dialog mehr frei sind. Der Benutzer würde dann erst *im* Fahrstuhl das gewünschte Ziel nennen.

Der Einsatz von „intelligenten Sprechstellen" kann die Spracherkennung hier wesentlich vereinfachen: Die Firma Sensory (`http://www.voiceactivation.com`) bietet für ca. 100 DM mit dem VoiceDirect Modul ein vorprogrammiertes DSP-Modul an, welches an ein analoges Sprachsignal oder Mikrofon angeschlossen direkt bis zu 15 verschiedene Phrasen von jeweils bis zu 3 Sekunden Länge erkennt. Das Modul verwendet dazu ein neuronales Netz , also ein ähnlich wie die Hidden Markov Chains von HTK fehlertolerantes und einfach an Beispielen lehrbares System.

Die Anwendung so eines Moduls kann z.B. darin bestehen, dem Modul vor Ort Schlüsselwörter wie „Fahrstuhl" vorzusprechen (Vorteil: die Raumakustik wird dadurch gleich mit berücksichtigt), die es dann später erkennt und ein einfaches binäres Signal schickt. Der Wortschatz von 15 Phrasen reicht bei kleineren Gebäuden sogar zur Erkennung aller relevanten Wörter.

Bei grösseren Systemen eignet sich eine Spracherkennungshardware besonders zur Erkennung von „Aufwachbegriffen" : Die Hardware erkennt, dass ein Benutzer den Fahrstuhl sprechen möchte, das entsprechende digitale Signal geht an den Rechner mit der komplexen Spracherkennungssoftware, der daraufhin einen Tonkanal zu dem jeweiligen Mikrofon öffnet und ggf. einen Prozess der Spracherkennungssoftware startet. Dadurch laufen nicht unnötig ständig viele Prozesse komplexer Software nur um auf ein dialogeinleitendes Stichwort zu warten.

Wenn keine Spracherkennungshardware in jeder Sprechstelle zur Verfügung steht, kann man sich wie folgt behelfen: die zentrale Spracherkennung wird reihum mit den Sprechstellen verbunden an denen überhaupt ein Geräusch erkannt wurde. Für die Geräuscherkennung reicht eine einfache Hardware, oder man lässt das zentrale System jeweils für Sekundenbruchteile an jeder Sprechstelle den Geräuschpegel messen (was natürlich ausgesetzt werden muss, wenn gerade keine Eingangskanäle frei sind).

Eine noch einfachere Methode wären Zeitfenster, die reihum auf alle Sprechstellen verteilt werden: Alle z.B. 5 Sekunden „hört" der Fahrstuhl jeder Sprechstelle kurz zu (kann dem Benutzer signalisiert werden), und wenn in dieser Zeit ein Dialog begonnen wird (

Aufwachbegriff), wird die Aufmerksamkeitsphase für die betreffende Sprechstelle verlängert. Da das zu z.B. nur 20 Prozent Chance für den Benutzer führt, ohne Signalisierung den Aufwachbegriff im richtigen Moment zu sagen (während 5 Sprechstellen sich 1 freien Kanal teilen), ist diese Methode nicht zu empfehlen.

Bei gleichem Verkabelungsaufwand wie für die Signalisierung nötig ist sind dann Rufknöpfe zur Aktivierung der Mikrofone die bessere Methode. Es ist auch denkbar, vorhandene Knöpfe und Signalleuchten zu verwenden: Wird der Rufknopf gedrückt, aber der Aufwachbegriff fällt nicht, kommt der Aufzug einfach in das betreffende Stockwerk. Oder man verzichtet auf den Stockwerken ganz auf Aufwachbegriffe: Wird der Knopf gedrückt, fragt der Aufzug per Sprachausgabe nach dem Wunsch des Benutzers.

Das hat gleich mehrere Vorteile: Man kann mit der Aufforderung warten, bis die Spracherkennungssoftware gestartet und das Mikrofon eingeschaltet ist, die Aufforderung durch den Aufzug erspart eine vorherige Einweisung der Benutzer (wer nicht mit einem sprechenden rechnet, drückt dennoch den Knopf) und Umstehende werden wahrscheinlich weniger Nebengeräusche verursachen – ein Mensch kann leichter mitten im Gespräch erkennen, dass der Aufzug einen Dialog beginnen will, als umgekehrt der Fahrstuhl erkennen kann, wenn ein Gespräch unterbrochen wird um den Aufzug anzusprechen.

7.3. Verfügbare Soundkarten

7.3.1. Mehrspur-Soundkarten

Hier vergleiche ich zwei professionelle Soundkarten, die jeweils mit 8 Eingängen und 8 Ausgängen ausgestattet sind. Ein Gerät reicht also bereits, um 7 Sprechstellen zu bedienen.

Die erste Karte, die Midiman Delta 1010, wurde praktisch getestet und lässt sich mit HTK unter Linux mit einem speziellen Hilfsprogramm oder einem getrennt zu kaufenden Treiber (Hilfsprogramm siehe 7.4.3) verwenden, unter Windows lassen sich wegen Treiberproblemen nur die Hälfte, also 4 Sprechstellen realisieren.

Die andere Karte, die Terratec EWS 88 MT, lässt sich unter Linux mit einem getrennt zu kaufenden Treiber verwenden. Unter Windows lassen sich auch hier nur 4 Sprechstellen realisieren, da der Lernout und Hauspie Spracherkenner auch hier den Treiber der 8 Sprechstellen erlauben würde nicht unterstützt. Die Terratec Karte ist gegenüber der Midiman die billigere Lösung, wurde aber nicht praktisch getestet, um die Ausgaben für den Kauf einer zweiten Profi-Soundkarte zu sparen.

Midiman Delta 1010 Mehrspur-Soundkarte

Die erste Karte die bei uns ankam war eine Midiman Delta 1010 (http : //www.midiman.de), eine MT-Soundkarte (MT: Mehrspur / Multitracking) für semiprofessionellen Studioeinsatz. Diese Karte bietet 8 analoge und 2 digitale Eingänge, ebensoviele Ausgänge und eine MIDI-Schnittstelle zum Anschluss von z.B. Synthesizern oder Effektgeräten. Dabei stehen beachtliche 96 kHz Samplingrate und 24 Bit Genauigkeit zur Verfügung, was einem Signal/Rausch-Verhältnis (SNR) von 109 dB entspricht. Bei 96 kHz Samplingrate können

Frequenzen bis zu 48 kHz verarbeitet werden (Nyquist-Theorem – beim Digitalisieren eines Signales bzw. allgemein beim Darstellen einer Kurve in diskreten Zeitschritten ist die maximal zu verarbeitende Frequenz die halbe Samplingrate): Menschen können allerdings keine Frequenzen über 20 kHz wahrnehmen. Die Karte bietet also absolute Studioqualität, zumal der Analogteil in einem abgeschirmten 19 Zoll Gehäuse getrennt aufgebaut ist. Mit einem Preis von 1600 DM ist sie allerdings auch die teuerste mir bekannte Alternative.

Ein wichtiger Vorteil der Midiman ist die Verfügbarkeit von Treibern für verschiedene Windowsversionen (mitgeliefert) und Linux (kostenlos: Advanced Linux Sound Architecture (ALSA) http://www.alsa-project.org kommerziell: Open Sound System (OSS) http://www.opensound.com).

Für Windows werden die Treibermodelle DirectSound, ASIO und MME unterstützt. Eine Spracherkennungssoftware ohne besondere Unterstützung der genannten Treiber kann immernoch jeweils ein Stereopaar als normale Sounddevice öffnen, damit sind für die unterstützten Windowsversionen (95, 98, 98 SE, ME, 2000, NT) unter allen Umständen 4 gleichzeitig laufende Spracherkenner und sehr häufig sogar volle 8 auf 1 Rechner möglich. Dabei droht natürlich leicht eine Überlastung oder Instabilität des Rechners: L & H empfehlen je nach Einsatzgebiet und Betriebssystem 32 bis 96 MByte RAM, für einzelne Erkenner einen Pentium, für mehrere Erkenner eine besondere Version ihres Programmes und ca. 300 MHz. Midiman empfiehlt bei 96 kHz allein zum „Datenschaufeln” mindestens einen Pentium II 350 MHz oder einen entsprechenden AMD Prozessor und 64 MByte RAM. Für 48 kHz werden immernoch 300 MHz MMX benötigt.

Sollen mehrere Erkenner gleichzeitig laufen, ergeben sich verschiedene Probleme: Bei HTK besteht die Gefahr, dass recht viel Rechenleistung verbraucht wird. Die Windowsversionen der verwendeten Erkenner unterstützen alle kein ASIO oder DirectSound und können daher nur Stereopaare öffnen, also können nur die Hälfte der Kanäle unabhängig voneinander benutzt werden. Der Erkenner von L & H ist unter Windows NT aus unklarem Grund nicht in der Lage, mehr als einen Kanal zu benutzen – will man einen weiteren Kanal öffnen, bricht der Zugriff mit einem internen Fehler ab. Unter Windows 98 ist das Problem schlicht, dass wir nur die Lizenz zur Benutzung von 2 Kanälen haben und zusätzliche Zugriffe abgelehnt werden.

Lernout und Hauspie bieten noch einen anderen Erkenner an (ASR 1500 TSO , der in Verbindung mit einem speziellen Telefonanschluss die Sprache von acht Telefonaten gleichzeitig erkennen soll. Dieser Erkenner ist für computergesteuerte Auskunfts- und Bestellsysteme gedacht und auf die schlechte Tonqualität von Telefonaten abgestimmt. Es ist allerdings fraglich, ob er ausreichend gut mit einer Soundkarte (z.B. über ASIO) verbunden werden kann. Lernout und Hauspie (die ohnehin in Schwierigkeiten stecken) haben eine entsprechende Anfrage noch nicht beantwortet, wir haben diesen Spezialerkenner nicht angeschafft.

Terratec EWS 88 MT

Eine preiswertere Alternative zur Midiman ist die EWS 88 MT von Terratec (http://www.terratec.de). Der Analogteil ist dort in einen 5.25 Zoll Laufwerkseinschub statt in ein 19 Zoll Gehäuse eingebaut, ausserdem fehlt die Midi-Schnittstelle zur Ansteuerung von Synthesizern und sonstigen elektronischen Musikinstrumenten und Effektgeräten in der Standardausführung. Die restlichen technischen Daten sind praktisch identisch, aber unter Linux wird die Karte nur vom kommerziellen Open Sound System Treiber (siehe 7.3.1 oben) unterstützt: Da dieser aber nur 60 Dollar kostet, bleibt die Kombination EWS 88 MT (1000 DM) plus Treiber trotzdem preiswerter als die Lösung von Midiman.

Praktische Tests haben wir mit dieser Karte nicht durchgeführt, da recht bald klar wurde, dass wir in der ersten Phase des Projektes nur eine Sprechstelle in der Kabine benutzen würden und deshalb die Experimente zunächst mit einer einfachen PC Soundkarte für unter 100 DM fortgesetzt wurden.

7.3.2. Standardsoundkarten

Bereits ab weit unter 100 DM bekommt man heute verschiedene einfache (z.B. SoundBlaster 16 kompatible) Soundkarten, wie sie sich in vielen PCs befinden. Sowohl Linux als auch Windows unterstützt diese Karten problemlos, Treiberprogramme sind vorhanden. Trotzdem besteht unter beiden Betriebssystemen die Gefahr, von einem Paar Eingangskanälen nur jeweils einen nutzen zu können, was die Anzahl gleichzeitig bedienbarer Sprechstellen halbiert. Unter Linux schafft ein Hilfsprogramm bzw. der kommerzielle OSS Treiber hier Abhilfe.

Ursprünglich für Geräuscheffekte oder Musik aus dem Computer vorgesehen, lassen sie sich auch für unsere Zwecke einsetzen. Normalerweise bieten solche einfachen Soundkarten 16 Bit Genauigkeit und Samplingraten bis 44100 Hz. 16 Bit sind bei sinnvoll eingestellter Lautstärke mehr als ausreichend zur Spracherkennung, 24 Bit Genauigkeit würde sogar rechnerisches Nachregeln der Lautstärke bei ausreichender Qualtiät erlauben, das heisst, man multipliziert das Signal NACH der Digitalisierung mit einem Faktor. Da aber praktisch alle Soundkarten einen digital steuerbaren Mischer haben, es ist nicht unbedingt erforderlich, am Signal herumzurechnen, um die Lautstärke zu korrigieren. Oft reicht es aus, den Mischer von Hand auf eine mittlere Laustärke zu programmieren und auf Nachregelung ganz zu verzichten.

Beim Einsatz einfacher Soundkarten kann gerade bei PCI Soundkarten oft problemlos mehr als eine Karte in einen Rechner eingebaut und so die Kanalzahl erhöht werden, sofern genügend Steckplätze vorhanden und frei sind. Oft sind 4 bis 6 PCI Steckplätze vorhanden. Grafikkarte, Netwerkkarte und Festplattencontroller können jeweils einen davon belegen, falls die entsprechende Funktionalität benötigt wird und nicht schon direkt auf der Hauptplatine vorhanden ist. Ausser der Anzahl freier Steckplätze im Computer begrenzt meistens die Zahl freier Interruptleitungen und DMA Kanäle die Zahl einbaubarer Karten:

Eine typische Soundblaster 16 Soundkarte (noch für die alten ISA Steckplätze mit 16 Bit Breite) benötigt eine solche Interruptleitung, um den Computer über Statusänderungen zu informieren (ein Interrupt löst ein Unterprogramm im PC aus, so dass dieser nicht ständig den Status abfragen muss). Ausserdem werden 2 DMA-Kanäle verbraucht, die das Kopieren der Daten zwischen Computer und Soundkarte automatisieren. Ein normaler

PC stellt 16 Interruptleitungen zur Verfügung und 8 DMA Kanäle, wovon jeweils einer nicht nutzbar ist. Festplatten mit DMA-Datenübertragung brauchen ggf. auch noch DMA Kanäle.

Bei einem normalen Heim-PC werden für Zeitgeber, Tastatur, Maus, Drucker, Uhr, Netzwerkkarte, Gleitkomma-Prozessoreinheit und Festplatten bereits insgesamt 9 Interruptleitungen verbraucht, Modem, Midi-Anschluss und ggf. besondere Einstellungen der Grafikkarte würden bis zu weiteren 3 Leitungen verbrauchen.

Bei neueren Steckkarten können sich mehrere Geräte eine Interruptleitung teilen („IRQ-sharing"), das funktioniert jedoch nur mit geeigneten Treibern. Gerade bei PCI sind die Chancen gut, aber es ist schwierig, einen DMA Kanal zwischen mehreren Geräten aufzuteilen.

Ein Interrupt ist ein einzelnes **Ereignis**, während bei DMA der **Prozess** des automatischen Kopierens eines ganzen Speicherbereichs im Mittelpunkt steht. Während für eine Karte also Daten bewegt werden, ist der ganze Kanal belegt. Bei einem Interrupt reicht es dagegen, den Verursacher herauszufinden (sofern der Interrupt von mehreren Geräten gemeinsam verwendet wird) und dem zugehörigen Gerätetreiber zu melden. Während dieser das Ereignis weiter verfolgt, ist die Interruptleitung schon wieder frei.

Im Normalfall macht es also aus verschiedenen Gründen kaum Sinn, mehr als zwei oder drei Soundkarten in einen einzelnen Rechner einzubauen. Da bei Spiele- und DVD-Freunden vierkanaliger Surround Sound zunehmend in Mode kommt, hat selbst eine einfache Soundkarte inzwischen oft 4 Ausgänge, aber nur 2 Eingänge. Drei einfache Soundkarten können also nicht die geforderten sieben Sprechstellen gleichzeitig bedienen – dieses Problem lässt sich aber meistens verschmerzen, da selten alle Sprechstellen gleichzeitig von Benutzern „besprochen" werden. Mehr dazu gleich im Anschluss.

7.3.3. Spezielle Wandler-Hardware

Für die Verwendung mehrerer Sprechstellen kann auch spezielle Hardware eingesetzt werden, die oft wesentlich preiswerter als entsprechende Mehrkanal-Studiosoundkarten ist. Dabei sind besonders integrierte Wandler zwischen analogen und digitalen Signalen interessant, die kaum zusätzliche Beschaltung brauchen und oft schon eine serielle oder parallele Schnittstelle zum Anschluss an einen Computer eingebaut haben. Praktische Tests wurden nicht durchgeführt, ich beschreibe also nur mögliche Alternativen zu klassischen Soundkarten.

Für Studioeinsatz gibt es auch direkt über Kabel oder Glasfaser anschliessbare Wandler. Entsprechende Schnittstellen sind teilweise auf Soundkarten auch der unteren Preisklasse schon vorhanden, insgesamt würde aber eine solche Lösung dennoch ähnlich teuer ausfallen wie eine Mehrspur-Studiosoundkarte, da mehrere Wandler (und Schnittstellen) benötigt werden.

Ebenfalls ungeeignet sind die meisten Oszilloskop- oder Mess-Karten für PC. Erstere bieten ihre enorme zeitliche Auflösung / Samplingrate oft nicht im Dauerbetrieb an, sondern können nur einzelne kurze Signale mit hoher Samplingrate in einem internen Speicher aufzeichnen und dann deutlich langsamer an den PC übertragen. Die Samplingrate bei durchgehender Aufzeichnung ist dann deutlich niedriger, ausserdem stehen nur wenige Kanäle zur Verfügung. Bei Messkarten sind zwar oft mehrere Kanäle verfügbar, aber diese teilen sich dann intern einen einzelnen hochgenauen aber langsamen Wandler. Versucht man, mehrere Kanäle gleichzeitig einzulesen, sinkt also die Samplingrate pro Kanal auf einen für Audioanwendungen zu niedrigen Wert.

Es gibt allerdings Messkarten, die sich mit 12 Bit Genauigkeit begnügen und dafür eine bessere sogenannte Summenabtastrate von etwa 100 kHz erreichen. Indem man diese Rate auf mehrere Kanäle aufteilt, lassen sich immerhin 4 bis 6 Eingänge mit ausreichender Qualität realisieren, zu einem angemessenen Preis von 250 DM. Diese Karten verwenden einen Wandler mit eingebauter Umschaltmöglichkeit zwischen mehreren (z.B. 8 oder 16) Eingängen, was eine Mehrfachnutzung erlaubt: Man schliesst zum Beispiel 8 Sprechstellen an einen Wandler an und beschränkt sich auf maximal 4 gleichzeitige Dialoge. Siehe oben unter 7.2.4 „scheinbare Skalierbarkeit" und unten unter 7.3.4 „Multiplexing".

Ein-Chip Wandler-ICs

Will man mehr Kanäle und trotzdem eine ausreichende Samplingrate, kann man mehrere integrierte Wandlerbausteine gleichzeitig verwenden. Integrierte 8 Bit Wandler sind zwar sehr preiswert (z.B. TLC 549, etwa 4 DM), für die Mikrofoneingänge sind aber weniger als 12 Bit Genauigkeit nicht sinnvoll. Entsprechende Bausteine kosten dann schon etwa 30 bis 50 DM, können aber mit einer Samplingrate von 50 kHz und mehr eventuell zwei oder mehr Kanäle bedienen. Entsprechende Umschalter zwischen mehreren Eingängen sind teilweise schon integriert.

Ein besonders schneller Baustein (LTC 1419, 70 DM) kann ganze 800000 Messungen pro Sekunde machen. Diese Rate auf viele Kanäle zu verteilen ohne das Messergebnis zu sehr zu verfälschen ist aber schwierig. Mit einer perfekten Verteilung der Messleistung könnten ganze 36 Kanäle über 22000 mal pro Sekunde gemessen werden! Für grosse Anlagen sollte diese Möglichkeit dennoch in Betracht gezogen werden: Zwei dieser Bausteine mit einer komplizierten Schaltung an 50 Sprechstellen anzubinden kann immernoch einfacher sein, als 50 einzelne Wandler gleichzeitig anzusteuern.

Sogar Studiosoundkarten stossen hier an ihre Grenzen: Beispielsweise sollte man nicht mehr als vier Midiman 1010 gleichzeitig in einen PC einbauen und ist damit auf 40 Kanäle begrenzt – allerdings dann mit 24 Bit Genauigkeit und 96 kHz Samplingrate pro Kanal, also perfekter Tonqualität. Gerade bei sehr vielen Sprechstellen ist es aber kaum noch machbar, auf allen gleichzeitig Dialoge zuzulassen. Hier sollte man unbedingt auf „scheinbare Skalierung" (siehe 7.2.4 oben) zurückgreifen, und zum Beispiel für 20 Sprechstellen an einem einzigen Fahrstuhl maximal Resourcen für 8 gleichzeitige Dialoge anschaffen.

Genauere Werte bekommt man durch Tests im praktischen Betrieb oder durch Simulation. Einerseits sollen die Benutzer nicht zu lange warten, bis der Fahrstuhl den Dialog akzeptiert, andererseits sollen nicht 95 Prozent der Hard- und Software die meiste Zeit mit dem Warten auf einen Dialog beschäftigt sein!

7.3.4. Multiplexing vieler Sprechstellen auf wenige Analoganschlüsse

Die hier beschriebene Methode erlaubt den Betrieb einer grösseren Anzahl von Sprechstellen mit einer KLEINEREN Anzahl von Kanälen bei der Audiohardware. Die Methode ist in Verbindung mit Soundkarten oder anderer Hardware anwendbar. Hier beschreibe ich die wenig komplexere Variante der Anwendung in Verbindung mit Spezialhardware.

Beschränkt man den Zugriff per Software auf wenige Kanäle gleichzeitig, sind zumindest schnellere Messkarten für den Einsatz in einem System denkbar, welches seine „Aufmerksamkeit” immer nur einer oder wenigen Sprechstellen zuteilt. Eine mittlere Messkarte für 250 DM erreicht bei ausreichenden 12 Bit Genauigkeit eine Summenabtastrate von 100 kHz, es sind mindestens 8 Eingänge vorhanden. Will man jeden Kanal mit einer Samplingrate von 22 kHz betreiben, lassen sich also maximal 4 der Eingänge gleichzeitig benutzen.

Eine Schaltung, die viele Kanäle und wenige Wandler verbindet (Multiplexer / Demultiplexer) lässt sich auch recht einfach auch „von Hand” passend für die jeweilige Anwendung konstruieren und beispielsweise an eine Soundkarte anschliessen. Die Steuerung kann dabei ohne Probleme über den Druckerport oder eine digitale Schnittstellenkarte geschehen. Ein Multiplexer ist dabei eine Schaltung, die einen von mehreren Eingängen mit einem Wandler verbindet, ein Demultiplexer verteilt das Ausgangssignal eines Wandlers auf mehrere Ausgänge. Die Kombination mehrerer (De-) Multiplexer nenne ich der Einfachheit halber im Folgenden einen Multiplexer.

Dazu ein Beispiel mit einfacher Hardware: Es sind 7 Sprechstellen vorhanden, aber nur eine Soundkarte mit 4 Ausgängen und 2 Eingängen. Es ist nicht praktikabel, einen Anschluss an zwei Geräte mit jeweils halber Samplingrate zu verteilen: Die Kanäle würden stark ineinander übersprechen, da Tiefpassfilter auf der Karte den vom Kanalwechsel bedingten Sprung im Signal verwischen. Der Fahrstuhl muss also während einem Dialog einen Teil seiner Resourcen jeweils einer Sprechstelle fest zuordnen. Auch während auf den Beginn eines Dialoges gewartet wird, ist es besser, jeweils mindestens für Zehntelsekunden an jeder Sprechstelle den Geräuschpegel zu messen oder sogar auf ein Stichwort zu warten, als zu versuchen, alle 8000tel Sekunde ein Sample von jeder Sprechstelle zu bekommen. Es wäre schwierig, die Samples mit ausreichender Qualität zu erfassen, so dass wirklich für jede Sprechstelle parallel ständig auf ein Stichwort gewartet werden könnte.

Es ergeben sich hier zwei Szenarios: Sofern die Sprechstellen **per Hardware** selber das Stichwort zum Dialogbeginn erkennen können oder man den Dialogbeginn vereinfacht **über einen Rufknopf** einleitet, sind alle Kanäle für Dialoge verfügbar. Andernfalls muss immer ein Kanal freigehalten werden, der die Sprechstellen reihum abhört und auf neu beginnende Dialoge wartet. Da mehr Ausgänge als Eingänge vorhanden sind, kann der Benutzer bereits zum Sprechen aufgefordert oder um noch etwas Geduld gebeten werden, bevor die Resourcen für einen neuen Dialog wirklich frei sind.

Insbesondere kann und sollte das Mikrofon einer Sprechstelle, an der gerade der Fahrstuhl selber spricht abgeschaltet werden. Dadurch besteht keine Gefahr, dass die Spracherkennung versucht, Äusserungen des Fahrstuhls zu interpretieren. Ausserdem kann der freigewordene Kanal währenddessen anderweitig verwendet werden.

Um alle Möglichkeiten offenzuhalten, sollten alle Kanäle frei den Sprechstellen zuzuordnen sein. Für die Auswahl von einer aus 7 Sprechstellen sind 3 digitale Steuerleitungen nötig, bei 2 Eingängen und 4 Ausgängen also 18 Leitungen. Man kann sich leicht klarmachen, dass 3 je Steuerleitungen je 8 Bitkombinationen / Binärwerte annehmen können. Der achte Wert kann dabei fest für mit „kein Signal" verbunden werden, um zum Beispiel Mikrofone per Hardware abzuschalten. Manche Spracherkenner bieten keine Softwareschnittstelle an, über die sie zum „Weghören" aufgefordert werden können, hier bietet sich also eine einfache Alternative zum Eingriff in die Software des Soundtreibers.

Ohnehin lohnt es sich kaum, durch komplizierte Codierungen Leitungen einsparen zu wollen: Mit jeder Sprechstelle ist zu jedem Zeitpunkt maximal ein Eingang oder ein Ausgang verbunden. Umgekehrt ist jeder Eingang oder Ausgang zu jedem Zeitpunkt mit genau einer Sprechstelle verbunden oder signalfrei geschaltet. Lässt man die Möglichkeit „kein Signal" weg, gibt es für den ersten Anschluss sieben mögliche Sprechstellen, für den zweiten sechs, und so weiter: Insgesamt gibt es dann 5040 sinnvolle Verschaltungen, zu deren Codierung immernoch 13 Leitungen nötig sind. Die Ansteuerung dieser 13 Leitungen ist aber deutlich komplizierter, als jedem von 6 Kanälen (oder jeder von 7 Sprechstellen) einfach jeweils drei Steuerleitungen zuzuordnen, die dann jeweils eine Sprechstelle (oder einen Kanal) auswählen.

Mit Bausteinen wie dem CMOS IC 4051 (analoger bidirektionaler 1 aus 8 Umschalter, unter 2 DM) lässt sich so ein digital steuerbares „Patchfeld" zur Verteilung und Verschaltung analoger Signale einfach realisieren. Eine ausreichende Zahl von Steuerleitungen bekommt man über preiswerte digitale PIO Schnittstellenkarten (zum Beispiel 48 Leitungen, in verschiedenen Gruppen als Eingänge, Ausgänge oder bidirektional konfigurierbar: 80 DM) oder einfach durch Verwendung von zwei Druckerports. Druckerports sind auf weniger als 100 Kilobyte pro Sekunde, einfache Schnittstellenkarten sind ähnlich langsam, die oben genannte Karte wird über einen ISA-Bus mit 8 Bit Breite und 8 MHz Takt an den PC angeschlossen, eine Übertragungsrate von maximal einigen Megabyte pro Sekunde ist zu erwarten.

Ein Druckerport hat je 8 Datenleitungen (bei modernen Ausführungen bidirektional, sonst in Richtung Drucker), 5 Statuseingänge (Drucker an PC, einer davon kann auch zusätzlich einen Interrupt auslösen) und 4 Steuerausgänge. Ohne besondere Anpassungen an den Treiber oder direkten Hardwarezugriff (braucht unter Linux root oder kmem Rechte) sollte man sich aber darauf beschränken, die Datenleitungen zu verwenden. Werden wirklich viele Anschlüsse benötigt, kann man an spezielle integrierte PIO-Bausteine wie den 8255 (PIO: parallel input/output) verwenden, der auch auf digitalen Schnittstellenkarten eingesetzt wird. Diese Bausteine werden z.B. über einen 8 Bit breiten Datenbus und zwei bis drei Adressleitungen angeschlossen und bieten dann z.B. jeweils 24 Anschlüsse. Gerade bei Anschluss über den Druckerport muss hier aber beachtet werden, dass oft nur eine Übertragungsrate von unter 100 Kilobyte pro Sekunde möglich ist.

Den PC um eine Druckerschnittstelle zu erweitern kostet etwa 30 bis 40 DM, mehr als drei Schnittstellen sind normal nicht sinnvoll. Für die vorgeschlagene Multiplexerschaltung ist es auch denkbar, einen einzelnen Druckerport zu verwenden: Drei Datenleitungen wählen eine Sprechstelle, drei weitere einen Kanal, der Rest kann zum Beispiel als Taktsignal dienen. Noch einfacher ist es, die vorhandene Steuerleitung „Strobe" zu verwenden, und mit zwei Datenleitungen eines von vier Registern zu wählen, das dann jeweils die 6 Bit der restlichen Datenleitungen speichert: Damit lassen sich über eine Druckerschnittstelle mit geringem Aufwand 24 Steuerleitungen ohne betriebssystemabhängige Spezialtreiber ansteuern. Ein solches Register für bis zu 8 Bit ist zum Beispiel der Standardbaustein (IC in Low Power Schottky TTL Bauweise) 74LS273 / 74LS373 für unter 2 DM. Viele andere Bauarten sind denkbar. Ein digitales „Patchfeld" für 6 Kanäle und 7 Sprechstellen und nur einen Druckerport wird insgesamt für etwa 30 bis 50 DM zu realisieren sein, vorverstärkte Pegel von den Mikrofonen vorausgesetzt.

7.4. Treiber für Mehrkanalzugriff

Hat man sich für eine Hardwarelösung entschieden, muss noch das Dialogsystem auf diese Hardware abgestimmt werden. Wenn die Spracherkenner zu viele Resourcen für einen Rechner brauchen, die Soundhardware sich aber in einem Rechner konzentriert, müssen die Audiodaten an mehrere Rechner per Netzwerk verteilt werden. Verteilt sich die Soundhardware ohnehin auf mehrere Rechner, empfiehlt es sich, die Spracherkenner auf allen Rechnern laufen zu lassen, damit nur noch der erkannte Text über das Netzwerk verschickt werden muss, beziehungsweise Status- und Steuer- Informationen mit dem Rechner ausgetauscht werden müssen, der mit der Fahrstuhlsteuerung verbunden ist.

Wenn Soundhardware UND Spracherkenner jeweils auf mehrere Rechner verteilt sind, können normale Soundtreiber verwendet werden (wenige Kanäle pro Rechner) und die Kommunikation zwischen den Rechnern beschränkt sich auf die Dialogkomponente: Alle Erkenner schicken ihre Ergebnisse an einen Rechner für die Dialogauswertung, der wiederum mit der Fahrstuhlsteuerung kommuniziert. Oder aber jeder Rechner wertet die Dialoge „seiner" Kanäle aus und schickt dann nur noch Fahrtziele über den entsprechenden Rechner oder sogar direkt an die Fahrstuhlsteuerung. Erfolgt die Steuerung des Fahrstuhls nämlich über die Simulation von Knopfdrücken, können über eine einfache Verschaltung mehrere Rechner gleichzeitig direkt mit der Fahrstuhlsteuerung / dem „Knopfsimulator" verbunden werden.

7.4.1. Soundtreiber für Windows

Ich beschränke mich im Folgenden auf die Probleme bei Bündelung der Soundhardware auf einem Rechner. Unter Windows stehen verschiedene Treibermodelle zur Verfügung: Klassische digital Wave Audio (PCM) Geräte , ASIO , DirectSound und MME . ASIO und MME sind dabei besonders für Mehrspur-Soundkarten aus dem Studiobereich vorgesehen. Gerade die Mehrspurkarten bringen oft Treiber für alle neueren Windowsversionen, also sowohl 95/98/ME als auch NT/2000 mit, wobei ASIO und DirectSound nur auf den Versionen für Spieler und Heimanwender (95/98/ME) verfügbar sind, die wiederum beim gleichzeitigen Betrieb mehrerer Spracherkennerprozesse leicht überfordert werden können. Der Betrieb des Systems ist dann nicht mehr so zuverlässig wie man das von einer

Fahrstuhlsteuerung erwartet. Die von uns verwendeten Spracherkenner unterstützen nur den einfachen PCM Treiber, was zur Folge hat, dass man linken und rechten Kanal eines Stereopaares nicht auf zwei Erkenner verteilen kann. Der ASR 1600 Erkenner liegt uns sowieso nur mit einer Lizenz zur Analyse von maximal zwei Kanälen gleichzeitig vor, wegen einem Fehler im Treiber oder im Erkenner geht sogar das nur mit Windows 98, nicht mit Windows NT, wenn man die Midiman Soundkarte verwendet.

Das klassische Gerätemodell wird von ALLEN Spracherkennern und Soundkarten unterstützt, oft lassen sich die Kanäle aber nur als Stereopaare ansprechen. Wenn der verwendete Spracherkenner mit halben Stereopaaren nicht richtig umgehen kann, kann man also nur die HÄLFTE der Kanäle verwenden! Mögliche Probleme: Der Erkenner kann nicht auf „nur links" oder „nur rechts" festgelegt werden, oder der Treiber erlaubt für jedes Stereopaar nur einen parallelen Zugriff, also nur einen Erkenner. Optimal wäre hier ein Erkenner, der ein Stereopaar öffnen und zugleich beide Kanäle getrennt auf einmal analysieren kann. Ein ähnliches Problem tritt unter Linux auf, mehr dazu gleich.

Die spezielleren Treibermodelle erlauben den Zugriff auf jeden Kanal einzeln. Soweit der verwendete Erkenner und die verwendete Hardware es also ermöglichen, führt diese Methode zuverlässiger zum Erfolg. Für speziell angefertigte Hardware stehen leider kaum Treiber zur Verfügung, zumindest Messkarten (siehe 7.3.3 oben) oder über den Druckeranschluss gesteuerte Multiplexer (siehe 7.3.4 oben) lassen sich aber mit erträglichem Aufwand ansteuern. Das Problem ist die zu erwartende Inkompatibilität zum Spracherkenner: Während ein Multiplexer ohnehin von der Dialogkomponente angesteuert wird, ist die Verbindung von Spracherkenner und Messkarte durch die nicht aufeinander abgestimmten Programmierschnittstellen (API, Application Programmer Interface) ein ernstes Problem.

7.4.2. Soundtreiber für Linux

Soll nur eine Standardsoundkarte verwendet werden, genügt prinzipiell der direkte Zugriff auf die Gerätedatei mit Standard-Dateifunktionen, weitere Befehle sind nur zum Beispiel zur Einstellung der Samplingrate nötig. Da wir jedoch mehrere Kanäle getrennt öffnen und getrennt der Spracherkennung zuführen wollen, ergeben sich weitere Schwierigkeiten. Bei einer einfachen Soundkarte kann es vorkommen, dass ein Stereopaar nicht als getrennter linker und rechter Kanal ansprechbar ist. Ein Hilfsprogramm bei ALSA oder der OSS-Treiber schafft hier Abhilfe. Grössere Probleme entstehen bei den Mehrspur-Soundkarten. Zunächst aber eine kurze Erklärung der Linux Treibermodelle.

Für Linux stehen hauptsächlich die zwei Treibermodelle ALSA und OSS (siehe 7.3.1 oben) zur Verfügung: ALSA bietet einen einfachen Gerätezugriff über die /dev/dsp Gerätedatei , komplexere Funktionen werden überwiegend über eine API realisiert, die die beim Compilieren von Programmen Einbindung der ALSA Bibliotheken erfordert. OSS verwendet ioctl Aufrufe , um komplexere Funktionen über Zugriffe auf die Gerätedatei abzubilden. Die ioctl-Aufrufe brauchen keine besondere Bibliothek, wohl aber besondere Headerdateien, da die verwendeten Funktionen über (nicht ausserhalb von OSS standardisierte) Nummern ausgewählt werden.

Der ALSA-Treiber für Mehrspur-Soundkarten wie Midiman Delta 1010 erlaubt nur das gleichzeitige Öffnen aller Kanäle, man muss also über ein spezielles Programm die Soundkarte zugreifen und die Kanäle dann einzeln an die verschiedenen Spracherkenner weiterleiten. Ein solches Programm stelle ich im nächsten Abschnitt vor: Es leitet die Kanäle über TCP/IP Netzwerkverbindungen weiter, ein kleines Hilfsprogramm erlaubt dann den Empfang. Die empfangenen Daten können in eine Datei oder eine „Pipeline” geschrieben werden. Sofern also der Spracherkenner Pipelines unterstützt, ist eine Weiterverarbeitung in Echtzeit möglich. Für den Spracherkenner HTK von Entropic lässt sich eine entsprechende Erweiterung über die Programmierschnittstelle (API) einbauen, mit zusätzlicher Verbesserung der Erweiterung könnte sogar das Hilfsprogramm entfallen.

Der OSS-Treiber unterstützt Mehrspur-Soundkarten nur in der kommerziellen Version (60 Dollar), bietet dann aber sehr umfangreiche Unterstützung des Konzepts von Gerätedateien: für jedes Stereopaar wird eine Gerätedatei eingerichtet, ausserdem kann beim Öffnen von /dev/dsp optional angegeben werden, wieviele und ggf. welche Kanäle man zugreifen will. Wenn man also aufzeichnet, welche Kanal wie zugeteilt wurde, kann man einfach mehrere Spracherkenner jeweils /dev/dsp mono öffnen lassen, um mehrere Kanäle gleichzeitig zu analysieren.

Soll spezielle Hardware zum Einsatz kommen, muss ggf. ein entsprechender Treiber erst noch geschrieben werden. Während unter Windows 95/98/ME ein Programm direkt auf die Hardware zugreifen kann, sind bei Linux und Windows NT besondere Rechte nötig. Bei Linux muss das Programm mit den erhöhten Rechten von root oder kmem laufen. Bei allen genannten Betriebssystemen kann alternativ ein normaler Treiber geschrieben und ins Betriebssystem eingebunden werden, was wegen des verfügbaren Quellcodes anderer Treiber unter Linux deutlich einfacher ist:

Zum Beispiel kann man den Treiber einer Messkarte und den einer Soundkarte als Vorlage nehmen, um einen Treiber zu schreiben, der die Messkarte über /dev/dsp auf OSS-kompatible Weise zugänglich macht. Verwendet man stattdessen den direkten Zugriff durch ein Programm, sollte nur ein Hilfsprogramm die Root-Rechte und direkten Hardwarezugriff bekommen, während die Weiterverarbeitung der Daten ohne besondere Rechte erfolgt. Das hat den Vorteil, bei einem Fehler im Hauptprogramm keine fehlerhafte Ansteuerung der Hardware oder Veränderung von wichtigen Daten zu riskieren. Also: Sicherheit vor Abstürzen, Systemveränderungen, unberechtigtem Zugriff durch Benutzer die Unsicherheiten in der Fahrstuhl- Software ausnutzen.

7.4.3. Treiber für die Übertragung von Audiodaten im Netz

Hier beschreibe ich ein kleines Hilfsprogramm, das die Verwendung der Midiman Mehrspur Soundkarte mit HTK erlaubt, so dass jeder Kanal getrennt angesprochen werden kann. Das Programm dient nur der grundlegenden Demonstration der Möglichkeiten mit ALSA und Netzwerk, ist aber dennoch schon in seinem aktuellen Zustand sinnvoll einsetzbar, auch wenn ich einige sehr empfehlenswerte Erweiterungen (und einige weniger wichtige Erweiterungen) vorstelle.

Für die Midiman Delta 1010 Soundkarte (jeweils 8 analoge Ein- und Ausgänge) habe ich eine spezielle Software für Linux entwickelt, die bisher im Zusammenhang mit HTK getestet wurde. Da der ALSA Treiber für diese Soundkarte nur erlaubt, alle Kanäle einer Richtung gleichzeitig zu öffnen, kann HTK nicht direkt auf den Treiber zugreifen. HTK kann immer nur einen Kanal pro Prozess untersuchen, muss also mehrfach gestartet werden, wenn mehrere Sprechstellen aktiv sind. Mein Server hält alle Kanäle ständig offen, liest alle Daten von ihnen und verteilt die Daten der einzelnen Kanäle an HTK Prozesse oder andere Clients nach Bedarf. Hat ein Kanal zu einem Zeitpunkt keinen Client, werden die aufgezeichneten Daten gleich wieder verworfen. Mein Programm bietet also die Sprachdaten zentral an (Server) und gibt sie auf Anfrage an „Kunden" (Clients) weiter.

Diese Aufteilung in Client und Server mit Verbindung übers Netzwerk hat den Vorteil, dass die Spracherkennung auf mehrere Rechner verteilt werden kann beziehungsweise ein Operator an übers Netzwerk mithören oder zur weiteren Untersuchung mitschneiden kann, was der Fahrstuhl hört. Sofern Client und Server auf dem selben Rechner laufen, funktioniert die Datenübertragung sehr schnell und resourcenschonend (über das sogenannte local loopback unter Linux), über eine externe Verbindung sind zum Beispiel bei einer Samplingrate von 22050 Hz und 16 Bit Genauigkeit pro Kanal 44100 Byte pro Sekunde zu übertragen, was in einem normalen Ethernet-Netzwerk bei den von uns angestrebten 7 Sprechstellen kein Problem sein sollte.

Für die Übertragung wird eine Verbindung auf Basis von TCP/IP aufgebaut, einem Standard-Protokoll zur Datenübertragung im Netzwerk. Nachdem der Server gestartet ist, kann jederzeit ein Client eine Verbindung aufbauen. Der Server antwortet dann mit dem Text „`Hello!`" und erwartet dann vom Client eine Anfrage. Es gibt bisher nur 2 Arten von Anfrage: Sendet der Client den Text „`CLOSE`", beendet sich der Server. Sendet der Client den Text „`GETx`", wobei an Stelle des `x` eine Zahl stehen muss, antwortet der Server mit einem Strom von signed short int Werten, die das Eingangssignal von Kanal `x` enthalten. Will der Client keine weiteren Daten empfangen, beendet er einfach die Verbindung. Tritt beim Server ein Fehler auf, beendet dieser seinerseits die Verbindung. Ausserdem ist der Server auf zwei Clients pro Kanal begrenzt. Beim anderen Anfragen als „`CLOSE`" gibt der Server Fehlermeldungen aus:
„`FAIL: Pardon?`" im Falle einer nicht interpretierbaren Anfrage, „`FAIL: Channel?`" wenn ein nicht vorhandener Kanal gelesen werden soll, und „`FAIL: Busy`" wenn ein Kanal bereits zwei Clients hat.

Das Programm ist bisher recht elementar aufgebaut, kann aber einfach erweitert werden. In `rec-sock-alsa` (L.1.1) werden von `main()` aus zuerst die Einstellungen für die Soundkarte gesetzt und alle Kanäle geöffnet (`open_sound`). Dann wird `open_socket` gerufen, welches einen Thread (nebenläufigen Verarbeitungsprozess, verwendet die pthread Funktions-Bibliothek von Linux) startet, der fortan auf eingehende Verbindungen auf dem beim Serverstart angegebenen Port (TCP/IP-Kanalnummer) wartet. Dieser Thread ist in `server-sockets` (L.1.2) implementiert und wird später noch genauer erklärt. Das Programm wird dann mit der Funktion `read_sound` fortgesetzt, die solange Daten von der Soundkarte liest, bis ein Fehler auftritt oder der Server eine „`CLOSE`" Aufforderung empfängt. Dann wird der Server beendet, vorher werden noch `close_socket` und `close_sound` aufgerufen, um restliche offene TCP/IP Verbindungen und die Verbindung zur Soundkarte zu beenden.

`open_sound` und `close_sound` sind in `alsa-ice` (L.1.3) implementiert und verwenden Funktionen der ALSA-Bibliothek zum Zugriff auf die Soundkarte. Die Funktionen dieser Bibliothek sind auf der ALSA-Homepage (http://www.alsa-project.org) detailliert beschrieben, die Beschreibung kann auch zusammen mit Treiber und Bibliothek dort heruntergeladen werden.

`read_sound` liest Daten in einen Puffer und setzt dann ein Signal, dass neue Daten eingetroffen sind. Nach kurzer Wartezeit werden die Daten für ungültig erklärt und neue Daten gelesen. Alte Daten werden also einfach überschrieben. Mehrere Puffer reihum als Ringpuffer zu verwenden um mehr Zeit zum Abholen der Daten anzubieten ist robuster, wurde hier aber noch nicht implementiert. Egal wie die Daten gepuffert werden, zu alte Daten sollten besser verworfen werden, damit die Spracherkennung möglichst zeitnah erfolgen kann.

`send_buffer` wandelt die Daten vor dem Senden um und filtert sie, so dass nur ein Kanal an den Client verschickt wird. Für komplexere Aufgaben dieser Art bietet ALSA ein Plugin-Konzept an, bei dem man vorgefertigte Funktionen zwischen den eintreffenden Rohdaten und die letztlich in `send_buffer` eintreffende Version „einklinken" kann. Damit würde `send_buffer` einfacher, aber die Initialisierung der Plugins und die Notwendigkeit von Puffern für jeden Kanal machen den Server insgesamt eher komplexer. Für einen Server mit mehreren Funktionen empfiehlt sich die Methode trotzdem, da die Verarbeitung dann modularer und übersichtlicher stattfindet.

Die Abwicklung von Verbindungen ist in `server-sockets` (L.1.2) implementiert und besteht aus drei Funktionen: `open_socket` öffnet einen TCP/IP „Socket" und startet einen weiteren Thread `accept_conn`, der auf eingehende Verbindungen wartet. Dann kehrt `open_socket` zurück, `accept_conn` läuft weiter bis der Server aufgefordert wird, sich zu beenden. Sobald eine Verbindung eingeht, wird ein neuer Thread `serve_partner` gestartet, der diese Verbindung dann bedient und bis solange läuft, wie die jeweilige Verbindung besteht. `serve_partner` ruft nach Bedarf `send_buffer` aus `rec-sock-alsa` auf, welches wartet, bis gültige Daten im Puffer sind und dann daraus die Daten für einen einzelnen Kanal ausliest und über die TCP/IP Verbindung in Form eines Feldes von signed short int Werten (16 Bit) abschickt.

Eine *wichtige Erweiterung* des Servers ist ein Befehl zur Stummschaltung eines Kanales: Oft liegen Lautsprecher und Mikrofon nahe beieinander, falls der Spracherkenner also nicht während einer Sprachausgabe des Aufzugs an den Benutzer auf Pause gestellt werden kann, wird der Spracherkenner sonst versuchen, die Sprache des Aufzuges zu analysieren. Kurz: Der Aufzug würde sich selber zuhören, was unnötig ist und leicht zu Fehlverhalten führt. Eine entsprechende Funktion muss in `serve_partner` aktiviert werden, indem dort ein Befehl erkannt wird (Syntax zum Beispiel `MUTEx` und `HEARx` wobei `x` eine Kanalnummer angibt). Dort wird dann eine Tabelle aktualisiert, die für jeden Kanal angibt, ob er gerade stummgeschaltet ist. Ebenfalls in `serve_partner` muss dann diese Tabelle abgefragt werden, um vor jedem Aufruf von `send_buffer` zu entscheiden, ob wirklich `send_buffer` aufgerufen werden soll oder eine Funktion `send_silence`, die einen Puffer gleicher Grösse aber ohne Sprachdaten sendet. Stattdessen werden Daten gesendet, die einem Signal der Lautstärke Null entsprechen.

Ein Beispielclient ist `client2file` (L.2) : Das Programm sendet sein drittes Argument als String an den im ersten und zweiten Argument angegebenen Rechner und Port des Servers. Die empfangenen Daten werden dann über die Standardausgabe ausgegeben und können per Umleitung dieser in eine Datei oder eine „Pipeline" weitergeleitet werden. Das Client-Programm ist sehr einfach. Der Spracherkenner kann entweder so eingestellt oder erweitert werden, dass er Mikrofondaten aus einer „Pipeline" akzeptiert, oder man kann den Spracherkenner direkt so erweitern, dass er direkt mit dem Server kommuniziert.

Wird die Ausgabe von `client2file` in eine sogenannte „named pipes" geschrieben (eine „Pipeline" die einen eigenen Dateinamen besitzt), kann der Spracherkenner die Daten sogar fast wie aus einer normalen Datei lesen – die Option, Daten aus einer Datei zu analysieren, ist in vielen Spracherkennern vorhanden. `client2file` kann so erweitert werden, dass die Daten in ein anderes Dateiformat umgewandelt werden, ausserdem kann man über die „Pipelines" alle Daten durch das Konvertierungsprogramm sox leiten und so ein anderes Datenformat erreichen. Durch das einfache Protokoll ist es nicht schwierig, einen Client aus Standardkomponenten in verschiedenen Programmiersprachen zu implementieren. Sogar von der Unix Kommandozeile kann man auf den Server zugreifen, indem man die Daten zum Beispiel mit netcat übers Netzwerk schickt (telnet ginge zur Not auch).

Hier nochmal ein Ablaufbeispiel. Zu Beginn:
`rec-sock-alsa 1234 &` (Server startet)
`client2file localhost 1234 GET5 | erkenner &` (ein Erkenner)
`client2file localhost 1234 GET3 | erkenner &` (weiterer Erkenner)
Auf dem Rechner des Operators:
`client2file soundserver 1234 GET5 >/dev/dsp &` (Mithören was der erste Erkenner hört)
Am Ende des Experiments:
`client2file soundserver 1234 CLOSE` (Server wird beendet)

Zum Abspielen von Sprache an einer Sprechstelle steht bisher nur ein einfaches Programm namens `play-alsa` (L.3) bereit, mit dem man eine an der Kommandozeile angegebene Audiodatei auf einem vorgegebenen Kanal abspielen kann. Da auch zum Abspielen alle Kanäle gleichzeitig zugegriffen werden müssen, ist es mit dem Programm nicht möglich, den Fahrstuhl an mehreren Sprechstellen zugleich sprechen zu lassen. Dazu und um die oben beschriebene MUTE Funktion zu vereinfachen, *sollte der Server um eine Abspielfunktion erweitert werden.* Eine denkbare Syntax ist `PLAYx dateiname`, um eine Datei auf einem Kanal abzuspielen und zu Beginn und Ende der Wiedergabe automatisch MUTE und HEAR aufzurufen.

Den Server auch beim Abspielen per Netzwerk mit Audiodaten zu versorgen ist weniger sinnvoll, da der Fahrstuhl nur wenige vorgegebene Sätze verwendet und die Bereitstellung neuer Audiodaten mit einem Sprachsyntheseprogramm nur wenig Resourcen verbraucht. Ausserdem kann man die Audiodateien mit Dateiübertragungsprogrammen wie ftp oder scp übers Netzwerk aktualisieren, wenn sie nicht sogar auf einem übers Netzwerk verbundenen Laufwerk liegen. Soll ein `PUT` für in Echtzeit erzeugte Audiodaten dennoch vorgesehen werden, braucht der Server auch zum Abspielen einen Puffer, um einen gleichmässigen

Datenstrom zur Soundkarte sicherzustellen.

Weitere Verbesserungsvorschläge:
Das Protokoll kann kompatibel zu http Version 1.0 und icecast gestaltet werden. Die Syntax lautet dann zum Empfangen `GET /5**` (* steht für einen Zeilenwechsel) und zum Senden (wie bei `PUT`) `SOURCE passwort /5**`, wobei im Beispiel die Sprechstelle 5 angesprochen wird. Entsprechende Anpassungen in `serve_partner` sind ohne grossen Aufwand zu machen. Die Rückmeldungen vom Server müssen für echte Kompatibilität allerdings auch geändert und erweitert werden, was aber auch eher geringen Aufwand bereitet.

Als weniger wichtige Erweiterung bietet sich MP3 oder GSM Unterstützung an: Dazu kann man in Server und Client Programme wie sox oder lame3 (LameENC) und mpeg123 aufrufen, um die Audiodaten in Formaten wie GSM oder MP3 komprimiert zu übertragen und so Netzwerkbandbreite zu sparen.

Das PLAY Kommando kann im Voraus auf bestimmte beim Serverstart genannte Dateien beschränkt werden, die dann gleich im Arbeitsspeicher eingelagert werden. Dadurch greift der Server nur beim Start auf die Festplatte zu, was manchmal wünschenswert ist (zum Beispiel wenn die Festplatte langsam ist oder sogar nur übers Netzwerk erreichbar und gar nicht im Rechner mit der Soundkarte eingebaut ist).