Projektabschlußbericht

Juli 1998

CTI-Ansteuerung einer TK-Anlage über CORBA und Java Telephony API am Beispiel eines Least-Cost-Routers

Friedemann Mattern, Gerd Meister, Marco Gruteser, Thomas M. Schoch

FG Verteilte Systeme
FB Informatik
Technische Universität Darmstadt
Alexanderstr. 6, 64283 Darmstadt

Email: mattern|meister@informatik.tu-darmstadt.de
http://www.informatik.tu-darmstadt.de/VS

Zusammenfassung

Im Wintersemester 1997/98 wurde am Fachgebiet Verteilte Systeme in komplementärer Weise ein Seminar und ein Praktikum im Hauptstudium mit dem Titel „Telekommunikationsdienste und verteilte Anwendungen mit Java“ durchgeführt. Ziel des Praktikums war die Realisierung eines Least-Cost-Routers in Java für eine moderne Telekommunikationsanlage (TK-Anlage). Die Routing-Entscheidung sollte dabei nicht in der TK-Anlage selbst, sondern auf einem angeschlossenen Rechner getroffen werden. Dadurch ergibt sich eine wesentlich größere Flexibilität. So kann der Algorithmus, der die Entscheidung trifft, dynamisch ausgetauscht werden und die Daten, welche die Grundlage der Entscheidung bilden, können regelmäßig beispielsweise über das Internet aktualisiert werden. Zudem ist eine Fernwartung der TK-Anlage und des Routers mittels eines Java-Applets über das Internet möglich.
Als Schnittstelle zur TK-Anlage nutzt der Router das neue Java Telephony API (JTAPI), welches teilweise selbst implementiert wurde. Dadurch ergibt sich eine hohe Portabilität für die Routing-Software. Sie kann mit jeder TK-Anlage, welche die nötige Routing-Funktionalität zur Verfügung stellt und für die eine JTAPI-Schnittstelle existiert, zusammenarbeiten. Zusätzlich ist sie durch den Einsatz von Java auch unabhängig von Hardware und Betriebssystem des Rechners.
Durch eine Kooperation mit der Firma Bosch Telecom, Frankfurt, stand für das Praktikum eine sich in der Entwicklung befindende TK-Anlage (Integral Communication Center) zur Verfügung. Die im Praktikum erarbeiteten Lösungen wurden schließlich auf der CeBIT '98 sowohl am Stand der hessischen Hochschulen als auch bei Bosch Telecom präsentiert und fanden reges Interesse.

Schlüsselworte: Computer Telephony Integration (CTI), CORBA, Java, Java Telephony API (JTAPI), Fernwartung, Least Cost Routing (LCR), Telekommunikation, TK-Anbieter (Provider)

1 Motivation

In den letzten Jahren kam es in der Telekommunikationsbranche zu bedeutenden Veränderungen, da im Zuge der Computer-Telefonie-Integration (CTI) die Grenze zwischen den Domänen Computer und Telefonie stark verwischt wurde. Auf der Seite der Telekommunikation begann dies vor einigen Jahren mit der Digitalisierung der Fernübertragungswege und dem Einsatz von digitalen Vermittlungsstellen. Mittels ISDN wurde schließlich auch die „letzte Meile“ überwunden und dem Telekommunikationskunden ein digitaler Anschluß bereitgestellt. Parallel dazu gab es auch bei den TK-Endgeräten und TK-Anlagen eine Entwicklung hin zu digitalen Geräten. Dies macht sich vor allem durch die Integration von Microcontrollern zur Konfiguration und Steuerung der Endgeräte - oder bei TK-Anlagen vielfältig sogar komplexen Standardprozessoren - und eine weitgehende Softwaresteuerung bemerkbar. Viele Geräte bieten darüber hinaus eine Schnittstelle, mit der eine Steuerung durch Computer möglich ist.

Auf der Seite der Computer fand eine analoge Entwicklung hin zur Telekommunikation statt. War anfänglich mit einem Akustikkoppler oder Modem nur eine langsame und eher umständliche Datenfernübertragung über die Telefonleitungen möglich, so bieten sich dem Benutzer inzwischen vielfältige Möglichkeiten. Mit modernen Modems bzw. ISDN steht weit mehr als die hundertfache Bandbreite verglichen mit der Anfangszeit zur Verfügung; darüber hinaus lassen sich mit heutigen Geräten aber auch andere Kommunikationsdienste wie beispielsweise Fax nutzen.

Sprach-, Fax- oder sogar Videodaten müssen heute jedoch nicht mehr zwangsläufig durch Telefonleitungen übertragen werden. Vielfach werden diese Daten auch in digitaler Form durch breitbandige Computernetze geschickt. Vor allem in letzter Zeit erzielte die kostengünstige Übertragung von Sprachdaten durch das Internet unter dem Schlagwort Internet-Telefonie große Aufmerksamkeit.

In Verbindung mit ISDN ist ein PC mit entsprechender Multimediaausstattung bereits in der Lage, die TK-Endgeräte (insbesondere Telefone) zu ersetzen. Mit einfachen Programmen ist das Telefonieren und Faxen möglich. Weiterhin gibt es Anrufbeantworter - vielfach mit erweiterter Funktionalität -, die in Software realisiert sind.

Bisher wurden beim Anwender jedoch TK-Geräte noch nicht durch Computer bzw. Multimedia-PCs abgelöst. Vielmehr wird durch eine intensive Koppelung und Kooperation beider Welten auf eine gegenseitige Ergänzung von TK-Gerät und Computer gesetzt. So übermitteln TK-Anlagen Anrufdaten zwecks einer Verarbeitung und eventuellen Speicherung an den Computer. Beispielsweise ist es möglich, daß in einem Call Center schon beim Klingeln eines Telefons der Anrufer durch den Computer identifiziert wird. Dem Telefonisten können vom Computer sofort Informationen über den Anrufer aus einer Datenbank angezeigt werden.

Der Computer wird auch zur Konfiguration und Steuerung von TK-Anlagen und Endgeräten eingesetzt. Die meisten Funktionen der TK-Geräte können ebenfalls vom Computer ausgelöst werden. So lassen sich für einfache Telefone Wahlhilfen - etwa mit integriertem Telefonbuch - realisieren. Weiterhin bietet diese Unterstützung aber auch wesentlich komplexere Funktionalitäten wie beispielsweise das Starten von Anrufen durch Computer in Call Centern für Anwendungen wie *predictive calling*. Dabei wird vom Rechner automatisch eine Liste mit Anrufzielen abgearbeitet. Erst bei einer zustandegekommenen Verbindung wird dies dem Telefonisten signalisiert, um unnötige Wartezeiten während des Wahlvorgangs und bei Anrufen, die nicht angenommen werden, zu vermeiden.

Parallel zur Integration von Computer und Telefonie auf der technischen Ebene wurde in der Bundesrepublik Deutschland in den letzten Jahren die Liberalisierung des Telekommunikationsmarktes vorangetrieben. Nach mehreren Stufen der Deregulierung wurde die Liberalisierung zum Beginn des Jahres 1998 abgeschlossen. Seitdem können auch andere Unternehmen neben der Deutschen Telekom TK- und insbesondere Sprachdienste außerhalb geschlossener Benutzergruppen anbieten. Diese Möglichkeit haben viele Unternehmen genutzt und kräftig in den Aufbau eigener TK-Netze investiert. Dabei wurden meist bestehende TK-Netze wie beispielsweise die der Energiekonzerne oder der Deutschen Bahn ausgebaut. Auch viele ausländische Unternehmen aus der TK-Branche sind an den neu entstandenen TK-Dienstleistern (Service Provider) beteiligt.

Infolge des Wettbewerbs ist davon auszugehen, daß dem Kunden in absehbarer Zeit verbesserte Leistungen und neue technische Möglichkeiten zu geringeren Preisen zur Verfügung stehen. Die Anbieter rechnen ihre Leistungen jedoch nach unterschiedlichen Tarifmodellen ab, was allgemein zu

einer größeren Unübersichtlichkeit bezüglich der Preise führt. Geschäfts- und Privatkunden können mittels des "Call by Call"-Verfahrens vor jedem Anruf den Provider bestimmen, über den der Anruf abgerechnet werden soll. Damit lassen sich durch eine geschickte Wahl der Provider Kosten in beträchtlichem Umfang einsparen.
Im Hinblick auf CTI stellt sich nun die Frage, inwieweit sich die Wahl des günstigsten Anbieters durch Einsatz von Rechnern automatisieren läßt. Zur Reduzierung der Komplexität der Realisierung ist dabei der Einsatz von Standardtechnologien wünschenswert. Durch den Einsatz der im Internet weit verbreiteten Sprache Java lassen sich die Softwarekomponenten plattformunabhängig implementieren, wodurch die Portierung wesentlich vereinfacht wird Weiterhin ist eine Anbindung an das Internet in Bezug auf die Fernwartung des TK-Systems interessant. Damit sind Einwählverbindungen zu TK-Anlagen überflüssig und die TK-Anlage kann von einem entfernten Rechner gewartet und überwacht werden.
Das Thema war für das Fachgebiet "Verteilte Systeme" der TU Darmstadt aufgrund seiner Aktualität in zweierlei Hinsicht interessant. Zum einen bot es die Möglichkeit, den im Wintersemester 1997/98 am Seminar und Praktikum teilnehmenden Studenten ein sehr aktuelles Themengebiet zu vermitteln. Die Studenten hatten so die Chance, theoretische und praktische Kenntnisse in einem Bereich zu erwerben, in dem in nächster Zeit starker Bedarf auf dem Arbeitsmarkt bestehen wird. Zum anderen ergab sich dadurch die Möglichkeit, Kontakte zu Firmen zu knüpfen und damit vielleicht den Weg zu weiteren Forschungsarbeiten zu ebnen.

2 Grundlagen

Bevor auf die Aufgabenstellung und ihre Umsetzung im Praktikum eingegangen wird, sollen in diesem Kapitel zunächst einmal die grundlegenden Technologien und Konzepte beschrieben werden, auf denen die entwickelten Lösungen basieren.

2.1 Least Cost Routing

Mit Least Cost Routing (LCR) bezeichnet man allgemein die Auswahl des für eine Nachricht oder Verbindung kostengünstigsten Weges durch ein Netz. Auf dem heutigen Telekommunikationsmarkt bieten viele verschiedene Provider ihre Dienste an und rechnen diese nach unterschiedlichen Tarifmodellen ab. Mittels des "Call by Call"-Verfahrens kann der Anrufer vor jedem Verbindungsaufbau den Provider auswählen, über den die Verbindung hergestellt und abgerechnet werden soll. Eine Verbindung kann dabei nicht nur zum Telefonieren, sondern auch zum Faxen oder einer beliebigen anderen Formen der Datenübertragung genutzt werden. In diesem Zusammenhang steht LCR für die Auswahl des optimalen Providers für jede einzelne Verbindung, um die Telekommunikationskosten des Kunden zu minimieren.
Größere Unternehmen verfügen oft über eigene Telekommunikationsnetze oder fest angemietete Leitungskapazitäten (Standleitungen). Wenn Kosteninformationen für Verbindungen in diesen Netzen vorhanden sind, können sie in die LCR-Entscheidung einbezogen werden.

2.1.1 Ermittlung des optimalen Providers

Betrachtet man die Abrechnungsmodelle der verschiedenen Provider, so stellt sich die Frage, wie sich der bezüglich der gesamten Telekommunikationskosten optimale Provider ermitteln läßt.
Der Preis eines Anrufs ist von einer Vielzahl von Parametern abhängig [1]. Neben den bekannten Parametern Dauer des Anrufs, Entfernung des Ziels sowie Wochentag und Uhrzeit, gibt es insbesondere verschiedene Mengenrabatte. So verbilligt sich der Tarif bei manchen Provider ab einer bestimmten Dauer eines Gespräches. Andere gewähren auf die gesamte Monatsrechnung (gestaffelte) Rabatte, sobald diese einen bestimmten Betrag übersteigt. Weiterhin sind auch noch Grundgebühren, Mindestumsätze, individuelle Rabatte und vor allem unterschiedliche Zeittakte zu berücksichtigen. Eine sehr kurze Verbindung kann beispielsweise bei einem Provider, der sekundengenau abrechnet, wesentlich günstiger sein, als bei einem Provider mit großem Zeittakt.
Da der Provider vor jedem Anruf ausgewählt werden muß, führt dies alles bei der Berechnung des optimalen Providers zu großen Schwierigkeiten. Denn weder die genaue Dauer einer bevorstehenden

Verbindung noch die genaue Zusammensetzung des monatlichen Telekommunikationsaufkommen sind im voraus bekannt.
Heute gängige Lösungen, die automatisch die LCR-Entscheidung treffen, beschränken sich daher auf die Minimierung der Kosten für jeden einzelnen Anruf. Zudem wird bei der Ermittlung des günstigsten Providers oft nur der Parameter "Entfernung des Ziels" berücksichtigt. Manche Lösungen beziehen auch die aktuellen Zeitdaten in ihre Entscheidung ein, und es existieren erste primitive Ansätze, auch komplexere Zusammenhänge wie Mindestumsätze zu berücksichtigen. Momentan nutzen fast alle Anbieter bei ihrer Abrechnung die Einteilung in Entfernungszonen der Deutschen Telekom. Unter der Annahme, daß die Provider ihre Tarife nicht allzu oft ändern, läßt sich die Berechnung des günstigsten Providers vor jedem Anruf vermeiden. Dazu wird für jede Entfernungs- und Zeitzone der günstigste Provider vorberechnet und in einer statischen LCR-Tabelle gespeichert.

2.1.2 Problemfälle bei LCR

Viele der neuen Provider sind des öfteren überlastet, d.h. alle ihre Leitungen sind belegt. In solch einem Fall kann der Router entweder ein vom normalen Besetztzeichen unterschiedliches Tonsignal (gassenbesetzt) an den Anrufer weiterleiten oder die Verbindung über den Standard-Provider Deutsche Telekom aufbauen (Fallback). Intelligentere Lösungen versuchen zunächst die Verbindung über den nächstgünstigsten Provider aufzubauen. Dieses Verfahren bezeichnet man als Rerouting. Möglich wird dies, indem man in die LCR-Tabelle zusätzlich noch weitere Provider in kostenoptimierter Reihenfolge aufnimmt.
Soll der Router für jeden möglichen Anruf die günstigsten Provider kennen, muß seine LCR-Tabelle eine gewisse Größe haben [2]. Betrachtet man nur nationale und internationale Entfernungs- und die jeweiligen Zeitzonen, dann läßt sich diese Tabelle schon nicht mehr mit vernünftigen Aufwand von jedem Benutzer von Hand eingeben. Zudem ändern sich die Tarife der Provider aufgrund der momentanen Dynamik im Telekommunikationsmarkt beinahe monatlich. Damit ist eine komfortable Möglichkeit, die LCR-Tabellen zu ändern und auf den neuesten Stand zu bringen, dringend notwendig.
Wenn die Tarife auf dem Telekommunikationsmarkt so unübersichtlich bleiben, ist davon auszugehen, daß LCR-Tabellen von Dienstleistern erstellt werden. Diese könnten Standard-LCR-Tabellen für jedes Ortsnetz verkaufen (Entfernungszonen sind ortsabhängig), oder auch mit Hilfe von Einzelverbindungsnachweisen eine nahezu optimale LCR-Tabelle für einen Kunden errechnen. Der Benutzer könnte diese Tabellen dann in seiner LCR-Lösung installieren. Falls die Lösung eine Fernwartungsschnittstelle besitzt, ist auch eine automatische Aktualisierung durch den Dienstleister denkbar.

2.1.3 Verfügbare Lösungen

Alle momentan verfügbaren Lösungen arbeiten mit LCR-Tabellen, wobei das Aktualisieren der Tabellen als Problem bestehen bleibt. Sie lassen sich grob einteilen in LCR-Boxen, die zwischen Telefon bzw. TK-Anlage und Telefonanschluß gesteckt werden [2], und TK-Anlagen mit in die Steuerungssoftware der Anlage integrierter und somit statischer LCR-Funktionalität. Daneben existieren noch Softwarelösungen für PCs [2, 3].

LCR-Box

Diese Geräte sind für Privatleute und Geschäftskunden, die keine TK-Anlage besitzen, gedacht. Neuere LCR-Boxen unterstützen auch kleinere TK-Anlagen, greifen aber zur Realisierung der Funktionalität sehr tief in die ISDN-Übertragungsprotokolle ein. LCR-Boxen bestehen aus einem Mikrocontroller und Speicher für eine LCR-Tabelle. Installiert wird ein solches Gerät zwischen den Endgeräten und dem Telefonanschluß. Damit kann es erkennen, wenn an einem Endgerät eine Rufnummer gewählt wird.
Aus der Rufnummer und dem aktuellen Zeitpunkt ermittelt das Gerät aus der LCR-Tabelle den günstigsten Provider und stellt dessen Netzkennzahl entsprechend dem "Call by Call"-Verfahren der Rufnummer voran. Dabei unterscheiden sich die Geräte u.a. darin, ob und wie genau der aktuelle Zeitpunkt in die Auswertung mit einfließt. Dies ist vor allem auch von der Größe der LCR-Tabelle abhängig. Weiterentwickelte Geräte beherrschen zusätzlich Rerouting oder Fallback und teilweise auch Callback-Verfahren.

Für die Aktualisierung der LCR-Tabellen werden je nach Gerätetyp verschiedene Verfahren angeboten. Manche Geräte müssen umständlich über ein angeschlossenes MFV-Telefon programmiert werden. Andere bieten eine Anschlußmöglichkeit für PCs (serielle Schnittstelle) und erlauben das Ändern der Tabellen über eine Konfigurationssoftware. Einige Geräte bieten sogar die Möglichkeit zur Fernwartung, so daß die LCR-Tabellen von einem Dienstleister (meist dem Hersteller) auf dem neuesten Stand gehalten werden können.

TK-Anlage mit LCR-Funktionalität

Für Geschäftskunden, die eine TK-Anlage benötigen, bieten verschiedene Hersteller eine direkt in die interne Software der TK-Anlagen integrierte LCR-Funktionalität an. Sie arbeiten nach dem gleichen Verfahren, das auch bei den LCR-Boxen zum Einsatz kommt. Wird an einem der angeschlossenen Telefone eine externe Rufnummer gewählt, so ermittelt die Anlage aus ihrer LCR-Tabelle den günstigsten Provider und stellt dessen Netzkennzahl der Rufnummer voran bzw. wählt die entsprechende Amtsleitung bei Preselection aus. Die Größe der LCR-Tabelle unterliegt auch bei dieser Vorgehensweise starken Beschränkungen, so daß meist nicht mehr als zehn verschiedene Provider unterstützt werden.

Software

Diese Lösung ist eher auf Privatkunden ausgerichtet und setzt einen PC und ein angeschlossenes Modem voraus. Das LCR funktioniert dabei nur, wenn die Anrufe vom PC aus gestartet werden. Die Software ermittelt dann den günstigsten Provider und wählt über das Modem die Netzkennzahl und die Rufnummer. Danach kann der Benutzer den Telefonhörer eines parallel zum Modem angeschlossenen Telefons abheben und das Gespräch führen.

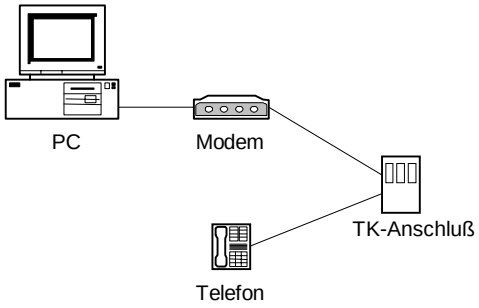


Abbildung 1: Softwarelösung ohne TK-Integration

2.2 Fernwartung

Mit der zunehmenden Vernetzung verschiedenster Systeme gewinnt der Aspekt der Fernwartung immer mehr an Bedeutung. Im Zuge der CTI ist es wünschenswert, mehrere TK-Anlagen zentral von einem Computer aus warten zu können. Dadurch kann ein Servicetechniker in vielen Fällen Konfigurationsfehler beheben, ohne selbst vor Ort sein zu müssen, und es lassen sich Zeit und Anfahrtskosten einsparen. Weiterhin ist es dadurch möglich, bestimmte regelmäßige Wartungsaufgaben zu automatisieren. Dies ist auch besonders im Hinblick auf die Aktualisierung von LCR-Tabellen oder sogar LCR-Algorithmen interessant.

Durch die Fernwartung entstehen jedoch auch zahlreiche Sicherheitsprobleme. Der Wartungszugang einer TK-Anlage könnte von Unbefugten zum Ausspähen von sensiblen Daten, wie beispielsweise Daten über Telefonverbindungen, bis hin zur Sabotage genutzt werden. Es besteht so die Gefahr, daß durch die Manipulation an der Konfiguration der TK-Anlage die Telekommunikationsmöglichkeiten einer Firma erheblich beeinträchtigt werden, was zu nicht unerheblichen Kosten führt.

In den heute gängigen TK-Anlagen erfolgt die Fernwartung - falls überhaupt möglich - meist über ISDN-Einwählverbindungen. Dabei kommt speziell auf die TK-Anlagen zugeschnitte Software zum Einsatz, die über proprietäre Protokolle mit der TK-Anlage kommuniziert. Zur Wartung sind also gegenwärtig keine Standardtechnologien wie beispielsweise das Intranet einsetzbar.

2.3 Java

Die objektorientierte Programmiersprache Java [W1] wurde in den letzten Jahren von Sun entwickelt und genießt inzwischen die Unterstützung von fast allen großen Softwareherstellern [W2]. Sie ist speziell auf den Einsatz im Internet zugeschnitten. Vor allem das Appletkonzept gestattet es, Programmcode über das Internet zu laden und auf dem Client auszuführen. Aufgrund der Plattformunabhängigkeit der Sprache kann das Applet auf beliebigen Rechnertypen mit jedem Betriebssystem ausgeführt werden. Einzige Voraussetzung ist dabei, daß auf dem Rechner ein *Java Runtime Environment* (JRE) installiert ist.

Dieses besteht zum einen aus der *Java Virtual Machine* (JVM), die den Java-Maschinencode (Bytecode) interpretiert bzw. in Maschinencode des entsprechenden Rechners übersetzt, was zu einem schlechteren Laufzeitverhalten führt. Alternativ dazu kann auch eine Übersetzung mit Hilfe eines *Just-in-Time-Compilers* (JIT) beim Programmstart durchgeführt werden. Dies resultiert in einer Verzögerung bei der erstmaligen Ausführung eines neu geladenen Codefragments, jedoch nähert sich dadurch das Laufzeitverhalten eines Java-Programms dem eines in C++ implementierten Programms an. In absehbarer Zukunft soll dies durch den Einsatz der *Hotspot*-Technologie [W3] weiter verbessert werden. Diese optimiert den Maschinencode während der Abarbeitung des Programms.

Weiterhin gehört zum JRE das Java API. Dieses API stellt einige nützliche Funktionen bereit, welche die Programmierung im Netz unterstützen. Dazu gehört auch *Remote Method Invocation* (RMI), welches es ermöglicht, Methoden von Objekten, die sich auf einem anderen Rechner befinden, über das Netz aufzurufen.

Voraussetzung für das Appletkonzept ist eine ausgefeilte Sicherheitsarchitektur [W4], die den Rechner, der das Applet lädt, vor unberechtigten Aktionen durch den fremden Code schützt. Applets, die über das Netz geladen wurden, werden dazu in einem von den anderen Programmen abgekapselten Bereich - der *Sandbox* - ausgeführt. Code, der innerhalb dieser *Sandbox* ausgeführt wird, hat u.a. keinen Zugriff auf das lokale Dateisystem und kann Netzwerkverbindungen nur zu dem Server, von dem das Applet stammt, aufbauen.

Die *Sandbox* besteht aus den Komponenten *Class Loader*, *Verifyer* und *Security Manager*. Der *Class Loader* lädt das Applet und sorgt dafür, daß es in einem von den lokalen Programmen getrennten Namensraum ausgeführt wird. Vor dem Start des Applets überprüft der *Verifyer*, ob der Bytecode der *Java Language Specification* entspricht. So werden u.a. unzulässige Typkonvertierungen und Stapelüberläufe verhindert. Während der Ausführung des Applets wacht der *Security Manager* darüber, daß es keine sicherheitskritische Funktionalität aus dem Java API nutzt. Schließlich trägt auch die Sprache selbst zur Sicherheit bei, indem sie keine Zeiger und damit keine direkten Speicherzugriffe zur Verfügung stellt und ein strenges Typkonzept besitzt.

Weiterhin bietet Java eine außerordentlich hohe Flexibilität beim dynamischen Nachladen von Programmcode. Klassen- und Methodennamen beispielsweise sind auch im Bytecode noch enthalten und werden erst zur Laufzeit aufgelöst. Dadurch können auf einfache Weise Module während der Laufzeit des Programms – sogar über das Netz – geladen und ausgeführt werden.

Ursprünglich war die Sprache einmal für die Steuerung von Haushaltsgeräten ("embedded systems") wie beispielsweise Waschmaschinen konzipiert worden. In letzter Zeit ist wieder ein Trend in diese Richtung zu erkennen. Es wurden (insbesondere hinsichtlich Systembibliotheken und APIs) abgespeckte Java-Versionen wie z.B. Embedded Java [W5] entwickelt, die für den Einsatz in Haushaltsgeräten und Geräten der Unterhaltungselektronik (Consumer electronics) vorgesehen sind. Damit lassen sich dann prinzipiell auch Telekommunikationsendgeräte mit Java programmieren.

2.4 JTAPI als abstraktes Telefonie-API

Eine weitere Entwicklung, die den Einsatz von Java im TK-Bereich forciert, war die Spezifikation des Java Telephony Application Programming Interface (JTAPI) [W6]. Dies ist eine Schnittstellendefinition, über die eine Java-Anwendung Telekommunikationsgeräte steuern kann. An der Entwicklung waren viele namhafte Hersteller aus der Telekommunikationsbranche, darunter auch die Entwickler proprietärer Telefonie-APIs wie XTL [W7] (Sun), TSAPI [W8] (Novell) und TAPI [W9] (u.a. INTEL) beteiligt. Bei Beginn unserer Arbeiten lag JTAPI in der Version 1.1 vor. Inzwischen gibt

es eine etwas überarbeitete und vor allem besser dokumentierte Version 1.2. Als weiterer etablierter Standard soll hier nur der Vollständigkeit halber Callpath [W10] (IBM) genannt werden.
Die Telefonie-APIs unterscheiden sich u.a. darin, daß sie auf eine first-party- oder eine third-party-Konfiguration zugeschnitten sind. Unter einer first-party-Konfiguration versteht man einen Rechner, an den die Telefonie-Hardware direkt angeschlossen ist. Telefonie-Applikationen greifen also mittels eines Telefonie-APIs auf die Hardware am gleichen Rechner zu. Eine third-party-Konfiguration besteht dagegen aus mehreren Rechnern. Die Telefonie-Hardware – beispielsweise eine TK-Anlage – ist an einen zentralen Telefonie-Server angeschlossen. Telefonie-Applikationen laufen auf den Arbeitsplatzrechnern und greifen dann mittels des Telefonie-APIs auf die Telefonie-Hardware am Telefonie-Server zu. Die Kommunikation mit dem Telefonie-Server über ein Netz wird dabei vom Telefonie-API transparent realisiert. JTAPI kann in beiden Konfigurationen eingesetzt werden.
JTAPI, wie es von Sun bereitgestellt wird, besteht zunächst nur aus der Schnittstellenspezifikation. Die Schnittstelle muß dann auf jeder Telefonieplattform geeignet implementiert werden. Bei der Implementierung kann die Telefoniehardware direkt angesteuert werden oder auf ein schon vorhandenes proprietäres Telefonie-API wie z.B. TSAPI aufgesetzt werden. In einem Schichtenmodell könnte man JTAPI also in einer Schicht zwischen den Anwendungen und den proprietären APIs einordnen.

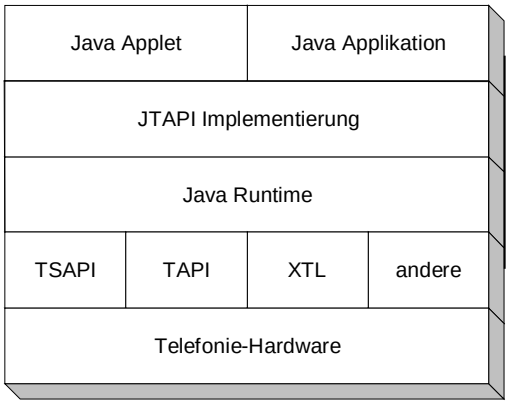


Abbildung 2: Zusammenhang der verschiedenen Telefonie-APIs

Damit besitzen Anwendungen, die das JTAPI nutzen, eine außerordentlich hohe Portabilität. Auf den verschiedenen Plattformen muß nur eine JRE und eine geeignete JTAPI-Implementation vorhanden sein. Dann ist es sogar möglich, den gleichen Bytecode auf völlig verschiedenen Systemen auszuführen, ohne spezifische Plattformanpassungen vornehmen zu müssen.
Das JTAPI wurde mit dem Blick auf eine gute Skalierbarkeit und damit für ein sehr breites Spektrum möglicher Anwendungen entworfen [W11]. Um diese Anwendungsgebiete optimal abzudecken, ist das JTAPI für einfache Anwendungen sehr übersichtlich und einfach zu programmieren, stellt aber trotzdem für große Anwendungen eine umfangreiche Funktionalität zur Verfügung. Das Szenario reicht von Applets mit einfachen Telefoniefunktionen bis zu Anwendungen in Call Centern.
Erreicht wird dies durch eine modulare Struktur des APIs. Es existiert ein zentrales *Core*-Modul; dieses stellt die für Telefonie benötigte Basisfunktionalität wie Anruf starten, Anruf annehmen und Verbindung beenden bereit. Diese Funktionalität wird dann durch zusätzliche Module, die *standard extension packages*, erweitert. Applikationen müssen so nicht das ganze API, sondern nur die Module, die sie wirklich benötigen, einbinden. Ebenso muß eine JTAPI-Implementation nicht alle Module, sondern neben dem *Core*-Modul nur die Module, dessen Funktionalität sie auch wirklich anbieten will, zur Verfügung stellen.
Eines der wichtigeren Erweiterungsmodule ist das Modul *Call Center*. Es bietet u.a. die Funktionalität für das Routing von Anrufen. Des weiteren werden die Module *Call Control*, *Phone*, *Media* und *Private Data* angeboten, die erweiterte Basisfunktionalität, Konfiguration von TK-Endgeräten, Zugriff auf Daten- und Sprachkanäle und herstellerspezifische Erweiterungen unterstützen.

3 Praktikumsverlauf

3.1 Entwicklung

Die ersten Bestrebungen sich mit dem Thema “Computer-Telefonie-Integration” (CTI) auseinanderzusetzen, sind auf den Mai 1997 datiert. Die Aktualität, sowie die Evaluation des Themengebietes CTI waren Motive, ein Hauptstudiumspraktikum in diesem Bereich anzubieten. Gerade im Hinblick auf Plattformunabhängigkeit und die angebotene Programmierschnittstelle JTAPI fiel der Entschluß, die Programmiersprache Java zu verwenden.
Nachdem das Thema grob umrissen feststand, wurden erste Gespräche mit Bosch Telecom Frankfurt bezüglich Kooperationsmöglichkeiten aufgenommen. Auf der Suche nach einer konkreten Problemstellung kam von Bosch Telecom der Vorschlag, ein externes Least Cost Routing zu realisieren.
Diese Idee wurde vom Fachgebiet sehr begrüßt, so daß folgende Vorgaben für das Praktikum und eine Zusammenarbeit nunmehr konkrete Gestalt annahmen:

- Bosch Telecom stellt eine Telefonanlage, die mit Java programmiert werden soll.
- Im Rahmen einer Praktikumsteilaufgabe soll eine externe LCR-Applikation entwickelt werden.

Der Titel des Praktikums lautet: *Telekommunikationsdienste und verteilte Anwendungen mit Java.*

3.2 Zielsetzung

Mit Bosch Telecom als Kooperationspartner wurden einige Kriterien festgelegt, welche es zu verwirklichen galt.

3.2.1 Transparenz

Der Routing-Vorgang soll transparent gegenüber möglichen Telefonieteilnehmer stattfinden. Auf die konkrete Situation bezogen bedeutet dies, daß der Anrufer seine Zielnummer wie gewöhnlich per Hand oder mit Hilfe eines Computerprogramms (z.B. unter Verwendung der TAPI-Schnittstelle) eingibt. Bei erfolgreichem Verbindungsaufbau erhält er das Freizeichen der Gegenseite. Sollte die Gegenstelle besetzt sein, so wird dem Anrufer das gewöhnliche Besetztzeichen signalisiert. In dem Fall, daß das Least Cost Routing aus anderen Gründen fehlschlug, muß ein unterschiedlicher Besetztton generiert werden, der auf diesen Zustand hinweist. Ansonsten würde dem Anrufer die Möglichkeit genommen, zu unterscheiden, ob die Gegenseite wirklich besetzt ist oder ob nur der Provider die Verbindung nicht schalten konnte. Sollte nämlich nur der von LCR vorgeschlagene Provider besetzt sein, würde ein sofortiges nochmaliges Wählen ggf. mit Auswahl eines anderen Providers Sinn machen. Im Gegensatz dazu werden bei einem echten Besetztzeichen der Gegenseite gewöhnlich in größeren Intervallen neue Anlaufversuche gestartet.

3.2.2 Externes LCR

Bisher bot Bosch Telecom für seine TK-Anlagen ausschließlich internes LCR an, welches einigen Restriktionen unterliegt. So ist bei der internen Variante der Speicherplatz relativ stark limitiert. Die Ermittlung der günstigsten Route erfolgt mit Hilfe einer statischen Methode, die weitgehend auf optimierten Zeit- und Entfernungsberechnungen beruht. Zusätzliche Merkmale, wie z.B. Zeitrabatte für längere Gespräche, Kontingente oder Mengenrabatte sind bei internen Lösungen nicht einfach zu integrieren. Einen Ausweg zeigt hier eine externe Variante an. Extern bedeutet, daß die Routing-Entscheidung außerhalb der TK-Anlage getroffen wird. Somit ist es möglich, Software zum Einsatz zu bringen, die überaus flexibel die LCR-Entscheidung kalkuliert.

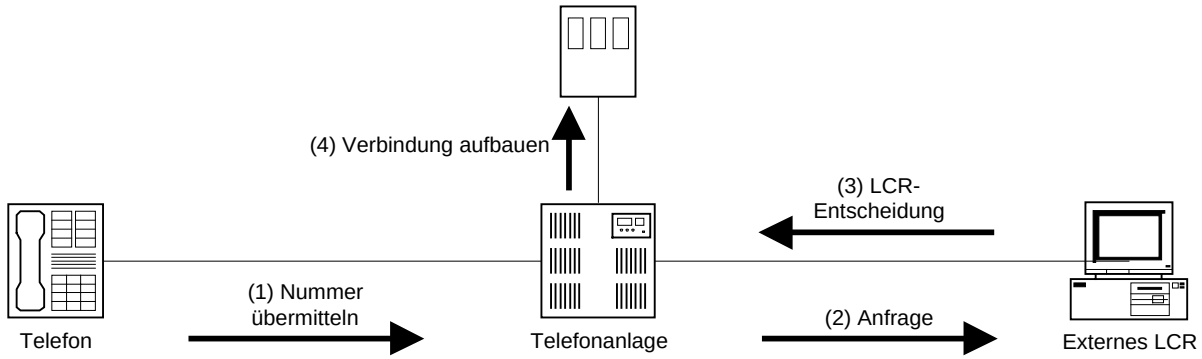


Abbildung 3: Schema des externen Routings einer Verbindung

3.2.3 Flexibilität

Flexibilität bedeutet, daß der Berechnungsalgorithmus austauschbar ist, d.h. ändern sich die Berechnungsmodelle der Anbieter, so muß nicht gleich die gesamte Routing-Applikation aktualisiert werden, sondern es wird nur die Methode, die für die Berechnung zuständig ist, ausgetauscht. Da der gesamte Provider-Markt sich erst langsam entwickelt, werden des öfteren Tarife und Berechnungsmodelle geändert, so daß einer simplen Möglichkeit, die Daten und die Berechnung aktuell zu halten, eine große Bedeutung beigemessen wird.

3.2.4 Fernwartung

Die konsequente Fortführung der im vorangegangen Abschnitt erwähnten Flexibilität besteht in der Fernwartung der Daten und der Berechnungsmethode. So bietet es sich an, eine Applikation zu entwerfen, die weltweit über das Internet die Daten sowie die Berechnungsmethode manipulieren kann. Besondere Sorgfalt muß aber, wie bereits erwähnt, der Sicherheit gelten.

3.2.5 Nutzung des Bosch Telecom Integral Communication Center (ICC)

Realisiert werden sollte das LCR schlußendlich auf Basis des TK-Servers ICC von Bosch Telecom. Dieser bietet die bezüglich ihrer Aufrufsyntax offengelegte Programmierschnittstelle TSAPI (Telephony Services Application Programming Interface) an, welche auf CSTA [4] beruht. Ein Vor- sowie ein Nachteil werden hier offensichtlich. Da TSAPI eine weitgehend offene Programmierschnittstelle darstellt, sind Applikationen, die TSAPI benutzen, unabhängig von der Anlage, d.h. eine solche Applikation kann auf allen TK-Anlagen betrieben werden, die TSAPI unterstützen. Der Nachteil besteht hier konkret darin, daß TSAPI keine Java-Sprachanbindung, sondern lediglich eine C-Schnittstelle bereitstellt. Zum Entwicklungszeitpunkt der LCR-Applikation existierte weiterhin keine Umsetzung von JTAPI auf TSAPI, so daß diese Schnittstelle zusätzlich im Praktikum programmiert werden mußte.

3.2.6 Einsatz von Java und JTAPI

Der Vorteil von Java, der den im vorherigen Abschnitt erwähnten Nachteil wieder kompensiert, ist zum einen die Plattformunabhängigkeit. Unabhängig davon, ob Windows, Unix oder andere Betriebssysteme zum Einsatz kommen: wenn sie Java unterstützen, können Java-Programme ausgeführt werden, wobei höchstens kleinere Einschränkungen in Abhängigkeit vom Betriebssystem (Threadscheduling, graphische Benutzeroberfläche) bestehen.
Ein weiterer sehr signifikanter Vorteil von Java ist die Möglichkeit, Programme als Applets im Internet zu starten. Dieser Aspekt ist gerade für die Fernwartung sehr wichtig. Einschränkungen gibt es auch an dieser Stelle. In der zum Entwicklungszeitpunkt zur Verfügung stehenden Version von Java sind die Sicherheitsrestriktionen für Applets so hoch angesetzt, daß viele Möglichkeiten zur Nutzung lokaler Rechnerressourcen nicht verwendet werden können. Allerdings existieren mittlerweile auch Möglichkeiten, Applets durch Signaturverfahren eine höhere Vertrauenswürdigkeit und somit auch mehr Rechte einzuräumen.

3.3 Organisation

Wie Abschnitt 3.1 bereits zu entnehmen ist, war der Rahmen, in dem die LCR-Applikation entwickelt werden sollte, ein universitätsinternes Praktikum im Hauptstudium, welches in Kooperation mit Bosch Telecom Frankfurt durchgeführt wurde. An der Veranstaltung nahmen 12 Studenten in vier Gruppen teil:

Gruppe 1: Lenhart, Michael Odignal, Frank Mink, Martin	Gruppe 2: Vest, Torsten Juhnke, Ralph Felck, Christoph
Gruppe 3: Holtschmidt, Jürgen Mössinger, Oliver Feldbusch, Piere	Gruppe 4: Schoch, Thomas De Cambray, Laetitia Gruteser, Marco

Tabelle 1: Teilnehmer des Praktikums

Als Bearbeitungszeitraum war die Vorlesungszeit des WS 97/98 von Mitte Oktober bis Mitte Februar vorgesehen. Bedingt durch fehlende Softwarekomponenten und zusätzlichen Einarbeitungsbedarf verschob sich der eigentliche Start auf Anfang Dezember. Die Fertigstellung eines Prototypen zur Präsentation auf der CeBIT 1998 zog sich bis kurz vor deren Beginn im März hin.
Zu Anfang war die Arbeit so organisiert, daß die Aufgaben von den einzelnen Gruppen parallel bearbeitet wurden. Gegen Ende hin mußte aus zeitlichen Gründen die restliche Arbeit zur Realisierung von JTAPI und der Integration der bis dahin fertiggestellten Teilkomponenten auf die Gruppen verteilt werden.

3.4 Aufgabenstellung

Zu Beginn waren fünf Teilaufgaben anvisiert, doch mußte wegen der anfänglichen Probleme die Anzahl auf drei gekürzt und dabei mehrere Teilaufgaben zusammengefaßt werden.

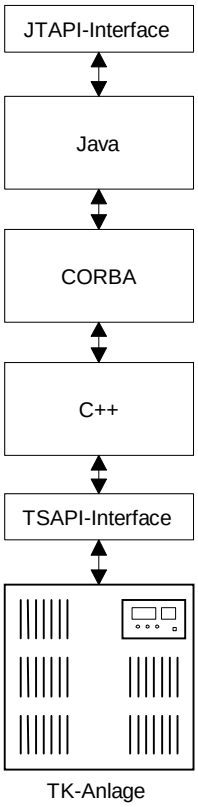
3.4.1 Aufgabe 1: GUI und Router

Die erste Aufgabenstellung [W12] bestand darin, eine Administrationsapplikation mit einem Graphical User Interface (GUI) zu entwickeln, das TK-Anlageneinstellungen sowie Daten zum Least Cost Routing verwaltet. Es sollte die Möglichkeit bieten, Dienstanbieter (Provider), Verkehrsgruppen zur Berechtigungsverwaltung (Traffic Groups), Dienstzugangsverfahren z.B. für Amtszugänge (Trunk Line Groups), Telefonbenutzer bzw. Endgeräte (Users) und Leitungsbündel (Bundles) zu verwalten. Ein anderer Aspekt der Applikation ist ein Monitormodus, der Statusinformationen insbesondere zum Least Cost Routing anzeigt.
Weiterhin sollte das LCR-Modul entwickelt werden, welches anhand der Daten, die durch die Administrationsapplikation verwaltet werden, das eigentliche LCR kalkuliert. Das LCR-Modul soll hierbei das JTAPI-Interface verwenden und speziell das JTAPI-Routing-Interface implementieren. Beide Komponenten sollten in Java als Applikation entwickelt werden. Eine Applet-Version des GUI konnte optional hinzugefügt werden.

3.4.2 Aufgabe 2: Basic Call und Routing-Anbindung der TK-Anlage

Die Intention der zweiten Aufgabe [W13] ist, die TK-Anlage mit TSAPI anzusteuern. Dazu muß zunächst eine Verbindung mit der TK-Anlage hergestellt werden. Im nächsten Schritt wird ein Basic Call, also der in diesem Fall softwaregesteuerte Aufbau einer Verbindung zwischen zwei an die Anlage angeschlossenen Telefonen, implementiert. Der interessante und für das Thema LCR wichtige Teil ist die Registrierung auf Routing-Ereignisse bei der TK-Anlagensoftware und die Auswertung und

Beantwortung der daraufhin von der Anlage vor jedem Verbindungsaufbau generierten Anfrage. Das Least Cost Routing soll im Praktikum ohne Verwendung von Amtsanschlüssen simuliert werden, indem ein Anruf innerhalb der Anlage auf verschiedene andere ebenfalls interne Telefone geroutet werden kann. Außerdem stellte das ICC innerhalb der Praktikumsdauer nicht die volle Anlagenfunktionalität für Routing externer Anrufe zur Verfügung, so daß externe Routing-Requests emuliert werden.



3.4.3 Aufgabe 3: JTAPI-Implementierung für das ICC

Die letzte der drei Aufgaben ist eine JTAPI-TSAPI-Umsetzung [W14]. Die Problematik ist folgende: In der ersten Aufgabe wurde eine LCR-Applikation entwickelt, welche JTAPI-konform ist. Die Anlage hingegen ist TSAPI-konform, d.h. es muß eine Umsetzung entwickelt werden, die JTAPI, wenigstens in dem benötigten Maße, auf TSAPI und umgekehrt abbildet. Da die direkte Kommunikation mit der Anlage in C programmiert wird und die LCR-Applikation in Java geschrieben ist, wird also eine Umsetzung benötigt, die auf der einen Seite TSAPI und C bzw. C++ und auf der anderen Seite JTAPI und Java "versteht". Hier sollte im Praktikum CORBA verwandt werden, was die Kommunikation von Objekten (verschiedener Programmiersprachen) in einem Rechnernetz ermöglicht. Bezogen auf unsere LCR-Applikation ergeben sich also folgende Anforderungen:

- Auf C++-Seite muß eine Registrierung bei der Anlage erfolgen.
- Im laufenden Betrieb müssen die anlagengenerierten Ereignisse dem CORBA-System bereitgestellt werden.
- CORBA liefert die Routing-Anfragen an die LCR-Komponente aus, die auch auf einem beliebigen anderen Rechner im Netz ablaufen kann.
- Auf Java-Seite müssen die Ereignisse abgearbeitet werden und dementsprechend Methoden der LCR-Applikation aufgerufen werden.
- Wurde eine Route ermittelt, so muß jene via CORBA und C++ der Anlage zurückgemeldet werden.

4 Umsetzung der Aufgabenstellung

Im folgenden wird die Realisierung nach dem Top-Down-Prinzip dargestellt. Diese Vorgehensweise entspricht zum einen der realen Herangehensweise im Praktikum, zum anderen wird die Möglichkeit geboten, ein zunächst grobes Bild zu vermitteln, das in der Folge weiter verfeinert wird[1].

4.1 Routing-Modell

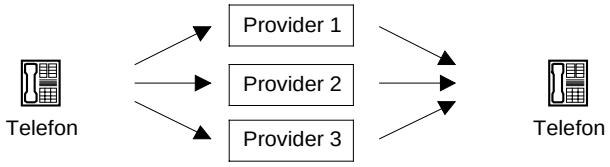


Abbildung 4: Telefon – LCR als Blackbox – Gegenseite

Im ersten Schritt stellt sich die Frage, wie das Least Cost Routing überhaupt stattfinden soll. Wie dem Einführungsteil bereits zu entnehmen war, besitzen interne Varianten einige Nachteile, die durch externe

[1] Bei der Beschreibung der Implementation werden folgende Schriftarten eingesetzt:
`cProvider, routeSelected`: C++-Klassen oder C++-Klasseninstanzen, sowie Attribute und Variable,
jRouting: Java-Interfaces sowie Java-Klassen und Java-Klasseninstanzen,
routeEvent(): Methodenaufrufe werden mit einem leeren Klammerpaar abgeschlossen.

Varianten aufgehoben werden können. Fällt die Entscheidung, wie hier vorgestellt, auf die externe Variante, so sind einige Bedingungen initial zu erfüllen. Die Telefone müssen an eine TK-Anlage angeschlossen sein, welche Daten mit einem Rechner austauschen können muß. Auf dem angeschlossenen Rechner muß Software in Betrieb sein, welche die Kommunikation mit der TK-Anlage ermöglicht.

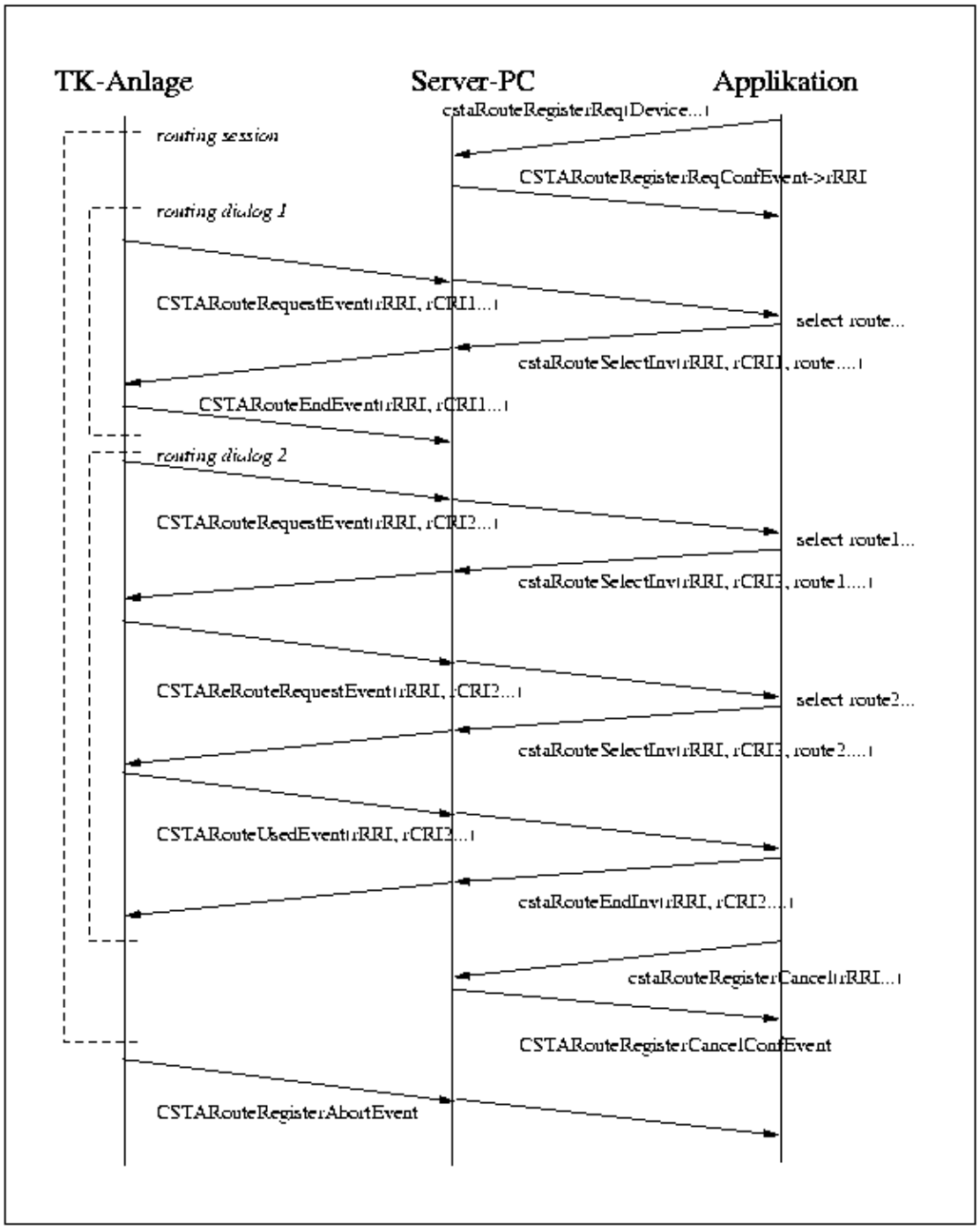


Abbildung 5: Ablauf des Routing unter CSTA

Die LCR-Entscheidung selbst soll nicht in der TK-Anlage selbst getroffen, sondern auf einen angeschlossenen Rechner ausgelagert werden. Diese externe Komponente stellt der TK-Anlage somit

einen LCR-Service zur Verfügung. Eventuell können mehrere Routing-Server in einem Netz parallel Anfragen bearbeiten und damit einen skalierbaren und transparenten Routing-Service bieten.
Die TK-Anlage sendet nun immer dann, wenn an einem ihrer Telefone eine externe Nummer gewählt wird, eine Nachricht (ROUTE REQUEST) an den LCR-Service. Dabei wird die gewählte Rufnummer übermittelt. Aus der Rufnummer und der aktuellen Uhrzeit kann der die Anfrage bearbeitende LCR-Server nun den optimalen Provider berechnen und die Rufnummer entsprechend dem Zugangsverfahren des ermittelten Anbieters modifizieren. Die so veränderte Nummer wird an die TK-Anlage zurückgesandt, die mit dem Rückgabewert die Verbindung über den optimalen Provider aufbaut.
Wenn in diesem Moment alle Leitungen des Providers besetzt sind (gassenbesetzt), wird ein Rerouting durchgeführt. Die TK-Anlage erkennt, daß der Provider momentan nicht erreichbar ist, und sendet erneut eine Nachricht (REROUTE REQUEST) an den LCR-Server. Diese hat dann die Möglichkeit, einen günstigen Alternativprovider zu finden und diesen wieder an die TK-Anlage zu übermitteln. Diese Vorgehensweise setzt natürlich voraus, daß der LCR-Server erkennt, zu welcher ursprünglichen Anfrage Reroute-Nachrichen gehören.
Schlägt der Verbindungsaufbau über alle vom LCR-Server vorgeschlagenen Provider fehl, so erhält der Anrufer von der TK-Anlage ein endgültiges (Gassen-)Besetztzeichen.

4.2 Architekturmodell

Der LCR-Server sollte (ggf. über mehrere Indirektionsstufen) über das JTAPI mit der TK-Anlage kommunizieren. Die von Bosch Telecom gestellte TK-Anlage Integral Communication Center ließ sich jedoch nur über Novells TSAPI ansteuern. Da JTAPI noch recht neu ist, ließ sich keine Implementation finden, die auf dieser Schnittstelle aufsetzt und damit die Anlage unter Java nutzbar macht. Deshalb mußten wir selbst eine solche Umsetzung realisieren. Aufgrund der fortgeschrittenen Zeit wurde jedoch diesbezüglich nur die für LCR absolut nötige Funktionalität (Details siehe Abschnitt 4.7.1) implementiert.
TSAPI stellt nur eine C/C++-Schnittstelle zur Verfügung. Deshalb mußte ein Weg gefunden werden, diese Schnittstelle aus einem Java-Programm heraus netzweit anzusprechen. Dafür wurde - vor allem zu Lehrzwecken - CORBA [W15] gewählt. Als Alternative wäre auch eine Anbindung über das Java Native Interface (JNI) denkbar, wobei die Netzwerkfähigkeit dann von TSAPI bereitgestellt werden kann. Damit ergibt sich insgesamt das in Abbildung 2 gezeigte Schichtenmodell.

4.2.1 TK-Anlage

Die TK-Anlage (ICC) wurde mit der Konfigurationssoftware und einer TSAPI-C-Bibliothek von Bosch Telecom gestellt. Die zugehörigen Include-Dateien wurden von Novells Webserver kopiert. Das ICC besteht aus zwei wesentlichen Komponenten:

- der Voice-Switch bietet die Telefonfunktionen einer modernen ISDN-Anlage und
- ein INTEL Pentium-PC mit Windows NT 4.0 Workstation als Betriebssystem fungiert als CTI-Applikationsserver zur softwaremäßigen Ansteuerung des Switch und zum Erbringen der CT-Dienste.

Beide, Switch und Rechner, verwenden als Kommunikationslink Ethernet, so daß die Kommunikation in einem lokalen LAN-Segment stattfinden kann.
Auf dem Switch selbst startet nach dem Einschalten ein Chorus-Mikrokernel, der nach dem Booten die eigentliche Anlagensoftware von einer lokalen Platte lädt und die Anlage in einem definierten Ausgangszustand für den Kunden bereitstellt. Anschließend besteht die Möglichkeit, das ICC mit kundenspezifischen Konfigurationsdaten anzupassen und es softwaremäßig in CTI-Applikationen einzubinden.
Die Kommunikation zwischen Voice-Switch und PC erfolgt in beiden Richtungen über Protokollstacks, welche die TSAPI-Aufrufe in CSTA-konformen (Computer Supported Telephony Application) [5] Paketen assemblieren und disassemblieren. Die Codierung der Datentypen und -inhalte erfolgt in ASN.1 [W16,W17,6]. Diese Daten werden anschließend in TCP/IP-Pakete verpackt und über das Ethernet

verschickt. Die eigentlichen CT-Dienste werden von einem auf dem Applikationsserver des ICC laufenden Serverprozeß weiteren Clients im LAN des Kunden bereitgestellt.

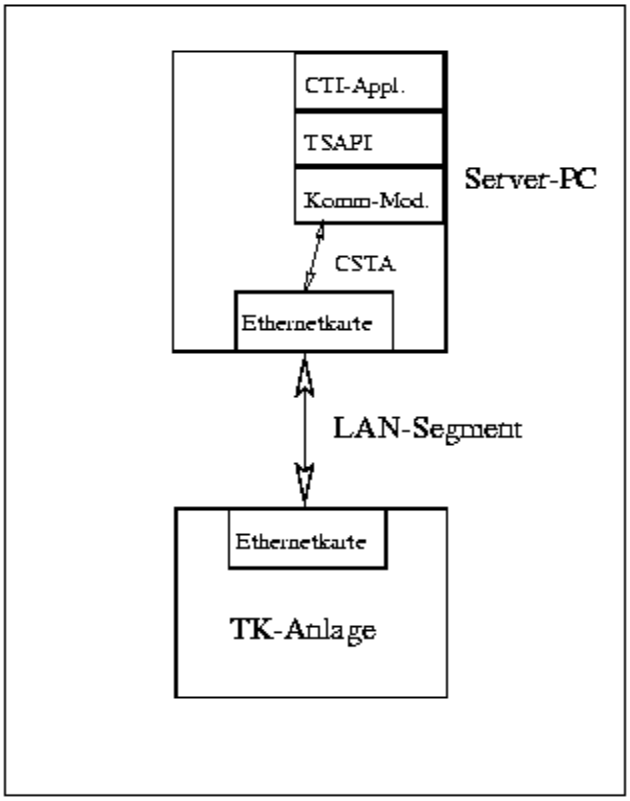


Abbildung 6: LAN-Einbindung des ICC

Die Kommunikation zwischen Voice-Switch und Applikationsserver basiert somit auf dem von der ECMA verabschiedeten Standard CSTA und Novells proprietären Standard TSAPI.

4.2.2 CSTA und TSAPI

CSTA beschreibt informell Dienste, die eine TK-Anlage und ein Rechner sich gegenseitig anbieten müssen, um CTI zu verwirklichen. So muß eine TK-Anlage dem Rechner u.a. den Dienst bereitstellen, eine Dreierkonferenz aufzubauen. Umgekehrt muß der Rechner z.B. den Dienst anbieten, ein Gespräch zu routen.
TSAPI ist eine Programmierschnittstelle (API), welche den informellen CSTA-Standard implementiert, d.h. mit der TSAPI-Schnittstelle ist es möglich, die TK-Anlage anzusteuern, um sie in CTI-Applikationen zu nutzen.
In unserem konkreten Fall bedeutet dies, daß der in Abschnitt 4.2.1 genannte Serverprozeß einen TSAPI-Server darstellt, der nach dem TSAPI-Standard das ICC ansteuert. Nach außen hin entspricht das ICC in dieser Betrachtungsweise dem TSAPI-Standard. Die Clients im LAN des Kunden werden dabei so konfiguriert, daß TSAPI-Aufrufe automatisch zur Bearbeitung an den Applikationsserver weitergeleitet und dort verarbeitet werden.

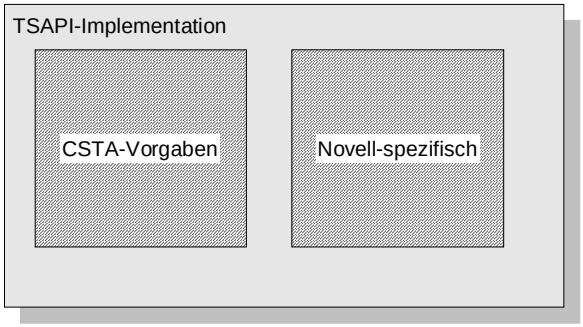


Abbildung 7: CSTA und TSAPI

4.3 Netzwerkfähigkeit von CTI-Server und CTI-Client

4.3.1 TSAPI

Zurückkommend auf die eigentliche Problematik des Least Cost Routing, wird ein Router in Form von Software benötigt, der mit dem ICC via TSAPI-Server kommunizieren kann. Diese Software muß nicht auf dem gleichen Rechner laufen, der den TSAPI-Server beherbergt, sondern sie kann sich auf einem Client-Rechner in einem beliebigen Rechnernetz – LAN oder WAN – befinden.

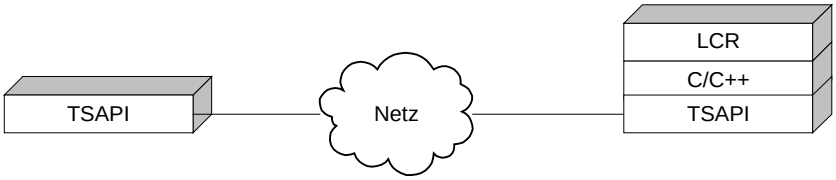


Abbildung 8: Netzwerkansatz von TSAPI

4.3.2 Realisierung einer netzwerkfähigen JTAPI-Version über CORBA

Die Einführung deutet aber bereits an, daß der Router der JTAPI-Schnittstelle entsprechen sollte. Das bedeutet für die aktuelle Konfiguration, daß auf dem Server- oder Client-Rechner eine Umsetzung von JTAPI nach TSAPI stattfinden muß.

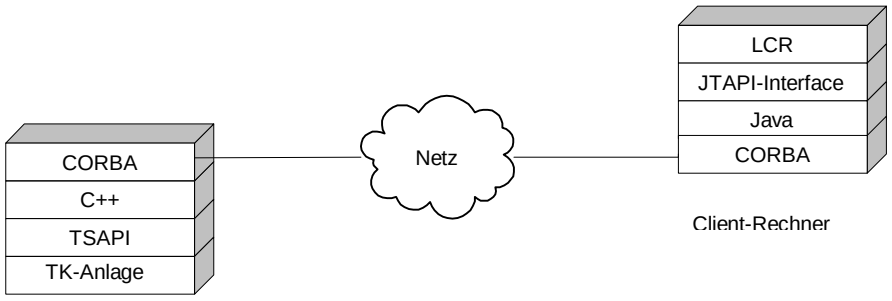


Abbildung 9: Netzwerkfähige JTAPI-Lösung mit CORBA

Zu Lehrzwecken wurde das Problem im Praktikum mit CORBA (Common Object Request Broker Architecture) gelöst [W15]. CORBA-Systeme stellen mit Hilfe eines ORB (Object Request Broker) Objekte unabhängig von der konkreten Programmiersprachanbindung, z.B. C++ und Java, dem Anwender zur Programmierung verteilter Anwendungen ortstransparent zur Verfügung. Ein Objekt, das in einem in C++ geschriebenen Programm erzeugt wurde, kann in einem Java-Programm angesprochen werden und umgekehrt. Außerdem besteht die Möglichkeit, daß das CORBA-System bei angebotenen Diensten vollständig von bestimmten Objekten, die diesen Dienst an bieten, abstrahiert. Ein Client

beschreibt lediglich die benötigten Dienstmerkmale und erhält ein beliebiges Objekt, das dieser Spezifikation genügt, vom CORBA-System zurück.
Genau diese Funktionaltät wird auch bei der JTAPI-TSAPI-Umsetzung benötigt. Auf C++-Seite wird die Anlage angesteuert. Objekte, die direkten Zugriff auf das ICC haben, werden via CORBA bereitgestellt, so daß sie auf Java-Seite angesprochen werden können.
Diese Teilung in C++ und Java wirft die Frage auf, wo die Verwaltung der Zustandsdaten der JTAPI-Implementation stattfinden soll. Es gibt zwei Philosophien. Die eine besagt, daß die Verwaltung auf C++-Seite stattzufinden habe. Der Vorteil ist, daß auf Java-Seite direkt über CORBA ein JTAPI-konformes Objekt zur Verfügung steht, das Java-seitig keinen Aufwand mehr erzeugt. Die andere Vorstellung geht den umgekehrten Weg und setzt den Schwerpunkt der Implementation auf Java-Seite. Diese Lösung ist nicht ganz in Java zu erzielen, da die Anlagenansteuerung schlußendlich via TSAPI in C++ erfolgen muß. Der Vorteil dieser heterogenen zweiten Lösung ist die höhere Performanz. Werden die Objekte auf Java-Seite gehalten, muß nicht bei jedem Zugriff auf ein solches Objekt eine CORBA-Verbindung über das Netz stattfinden. In der hier vorliegenden Realisation sind beide Herangehensweisen anzutreffen. Dies liegt nicht zuletzt darin begründet, daß mehrere Entwicklungsgruppen kooperierten und keine expliziten Vorgaben bezüglich der Vorgehensweise gemacht wurden.
Da komplexere Datentypen, Objekte und Zugriffsmethoden auf Objekte, wie sie von JTAPI benötigt werden, CORBA nicht von vornherein bekannt sind, müssen die Datenstrukturen in einer programmiersprachenunabhängigen Weise definiert werden. Dazu dient die Interface Definition Language (IDL). Die Notation erinnert stark an C++, darf aber mit diesem nicht verwechselt werden. Alle Datentypen, Methoden und Objekte, die für die Kommunikation über das CORBA-System benötigt werden, müssen zunächst in IDL definiert werden. Für diverse Programmiersprachen existieren Compiler, die entsprechende Stellvertreterklassen erzeugen. Für die konkrete Implementation werden Klassen mit dem Suffix `skel` (skeleton) erstellt. Eine Implementation wird von dieser Klasse abgeleitet. Für andere Objekte, die auf die CORBA-Implementationen zugreifen müssen, werden Klassen mit dem Präfix oder Suffix `stub` und `helper` erzeugt. Die genannten Hilfsklassen realisieren für den Benutzer transparent den Aufruf von Methoden anderer Objekte, ohne deren genaue Lokation kennen zu müssen. Für den Anwender stellt sich jeder Aufruf wie ein lokaler Methodenaufruf dar. Die Verwendung verteilter Komponenten wird ggf. nur durch Laufzeiten bemerkbar. In der jetzigen Konfiguration ist es möglich, einen Router zu betreiben, der JTAPI-Routing implementiert.
Im Praktikum wurden insgesamt vier verschiedene GUIs und LCR-Komponenten entwickelt. Für die Demonstration auf der CeBIT verwendeten wir die Versionen der Gruppe 4, da diese Gruppe in der letzen Phase des Praktikums für die Routing-Anbindung des LCR-Moduls an die TK-Anlage verantwortlich war und die Integration in eine Gesamtlösung sich deshalb mit deren GUI und LCR-Algorithmus einfacher gestaltete. Die anderen Gruppen entwickelten ebenfalls voll funktionsfähige Komponenten, die aber noch nicht auf das korrekte Zusammenspiel mit der vorliegenden Lösung getestet wurden. Im folgenden wird deshalb exemplarisch die LCR-Lösung und GUI der Gruppe 4 beschrieben.

4.4 Der LCR-Server (Router)

Die beiden Hauptaufgaben des LCR-Servers liegen in der Verwaltung der Konfigurationsdaten der TK-Anlage, die durch das GUI verändert werden können, und in der Beantwortung der Routing-Anfragen der TK-Anlage.
Die LCR-Applikation besteht entsprechend dem Client/Server-Modell aus zwei Teilen. Dies sind im einzelnen der Router und die graphische Wartungsoberfläche / GUI (*LCRMonitor*), wobei der Router den Server und die GUI den Client repräsentiert. Der Router ist somit für die eigentliche Routing-Funktionalität - also die Kommunikation mit der Telefonanlage über JTAPI - und das Speichern der Konfigurationsdaten verantwortlich. Die GUI ist für die Anzeige von Ereignissen, der Konfiguration sowie die Veränderung der Konfigurationsdaten durch den Benutzer und die Überprüfung der Veränderungen zuständig.
Die Konfigurationsinformationen des ICC weisen eine hohe Komplexität auf, die sich auch in den Datenstrukturen, in denen die Konfigurationsdaten der LCR-Applikation gespeichert werden, widerspie-

gelt. Sie versuchen die Objekte, die das ICC intern für seine Arbeit benutzt, weitgehend nachzumodellieren. Die Datenstrukturen setzen sich zusammen aus Informationen über die

- Dienste, die verfügbar sind (Sprache, Fax, Daten),
- Zugangsverfahren (Ausscheidungskennziffern) zu einzelnen Diensten (Trunk Line Groups),
- externen Anschlüsse der TK-Anlage (Bundles),
- die einzelnen Benutzer (Users) und
- Berechtigungsdaten (Traffic Groups).

Zusätzlich werden für das LCR der TK-Anlage Daten über die verschiedenen Anbieter und deren Tarife benötigt. Diese Daten werden vom LCR-Server mittels der von Java bereitgestellten Objektserialisation persistent auf dem Dateisystem des Rechners gespeichert.

4.4.1 Objekte zur Anlagenverwaltung und zugehörige Datenstrukturen

Alle Daten sind durch das Objekt *LCRData* zugänglich. Es verwaltet folgende Objekte:

- *TrunkLineGroup,*
- *Bundle,*
- *Service,*
- *User,*
- *TrafficGroup,*
- *Provider.*

Ein *Bundle* repräsentiert mehrere physikalisch nach außen führende Leitungen der TK-Anlage (z.B. Amtsanschlüsse), welche die gleichen Eigenschaften besitzen. Im Kontext des LCR bedeutet dies insbesondere, daß alle Leitungen eines *Bundles* nach dem gleichen Tarifmodell abgerechnet werden müssen. Zwischen *Bundles* und *Providern* besteht in den Datenstrukturen eine 1:1-Beziehung, was für die meisten Anwendungen ausreichend ist. In den *Bundle*-Objekten wird lediglich der Name und die Rufnummer des *Bundles*, unter dem es vom Switch angesprochen wird, gespeichert.
Im *Provider*-Objekt werden Informationen über den *Provider* und vor allem über das Verfahren, mit dem dieser *Provider* genutzt werden kann, festgehalten. Der *Provider* wird über einen Namen und eine ID identifiziert. Für *Dial-in-Provider*, die Leitungen anderer TK-Netzbetreiber benötigen, weil sie über kein eigenes Festnetz bis zum Kunden verfügen, wird die "Call by Call"-Rufnummer und für Provider, die eigene Leitungen zum Kunden besitzen, die *Bundle*-Nummer gespeichert. Weiterhin kann die interne LCR-Komponente der TK-Anlage die vom Benutzer gewählte Rufnummer mit speziellen Algorithmen modifizieren. Die Konfiguration dieser Funktionalität wird von der hier entwickelten externen Lösung bereitgestellt. Die Parameter zu diesen Algorithmen werden ebenfalls im *Provider*-Objekt gespeichert.
Die *TrunkLineGroup*-Objekte enthalten Daten über die einzelnen *Trunk Line Groups* (TLG) der TK-Anlage. Im wesentlichen sind dies ein Name, die Rufnummer der TLG und die Dienste, die auf dieser TLG genutzt werden können. Darüber hinaus werden auch LCR-Informationen für die einzelnen TLGs gespeichert. Diese bestehen aus einem *Default-Provider* und einer *Routing*-Tabelle. Der Default-Provider wird benutzt, wenn LCR nicht möglich ist oder kein befriedigendes Ergebnis gebracht hat.
Die Routing-Tabelle wird in einem *RoutingTable*-Objekt verwaltet. Dieses besteht wiederum aus einzelnen *RoutingTableEntry*-Objekten. In diesen sind die eigentlichen LCR-Informationen gespeichert. Dazu enthält jedes *RoutingTableEntry*-Objekt den Zeitraum, in dem es gültig ist. Dieser kann mittels Wochentag- und Uhrzeitangaben minutengenau angegeben werden. Weiterhin ist der Präfix des Rufnummernbereichs, für die dieser Eintrag zuständig ist, angegeben. Zu diesem Zeitraum und dem Präfix sind dann der günstigste und zweitgünstigste *Provider* gespeichert. Durch die Verwendung von Präfixen läßt sich die Größe von Routing-Tabellen stark reduzieren.
Weiterhin enthält jede TLG noch ein *RoutingData*-Objekt, das zusätzliche LCR-Einstellungen ermöglicht. Darin ist das für die LCR-Berechnung zu benutzende Modul (die Klasse, die den Routing-Algorithmus implementiert) angegeben und es kann *Rerouting* aktiviert bzw. deaktiviert werden. Im zweiten

Fall wird bereits nach dem Fehlschlagen des ersten Verbindungsaufbaus über den Primärprovider ein Besetztzeichen erzeugt.
Die Berechtigungen der Endgeräte werden in der TK-Anlage durch *User*-Objekte und über Beziehungen zwischen *Traffic Groups* repräsentiert. Jedem Endgerät kann (mindestens) ein *User*-Objekt zugeordnet werden, in dem ein Name und die Rufnummer des Geräts gespeichert ist. Endgeräte mit gleichen Berechtigungen werden dann zu *Traffic Groups* zusammengefaßt. In jeder *Traffic Group* ist festgelegt, zwischen welchen *Traffic Groups* (inklusive der eigenen) die Endgeräte Verbindungen aufbauen dürfen. Zusätzlich kann jeder *Traffic Group* eine TLG zugeordnet werden, die z.B. für externe Verbindungen genutzt werden darf. Die Abhängigkeiten bilden einen gerichteten Graph, der allerdings nach Diensttypen getrennt aufgebaut wird. Es existiert also für jeden Dienst ein eigener Graph und ein *TrafficGroup*-Objekt ist somit nur für einen Dienst gültig.

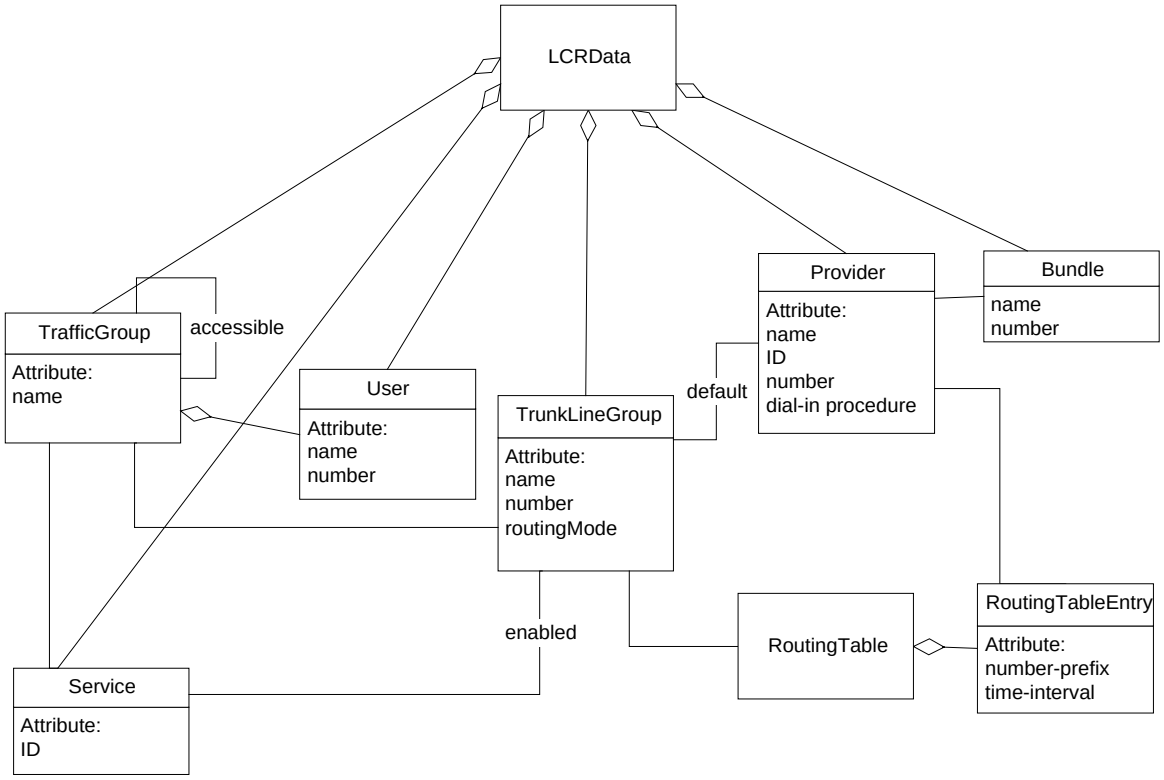


Abbildung 10: Klassendiagramm der Router-Datenstrukturen

4.4.2 Struktur einer JTAPI-Implementation

Die Initialisierung einer JTAPI-Implementation geschieht im allgemeinen über die *JtapiPeerFactory*-Klasse. Da auf einem Rechner JTAPI-Implementationen von verschiedenen Herstellern installiert sein können (z.B. zur Ansteuerung verschiedener angeschlossener TK-Anlagen), wird mittels dieser Klasse unter Angabe eines eindeutigen Namens eine Auswahl getroffen. Die *JtapiPeerFactory* instanziiert für eine Applikation, die Telefoniefunktionalität nutzen möchte, ein Objekt einer Klasse, die das Interface *JtapiPeer* implementiert und liefert es als Referenz auf die reale JTAPI-Implementation zurück. Zur Garantie der Eindeutigkeit legt JTAPI fest, daß der Name als Konkatenation von Internet-Domainname des Herstellers in inverser Notation und einem frei wählbaren Suffix gebildet wird. Die von uns realisierte Implementation trägt somit den Namen:

de.tu_darmstadt.informatik.isa.boschpabx.boschpabx.

Über diese Referenz wird schließlich auf Provider für verschiedene Diensttypen zugegriffen. Unter anderem kann, sofern implementiert, auch ein *CallCenterProvider*-Objekt angefordert werden kann. Dieses bietet schließlich den Ausgangspunkt, von dem die gesamte Funktionalität einer JTAPI-Implementation zugänglich ist.

4.4.3 Routing unter JTAPI

Um zu erläutern, wie der LCR-Server und die TK-Anlage bei der Beantwortung von Routing-Anfragen kooperieren, muß zunächst auf die JTAPI-Schnittstelle eingegangen werden. Für Routing sind in JTAPI einige *Interfaces* im Modul *Call Center* vorgesehen. Dies sind im einzelnen *CallCenterProvider*, *RouteAddress*, *RouteCallback* und *RouteSession*. Weiterhin existieren noch einige spezielle Objekte, die Informationen über Ereignisse kapseln.
Die JTAPI-Implementation muß zunächst einmal wissen, für welche Anrufe Routing aktiviert werden soll und welche Applikation dafür zuständig ist. Dazu existiert im *RouteAddress*-Interface, welches die Rufnummer eines Endgeräts repräsentiert, eine Methode, mit der sich eine Applikation für das Routing zu dieser (externen) Zielnummer registrieren kann. Dies bedeutet, daß die JTAPI-Implementation für jede Verbindung, die zu dem Endgerät mit dieser Rufnummer aufgebaut werden soll, eine Routing-Anfrage an die registrierte Applikation stellt. Sollen generell alle Verbindungen vom LCR-Server geroutet werden, registriert sich die Applikation auf einer speziellen Default-Adresse, die alle möglichen Ziele bezeichnet.
Damit die JTAPI-Implementation diese Anfragen an die Applikation übermitteln kann, muß die Applikation das *RouteCallback*-Interface implementieren und bei der Registrierung eine Referenz darauf an die JTAPI-Implementation übergeben. In diesem Interface gibt es für jedes Ereignis, das beim Routing auftreten kann, eine Methode, die dann von der JTAPI-Implementation mit entsprechenden Parametern aufgerufen wird. Die Applikation selbst muß diese Methoden geeignet implementieren.
Der Routing-Vorgang für einen einzelnen Anruf ist nicht mit einer einzelnen Anfrage und deren Antwort abgeschlossen. Er durchläuft mehrere Zustände und jeder Zustandswechsel innerhalb der JTAPI-Implementation wird als Ereignis an das *RouteCallback*-Interface übermittelt. Die einzelnen Ereignisse müssen also einem bestimmten Verbindungsaufbau zugeordnet werden. Dazu wird jeder Vorgang durch ein *RouteSession*-Objekt repräsentiert, welches die Zustandsinformationen enthält und bei Auftreten eines *Route*-Ereignisses (als Initialkennzeichen einer neuen Verbindung) generiert wird. Dieses Objekt stellt auch die Methoden *selectRoute()* und *endRoute()* bereit, mit denen die Applikation die Antwort an die JTAPI-Implementation übermitteln kann. Dazu wird bei jedem Aufruf einer Methode des *RouteCallback*-Interfaces eine Referenz auf das zugehörige *RouteSession*-Objekt mitgegeben.
Die einzelnen Zustände einer *RouteSession* sind *Route*, *Reroute*, *RouteUsed*, *RouteEnd* und *RouteCallbackEnded*. Dementsprechend gibt es im *RouteCallback*-Interface die Methoden *routeEvent()*, *rerouteEvent()*, *routeUsedEvent()*, *routeEndEvent()* und *routeCallbackEndedEvent()*. Dabei wird *routeEvent()* zu Beginn eines Routing-Vorgangs aufgerufen. Die Applikation antwortet darauf mit einer Zielnummer für den entsprechenden Anruf (*RouteSession.selectRoute()*). Ist das Routing zu dieser Zielnummer nicht durchführbar, so wird von der JTAPI-Implementation die Methode *rerouteEvent()* aufgerufen. Die Applikation kann dann eine alternative Zielnummer ermitteln und diese wieder an die JTAPI-Implementation zurückgeben. Kann eine dieser Zielnummern benutzt werden, so wird dies der LCR-Applikation durch den Aufruf der Methode *routeUsedEvent()* signalisiert. Nach Bearbeitung des Vorgangs wird schließlich noch die Methode *routeEndEvent()* aufgerufen. Falls die Registrierung der Applikation auf einer Rufnummer für Routing von der TK-Anlage aufgehoben wird, so ruft die JTAPI-Implementation die *routeCallbackEndedEvent()*-Methode auf. Die Vorgehensweise der Abarbeitung einer Routing-Anfrage erfolgt dabei weitgehend wie in Abschnitt 4.1 (Abbildung 5) beschrieben.

4.4.4 Realisierung des JTAPI-Routing im LCR-Server

Der LCR-Server implementiert das *RouteCallback*-Interface mit der Klasse *Callback*. Darin findet die Bearbeitung der Ereignisse und nochmals eine eigene Verwaltung der Vorgänge (*Session*) statt. Dies ist nötig, da es bei einem *Reroute*-Ereignis nicht möglich ist, die Rufnummer zu ermitteln, die der Anrufer ursprünglich gewählt hat. Sie muß also beim Auftreten eines *Route*-Ereignisses für die spätere Verwendung gespeichert werden. Zudem erlaubt die eigene Vorgangsverwaltung ein einfaches Protokollieren der Bearbeitungsvorgänge für die einzelnen Anrufe.
Die Vorgänge werden durch die Klasse *Session* repräsentiert. Sie speichert die vom Anrufer gewählte Nummer und die Zielnummern, welche die Applikation an die JTAPI-Implementation übergeben hat.

Die *Callback*-Klasse erzeugt beim Auftreten eines *routeEvent* ein neues *Session*-Objekt und speichert die vom Anrufer gewählte Rufnummer darin. Anschließend wird diese Rufnummer an das LCR-Modul übergeben, um die von der Anlage endgültig zu wählende Zielnummer zu berechnen. Diese Zielnummer wird dann ebenfalls in dem *Session*-Objekt gespeichert und mittels *selectRoute()* an die TK-Anlage weitergegeben. Konnte keine Zielnummer ermittelt werden, so bricht der LCR-Server den Vorgang durch *endRoute()* ab.
Bei einem *reRouteEvent* muß zunächst das zugehörige *Session*-Objekt identifiziert werden. Aus diesem kann die vom Anrufer gewählte Nummer ermittelt und an das LCR-Modul übergeben werden. Dieses berechnet dann eine alternative Zielnummer, welche wieder im *Session*-Objekt eingetragen und an die TK-Anlage übermittelt wird.

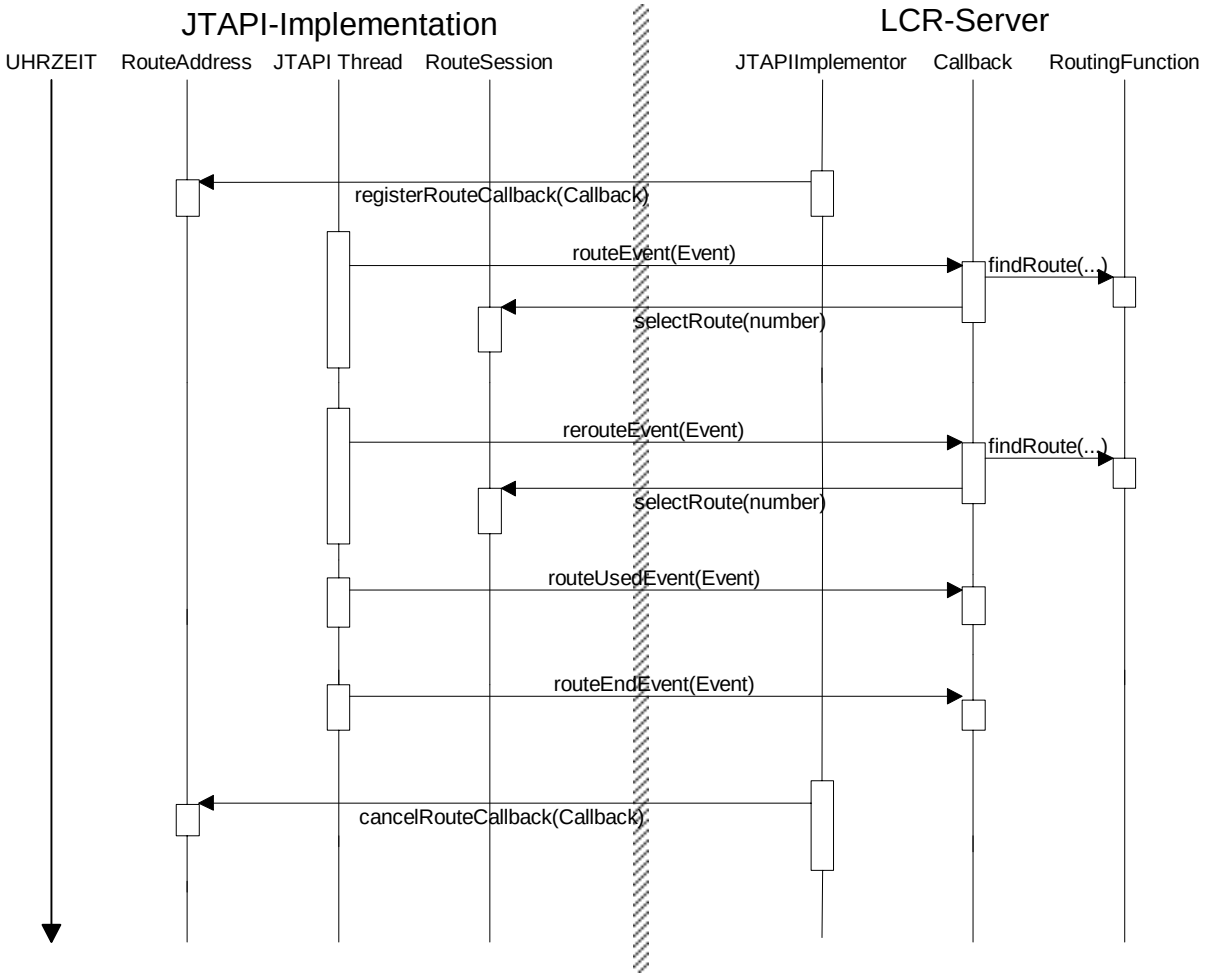


Abbildung 11: Kommunikation zwischen JTAPI-Implementation und LCR-Server

Auf ein *routeUsedEvent* reagiert der LCR-Server lediglich mit einem Eintrag in einer Protokolldatei (*Log)*. Zunächst wird jedoch wieder das *Session*-Objekt identifiziert, aus dem die Daten für den Eintrag ausgelesen werden.

Tritt schließlich ein *routeEndEvent* auf, so wird erneut die zugehörige *Session* herausgesucht und ein Eintrag im *Log* vorgenommen, falls dies nicht schon bei einem *routeUsedEvent* geschehen ist. Danach kann das *Session*-Objekt gelöscht werden.

Bevor die JTAPI-Implementation die Methoden in der *Callback*-Klasse aufrufen kann, muß sie jedoch initialisiert werden und der LCR-Server muß sich über das *RouteAddress*-Interface für Routing registrieren.

4.4.5 Simulationsschnittstelle zur Generierung von Routing-Ereignissen

Die JTAPI-Initialisierung wird im realisierten LCR-Server von einem zusätzlichen Java-Interface *JTAPIInterface* gekapselt. Zu diesem existieren in Anlehnung an das *Bridge*-Entwurfsmuster [7] das *APIImplementor*-Interface und dessen Implementationen *DebugImplementor* und *JTAPIImplementor*. Durch diese Architektur ist der LCR-Server nicht statisch an das JTAPI gebunden, sondern könnte durch Hinzufügen eines neuen Implementors auch mit anderen Telefonie-APIs arbeiten. Das wichtigere Argument für diese Architektur war jedoch, daß der LCR-Server auf einfache Weise vom JTAPI getrennt getestet werden konnte. Um für die Programmtests nicht direkt mit der TK-Anlage kommunizieren zu müssen, wurde der *DebugImplementor* realisiert, der in einem eigenen Thread aus der Datei *test.txt* TLG- und Rufnummern ausliest und mit diesen Routing-Ereignisse für den LCR-Server generiert. Die *JTAPIImplementor*-Klasse stellt dagegen eine echte JTAPI-Anbindung dar. Sie übernimmt die Initialisierung der JTAPI-Implementation über die JtapiPeerFactory und registriert den LCR-Server für Routing-Ereignisse bei der TK-Anlage.

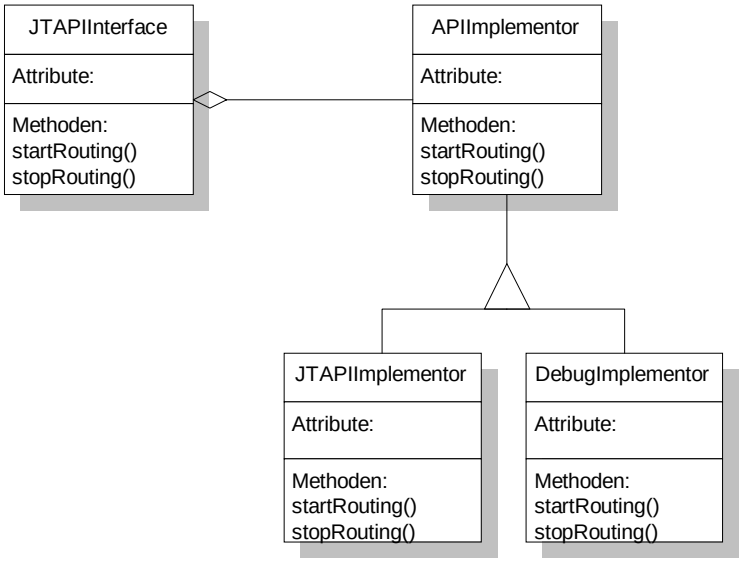


Abbildung 12: Verwendung unterschiedlicher Routing-Clients

4.4.6 Austauschbare Routing-Module

Die eigentliche LCR-Entscheidung wird im LCR-Server von einem Routing-Modul getroffen. Dieses ist – auch während der Laufzeit des Servers – beliebig austauschbar. Ein Routing-Modul muß das *RoutingFunction*-Interface implementieren. Dieses Interface enthält u.a. eine *findRoute()*-Methode, in der jedes Modul seine Routing-Funktionalität implementiert und die ermittelte Route als Rückgabewert übergibt. Als Parameter bekommt die *findRoute()*-Methode ein *RoutingInfo*-Objekt. Dieses enthält die angewählte TLG, die vom Anrufer gewählte Rufnummer, den aktuellen Zeitpunkt und einen booleschen Wert der angibt, ob in der aktuellen Anfrage die Alternativroute für Rerouting ermittelt werden soll (z.Z. wird aus zwei Providern ausgewählt, so daß ein boolescher Wert zur Festlegung des Auswahlkriteriums genügt). Durch das Zusammenfassen aller Parameter in der *RoutingInfo*-Klasse, kann die Schnittstelle leichter um zusätzliche Parameter ergänzt werden.
Der LCR-Server untersucht alle Java-Klassendateien (*.class*) im Unterverzeichnis *functions*, ob sie das *RoutingFunction*-Interface implementieren. Diejenigen, für die dies zutrifft, werden als Routing-Module betrachtet und jeweils eins kann über das GUI für die entsprechenden TLGs aktiviert werden. Die *Callback*-Klasse leitet dann alle Routing-Anforderungen an dieses Modul weiter.
Derzeit existieren zwei beispielhafte Implementationen eines Routing-Moduls:

- *RandomRoutingFunction* und
- *DefaultRoutingFunction*.

Die Klasse *RandomRoutingFunction* dient nur zur Demonstration der Austauschbarkeit der Module. Sie ermittelt für jede Routing-Anfrage ein zufälliges Ziel.
Die *DefaultRoutingFunction*-Klasse implementiert LCR-Funktionalität, wie sie auch in leistungsfähigeren LCR-Geräten realisiert ist. Sie nutzt die Routing-Tabellen aus den vom LCR-Server verwalteten Konfigurationsdaten der TK-Anlage, die über das GUI verändert werden können. In der *findRoute()*-Methode wird zunächst aus der übergeben TLG die Routing-Tabelle angefordert. Die Tabelle wird nach einem Eintrag durchsucht, dessen Präfix zu der gewählten Rufnummer paßt und dessen Zeitrahmen den aktuellen Zeitpunkt einschließt. Falls mehrere Einträge zutreffen, wird derjenige mit dem längsten Präfix gewählt. Ist kein passender Eintrag vorhanden, so wird der *Default-Provider* der TLG benutzt. Ansonsten wird abhängig davon, ob es sich um eine Route- oder Reroute-Anforderung handelt, die *Bundle*-Nummer des Primär- oder Sekundärproviders zurückgegeben.
Die Manipulationsalgorithmen sind z.Z. nur rudimentär implementiert und ersetzen die Rufnummer durch die *Bundle*-Nummer, die zur Simulation einer Amtsleitung auf eine interne Nebenstellennummer abgebildet wird. Diese Vorgehensweise war für das Praktikum ausreichend und stellt prinzipiell keine Einschränkung der Routing-Funktionalität dar. Die Manipulationsverfahren sind elementare Operationen auf Strings, deren Schnittstelle bereits realisiert wurde. Der Zugriff auf reale Amtsleitungen wird über das GUI durch die konkrete Zuordnung der Bundles zu internen (System-)Rufnummern der TK-Anlage konfiguriert.

4.4.7 Die Protokolldatei

Der LCR-Server speichert Statusinformationen und Fehlermeldungen in einer Protokolldatei mit dem Namen *lcr.log*. Zudem werden in dieser Datei detaillierte Informationen über alle Routing-Vorgänge festgehalten, so daß die Korrektheit der Routing-Tabellen und die Effektivität des LCR anhand dieser Daten überprüft werden kann.
Realisiert ist dies mittels einer *LogFile*-Klasse. Sie bietet Methoden zum Anfügen von Strings und zum Löschen der Daten. Angefügte Daten werden automatisch mit einem Zeitstempel versehen und in der Datei persistent gespeichert. Darüber hinaus kann bei der Initialisierung des Objekts die Anzahl der Einträge in der Protokolldatei beschränkt werden. Wenn die Grenze überschritten wird, löscht die LogFile-Klasse den jeweils ältesten Eintrag.

4.4.8 Die Schnittstelle zum GUI

Die Trennung von GUI und Administrationsprogramm der LCR-Daten und insbesondere die ausschließliche Verantwortung für die Speicherung durch letzteres erfordert, daß Konfigurationsdaten (über das Netz) vom GUI-Modul gelesen und wieder gespeichert werden können. Dazu wird Javas *Remote Method Invocation* (RMI) eingesetzt, welches den transparenten Aufruf von Methoden über ein Netz erlaubt.
Der LCR-Server stellt die RMI-Schnittstelle *RoutingServer* zur Verfügung, über die GUI-Module mit dem Server kommunizieren können (die Funktionalität ähnelt der von CORBA). Diese Schnittstelle erlaubt vor allem das Lesen eines *LCRData*-Objektes aus dem LCR-Server und das Zurückschreiben eines (in der GUI) veränderten *LCRData*-Objektes. Eine weitere Methode ermittelt alle im LCR-Server installierten LCR-Module, welche die verschiedenen Routing-Algorithmen implementieren. Damit können vom GUI-Modul aus die LCR-Module ausgetauscht werden. Weiterhin kann durch die *RoutingServer*-Schnittstelle mit entsprechenden Befehlen das Routing allgemein aktiviert bzw. deaktiviert und der LCR-Server beendet werden. Schließlich ist es auch möglich die Protokolldatei des Servers, in der Fehler und Routing-Ereignisse aufgezeichnet werden, auszulesen und anzuzeigen.

4.4.9 Die Initialisierungsphase des LCR-Servers

Die *Router*-Klasse enthält die *main()*-Methode und übernimmt die Initialisierung des LCR-Servers. Weiterhin ist *Router* von *UnicastRemoteObject*(aus der RMI-Package) abgeleitet und implementiert das *RoutingServer*-Interface. Damit stellt sie die RMI-Schnittstelle zu den GUI-Modulen zur Verfügung. Die Initialisierungsphase gliedert sich in folgende Abschnitte:

- Initialisierung des RMI,

- Initialisierung der JTAPI-Implementation,
- Laden der Konfigurationsdaten und
- Aktivieren des Routings.

Bei der Initialisierung der RMI-Schnittstelle ist zuerst der *RMISecurityManager* festzulegen. Dieser implementiert eine Standard Sicherheitskonzept (security policy) für die RMI-Applikation. Danach registriert sich der LCR-Server im RMI-Namensdienst unter dem Namen *LCRouter*. Mittels dieses Namens können dann GUI-Module beim Namensdienst IP-Adresse und Port des LCR-Servers ausfindig machen. Dies setzt jedoch voraus, daß vor dem Start des LCR-Servers ein entsprechender Namensdienst geladen wurde, der beispielsweise vom Programm *rmiregistry* aus dem Java Developers Kit (JDK) von Sun realisiert wird.
Als nächstes erfolgt die Initialisierung des JTAPI durch die Klassen *JTAPIInterface* und *JTAPIImplementor*. Dieser Vorgang wurde bereits in den Abschnitten 4.4.2 und 4.4.5 erläutert.
Die Klasse *LCRData* bietet die Methode *load()*, welche die Objektserialisation nutzt, um die Daten aus einer Datei zu lesen. Dabei wird auf die Datei *lcrdata.dat* zugegriffen. Ist diese Datei nicht vorhanden oder schlägt das Laden der Daten aus einem anderen Grund fehl, so werden die Datenstrukturen mit (sinnvollen) Default-Werten belegt.
Schließlich wird noch das Routing mittels des *JTAPIInterface* aktiviert. Dies wird durch die Registrierung der *Callback*-Klasse in der JTAPI-Implementation für die alle Ziele repräsentierende Default-Adresse realisiert.

4.5 Die Architektur des GUI (Klasse *LCRMonitor*)

Das GUI-Modul beinhaltet alle Methoden zum Editieren der Einstellungen und zum Anzeigen der Routing-Informationen. Es benutzt dabei nur grafische Elemente des von Sun mit JDK gelieferten Abstract Windowing Toolkit (AWT) der Version 1.1. Somit ist die Applikation universell über jeden Browser, d.h. ohne zusätzliche Klassen zu starten.

Um die Daten darzustellen, werden folgende Objekte aus der Package AWT benutzt:

- TextField,
- ListBox,
- Choice (Auswahlbox) und
- RadioButtons.

Um Aktionen auszulösen, werden Menü, Menüeinträge und Buttons verwandt. Dabei wird sichergestellt, daß Buttons deselektiert werden, wenn sie momentan nicht einsetzbar sind. Weiterhin muß darauf geachtet werden, wenn sich Daten ändern, auf die andere Objekte angewiesen sind, diese zu aktualisieren. Wird ein Provider entfernt, der durch eine TrunkLineGroup referenziert wurde, so wird eine Warnung ausgegeben und ein Standardprovider selektiert. Dies ist wichtig, damit keine Inkonsistenzen entstehen, die schließlich ein Programm durch fehlerhafte Referenzen abstürzen lassen. Auch wenn nur einzelne Einträge für Objekte verändert werden, muß die Konsistenz gewahrt werden. Wird z.B. der Name eines Providers geändert, so muß eine Auswahlbox, die diese Namen zur Selektion anbietet, entsprechend aktualisiert werden. Zu diesem Zweck wurden verschieden Update-Methoden definiert. Zum Teil werden aber auch nur Initialisierungen benötigt, diese werden von entsprechenden *Init()*-Methoden übernommen. Eine weitere Gruppe von Methoden sind die *Conn()*-Methoden, die von den Eventhandlern bei GUI-Aktionen aufgerufen werden, um entsprechende Aktionen auszuführen. *Update()*-, *Init()*- und *Conn()*-Methoden bilden den größten Teil der Applikation. Zu jedem grafischen AWT-Objekt existiert eine *get()*-Methode, die dafür zuständig ist, daß das Objekt initialisiert wird. Die Daten wurden gemäß der OOP-Philosophie vollkommen gekapselt, so daß sie nur via *get()*- und *set()*-Methoden les- und schreibbar sind.

Beim Entwurf der GUI wurde darauf geachtet, daß alle Fenster einheitlich gestaltet sind, um sich besser zurechtzufinden. Die Fenster öffnen sich entsprechend ihrer Hierarchieebene nach von links oben nach rechts unten, so daß man immer den Überblick behält, auf welcher Ebene man sich befindet.
Schon bei der Eingabe wird überprüft, ob legale Zeichen eingegeben werden, ansonsten werden sie ignoriert. Beim Beenden eines Dialoges mit `OK` werden die Daten nochmals auf ihre Gültigkeit hin überprüft. Sollten sie inkonsistent sein, so wird eine entsprechende Meldung auf dem Bildschirm ausgegeben und der Dialog kann nicht beendet werden. Mit `Cancel` innerhalb jedes Fensters ist es jederzeit möglich, Änderungen und der aktuellen und aller davon abhängigen Ebenen rückgängig zu machen. Dazu wurde eine Containerklasse eingeführt, die alle Daten enthält und die vor Änderungen dupliziert werden kann. Damit ist es möglich, zunächst auf einer Kopie zu arbeiten, ohne an den Originaldaten Änderungen vorzunehmen; erst beim Speichern auf der obersten Ebene werden die veränderten Daten in das Ausgangsobjekt zurückgeschrieben.
Um die Routing-Informationen anzuzeigen, wird anfangs ein Thread gestartet, der die neuen Routing-Informationen beim Routing-Server abfragt und sich dann drei Sekunden schlafen legt, um schließlich wieder Informationen vom Server anzufordern. Sollten Probleme mit dem Zugriff auf die Routing-Komponente auftreten, wird auch hier eine Fehlermeldung ausgegeben.

4.6 Bedienung der GUI

4.6.1 Kompilierung

Als erstes müssen alle *.java*-Dateien übersetzt werden. Danach sind noch Stub- und Skeleton-Dateien für die RMI-Schnittstelle des *RoutingServer*-Interface mit dem Aufruf *rmic Router* zu erzeugen.

4.6.2 Starten der Applikation

Da die Kommunikation der GUI mit dem Router mittels RMI implementiert ist, muß zunächst das Programm *rmiregistry* gestartet werden. Im `CLASSPATH` der *rmiregistry* müssen die Dateien *Router_Stub.class* und *Router_Skel.class* liegen. Danach kann mit *java Router* der Router gestartet werden. Findet dieser ein Konfigurationsfile vor, so ist er sofort betriebsbereit, ansonsten wartet er darauf, daß er mittels der GUI konfiguriert wird. Die GUI kann mit *java LCRMonitor* aufgerufen werden. Da die GUI auch als Applet lauffähig ist, kann man sie auch mit *appletviewer lcrmon.html* starten.

4.6.3 Komponenten der GUI

Das Monitor- und Wartungsprogramm wird entweder komfortabel über einen Java-fähigen Browser gestartet oder über den Java-Interpreter als Applikation.

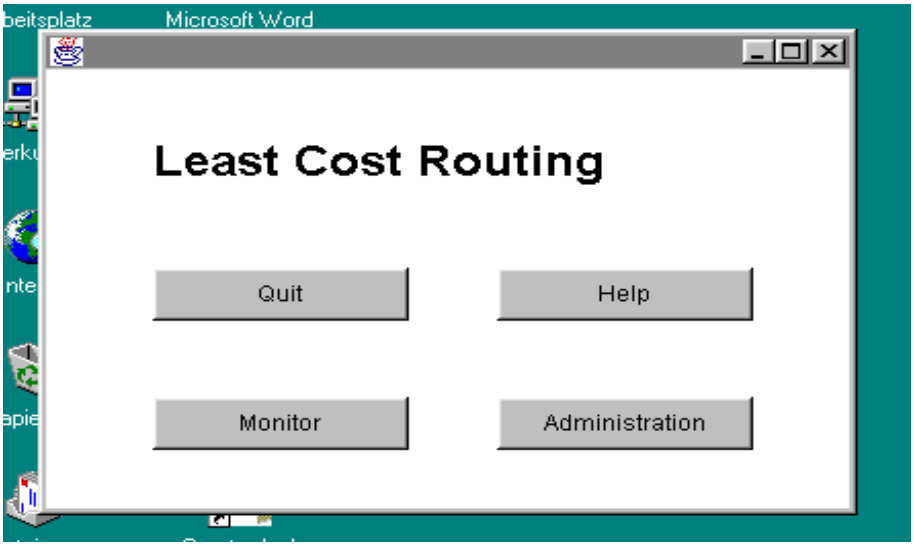


Abbildung 13: Startfenster des GUIs

Beim Startbildschirm (Abbildung 13) besteht die Möglichkeit, im Monitor- oder Wartungsmodus zu starten.

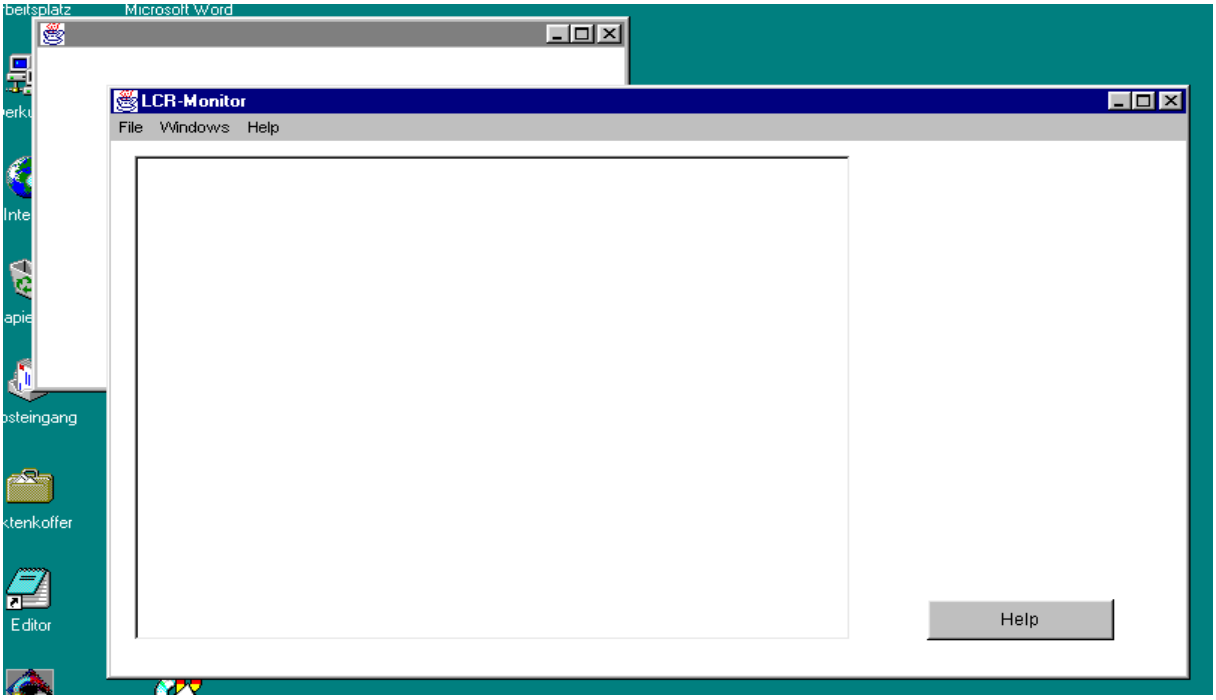


Abbildung 14: Monitoranzeige

Der Monitor-Dialog (Abbildung 14) besteht aus einer Liste, die LeastCostRouting-Informationen anzeigt. Ein Thread aktualisiert alle drei Sekunden die Liste.

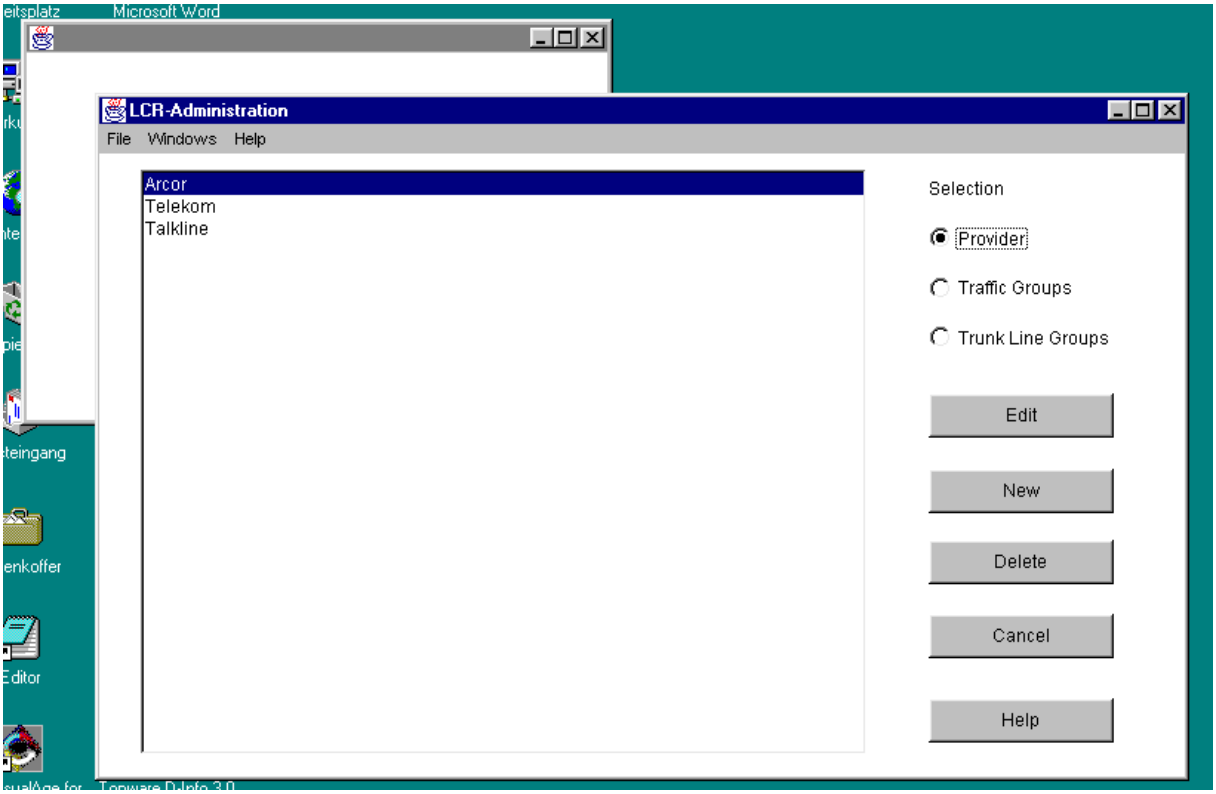


Abbildung 15: Startfenster der Wartungsapplikation

Abbildung 15 zeigt das Kernstück der GUI. Hier ist es möglich, via RadioButtons zwischen Provider, TrafficGroups und TrunkLineGroups zu wechseln, diese zu löschen, zu erstellen oder zu editieren. Dazu steht jeweils ein Button zur Verfügung.

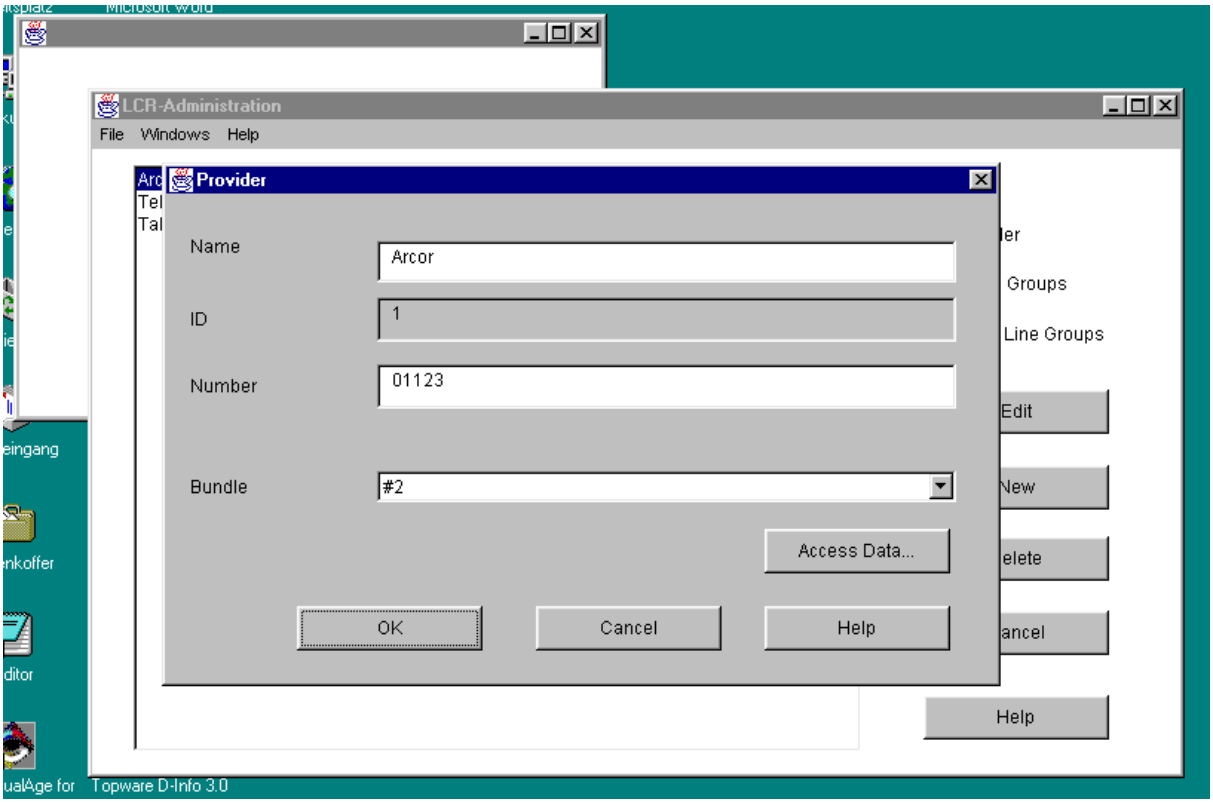


Abbildung 16: Provider-Eingabemaske

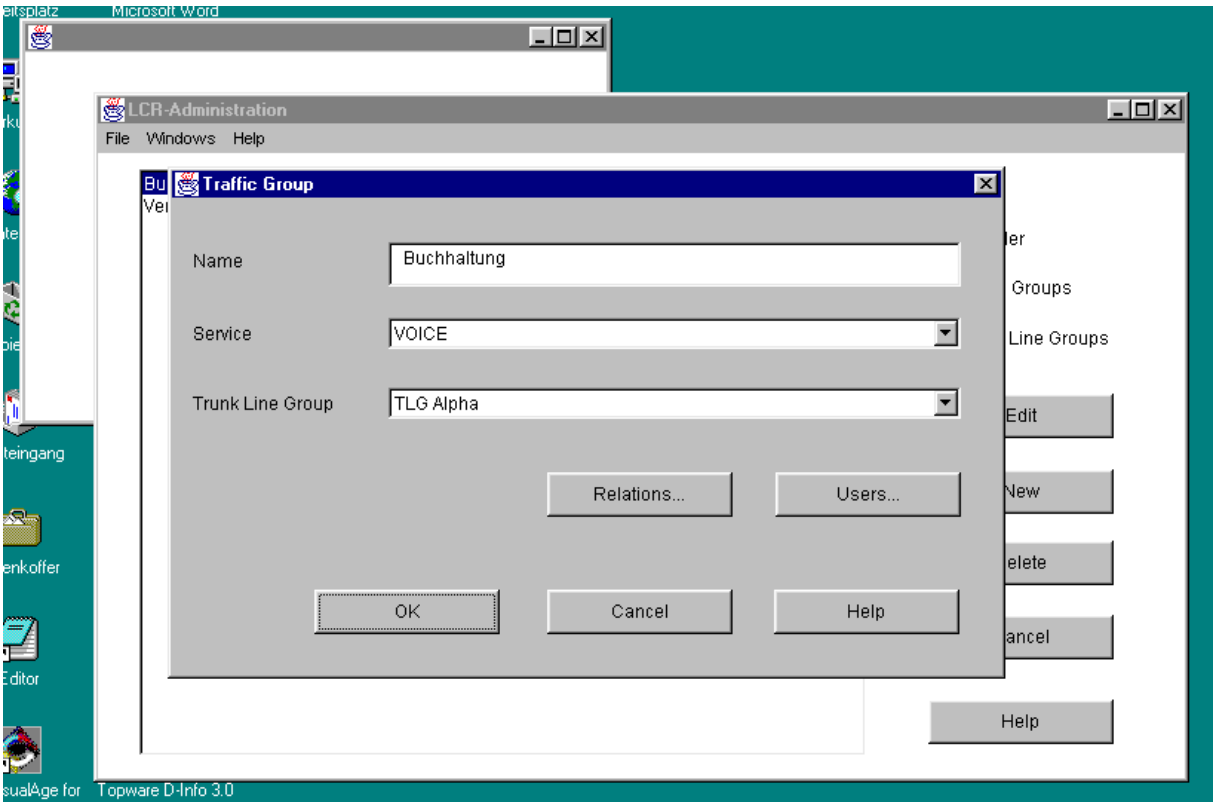


Abbildung 17: TrafficGroup Dialog

In dem in Abbildung 16 gezeigten Fenster ist es möglich, Provider-Daten zu editieren. `ID` läßt sich, wie in der Aufgabenstellung gefordert, nicht editieren, sondern wird eindeutig vergeben. `Name` sowie `Nummer` lassen sich über Tastatur eingeben, wobei bei der Eingabe der Nummer nur Zeichen erlaubt sind, die gültigen Eingaben auf einer Telefontastatur entsprechen. Die Auswahlbox Bundle enthält eine Liste aller Bündel, aus der man eine selektiert. Wird der Button `Access Data...` gedrückt, so kann man weitere Einstellungen am Provider vornehmen. Der Dialog läßt sich nur schließen, wenn die Daten gültig sind. Mit `Cancel` können jederzeit Änderungen widerrufen werden.
Der Dialog TrafficGroup (Abbildung 17) zeigt Name sowie selektierten Service und selektierte TrunkLineGroup an. Der Name muß eindeutig sein, erst dann kann der Dialog mit `OK` beendet werden. Die Buttons `Relations` und `Users` öffnen jeweils einen Dialog (), mit dem es möglich ist, User sowie Beziehungen zu anderen TrafficGroups mit Hilfe zweier Listen auszuwählen.

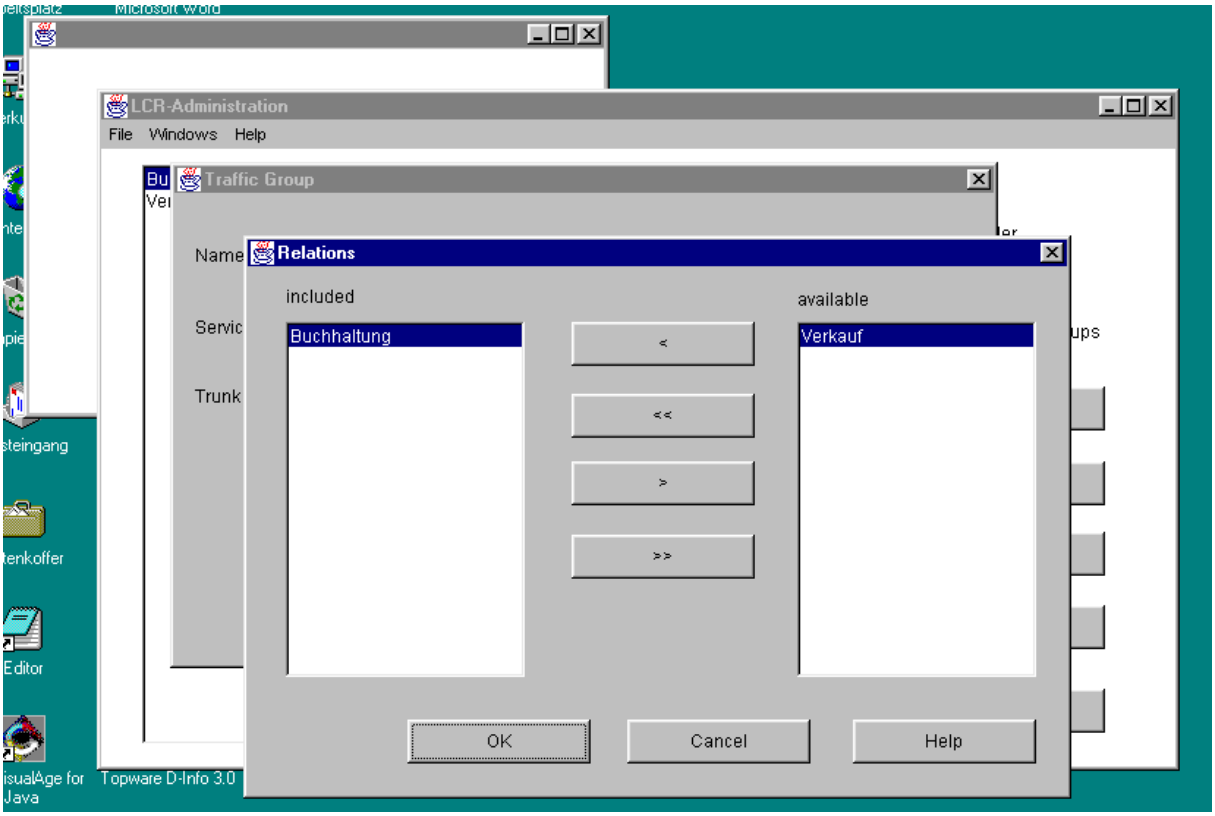


Abbildung 18: Berechtigungsvergabe zwischen TrafficGroups

Abbildung 19 – TrunkLineGroup – zeigt Name, Zielrufnummer, sowie Default-Provider an. Auch ist es mit RadioButtons möglich, zwischen internem und externem Routing zu wählen. Der Button `Service` eröffnet ein Fenster ähnlich wie `Relations` in Abbildung 18, um die zugehörigen Services auszuwählen.
Wurde im Dialog TrunkLineGroup (Abbildung 19) der Button `Routing` gedrückt, so erscheint die in Abbildung 20 gezeigte Maske. Hier besteht die Möglichkeit, RoutingTableEntries zu löschen, zu erstellen oder zu editieren. Weiterhin kann die ganze RoutingTable importiert oder exportiert werden.

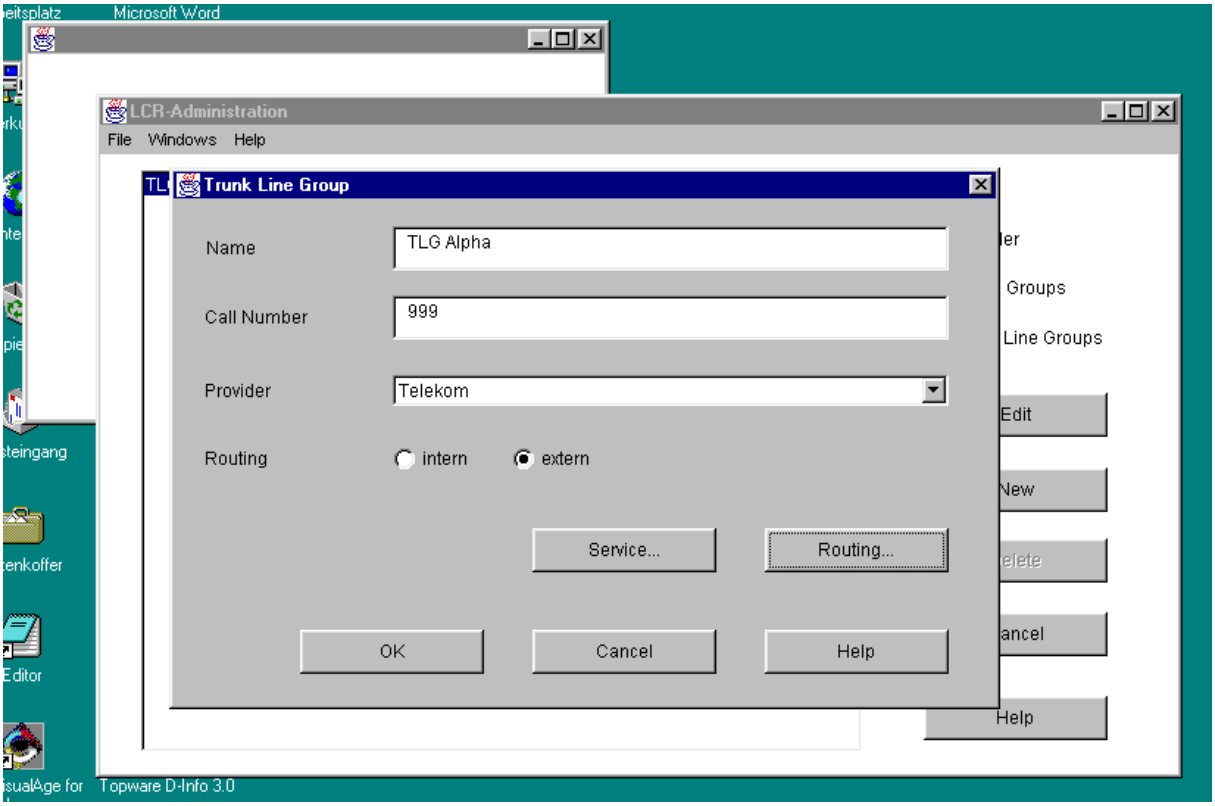


Abbildung 19: Daten einer TrunkLineGroup

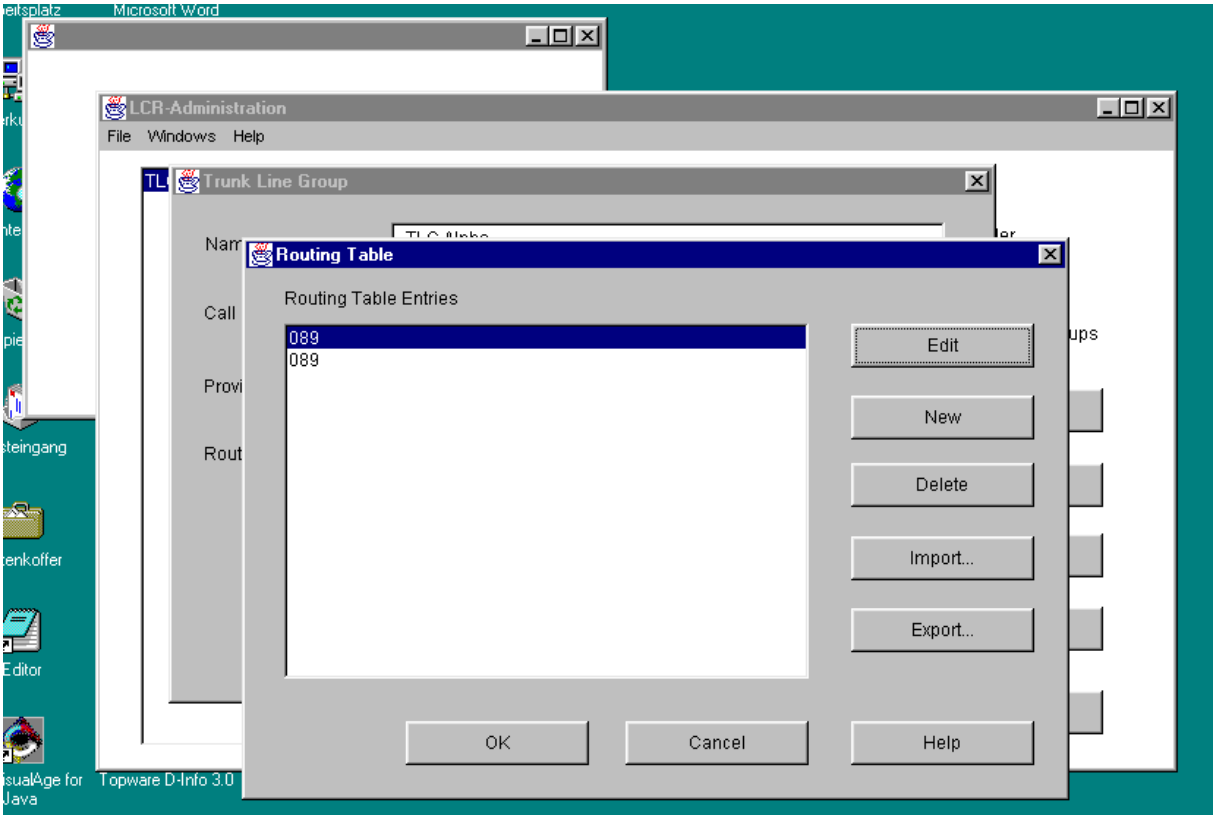


Abbildung 20: Verwaltung der Routing-Tabelle

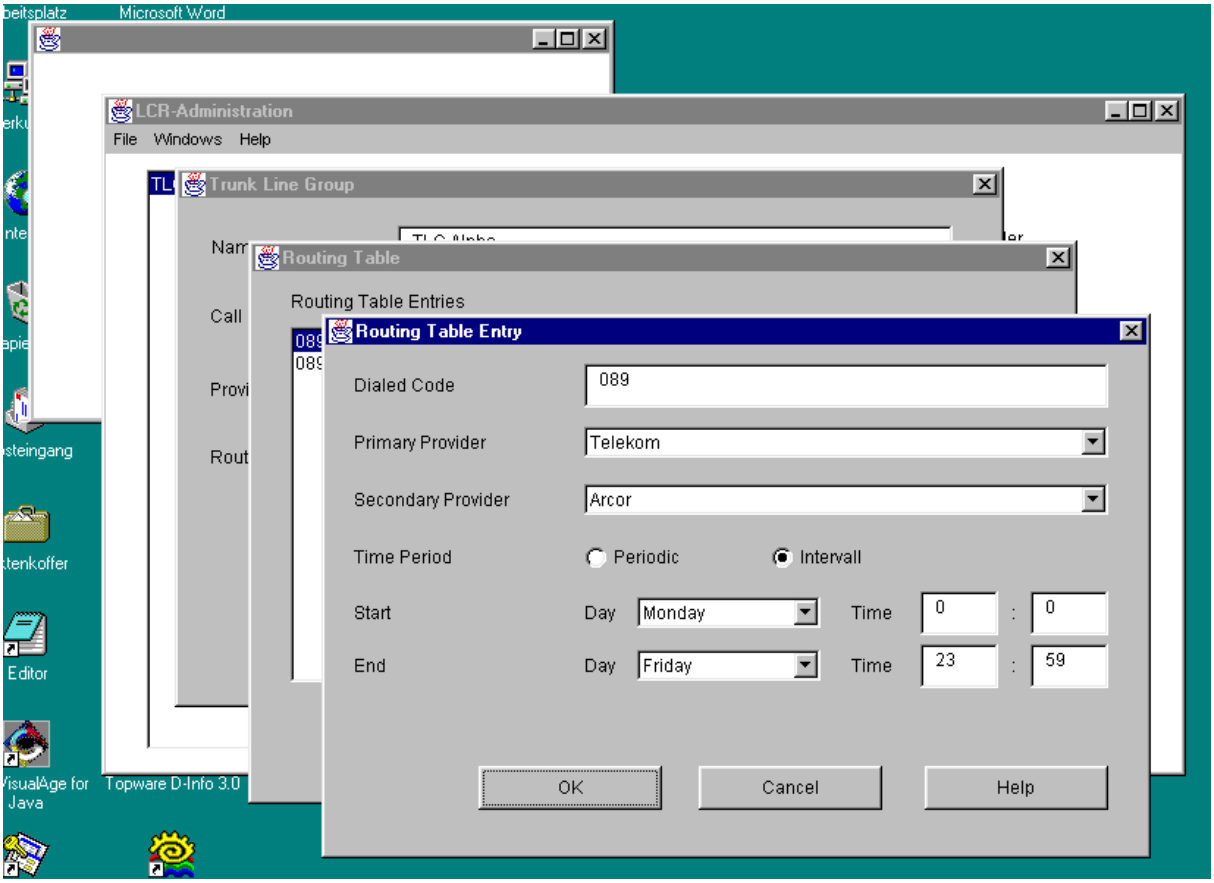


Abbildung 21: Editierfunktion der Routing-Tabelleneinträge

Abbildung 21 offenbart das Herzstück aller Informationen. Die hier angegebenen Informationen werden zum Least Cost Routing verwandt. Wichtig ist, daß stets ein PrimaryProvider ausgewählt ist, der Secondary darf auch <none> sein. Wird als PrimaryProvider der gerade als SecondaryProvider selektierte benutzt, so wird SecondaryProvider automatisch auf <none> gesetzt.
Beim Öffnen des Router-Dialoges (Abbildung 22) über das Windows-Menü erhält man die Möglichkeit, das Routing an- oder abzuschalten sowie das Verhalten beim ReRoute zu bestimmen. Dabei kann entweder die Signalisierung eines Besetztzustands oder eine Behandlung der ReRoute-Anfrage gestartet werden. Auch ist der Routing-Algorithmus (RoutingFunction) auswählbar, der die LeastCostRouting-Berechnung durchführt.

4.7 JTAPI für das ICC

Da zum Entwicklungszeitpunkt keine Umsetzung von JTAPI auf TSAPI erhältlich war, mußte die Umsetzung selbst erstellt werden. Die Mächtigkeit JTAPIs und TSAPIs machten es im vorgegebenen Zeitrahmen unmöglich, sie im gesamten Umfang umzusetzten, so daß das Hauptaugenmerk auf die Funktionalität gesetzt wurde, die für das Routing absolut notwendig ist.
Problematisch ist, daß der TSAPI-Server in C++ anzusprechen ist und die JTAPI-Implementierung in Java zu erfolgen hat. Im Praktikum erfolgte die Anbindung der beiden Komponenten über die JTAPI-konforme CORBA-IDL-Schnittstelle. Eine effizientere Methode ggf. unter Einschränkung der Portabilität wäre, den C++-Code als native-Methoden in Java einzubinden.

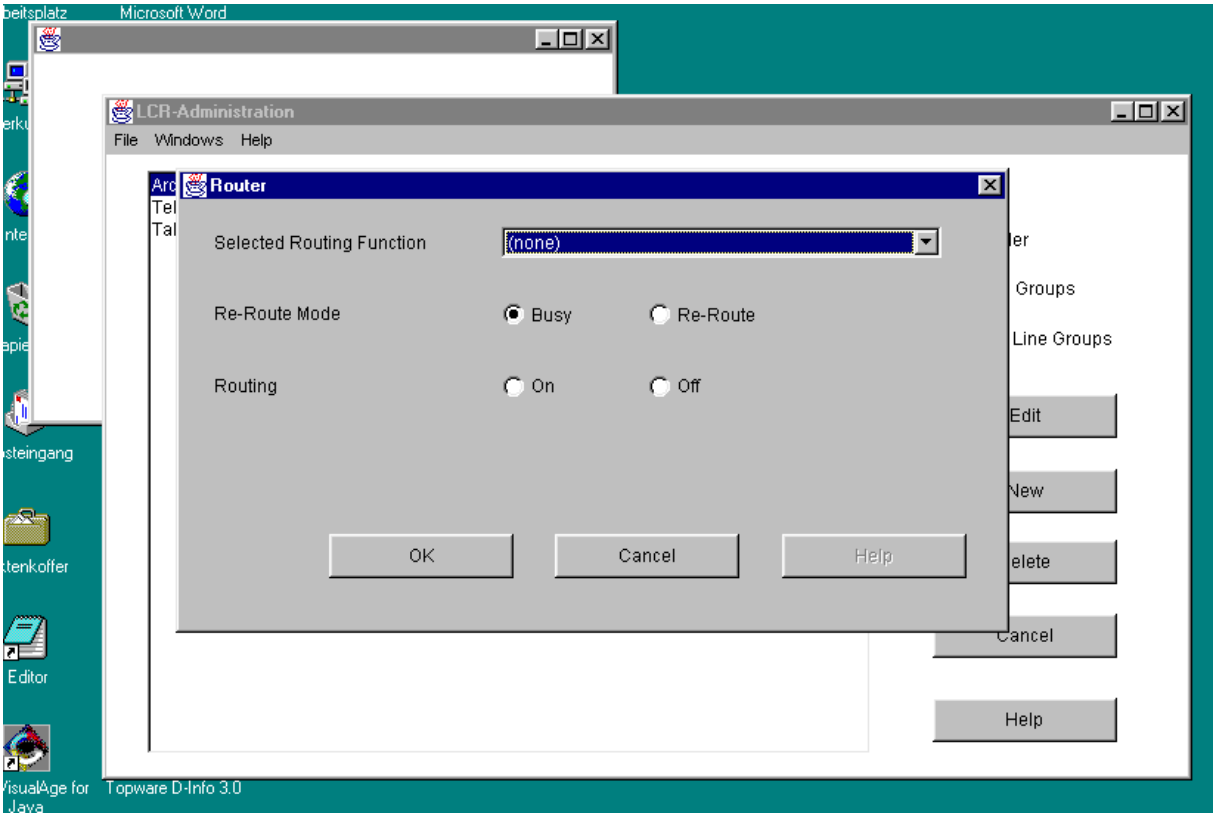


Abbildung 22: Einstellung für den Routing-Algorithmus

4.7.1 Implementierte Funktionalität

Bevor auf die konkrete Umsetzung eingegangen wird, geben wir zuerst eine Übersicht über die JTAPI-Interfaces, die implementiert werden müssen:

I. Call
Address / RouteAddress
Terminal / Connection
II. Provider / CallCenterProvider
III. RouteSession / RouteCallback / RouteEvents

Die Realisierung der Interfaces unter Teil I erfolgte dabei durch Praktikumsgruppe 1. Gruppe 3 implementierte den *Provider* und *CallCenterProvider* sowie die Anlagenansteuerung mit C++ und TSAPI (Teil II), während die Gruppen 2 und 4 jeweils eine Routing-Anbindung über JTAPI zwischen dem LCR-Modul und dem ICC programmierten (Teil III).
Im folgenden werden die beiden großen Aufgabenblöcke allgemeine Ansteuerung des Voice-Switch mit JTAPI und die Vorgehensweise bei der Bearbeitung von Routing-Ereignissen beschrieben.

4.7.2 Anlagenansteuerung

Die Anlagenansteuerung wird in ein Programm mit graphischer Benutzerschnittstelle eingebettet, mit dessen Hilfe es möglich ist, Telefonverbindungen aufzubauen, bzw. das Routing zu simulieren, sowie das Datum und die Uhrzeit für das Routing zu modifizieren. Dieses GUI wurde mit Microsoft Visual C++ 5.0 und den Microsoft Foundation Classes (MFC) erstellt. Auf die Zusammenhänge der von der integrierten Entwurfsumgebung (IDE) generierten Klassen für eine MFC-Applikation wird nur rudimentär eingegangen, da dies keine Bedeutung für die Umsetzung und das Routing hat. Ansonsten wird auf die Primär,- sowie Sekundärliteratur zur Programmierung von Visual C++ verwiesen (z.B. [8]).

Der Name der erstellten Anwendung lautet *ICCProvider*. Dieses Programm lief im Praktikum auf dem Applikationsserver. Es nutzt die unterliegende TSAPI-Schnittstelle zur Ansteuerung des Voice-Switch und stellt nach oben hin die JTAPI-konformen CORBA-Interfaces bereit. Der Quelltext ist auf mehrere mit Hilfe der IDE von Visual C++ erstellte und teilweise automatisch generierte Dateien verteilt. Dabei lassen sich die Klassen in folgende vier Gruppen einteilen:

I. Für die konkrete Realisierung selbst erstellte Klassen:

- EventMessage,
- cProvider,
- Variableninterface,
- EventHandlerThread,
- MainHandlerThread und
- WorkerThread.

II. MFC-generierte Klassen, welche angepaßt wurden:

- CICCProviderApp (applikationsspezifische Ereignisbehandlung und Funktionalität) und
- CICCProviderView (stellt eine spezifische Sicht auf zu bearbeitende Daten bereit, z.B. in Form einer GUI).

III. Klassen, die einen Dialog zur Dateneingabe bereitstellen:

- DialogMakeCall (baut eine Telefonverbindung zwischen zwei Endgeräten auf),
- DialogWaehlen (wählt die Nummer eines Endgeräts) und
- TimeEdit (gestattet das Modifizieren der voreingestellten Zeit für die Demonstration des Routings).

IV. MFC-generierte Klassen, welche nicht modifiziert wurden:

- CICCProviderDoc (repräsentiert die zu verarbeitenden Daten bzw. Dokumente, auf die es unterschiedliche Sichten (Views) geben kann),
- CmainFrame (Verwaltung des Hauptfensters der Windows-Applikation),
- Resource (Ressourcen für das Windows-Programm, z.B. Icons, usw.) und
- StdAfx (wird von MFC benötigt).

Folgende Klassen stellen die Funktionalität für das benötigte Multithreading zur Verfügung:

- CGDIThread,
- CThreadApp,
- CthreadDoc und
- CthreadView.

Das Hauptaugenmerk wird auf die Klassen der Gruppe I und II gelegt, da in ihnen die JTAPI-Implementation enthalten ist bzw. Teile eingebettet wurden. Die Klassen, die in Gruppe III und IV aufgeführt sind, werden im wesentlichen nur intern von MFC genutzt.
Die folgende Funktionalität muß die Anlagenansteuerung bereitstellen, um darauf aufbauend das Routing zu ermöglichen:

- Registrierung als CTI-Applikation: In der aktuellen Version erfolgt keine Sicherheitskontrolle innerhalb der Applikation. Das Freischalten der CSTA-Funktionalität erfolgt über ein Konfigurationsmodul der Anlagensteuerungssoftware;
- Registrierung auf relevante CSTA-Events;
- Eventhandling: CSTA generiert asynchrone Ereignisse als Bestätigung für zuvor angestoßene Funktionsaufrufe. Diese Ereignisse müssen entgegengenommen und der aufrufenden Applikation als Rückgabeparameter übergeben werden;
- Aufruf von Anlagenfunktionen
  - Verbindungsaufbau und

- Routing-Ergebnisse melden.

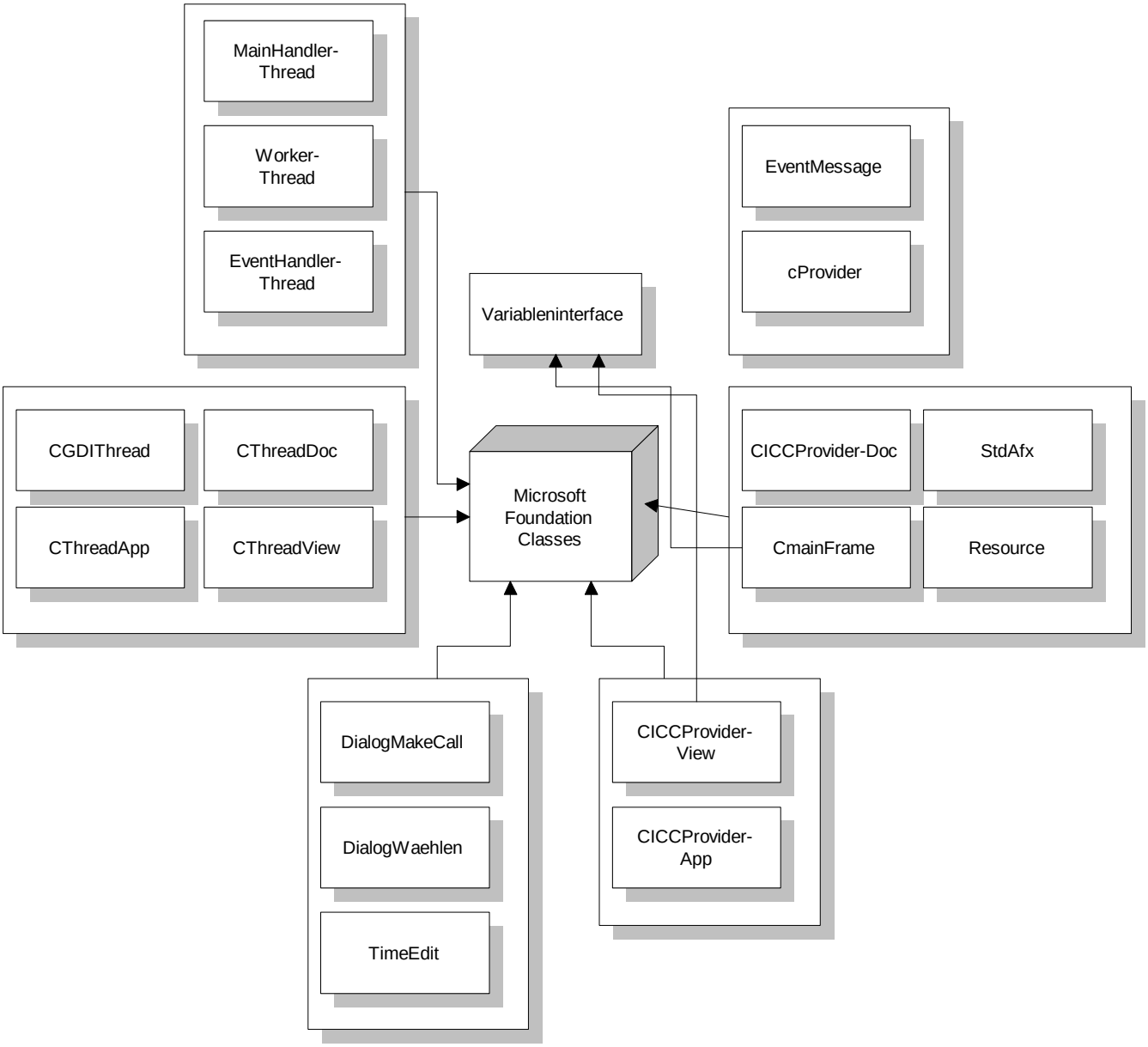


Abbildung 23 : JTAPI-C++-Provider

4.7.2.1 EventMessage

Das Handling der vom Voice-Switch generierten CSTA-Events wird mit Hilfe der Klasse `EventMessage` realisiert. Ein Objekt dieser Klasse verwaltet sämtliche Ereignisse, welche die Anlage erzeugt. Andere Objekte registrieren sich bei `EventMessage` für die Zustellung bestimmter für sie relevanter Events. Dabei wird für jedes Objekt ein sogenannter *EventChannel* (Registrierungshandle) generiert. Diese Handles werden zusammen mit dem Typ der interessierenden Ereignis in einer Liste verwaltet.
Erhält ein `EventMessage`-Objekt nun ein Event zur Bearbeitung, das einem EventChannel zuzuordnen ist, so wird das Event in die von ihm ebenfalls verwaltete Liste generierter aber noch nicht verarbeiteter Ereignisse (Eventliste) aufgenommen. Diese Eventliste kann dann blockierend oder im Polling-Verfahren von den registrierten Objekten abgefragt werden.

EventMessage
Attribute: CList<EventMsg*,EventMsg*> **RegisterList**; CList<EventData*,EventData*> **EventList**; static CRITICAL_SECTION **RegisterListLock**; static CRITICAL_SECTION **EventListLock**;
Operation: **EventMessage**(); virtual **~EventMessage**(); bool **EventChanneling**(CSTAEvent_t *); HANDLE **CreateEventChannel**(); bool **BlockEvent**(CSTAEvent_t &, HANDLE &); bool **PollEvent**(CSTAEvent_t &, HANDLE &); void **SetChannels**(); EventMsg * **RegisterEventMsgSystem**(EventMsg *); EventMsg * **RegisterEventMsgSystem**(HANDLE &, InvokeID_t); EventMsg * **RegisterEventMsgSystem**(HANDLE &, EventType_t); EventMsg * **RegisterEventMsgSystem**(HANDLE &, EventType_t, EventClass_t); EventMsg * **RegisterEventMsgSystem**(HANDLE &, InvokeID_t, EventType_t, EventClass_t);

Die registrierten Objekte werden über Standardereignismechanismen, die von Windows zur Verfügung gestellt werden, über neue CSTA-Ereignisse informiert. Dazu bietet Windows den Typ `HANDLE` an, an den Windows-Ereignisse geschickt werden können (*SetEvent()*) und an denen auf das Eintreffen von Windows-Ereignissen (ggf. mit einem vorgegebenen TimeOut) gewartet werden kann (*WaitForSingle-Object()*). Der Aufruf von *WaitForSingleObject()* blockiert, sofern kein *SetEvent()* auf dem entsprechenden Handle aufgerufen wurde. Ein nachfolgendes *SetEvent()* führt zur Freigabe des blockierten Threads.
Da mehrere Threads zeitgleich modifizierende Operationen auf `EventMessage` ausführen können, werden im Konstruktor kritische Abschnitte eingerichtet, die den Zugriff der Threads auf die Daten zur Verwaltung der Ereignis- und Registrierungslisten synchronisieren. Dazu werden Standardklassen aus der MFC-Bibliothek verwendet:

```
EventMessage::EventMessage(){
  InitializeCriticalSection(&EventListLock);
  InitializeCriticalSection(&RegisterListLock);
}
```

Andere Objekte können sich mit Hilfe der im folgenden aufgeführten Methoden auf diverse Anlagenevents registrieren. Es besteht die Möglichkeit, sich auf einzelne Events, Eventtypen oder Eventklassen zu registrieren. `HANDLE` bezeichnet einen EventChannel, der mit der Methode *CreateEventChannel()* erzeugt wird.

```
EventMsg *RegisterEventMsgSystem(EventMsg *);
EventMsg *RegisterEventMsgSystem(HANDLE&, InvokeID_t);
EventMsg *RegisterEventMsgSystem(HANDLE&, EventType_t);
EventMsg *RegisterEventMsgSystem(HANDLE&, EventType_t, EventClass_t);
EventMsg *RegisterEventMsgSystem(HANDLE&, InvokeID_t, EventType_t,
                                 EventClass_t);
```

Die Registrierung erfolgt dabei nach unterschiedlichen Kriterien: für einen einzelnen Vorgang (mittels `InvokeID`), einen bestimmen Ereignistyp (z.B. für Monitoring) oder Klassen von Ereignistypen (CSTARequest, CSTAConfirmation, ...) [5]. Die *RegisterEventMsgSystem()*-Methode erzeugt eine neue Instanz der Klasse `EventMsg` zur Speicherung der Registrierungsdaten und setzt deren Attribute auf die übergebenen Parameter. Die `is_*`-Attribute geben dabei an, ob das jeweilige Kriterium berücksichtigt werden soll. Das `Event`-Attribut beinhaltet den EventChannel der registrierten Komponente.

```
EventMsg *EventMessage::RegisterEventMsgSystem(HANDLE &Event,
          InvokeID_t ID, EventType_t EventType, EventClass_t EventClass){
  EventMsg *Msg      = new EventMsg;
```

```
  Msg->Event         = &Event;
  Msg->is_InvokeID   = true;
  Msg->InvokeID      = ID;
  Msg->is_EventType  = true;
  Msg->EventType     = EventType;
  Msg->is_EventClass = true;
  Msg->EventClass    = EventClass;
  return RegisterEventMsgSystem(Msg);
}
```

Dieses Objekt wird nun schließlich in die Liste `RegisterList` aufgenommen. Damit wird sichergestellt, daß später jede Registrierung eine eigene Kopie jedes relevanten Events erhält. Das Einfügen einer neuen Registrierung erfolgt dabei durch einen kritische Bereich geschützt, um Inkonsistenzen durch zeitgleichen Zugriff anderer Threads zu vermeiden. Durch die Bereitstellung einer Kopie für jede Registrierung wird ein Konflikt mehrerer Threads bei der Eventverarbeitung ebenfalls vermieden.

```
EventMsg *EventMessage::RegisterEventMsgSystem(EventMsg *Msg){
  EnterCriticalSection(&RegisterListLock); {
    RegisterList.AddHead(Msg);
  }
  LeaveCriticalSection(&RegisterListLock);
  return Msg;
}
```

Um nun neue CSTA-Ereignisse in `EventMessage` einzutragen, wird die Methode *EventChanneling()* benutzt. Die for-Schleife sorgt dafür, daß jede Registrierung einen eigenen Eintrag in der Eventliste erhält. So kann ein und dasselbe CSTA-Event von mehreren Objekten unabhängig und zu verschiedenen Zeitpunkten ausgewertet werden. Qualifiziert sich das neue Event nach den Auswahlkriterien einer Registrierung nicht, so liefert die boolesche Methode `EventMessage`.*compare()* den Wert `false`, ansonsten `true`.

```
bool EventMessage::EventChanneling(CSTAEvent_t *Event){
  bool Result = false;
  EnterCriticalSection(&RegisterListLock); {
    for (POSITION pos = RegisterList.GetHeadPosition(); pos != NULL; ){
      EventMsg *Msg = RegisterList.GetNext(pos);
      if(compare(Msg, Event)){
        EventData *TEMPEvent = new EventData;
        TEMPEvent->Event = Msg->Event;
        memcpy(&(TEMPEvent->CSTAEvent), Event, sizeof(CSTAEvent_t));
        EnterCriticalSection(&EventListLock); {
          EventList.AddTail(TEMPEvent);
        }
        LeaveCriticalSection(&EventListLock);
        SetEvent(*(Msg->Event));
        Result = true;
      }
    }
  }
  LeaveCriticalSection(&RegisterListLock);
  return Result;
}
```

Das Sperren der `RegisterList` dient ebenfalls zum Vermeiden von Inkonsistenzen bei der Ereignisverteilung.
Die Methoden *BlockEvent()* (blockierend) und *PollEvent()* (im Poll-Verfahren) überprüfen anhand des übergebenen EventChannel, ob ein Event vorliegt. Dabei wird die gesamte Eventliste daraufhin überprüft, ob die mit dem Events gespeicherten Handles mit dem EventChannel übereinstimmen. In diesem Fall wird das erste gefundene Event in `Return` kopiert und aus der Liste gelöscht.

```
bool EventMessage::BlockEvent(CSTAEvent_t &Return, HANDLE &Event){
  // warten bis Signal eintrifft
  WaitForSingleObject(Event, INFINITE);
  bool geloescht = false;
  bool removeSignal  = true;
  POSITION vor_pos;
  EnterCriticalSection(&EventListLock); {
    POSITION pos = EventList.GetHeadPosition();
    while(pos != NULL){
      // alle gespeicherten Events überprüfen
      vor_pos = pos;
      EventData *TEMPEvent = EventList.GetNext(pos);
      if(TEMPEvent->Event == &Event){
        if(!geloescht){
          // entsprechendes Event übergeben und löschen
          if (&Return != NULL)
            memcpy(&Return, &(TEMPEvent->CSTAEvent), sizeof(CSTAEvent_t));
          delete TEMPEvent;
          EventList.RemoveAt(vor_pos);
          geloescht = true;
        }
        else{
         // es sind noch Events in der Liste
          removeSignal = false;
        }
      }
    }
  }
  LeaveCriticalSection(&EventListLock);
  if (removeSignal)     // falls alle Events gelöscht sind:
    ResetEvent(Event);  // Eventsignal beseitigen
  return geloescht;
}
```

Da ggf. mehrere Events für eine Registrierung vorhanden sind, jeder Aufruf von *BlockEvent()* und *PollEvent()* aber höchstens ein Ereignis entfernt, wird der EventChannel, der nur die Werte "gesetzt" oder "nicht gesetzt" annehmen kann, erst zurückgenommen nachdem das letzte vorhandene Ereignis bearbeitet ist.

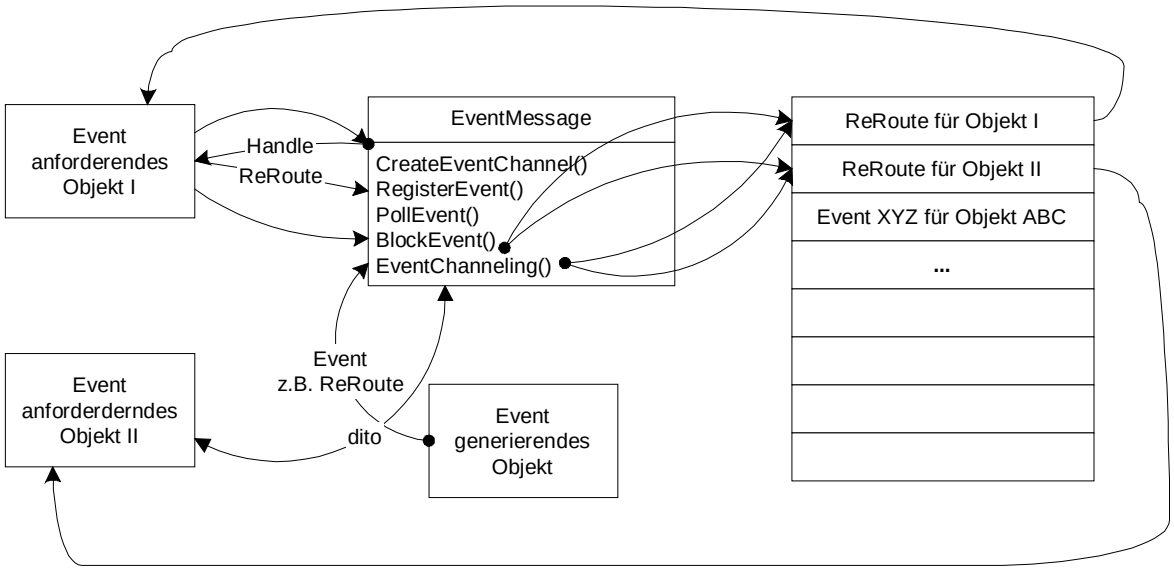


Abbildung 24 : Verwaltung der CSTA-Events in der Eventliste

4.7.2.2 cProvider

cProvider

Attribute:
ACSHandle_t **acsHandle**;
CSTAEvent_t **eventBuf**;
EventMessage ***EventChannel**;
InvokeID_t **invokeID**;
unsigned short **eventBufSize**;
CSTAMonitorFilter_t **MonitorFilter**;
bool **openStream**;
int **nr_pending_calls**;
int ***pending_calls**;
char ****pending_callee_ids**;

Operation:
cProvider();
cProvider(EventMessage *);
cProvider(EventMessage *, unsigned short, unsigned short);
virtual ~**cProvider**();
int **getEventString**(char *);
void **getRetCodeString**(char *, RetCode_t);
EventType_t **getEventType**();
EventClass_t **getEventClass**();
InvokeID_t **getInvokeID**();
RetCode_t **OpenStream**(InvokeID_t, unsigned short, unsigned short);
RetCode_t **BlockOpenStream**(CSTAEvent_t &, HANDLE &, unsigned short, unsigned short);
RetCode_t **CloseStream**(InvokeID_t);
RetCode_t **BlockCloseStream**(CSTAEvent_t &, HANDLE &);
RetCode_t **getEventBlock**(unsigned short *);
RetCode_t **getAPICaps**(InvokeID_t);
RetCode_t **BlockgetAPICaps**(CSTAEvent_t &, HANDLE &);
RetCode_t **GetDeviceList**(InvokeID_t);
RetCode_t **MakeCall**(InvokeID_t, DeviceID_t *, DeviceID_t *);
RetCode_t **BlockMakeCall**(CSTAEvent_t &, HANDLE &, const char *, const char *);
RetCode_t **MonitorDevice**(InvokeID_t, char *);
RetCode_t **BlockMonitorDevice**(CSTAEvent_t &, HANDLE &, char *);
RetCode_t **MonitorCall**(InvokeID_t);
RetCode_t **MonitorCallsViaDevice**(InvokeID_t, char *);
RetCode_t **RouteRegisterReq**(InvokeID_t, DeviceID_t *);
RetCode_t **BlockRouteRegisterReq**(CSTAEvent_t &, HANDLE &,char *);
RetCode_t **RouteRegisterCancel**(InvokeID_t, RouteRegisterReqID_t);
RetCode_t **BlockRouteRegisterCancel**(CSTAEvent_t &, HANDLE &,
RouteRegisterReqID_t);
RetCode_t **RouteSelectInv**(InvokeID_t, RouteRegisterReqID_t, RoutingCrossRefID_t,
DeviceID_t *, RetryValue_t, SetUpValues_t *, Boolean);
RetCode_t **BlockRouteSelectInv**(CSTAEvent_t &, HANDLE &, RouteRegisterReqID_t,
RoutingCrossRefID_t, DeviceID_t *, RetryValue_t, SetUpValues_t *, Boolean);
RetCode_t **RouteEndInv**(InvokeID_t, RouteRegisterReqID_t, RoutingCrossRefID_t,
CSTAUniversalFailure_t);
RetCode_t **BlockRouteEndInv**(CSTAEvent_t &, HANDLE &, RouteRegisterReqID_t,
RoutingCrossRefID_t, CSTAUniversalFailure_t);
bool **insertPendingCall**(long, char *);
bool **deletePendingCall**(long);
char * **getPendingCallString**(long);

Die konkrete Ansteuerung der Anlage übernimmt die Klasse `cProvider`. Sie kapselt den C++-spezifischen Teil der JTAPI-Implementation. Der Konstruktor der Klasse ruft die Methode *OpenStream()* auf, welche die Registrierung bei der Anlage als CTI-Applikation übernimmt. Alle Registrierungen von Events werden über das zu Programmstart instanziierte `EventMessage`-Objekt `EventChannel` abgewickelt, das dem Konstruktor als Parameter übergeben wurde. Die Variable `invokeID` dient dazu, jedem Vorgang (Inanspruchnahme des JTAPI-C++-Providers) eine eindeutige Kennung zuordnen, über welche die asynchron eintreffenden CSTA-Events den Vorgängen zugeordnet werden können.

`nr_pending_calls` dient zur Verwaltung von Verbindungen, die ihre Routing-Anfragen noch nicht vollständig abgewickelt haben. Die nächsten beiden Attribute liefern eine Zuordnung zwischen der von CSTA beim Verbindungsaufbau gelieferten CallID und der gewählten Rufnummer. Die zusätzliche Verwaltung von Verbindungen und eine Zuordnung zu Routing-Events erscheint zunächst überflüssig. Sie ist jedoch notwendig, da das ICC noch keine Unterstützung für das Routing von Anrufen, welche die TK-Anlage verlassen, anbietet. Für interne Automatische Rufverteilung (ACD) werden jedoch bereits Routing-Ereignisse generiert. Allerdings wird als Zielnummer nur die gewählte ACD-Nebenstelle geliefert.
Um dennoch korrekt mit der Anlage zu kooperieren, wird die gewählte Nummer bei einem *MakeCall()* zunächst in `pending_callee_ids` gespeichert und an die Anlage ein Ruf auf eine ACD-Nummer ausgelöst. Die zugehörige von CSTA gelieferte CallID wird ebenfalls in `pending_calls` unter demselben Index abgelegt. Alle RouteRequests beziehen sich nun auf eine CallID, so daß die ursprünglich gewählte Nummer bei Eintreffen eines Routing-Events rekonstruiert und vor der Weiterleitung an den Router wieder eingetragen werden kann.
Diese Vorgehensweise stellt keinerlei Einschränkung der prinzipiellen Funktionalität dar, da, sobald die Anlagensoftware auch für ausgehende Rufe Routing-Ereignisse generiert, die volle Rufnummer im Request enthalten ist, diese Indirektion einfach entfernt werden kann.

```
cProvider::cProvider(EventMessage *Event, unsigned short sendQSize,
                     unsigned short recvQSize) {
  EventChannel       = Event;
  invokeID           = 0;
  acsHandle          = NULL;
  nr_pending_calls   = 0;
  pending_calls      = new int[MAX_PENDING_CALLS];
  pending_callee_ids = new char *[MAX_PENDING_CALLS];
  for (int i = 0; i < MAX_PENDING_CALLS; i++) {
    pending_calls[i]     = -1;
    pending_callee_ids[i]   = NULL;
  }
  OpenStream (getInvokeID(), sendQSize, recvQSize);
}
```

OpenStream() ruft im wesentlichen nur die TSAPI-Methode *acsOpenStream()* mit den entsprechenden Parametern auf und liefert als boolesches Ergebnis, ob die Registrierung bei der Anlage als CTI-Applikation erfolgreich war. Das `acsHandle` wird als Referenzparameter übergeben und entsprechend gesetzt. Es wird später für jeden Zugriff auf die Anlage benötigt.

```
RetCode_t cProvider::OpenStream(InvokeID_t ID, unsigned short sendQSize,
                                unsigned short recvQSize){
  RetCode_t Result;
  ServerID_t serverID = "Bosch CTI-Server tcp-95";
  LoginID_t loginID   = "";
  Passwd_t passwd     = "";
  AppName_t appName   = "cProvider";
  Version_t apiVer    = "TS1-2";
  if (0 > (Result = acsOpenStream(
            &acsHandle,  // acsHandle
            APP_GEN_ID,  // invokeIDType
            ID,          // InvokeID
            ST_CSTA,     // streamType
            &serverID,   // serverID
            &loginID,    // loginID
            &passwd,     // passwd
            &appName,    // applicationName
            ACS_LEVEL1,  // acsLevelReq
            &apiVer,     // apiVer
            sendQSize,   // sendQSize
            0,           // sendExtraBufs
            recvQSize,   // recvQSize
            0,           // recvExtraBufs
```

```
            NULL))){     // privateData
 }
  ...
  return Result;
}
```

Die Methode *RouteRegisterReq()* erlaubt einer CTI-Applikation sich als Router für ein bestimmtes `Device` zu registrieren. Dabei wird der Applikation eine für jede Registrierung eindeutige `RouteRegisterReqID` zurückgegeben.

```
RetCode_t cProvider::RouteRegisterReq(InvokeID_t ID, DeviceID_t *Device) {
  RetCode_t Result;
  Result = cstaRouteRegisterReq(acsHandle, // ACSHandle_t
                                ID,        // invokeID
                                Device,    // DeviceID_t
                                NULL);     // PrivateData_t
  return Result;
}
```

Wurde der Routing-Vorgang vom LCR-Modul abgebrochen, so muß dies der Anlage gemeldet werden. Hierzu dient die Methode *RouteEndInv()*, welche die TSAPI-Methode *cstaRouteEndInv()*aufruft. `routeRegisterReqID` bezeichnet das registrierte Device. `routingCrossRefID` den konkreten Routing-Dialog, der beendet werden soll. Es können zeitgleich mehrere Routing-Dialoge auf einem Device aktiv sein.

```
RetCode_t cProvider::RouteEndInv(InvokeID_t ID,
                                 RouteRegisterReqID_t routeRegisterReqID,
                                 RoutingCrossRefID_t routingCrossRefID,
                                 CSTAUniversalFailure_t errorValue){
  RetCode_t Result;
  Result = cstaRouteEndInv (acsHandle,          // ACSHandle_t
                            ID,                 // InvokeID_t
                            routeRegisterReqID, // RouteRegisterReqID_t
                            routingCrossRefID,  // RoutingCrossRefID_t
                            errorValue,         // CSTAUniversalFailure_t
                            NULL);              // PrivateData_t
  return Result;
}
```

Wenn die Routing-Komponente ein Ergebnis ermittelt hat, muß dies der Anlage mitgeteilt werden. Dazu dient die Methode *RouteSelectInv()*, die das Ergebnis im Parameter `routeSelected` an die Anlage übergibt.

```
RetCode_t cProvider::RouteSelectInv(InvokeID_t ID,
                                RouteRegisterReqID_t routeRegisterReqID,
                                RoutingCrossRefID_t routingCrossRefID,
                                DeviceID_t *routeSelected,
                                RetryValue_t remainRetry,
                                SetUpValues_t *setupInformation,
                                Boolean routeUsedReq){
  RetCode_t Result;
  Result = cstaRouteSelectInv (acsHandle,        // ACSHandle_t
                                ID,          // InvokeID_t
                                routeRegisterReqID, // RouteRegisterReqID_t
                                routingCrossRefID,  // RoutingCrossRefID_t
                                (DeviceID_t *) routeSelected, // DeviceID_t*
                                remainRetry,        // RetryValue_t
                                setupInformation,   // SetUpValues_t *
                                true,               // Boolean
                                NULL);              // PrivateData_t
  return Result;
}
```

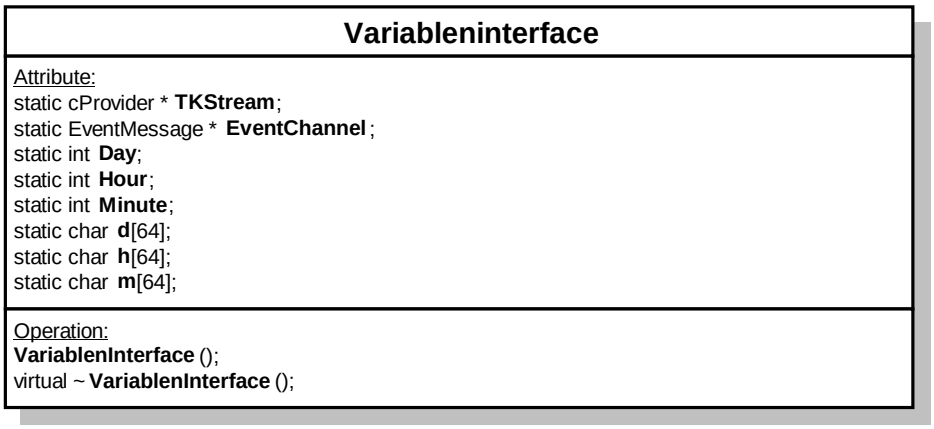
Während die letzten beiden Methoden Informationen an die Anlage übergaben, wird mit der Methode *getEventBlock()* blockierend auf CSTA-Events der Anlage gewartet. Ob der Aufruf erfolgreich war, kann in `Result` überprüft werden. Das Event wird in `eventBuf` geschrieben. Diese Methode wird nur von der Klasse `EventMessage` zum Abholen von Anlagenereignissen vor der anschließenden Duplizierung für jedes registrierte Objekt genutzt.

```
RetCode_t cProvider::getEventBlock(unsigned short *numEvents){
  RetCode_t Result;
  eventBufSize = sizeof(CSTAEvent_t);
  Result = acsGetEventBlock(acsHandle,     // acsHandle
                            &eventBuf,     // *eventBuf
                            &eventBufSize, // *eventBufSize
                            NULL,          // *privateData
                            numEvents);    // *numEvents
  return Result;
}
```

MakeCall() ist eine Methode, die es erlaubt softwareseitig einen Verbindungsaufbau zu initiieren. Dabei wird die TSAPI-Methode *cstaMakeCall()* aufgerufen. Als Parameter werden die Nummer des Anrufers (`callingDevice`) und die Nummer der Angerufenen (`calledDevice`) übergeben.

```
RetCode_t cProvider::MakeCall(InvokeID_t ID, DeviceID_t *callingDevice,
                              DeviceID_t *calledDevice){
  return cstaMakeCall(acsHandle,     // acsHandle
                      ID,            // invokeID
                      callingDevice, // *deviceID
                      calledDevice,  // *deviceID
                      NULL);         // *privateData
}
```

4.7.2.3 Variableninterface



Die Klasse `Variableninterface` besitzt ausschließlich statische Attribute, um globale Objekte, bzw. Informationen zur Verfügung zu stellen. Die Lösung unter Verwendung globaler Daten ist zugegebenermaßen nicht sehr elegant und sollte nochmals überarbeitet werden.
Die Variable `TKStream` speichert eine Referenz auf ein `cProvider`-Objekt, das genutzt wird, um die Anlage anzusteuern.

```
static cProvider  *TKStream;
```

`EventChannel` speichert eine Referenz auf das zu benutzende `EventMessage`-Objekt, um Events von der TK-Anlage abzufragen:

```
static EventMessage *EventChannel;
```

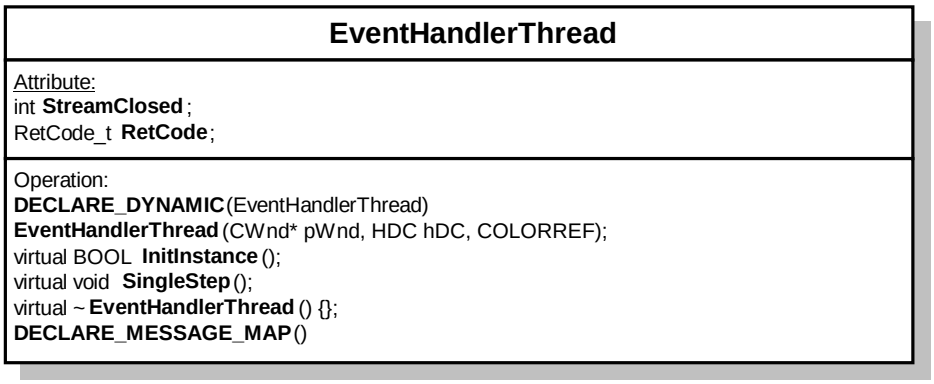
Die folgenden drei Integer-Zahlen speichern den für das Routing zugrunde liegenden Wochentag und die Uhrzeit, die für die Demoversion zum Einstellen eines festen Routing-Zeitpunkts benötigt wurde. Gewöhnlicherweise werden diese Informationen in einem Produkt nicht benötigt sondern direkt über die Systemuhrzeit vom Router ermittelt:

```
static int Day;
static int Hour;
static int Minute;
```

Die letzten drei Strings speichern die gleiche Information wie oben nur in textueller Form zur Verwendung im GUI:

```
static char d[64];
static char h[64];
static char m[64];
```

4.7.2.4 EventHandlerThread



Bei den folgenden drei Klassen handelt es sich um MFC-Threads. Unter MS Windows NT werden Threads zeitscheibengesteuert verwaltet, d.h. es ist sichergestellt, daß alle MFC-Threads Rechenzeit erhalten. Der `EventHandlerThread` hat ausschließlich die Aufgabe, blockierend Events von der Anlage zu holen und sie in `eventBuf` zu speichern. Dazu bedient er sich der *getBlockEvent()*-Methode der Klasse `cProvider`. Die Referenz auf das `cProvider`-Objekt `TKStream` erhält er aus dem Attribut `TKStream` der Klasse `Variableninterface`. Ebenso wird die statische Referenz `EventChannel` genutzt, um mit Hilfe des `EventMessage`-Objektes und der Methode *EventChanneling()* ausgelesene Events zu verteilen.

```
void EventHandlerThread::SingleStep(){
 unsigned short count;
  RetCode_t getEventResult;
  if (!StreamClosed){
    getEventResult = TKStream->getEventBlock(&count);
    if (getEventResult < 0){
      if (getEventResult != RetCode){
        ... // Error
       RetCode = getEventResult;
      }
    }
    else{
      if ((TKStream->getEventType() == ACS_CLOSE_STREAM_CONF) &&
          (TKStream->getEventClass() == ACSCONFIRMATION)){
        StreamClosed = true;
      }
      EventChannel->EventChanneling(&(TKStream->eventBuf));
      ...
    }
  }
```

```
}
```

4.7.2.5 MainHandlerThread

Der MainHandlerThread dient dazu ein JTAPI-konformes CORBA-Objekt, das die Funktionalität des *CallCenterProvider*-Interface anbietet, zu generieren und es JTAPI-Clients zur Verfügung zu stellen. Es ist dazu ein eigener Thread notwendig, da eine CORBA-Serverimplementation blockierend auf Methodenaufrufe ihres Interface wartet. Bei der Instanziierung werden zuerst die ORB- und BOA-Objekte erzeugt. Danach wird eine Instanz des zur Verfügung zu stellenden *CallCenterProvider*-Objekts generiert. Um es von außerhalb ansprechbar zu machen, wird seine (eindeutige) Referenz in eine Datei geschrieben, die andere CORBA-Objekte auslesen können. Schlußendlich wird mit der Methode *impl_is_ready()* des BOA die Implementation des Service anderen Objekten im System bereitgestellt.

MainHandlerThread
Attribute: HANDLE **MainHandlerEvent**;
Operation: **DECLARE_DYNAMIC**(MainHandlerThread) **MainHandlerThread**(CWnd* pWnd, HDC hDC, COLORREF); virtual BOOL **InitInstance**(); virtual void **SingleStep**(); virtual ~**MainHandlerThread**(); **DECLARE_MESSAGE_MAP**();

```
BOOL MainHandlerThread::InitInstance(){
  CSTAEvent_t CSTAEvent;  // Speicher um CSTAEvents zu empfangen
 // Kommunikationskanal öffnen
  MainHandlerEvent = EventChannel->CreateEventChannel();
 TKStream->BlockOpenStream(CSTAEvent, MainHandlerEvent, 30, 30);
  if (CSTAEvent.eventHeader.eventClass == ACSCONFIRMATION &&
      CSTAEvent.eventHeader.eventType  == ACS_OPEN_STREAM_CONF){
    try{
      // Corba-Server starten
      // ORB und BOA erzeugen
      int argc = 0;
      CORBA_ORB_var orb = CORBA_ORB_init(argc, NULL);
      CORBA_BOA_var boa = orb -> BOA_init(argc, NULL);

      // Implementationsobjekt generieren
      // im Konstruktor übergeben:
      //       1. EventChannel
      //       2. TKStream
      // Diese Daten werden dann an alle Addresses,
      // Terminals etc. weitergegeben !
      java_telephony_callcenter_CallCenterProvider_var p =
           new java_telephony_callcenter_CallCenterProvider_Impl (TKStream,
                                                        EventChannel);
      // Objektreferenz auf Datei schreiben -> Ändern: CORBA-NAMING-SERVICE
      CORBA_String_var s = orb -> object_to_string(p);
      const char* refFile = "C:\\public\\Provider.ref";
      ofstream out(refFile);
      if(out.fail()){
        println("Kann Datei nicht öffnen");
        return false;
      }
      out << s << endl;
      out.close();

      // Implementation starten
      boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
    }
```

```
    ...
  }
  else{
    println("Fehler bei ACS_OPEN_STREAM");
  }
  ...
  // FALSE zurückgeben, um Thread zu beenden
  return FALSE;
}
```

4.7.2.6 WorkerThread

WorkerThread
Attribute: HANDLE **WorkerThreadEvent**;
Operation: **DECLARE_DYNAMIC**(WorkerThread) **WorkerThread**(CWnd* pWnd, HDC hDC, COLORREF); virtual BOOL **InitInstance**(); virtual void **SingleStep**(); virtual ~**WorkerThread**() {}; **DECLARE_MESSAGE_MAP**()

Ebenso wie der MainHandlerThread ist der WorkerThread nur dazu da, ein CORBA-Serverobjekt jedoch mit JTAPI-Routing-Funktionalität bereitzustellen. Die eigentliche CORBA-Initialisierung findet in der Klasse `CRoutingServer` statt, deshalb wird an dieser Stelle nur ein Objekt dieser Klasse gegründet, welches als Parameter `TKStream` und `EventChannel` entgegennimmt, um sich später auf Routing-Events zu registrieren und die Anlage zu informieren, wenn das Routing beendet oder eine Route gefunden wurde.

```
BOOL WorkerThread::InitInstance(){
  // Starten der CORBA-Implementation zur Uebergabe von
  //    RouteSelectInv  und
  //    RouteEndInv
  // an die TK-Anlage  und
  // zum Bereitstellen und Abholen von Route*Events von der
  // Anlage durch die Java-Routing-Komponente

  new CRoutingServer(VariablenInterface::TKStream,
                     VariablenInterface::EventChannel);

  ...
  // FALSE zurückgeben, um Thread zu beenden
  return FALSE;
}
```

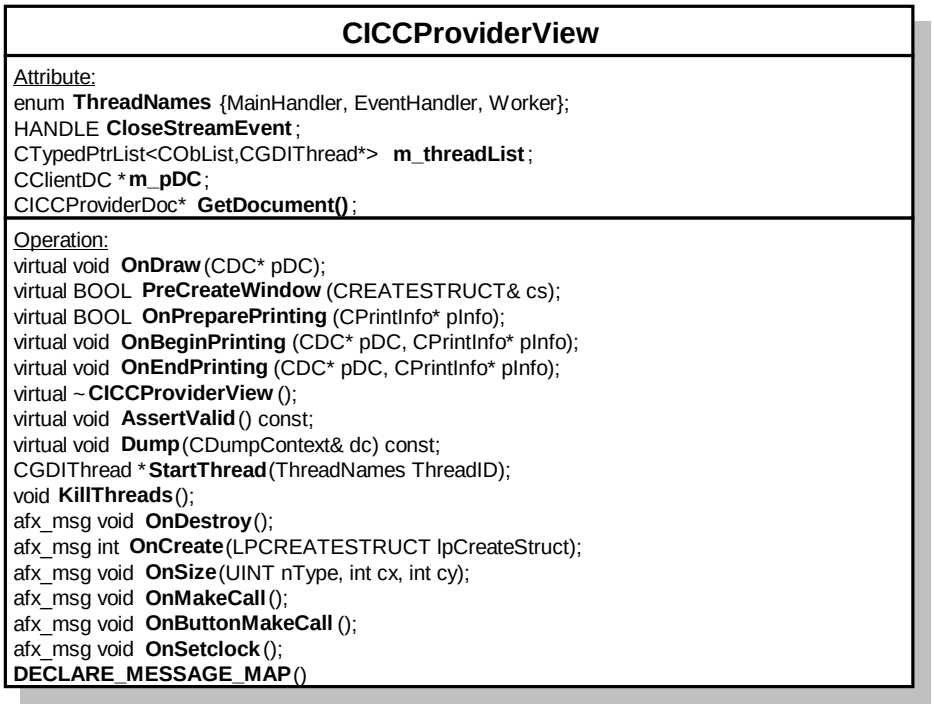
4.7.2.7 CICCProviderApp

CICCProviderApp
Attribute: cProvider * **TKStream**; EventMessage * **EventChannel**;
Operation: **CICCProviderApp**(); virtual BOOL **InitInstance**(); virtual int **ExitInstance**(); afx_msg void **OnAppAbout**(); afx_msg void **OnWaehle**(); **DECLARE_MESSAGE_MAP**()

CICCProviderApp wurde automatisch generiert. Hinzugefügt wurden einige Initialisierungen in der Methode *InitInstance()*. An dieser Stelle werden der `EventChannel`, sowie der `TKStream` erzeugt und in die statischen Variablen der Klasse `Variableninterface` geschrieben. Danach wird das Datum in der Klasse `Variableninterface` als Grundeinstellung auf Montag 12:00 Uhr gesetzt und das GUI (für die Demo) über die Klasse CICCProviderView instanziiert.

```
BOOL CICCProviderApp::InitInstance(){

  // Eventkanal öffnen
  EventChannel = new EventMessage();
 // Stream öffnen
  TKStream = new cProvider(EventChannel);
...
  // Initialize static members of CGDIThread
  InitializeCriticalSection(&CGDIThread::m_csGDILock);

  // Initialisierung des VariablenInterface
  VariablenInterface::TKStream     = TKStream;
  ...
  strcpy(VariablenInterface::m, "00");
 CRuntimeClass *ICCProviderView = RUNTIME_CLASS(CICCProviderView);
 ...
 // Das einzige Fenster ist initialisiert und kann jetzt
  //angezeigt und aktualisiert werden.
  m_pMainWnd->ShowWindow(SW_SHOW);
  m_pMainWnd->UpdateWindow();
 return TRUE;
}
```

4.7.2.8 CICCProviderView



Ebenso wie CICCProviderApp wurde CICCProviderView automatisch generiert. Dieser Klasse fällt die Aufgabe zu, die drei oben erwähnten Threads zu starten, damit sie ihre Arbeit aufnehmen können, d.h. Events auslesen und zwei CORBA-Implementationen bereitstellen.

```
int CICCProviderView::OnCreate(LPCREATESTRUCT lpCreateStruct){

  if (CView::OnCreate(lpCreateStruct) == -1) return -1;
  ...
```

```
  // Threads starten
  StartThread(EventHandler);  // Monitor starten
  StartThread(MainHandler);   // Hauptprogramm starten
  StartThread(Worker);
 return 0;
}
```

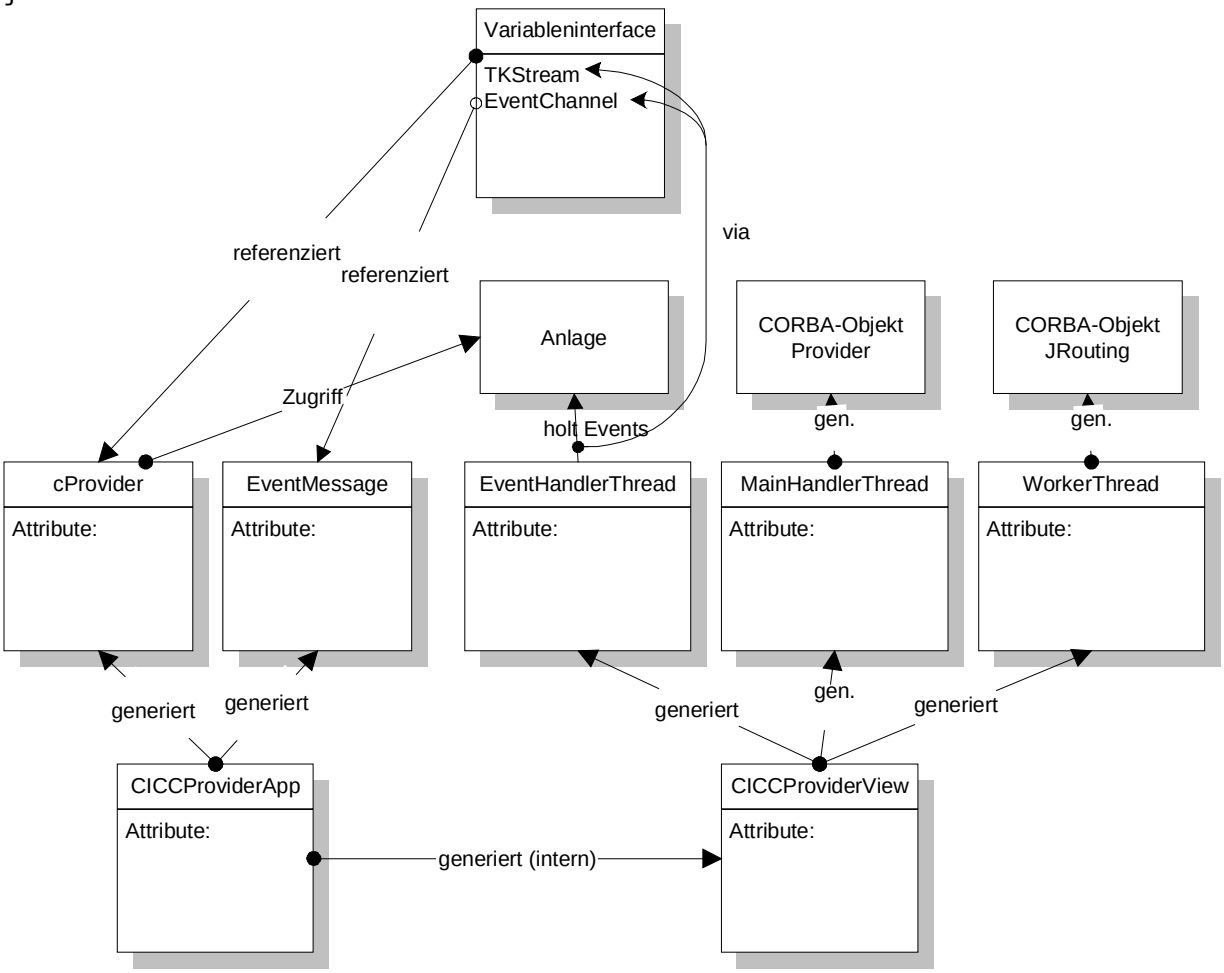


Abbildung 25 : Interaktion der Klassen im C++-Providerteil

4.7.3 JTAPI-Implementation

Wie den JTAPI-Ausführungen in Abschnitt 2.4 zu entnehmen ist, müssen die JTAPI-Interfaces erst implementiert werden. Um LCR realisieren zu können, müssen zumindest die folgend aufgeführten Interfaces implementiert werden. Aus der Package *jtapi.telephony*:

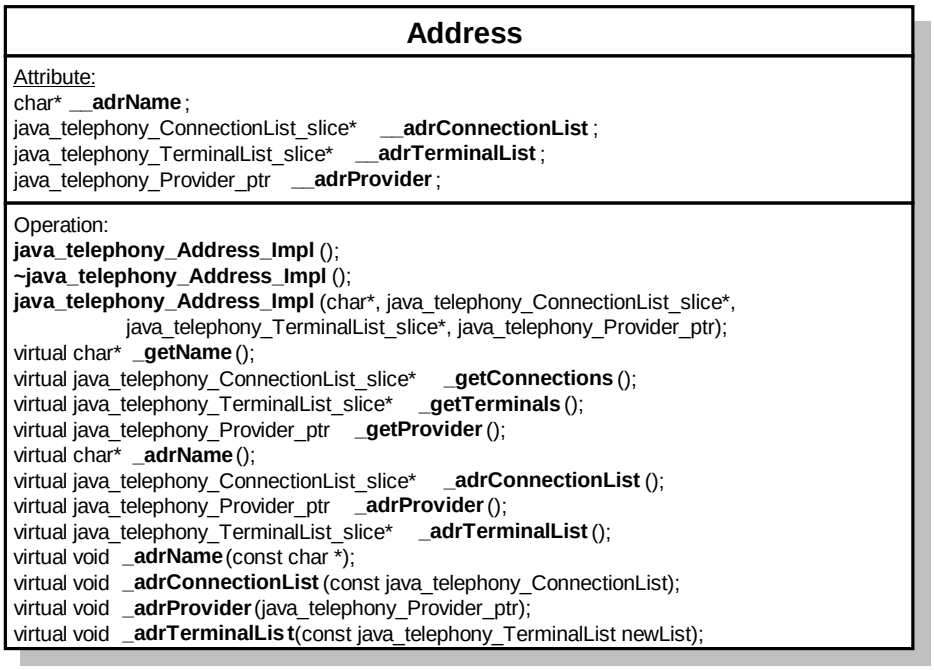
- *Address,*
- *Call,*
- *Connection,*
- *Provider,*
- *Terminal,*

sowie aus der Package *jtapi.telephony.callcenter*:

- *CallCenterProvider,*
- *RouteAddress.*

Die konkrete Implementierung der Klassen erfolgt in C++. Damit sie auch unter Java ansprechbar sind, müssen sie dem ORB bekannt sein, d.h. alle Klassen werden zunächst in IDL beschrieben. Mit dem IDL-Compiler (*idl* für C++, *jidl* für Java) werden Skeleton-Klassen erzeugt, von denen die Implementationsklassen abgeleitet werden. Die Methoden der diversen Klassen wurden in der IDL-Beschreibung mit einem voranstehenden Unterstrich definiert. Der Vorteil ist, daß auf Java-Seite kleine Wrappermethoden mit den JTAPI-konformen Namen die so definierten Methoden aufrufen und Exceptions werfen können, die sonst zusätzlich in IDL hätten spezifiziert werden müssen.
Die meisten Klassen besitzen als Attribut eine Referenz auf die zugehörige Instanz der Provider-Klasse. Details zu weiteren Attributen können, sofern sie nicht selbsterklärend sind, der JTAPI-Spezifikation entnommen werden.

4.7.3.1 Address



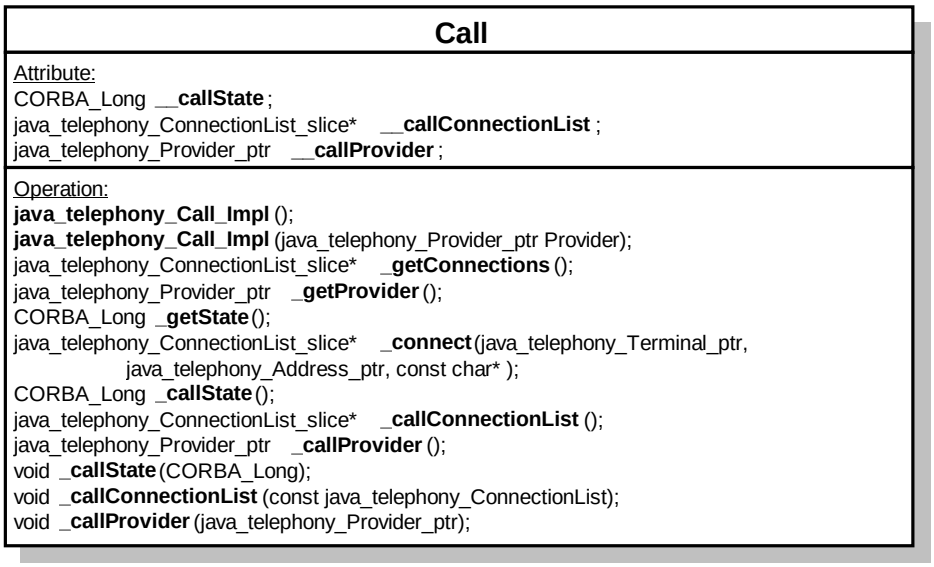
Das `Address`-Objekt bezeichnet im allgemeinen eine Telefonnummer. Die `Address`-Klasse besitzt vier Attribute, welche im Konstruktor mit den Parametern gesetzt werden. Für jedes Attribut existiert eine Methode zum Lesen und Schreiben. Die beiden Listentypen, sowie Provider wurden bereits in IDL definiert. Nur der Name ist vom einfachen Datentyp `String` bzw. `char*` in C++. Auf Java-Seite stellt die Klasse *StubForAddress* die entsprechenden Methodenaufrufe der Implementation bereit. `ConnectionList` beinhaltet alle von dieser Klasse ausgehenden Verbindungen. `TerminalList` repräsentiert die "Endgeräte", denen `Address` zugeordnet ist.

```
class java_telephony_Address_Impl : public java_telephony_Address_skel
{
  java_telephony_Address_Impl(char*,
                              java_telephony_ConnectionList_slice*,
                              java_telephony_TerminalList_slice*,
                              java_telephony_Provider_ptr);

  // Methoden zum Lesen der lokalen Variablen
  virtual char* _adrName();
  virtual java_telephony_ConnectionList_slice* _adrConnectionList();
  virtual java_telephony_Provider_ptr _adrProvider();
  virtual java_telephony_TerminalList_slice* _adrTerminalList();

  // Methoden zum Setzen der lokalen Variablen
  virtual void _adrName(const char *);
  virtual void _adrConnectionList(const java_telephony_ConnectionList);
```

```
  virtual void _adrProvider(java_telephony_Provider_ptr);
  virtual void _adrTerminalList(const java_telephony_TerminalList newList);
};
```

4.7.3.2 Call

Das `Call`-Objekt wird dazu eingesetzt, um ein Gespräch logischals Menge von `Connection`-Objekten zu repräsentieren. Ähnlich dem `Address`-Objekt, besitzt die `Call`-Klasse drei Attribute, welche mit Hilfe von jeweils drei Methoden gesetzt bzw. ausgelesen werden können. Der Konstruktor definiert nur den Provider. Auf Java-Seite ruft die Klasse *StubForCall* die entsprechenden Methoden der Implementation auf.




```
class java_telephony_Call_Impl : public java_telephony_Call_skel {

  java_telephony_Call_Impl(java_telephony_Provider_ptr Provider);

  // Lesen und Setzen von lokalen Variablen
  CORBA_Long _callState();
  java_telephony_ConnectionList_slice* _callConnectionList();
  java_telephony_Provider_ptr _callProvider();
  void _callState(CORBA_Long);
  void _callConnectionList(const java_telephony_ConnectionList);
  void _callProvider(java_telephony_Provider_ptr);
};
```

4.7.3.3 Connection

Das `Connection`-Objekt stellt eine Verbindung dar und repräsentiert eine Beziehung zwischen einem `Call`- und einem `Address`-Objekt. Auch in der Klasse `Connection` werden drei Attribute durch entsprechende Methoden ausgelesen oder geschrieben. Der Konstruktor initialisiert die Referenz auf den `Call` und den `Provider`. Auf Java-Seite ruft die Klasse *StubForConnection* die entsprechenden Methoden der Implementation auf.

```
class java_telephony_Connection_Impl : public
                                       java_telephony_Connection_skel {

  java_telephony_Connection_Impl(java_telephony_Call_ptr,
                                 java_telephony_Address_ptr);
  // lokale Variable lesen /setzen
  virtual java_telephony_Call_ptr _conCall();
  virtual java_telephony_Address_ptr _conAddress();
  virtual CORBA_Long _conState();
  virtual void _conCall(java_telephony_Call_ptr);
```

```
  virtual void _conAddress(java_telephony_Address_ptr);
  virtual void _conState(CORBA_Long);
};
```

Connection
Attribute: CORBA_Long __**conState**; java_telephony_Address_ptr __**conAddress**; java_telephony_Call_ptr __**conCall**;
Operation: **java_telephony_Connection_Impl**(java_telephony_Call_ptr, java_telephony_Address_ptr); **java_telephony_Connection_Impl**(); **~java_telephony_Connection_Impl**(); virtual java_telephony_Call_ptr _**getCall**(); virtual java_telephony_Address_ptr _**getAddress**(); virtual CORBA_Long _**getState**(); virtual void _**disconnect**(); virtual java_telephony_Call_ptr _**conCall**(); virtual java_telephony_Address_ptr _**conAddress**(); virtual CORBA_Long _**conState**(); virtual void _**conCall**(java_telephony_Call_ptr); virtual void _**conAddress**(java_telephony_Address_ptr); virtual void _**conState**(CORBA_Long);

4.7.3.4 Provider

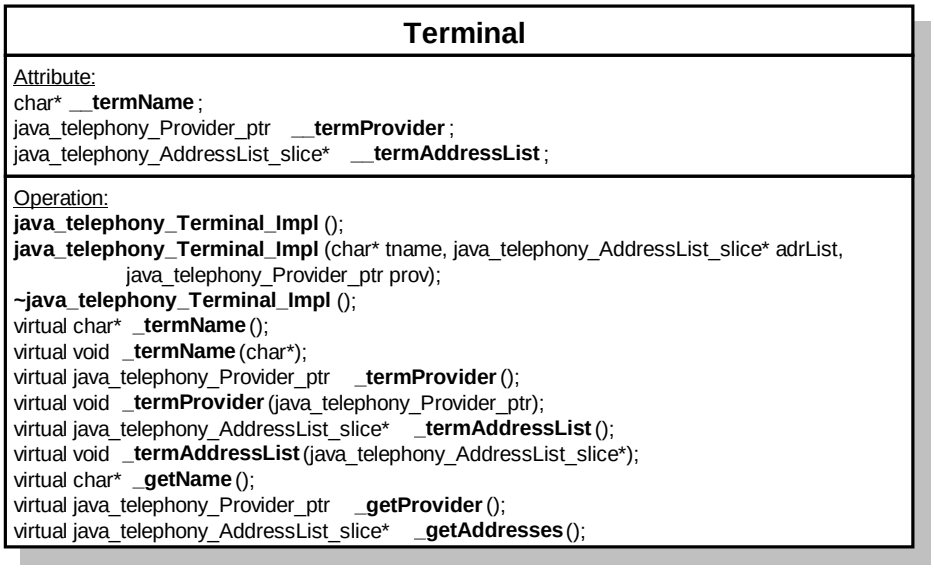
Provider
Attribute: CORBA_Long **State**; char* **Name**; cProvider* **TKStream**; EventMessage* **EventChannel**; java_telephony_AddressList_slice * **AddressList**; java_telephony_TerminalList_slice * **TerminalList**; java_telephony_CallList_slice * **CallList**;
java_telephony_Provider_Impl (); **java_telephony_Provider_Impl** (cProvider*, EventMessage*); **virtual ~java_telephony_Provider_Impl** (); virtual CORBA_Long _**getState**(); virtual char* _**getName**(); virtual java_telephony_CallList_slice* _**getCalls**(); virtual java_telephony_Address_ptr _**getAddress**(const char* number); virtual java_telephony_AddressList_slice* _**getAddresses**(); virtual java_telephony_TerminalList_slice* _**getTerminals**(); virtual java_telephony_Terminal_ptr _**getTerminal**(const char* name); virtual void _**shutdown**(); virtual java_telephony_Call_ptr _**createCall**(); virtual cProvider * _**get_cProvider**();

Der `Provider`, von dem sich der `CallCenterProvider` ableitet, stellt das wichtigste Objekt dar. Vom `Provider` ausgehend sind alle anderen Objekte erreichbar, da er Referenzen auf jene in Form von Listen parat hält. So verwaltet er die `Address-`, `Call-`, `Connection`,- und `Terminal`-Objekte. Der Konstruktor erhält auf C++-Seite als (nicht in JTAPI definierten) Parameter eine Referenz auf den `cProvider` und den `EventChannel`. Damit ist dieses Objekt in der Lage, auf die Anlage zuzugreifen. Die restlichen Methoden dienen der Datenmanipulation.

```
class java_telephony_Provider_Impl : public java_telephony_Provider_skel {

  java_telephony_Provider_Impl(cProvider*, EventMessage*);
  virtual CORBA_Long _getState();
  virtual char* _getName();
  virtual java_telephony_CallList_slice* _getCalls();
```

```
  virtual java_telephony_Address_ptr _getAddress(const char* number);
  virtual java_telephony_AddressList_slice* _getAddresses();
  virtual java_telephony_TerminalList_slice* _getTerminals();
  virtual java_telephony_Terminal_ptr _getTerminal(const char* name);
  virtual void _shutdown();
  virtual java_telephony_Call_ptr _createCall();
  virtual cProvider* _get_cProvider();
};
```

4.7.3.5 Terminal



Ein `Terminal`-Objekt stellt physikalisch oder logisch eines oder mehrere Endgeräte dar. Die drei Attribute werden auch hier durch entsprechende Methoden manipuliert. Im Konstruktor wird bereits der Name, der `Provider`, sowie eine `Address`-Liste übergeben, da ein Endgerät auch über mehrere Telefonnummern erreichbar sein kann.

```
class java_telephony_Terminal_Impl : public java_telephony_Terminal_skel {

  java_telephony_Terminal_Impl(char* tname,
                               java_telephony_AddressList_slice* adrList,
                               java_telephony_Provider_ptr prov);
  virtual char* _termName ();
  virtual void _termName (char*);
  virtual java_telephony_Provider_ptr _termProvider ();
  virtual void _termProvider (java_telephony_Provider_ptr);
  virtual java_telephony_AddressList_slice* _termAddressList ();
  virtual void _termAddressList (java_telephony_AddressList_slice*);
  virtual char* _getName ();
  virtual java_telephony_Provider_ptr _getProvider ();
  virtual java_telephony_AddressList_slice* _getAddresses ();
};
```

4.7.3.6 CallCenterProvider

`CallCenterProvider` erbt von `Provider` und bietet zusätzlich die Möglichkeit, eine Liste von Referenzen auf `RouteAddress`-Objekte zu speichern. Eine Instanz von `CallCenterProvider` wird vom MainFrameThread erzeugt und dem ORB zur Verfügung gestellt. Die Passagen, die sich auf `ACDAddress`-Objekte beziehen, wurden durch ein entsprechendes Makro auf `Address`-Objekte abgebildet, da eine Definition aber keine Implementierung benötigt wird.
Der Konstruktor erhält auch eine Referenz auf den `cProvider` und `EventMessage`, um sie beim impliziten Aufruf des `Provider`-Konstruktors zu übergeben.

CallCenterProvider
Attribute: java_telephony_callcenter_RouteAddressList_slice * **routeAddressList**; java_telephony_callcenter_ACDAddressList_slice * **ACDAddressList**; java_telephony_callcenter_ACDManagerAddressList_slice * **ACDManagerAddressList**;
Operation: **java_telephony_callcenter_CallCenterProvider_Impl**(); **java_telephony_callcenter_CallCenterProvider_Impl**(cProvider*, EventMessage*); ~**java_telephony_callcenter_CallCenterProvider_Impl**(); virtual java_telephony_callcenter_RouteAddressList_slice* _**getRouteableAddresses**(); virtual java_telephony_callcenter_ACDAddressList_slice* _**getACDAddresses**(); virtual java_telephony_callcenter_ACDManagerAddressList_slice* _**getACDManagerAddresses**();

```
class java_telephony_callcenter_CallCenterProvider_Impl :
                  public java_telephony_Provider_Impl,
                  public java_telephony_callcenter_CallCenterProvider_skel{

  // to be implemented, now substituted by Address
  java_telephony_callcenter_ACDAddressList_slice *ACDAddressList;
  java_telephony_callcenter_ACDManagerAddressList_slice
                                                *ACDManagerAddressList;
  virtual java_telephony_callcenter_RouteAddressList_slice*
                                                _getRouteableAddresses();

  // keine Rueckgabewerte bis Dato, da nicht implementiert
  virtual java_telephony_callcenter_ACDAddressList_slice*
                                                _getACDAddresses();
  virtual java_telephony_callcenter_ACDManagerAddressList_slice*
                                                _getACDManagerAddresses();

  java_telephony_callcenter_CallCenterProvider_Impl(cProvider*,
                                                      EventMessage*);
};
```

4.7.3.7 RouteAddress

RouteAddress
Attribute: CORBA_Long __**adrRoutingId**; java_telephony_callcenter_RouteCallback_ptr **cb**;
Operation: **java_telephony_callcenter_RouteAddress_Impl** (); **java_telephony_callcenter_RouteAddress_Impl** (char*, java_telephony_Provider_ptr); **java_telephony_callcenter_RouteAddress_Impl** (char*, java_telephony_ConnectionList_slice*, java_telephony_TerminalList_slice*, java_telephony_Provider_ptr); void _**registerRouteCallback** (java_telephony_callcenter_RouteCallback_ptr); void _**cancelRouteCallback** (java_telephony_callcenter_RouteCallback_ptr); void _**adrRoutingId** (CORBA_Long); CORBA_Long _**adrRoutingId** (); static const CORBA_Long **ROUTING_ID_NULL**;

`RouteAddress` erbt von `Address` und stellt zusätzlich ein Attribut zur Verfügung, das eine Referenz auf das `Callback` speichert. Der Konstruktor übernimmt Parameter für Namen, `ConnectionList`, `TerminalList` und `Provider`. Die für das Routing elementaren Methoden sind *_registerRouteCallback()* sowie *_cancelRouteCallback()*, die direkt auf die Anlage zugreifen. Das Attribut `RoutingID` speichert dabei die Nummer des gelieferten Routing-Dialogs.

```
class java_telephony_callcenter_RouteAddress_Impl :
                      public java_telephony_Address_Impl,
                      public java_telephony_callcenter_RouteAddress_skel {
```

```
  java_telephony_callcenter_RouteAddress_Impl(char*,
                        java_telephony_ConnectionList_slice*,
                        java_telephony_TerminalList_slice*,
                        java_telephony_Provider_ptr);

  void _registerRouteCallback(java_telephony_callcenter_RouteCallback_ptr);
  void _cancelRouteCallback  (java_telephony_callcenter_RouteCallback_ptr);

  void _adrRoutingId (CORBA_Long);
  CORBA_Long _adrRoutingId ();
  static const CORBA_Long ROUTING_ID_NULL;
};
```

_cancelRouteCallback() ruft die Methode *BlockRouteRegisterCancel()* des Objektes `TKStream` auf und wartet das Event als Antwort ab.

```
void java_telephony_callcenter_RouteAddress_Impl::_cancelRouteCallback (
                          java_telephony_callcenter_RouteCallback_ptr rcb) {
  // Cancel routing for this address
  if (_adrProvider() == NULL){
    throw new java_telephony__PlatformException ();
  }
  RetCode_t Result;
  CSTAEvent_t cstaEvent;

  cProvider *cP = VariablenInterface::TKStream;
 HANDLE Ev = (VariablenInterface::EventChannel)->CreateEventChannel();

  Result = cP->BlockRouteRegisterCancel(cstaEvent, Ev, _adrRoutingId());
  if (Result < 0 ||
      cstaEvent.eventHeader.eventType != CSTA_ROUTE_REGISTER_CANCEL_CONF){
    throw new java_telephony__PlatformException();
  }
  _adrRoutingId (ROUTING_ID_NULL);
  if (cb == rcb){
    cb = java_telephony_callcenter_RouteCallback::_nil();
  }
}
```

Ähnlich der Methode *_cancelRouteCallback()* ruft *_registerRouteCallback()* mit Hilfe des Objektes `TKStream` die Methode *BlockRouteRegisterReq()* auf und wartet auf das Event, das als Bestätigung von der Anlage gesandt wird.

```
void java_telephony_callcenter_RouteAddress_Impl::_registerRouteCallback(
                          java_telephony_callcenter_RouteCallback_ptr rcb) {
  // Register this address for routing via cstaRegisterRouteReq
  if (_adrProvider() == NULL) {
    throw new java_telephony__PlatformException ();
  }
  RetCode_t Result;
  CSTAEvent_t cstaEvent;

  cProvider *cP = VariablenInterface::TKStream;
 HANDLE Ev = (VariablenInterface::EventChannel)->CreateEventChannel();

  Result = cP->BlockRouteRegisterReq(cstaEvent, Ev, _adrName());
  if (Result < 0 ||
      cstaEvent.eventHeader.eventType != CSTA_ROUTE_REGISTER_REQ_CONF) {
    throw new java_telephony__PlatformException();
  }
  _adrRoutingId(
      cstavent.event.cstaConfirmation.u.routeRegister.registerReqID);
}
```

Aufgrund der Restriktionen, die die TK-Anlagensoftware zur Zeit noch auferlegt, registriert sich die Routing-Komponente lediglich auf eine ACD-Gruppe und führt über die bereits beschriebene Indirektionsstufe eine nachträgliche Zuordnung zwischen Routing-Ereignissen und gewählter Nummer durch.

4.7.3.8 Restliche IDL-Definitionen

In zwei weiteren IDL-Dateien (TypsDefs.idl und Exceptions.idl) sind noch weitere Hilfskonstrukte definiert. So befinden sich in TypsDefs.idl die folgenden Einträge für Listen aus Objekten anderer IDL-Klassen:

- `AddressList`,
- `ConnectionList`,
- `TerminalList`,
- `CallList`,
- `RouteAddressList`,
- `RouteCallbackList` und
- `RouteSessionList`.

In Exceptions.idl befinden sich die Definitionen der Ereignisse zur Ausnahmebehandlung:

- `PlatformException`,
- `InvalidArgumentException`,
- `InvalidStateException`,
- `InvalidPartyException`,
- `ResourceViolationException`,
- `MethodNotSupportedException` und
- `InvalidObjectException`.

4.7.4 Routing via CORBA-Implementation

Die im vorherigen Abschnitt dargestellten CORBA-Implementationen bezogen sich im wesentlichen auf die Interface-Definitionen der Package *java.telephony* und nahmen nur wenig Bezug auf das Routing. Die Implementation war so ausgelegt, daß die gesamte Verwaltung auf C++-Seite stattfand und auf Java-Seite nur kleine Wrapper-Methoden die CORBA-Implementationsmethoden aufriefen. Die Implementation des Routings legt die andere Philosophie zugrunde. Auf C++-Seite wird nur das wesentliche erledigt, um die Verwaltung auf Java-Seite auszulagern. Dafür spricht die höhere Performanz, da weniger Netzverbindungen aufgebaut werden müssen.

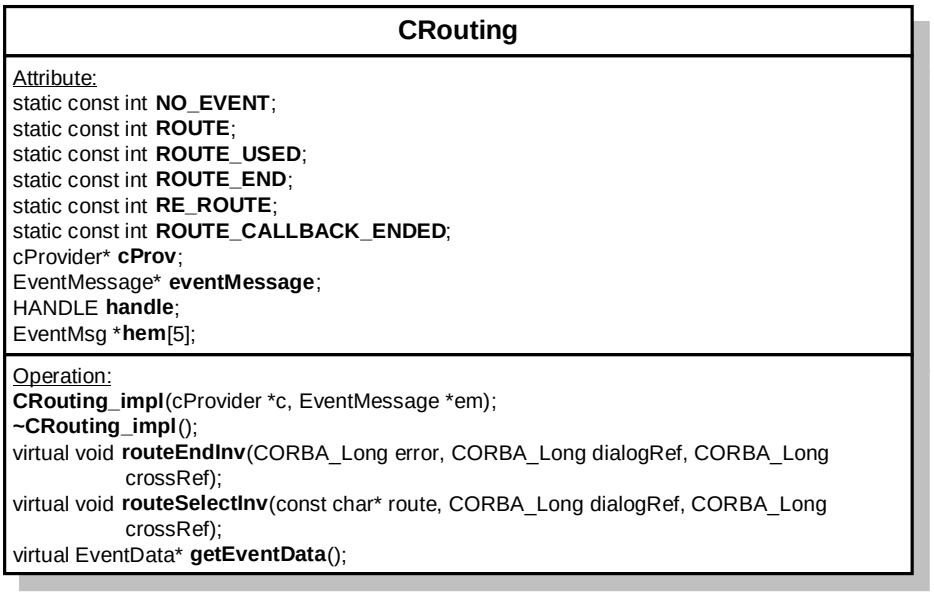
4.7.4.1 EventData

`EventData` dient dazu, Routing-Events zwischen C++- und Java-Seite auszutauschen. Dabei enthält die IDL-Datenstruktur alle Attribute, die für die Routing-Events benötigt werden.

```
struct EventData {
  long eventType;
  long dialogRef;
  long crossRef;
  string currentRoute;
  long day;
  long hour;
  long minute;
 };
```

4.7.4.2 CRouting

Die Implementation der Klasse `CRouting_impl` dient dazu, Events von der Anlage abzufragen, sowie Methoden der Klasse `cProvider` aufzurufen. Mehr Funktionalität wird für das Routing auf C++-Seite nicht benötigt.




```
class CRouting_impl : public CRouting_skel {
 CRouting_impl(cProvider *c, EventMessage *em);
void routeEndInv(CORBA_Long error, CORBA_Long dialogRef,
                                CORBA_Long crossRef);
  void routeSelectInv(const char* route, CORBA_Long dialogRef,
                                CORBA_Long crossRef);
  EventData* getEventData();
};
```

Der Konstruktor initialisiert die Referenzen für `cProvider` und `EventMessage` und erzeugt über letzteres ein `HANDLE`, das den EventChannel darstellt. Auf diesem EventChannel registriert sich `CRouting` schließlich auf die fünf relevanten Routingevents.

```
CRouting_impl::CRouting_impl(cProvider* c, EventMessage* em) {
  cProv = c;
  eventMessage = em;
  handle = em->CreateEventChannel();
 hem[0] = em->RegisterEventMsgSystem(handle,
                                (EventType_t) CSTA_ROUTE_REGISTER_ABORT);
  hem[1] = em->RegisterEventMsgSystem(handle,
                                (EventType_t) CSTA_ROUTE_REQUEST);
  hem[2] = em->RegisterEventMsgSystem(handle,
                                (EventType_t) CSTA_RE_ROUTE_REQUEST);
  hem[3] = em->RegisterEventMsgSystem(handle,
                                (EventType_t) CSTA_ROUTE_USED);
  hem[4] = em->RegisterEventMsgSystem(handle,
                                (EventType_t) CSTA_ROUTE_END);
}
```

Die Methode *EventData()* ruft die Methode *PollEvent()* von `cProvider` auf, um nichtblockierend ein neues Event abzuholen. Liegt eines vor, so werden die relevanten Daten ausgelesen und entsprechend in das zuvor gegründete `EventData` geschrieben, um dieses als Funktionsergebnis zurückzuliefern. Um das Datum zu setzen, werden die entsprechenden Variablen aus der Klasse `Variableninterface` ausgelesen. Die alternative Berechnungen sowohl mit als auch ohne Realisierung der Routing-Eventzuordnung über ACD-Gruppen ist im folgenden Codefragment enthalten.

```
EventData *CRouting_impl::getEventData() {
  // EventMessage - Events abholen, Parameter isolieren, EventData
  // zusammenbauen
  // return EventData-Struct*
  long callid;

  CSTAEvent_t eventBuf;
  EventData* eventData = new EventData;
  eventData->day = VariablenInterface::Day; ...
  if (eventMessage->PollEvent (eventBuf, handle)) {
    switch(eventBuf.eventHeader.eventType){
      case(CSTA_ROUTE_REQUEST): {
      eventData->eventType = ROUTE;
      eventData->dialogRef =
            eventBuf.event.cstaRequest.u.routeRequest.routeRegisterReqID;
      eventData->crossRef =
            eventBuf.event.cstaRequest.u.routeRequest.routingCrossRefID;

      // Since we use ACD-RouteRequests we do not obtain the destination
      // string and insert it from the list of pending calls from cProvider

      callid = eventBuf.event.cstaRequest.u.routeRequest.routedCall.callID;
      if (cProv->getPendingCallString(callid) == NULL) {
        eventData->currentRoute = CORBA_string_dup("0");
      }
      else {
        eventData->currentRoute = CORBA_string_dup(
                                    cProv->getPendingCallString(callid));
        cProv->deletePendingCall(callid);
      }

//       This should be used if real LCR-RouteReq are available from ICC
//       eventData->dialogRef =
//       eventBuf.event.cstaRequest.u.routeRequestExt.routeRegisterReqID;
//       eventData->crossRef =
//         eventBuf.event.cstaRequest.u.routeRequestExt.routingCrossRefID;
//       eventData->currentRoute = CORBA_string_dup(
          eventBuf.event. ... .routeRequestExt.currentRoute.deviceID);
        break;
      }
      case (CSTA_RE_ROUTE_REQUEST):{
        eventData->eventType = RE_ROUTE;
        ...
        break;
      }
      case(CSTA_ROUTE_USED):{
        eventData->eventType = ROUTE_USED;
        ...
        break;
      }
      case (CSTA_ROUTE_END):
        ...
      case (CSTA_ROUTE_REGISTER_ABORT):
        ...
      }
    }
  }
  else { // es lag kein Event vor
    eventData->eventType = NO_EVENT;
    eventData->currentRoute = CORBA_string_dup("NO_EVENT");
  }
  return eventData;
}
```

4.7.4.3 CRoutingServer

Diese Klasse dient lediglich dazu, ein Objekt der Klasse `CRouting_impl` zu erzeugen und dessen Funktionalität über CORBA anderen CORBA-Objekten zur Verfügung zu stellen. Dementsprechend besitzt die Klasse nur einen Konstruktor, der die gesamte Arbeit übernimmt. Da `CRouting` einen Verweis auf das `cProvider`- und das `EventMessage`-Objekt benötigt, werden beide Referenzen als Parameter übergeben. Instanziiert wird `CRoutingServer` wie bereits erwähnt im WorkerThread.

CRoutingServer
Operation: **CRoutingServer**(); **CRoutingServer**(cProvider* c, EventMessage* em); virtual **~CRoutingServer**();

```
CRoutingServer::CRoutingServer(cProvider* c, EventMessage* em)
{
  int argc = 0;
  char** argv = NULL;

  CORBA_ORB_var orb = CORBA_ORB_init(argc, argv);
  CORBA_BOA_var boa = orb -> BOA_init(argc, argv);

  CRouting_var p = new CRouting_impl(c, em);

  CORBA_String_var s = orb -> object_to_string(p);
  const char* refFile = "C:\\public\\CRouting.ref";
  ofstream out(refFile);
  out << s << endl;
  out.close();

  boa -> impl_is_ready(CORBA_ImplementationDef::_nil());
}
```

4.7.4.4 EventThread

EventThread

Attribute:
private JRouting **jRouting** = null;
private CRouting **cRouting** = null;

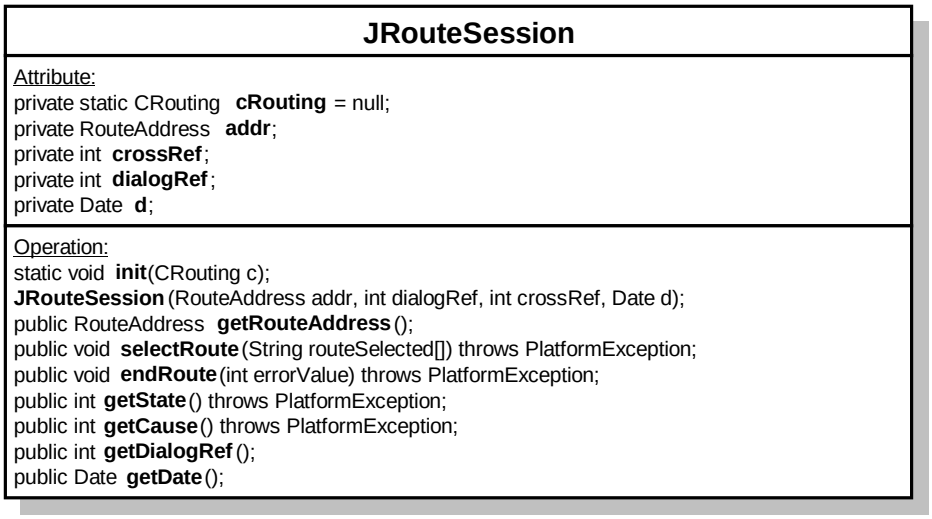
Operation: public **EventThread**(JRouting jRouting, CRouting cRouting); public void **run**();

Bei *EventThread* handelt es sich um eine Java-Klasse, die das *java.lang.Thread*-Interface implementiert. Es wird vom Router instanziiert und fragt in definierten Intervallen Routing-Events auf C++-Seite bei der JTAPI-Implementation ab. Dazu bedient er sich des `CRouting`-Objekts, das durch `CRoutingServer` als CORBA-Implementation von Java aus ansprechbar ist. Der Konstruktor setzt nur die beiden Attribute für Referenzen auf `cRouting` und *jRouting*. Mit Hilfe der Methode *getEventData()* von `CRouting` wird ein Event abgeholt und in Abhängigkeit des Typs eine entsprechende Methode in *JRouting* aufgerufen.

```
public void run() {
  while (true) {
    //endless loop is used to get all events as long the thread is running
    try {
      eventData = cRouting.getEventData();  // get the event
    }
    Date d = new Date(eventData.day, eventData.hour, eventData.minute);
    switch (eventData.eventType) {
```

```
      case RouteSession.ROUTE:  // cstaRouteRequestEvent
        jRouting.handleRouteRequest (eventData.dialogRef,
                                     eventData.crossRef,
                                     eventData.currentRoute, d);
        break;
      case RouteSession.RE_ROUTE: // cstaReRouteRequestEvent
        jRouting.handleRerouteRequest (eventData.crossRef, d);
        break;
      case RouteSession.ROUTE_USED: // cstaRouteUsedEvent
        jRouting.handleRouteUsedEvent (eventData.crossRef);
        break;
      case RouteSession.ROUTE_END: // cstaRouteEndEvent
        jRouting.handleRouteEndEvent (eventData.crossRef);
        break;
      case RouteSession.ROUTE_CALLBACK_ENDED:
        // cstaRouteRegisterCancelConfEvent
        jRouting.handleRouteRegisterAbortAndCancel (eventData.dialogRef);
        break;
    }
    Thread.sleep(100);
  }
}
```

4.7.4.5 JRouteSession



JRouteSession implementiert das JTAPI-Interface *RouteSession*. Wie der Konstruktor zeigt, speichert es alle relevanten Informationen für eine *RouteSession*: die Zieltelefonnummer, das aktuelle Datum, sowie die Referenzen für den gesamten Routing-Dialog einer CTI-Applikation (`dialogRef`) und die aktuelle Routing-Anfrage (`crossRef`). Diese Daten können mit Hilfe von entsprechenden Methoden gelesen und geschrieben werden.

```
JRouteSession(RouteAddress addr, int dialogRef, int crossRef, Date d){
  this.addr=addr;
  this.crossRef=crossRef;
  this.dialogRef= dialogRef;
  this.d= d;
}
```

Die Methode *selectRoute()* wird mit dem Routing-Ergebnis aufgerufen. Mit dieser Information, sowie Dialog- und Routing-Referenz wird die `cRouting`-Methode *routeSelectInv()* aufgerufen.

```
public void selectRoute(String routeSelected[]) throws PlatformException{
  if (routeSelected.length!=1)
    throw new PlatformException("RouteSession: length of array
                                 routeSelected is not 1");
```

```
  cRouting.routeSelectInv(routeSelected[0], dialogRef, crossRef);
}
```

Ähnlich verhält sich die Methode *endRoute()*, welche mit einem Fehlerwert anstatt des Routing-Ergebnisses die Methode *routeEndInv()* von `cRouting` aufruft.

```
public void endRoute(int errorValue) throws PlatformException{
  cRouting.routeEndInv(errorValue, dialogRef, crossRef);
}
```

4.7.4.6 JRoutingSessionEvent-Klassen

JRouteSessionEvent
Attribute: protected java.telephony.callcenter.RouteSession **session**; private Date **date**;
Operation: **JRouteSessionEvent** (java.telephony.callcenter.RouteSession s); **JRouteSessionEvent** (java.telephony.callcenter.RouteSession s, Date d); public java.telephony.callcenter.RouteSession **getRouteSession** (); public Date **getDate**();

Für jeden Typ von Routing-Events existiert in JTAPI ein eigenes Interface, d.h. es werden im folgenden fünf Implementationen benötigt.
Für einen *ReRouteRequest* und als Basisklasse für die anderen Event-Klassen, die sich davon ableiten, wird ein *JRouteSessionEvent*-Objekt erstellt, welches das JTAPI-Interface *RouteSessionEvent* implementiert. Es besitzt nur zwei Attribute, eine Referenz auf das zugehörige *RouteSession*-Objekt, sowie ein Datumsobjekt. Außer dem Konstruktor, der beide Attribute initialisiert, existieren zwei Methoden, welche die beiden Attribute auslesen.

```
class JRouteSessionEvent implements RouteSessionEvent{
  JRouteSessionEvent(java.telephony.callcenter.RouteSession s){
    session=s;
    date = new Date();
  }
  JRouteSessionEvent(java.telephony.callcenter.RouteSession s, Date d){
    session=s;
    date = d;
  }
  public java.telephony.callcenter.RouteSession getRouteSession(){
    return session;
  }
  public Date getDate(){
    return date;
  }
}
```

Bei Auftreten eines *RouteRequest* wird ein *JRouteEvent*-Objekt erstellt, welches *RouteEvent* implementiert und von *JRouteSessionEvent* erbt. Der Konstruktor ruft nur den seiner Superklasse *JRouteSessionEvent* auf. Die zusätzlichen Methoden liefern zum größten Teil `null`, da sie für die Implementation des Routings nicht notwendig sind. Die Methode *getCurrentRouteAddress()* liefert über das *RouteSession*-Objekt und dessen Methode *getRouteAddress()* die Nummer, die geroutet wird. Die Methode *getRouteSelectAlgorithm()* liefert (passend zur Anwendung) die Konstante `SELECT_LEAST_COST`.

```
class JRouteEvent extends JRouteSessionEvent implements
                          java.telephony.callcenter.events.RouteEvent{
  JRouteEvent(java.telephony.callcenter.RouteSession s){
    super(s);
  }
```

```
  public RouteAddress getCurrentRouteAddress(){
    return session.getRouteAddress();
  }
  public Address getCallingAddress(){return null;}
  public Terminal getCallingTerminal(){return null;}
  public int getRouteSelectAlgorithm(){return SELECT_LEAST_COST;}
  public String getSetupInformation(){return null;}
}
```

Entsprechend wird bei Auftreten eines *RouteUsedEvent* ein *JRouteUsedEvent*-Objekt generiert, welches das *RouteUsedEvent*-Interface implementiert und von *JRouteSessionEvent* abgeleitet ist. Der Konstruktor ruft den des Vorgängers auf und die Methoden liefern als Wert `null` oder `false`, da sie nicht benötigt werden.

```
class JRouteUsedEvent extends JRouteSessionEvent implements
                      java.telephony.callcenter.events.RouteUsedEvent{
  JRouteUsedEvent(java.telephony.callcenter.RouteSession s){
    super(s);
  }
  public Terminal getRouteUsed(){return null;}
  public Terminal getCallingTerminal(){return null;}
  public Address getCallingAddress(){return null;}
  public boolean getDomain(){return false;}
}
```

Die *JRouteEndEvent*-Klasse ist noch simpler strukturiert als die vorhergehenden. Sie besitzt ausschließlich einen Konstruktor.

```
class JRouteEndEvent extends JRouteSessionEvent implements
                     java.telephony.callcenter.events.RouteEndEvent{
  JRouteEndEvent(java.telephony.callcenter.RouteSession s){super(s);}
}
```

Nun verbleibt also nur noch das *JRouteCallbackEndedEvent*, das einen Konstruktor und eine Methode besitzt, welche `null` liefert.

```
class JRouteCallbackEndedEvent extends JRouteSessionEvent implements
                   java.telephony.callcenter.events.RouteCallbackEndedEvent{
  JRouteCallbackEndedEvent(java.telephony.callcenter.RouteSession s){
    super(s);
  }
  public RouteAddress getRouteAddress(){return null;}
}
```

4.7.4.7 JRouting

JRouting
Attribute: private CallCenterProvider **prov**; private Hashtable **sessions**; private CRouting **cRouter**; private Thread **eventThread**;
Operation: **JRouting**(CallCenterProvider p); public void **handleRouteRequest** (int dialogRef, int crossRef, String currentRoute, Date d); public void **handleRerouteRequest** (int crossRef, Date d); public void **handleRouteUsedEvent** (int crossRef); public void **handleRouteEndEvent** (int crossRef); public void **handleRouteRegisterAbortAndCancel** (int crossRef);

Diese Klasse dient dazu, die diversen *RouteSessions* zu verwalten und den *EventThread* zu starten. *JRouting* wird über den *JTAPIImplementor* instanziiert, welcher wiederum über den *Router* gestartet wird.
Der Konstruktor verwaltet die *RouteSessions* mit Hilfe einer Hashtable. Eine Referenz auf das `cRouting`-Objekt wird über CORBA besorgt. Weiterhin wird der *EventThread* gestartet, der in Intervallen von 100 ms Events abfragt.

```
class JRouting{
 StubForCRouting cRouter;
 JRouting(CallCenterProvider p){
    sessions = new Hashtable();
    // JProvider wird benoetigt, um getAddress aufzurufen
    prov=p;
    // CRouting via CORBA suchen
    String args[] = null;
    ORB orb = ORB.init(args, new java.util.Properties());
    String ref = null;
    String refFile = "C:\\public\\CRouting.ref";
    BufferedReader in = new BufferedReader (new FileReader (refFile));
    ref = in.readLine();
    org.omg.CORBA.Object obj = orb.string_to_object(ref);
    if (obj == null) throw new RuntimeException();
    cRouter = CRoutingHelper.narrow(obj);

    // JRouteSession initialisieren, damit jede Instanz auf
    // CallCenterProvider.getAddress zugreifen kann
    JRouteSession.init(cRouter);
    eventThread = new Thread (new EventThread (this, cRouter));
    eventThread.start();
  }
}
```

Die Methode *handleRouteRequest()* wird von *EventThread* aufgerufen, falls ein *RouteRequest* auftrat. Da zu diesem (neuen) Routing-Dialog noch kein *RouteSession*-Objekt existiert, wird ein solches angelegt und anhand der `crossRef` in der Hashtable `sessions` abgelegt. Über den *Provider* wird anhand der Telefonnummer das dazugehörige *RouteAddress*-Objekt abgefragt, um über dessen Methode *getRouteCallback()*, das zugehörige *Callback*-Objekt zu erhalten. Schließlich wird nun die Methode *routeEvent()* aufgerufen, welche als Parameter ein *JRouteEvent* erhält.

```
public void handleRouteRequest(int dialogRef, int crossRef,
                               String currentRoute, Date d){

  RouteAddress addr =((RouteAddress)prov.getAddress(currentRoute));
  JRouteSession s = new JRouteSession(addr, dialogRef, crossRef, d);
  RouteCallback[] callbacks = addr.getRouteCallback();
    sessions.put(new Integer(crossRef),s);
  callbacks[0].routeEvent(new JRouteEvent(s));
}
```

handleReRouteRequest() wird vom *EventThread* aufgerufen, wenn er einen *ReRouteRequest* erhält. Anhand der `crossRef` wird die dazugehörige *RouteSession* ermittelt. Somit erhält man wieder das *Callback*-Objekt, welches die Methode *reRouteEvent()* bereitstellt. Als Parameter wird zuvor ein *JRouteSessionEvent* instanziiert.

```
public void handleRerouteRequest(int crossRef, Date d){
  JRouteSession s = (JRouteSession)sessions.get(new Integer(crossRef));
  int dialogRef = s.getDialogRef();
  RouteCallback[] callbacks = s.getRouteAddress().getRouteCallback();
  callbacks[0].reRouteEvent(new JRouteSessionEvent(s, d));
}
```

Wie zuvor bei *handleRerouteRequest()* wird auch in der Methode *handleRouteUsedEvent()* anhand der `crossRef` das *RouteSessionEvent* ermittelt. Mit einem *JRouteUsedEvent* wird die *routeUsedEvent()*-Methode des *Callback*-Objektes aufgerufen.

```
public void handleRouteUsedEvent(int crossRef){
  JRouteSession s = (JRouteSession)sessions.get(new Integer(crossRef));
  int dialogRef = s.getDialogRef();
  RouteCallback[] callbacks = s.getRouteAddress().getRouteCallback();
  callbacks[0].routeUsedEvent(new JRouteUsedEvent(s));
}
```

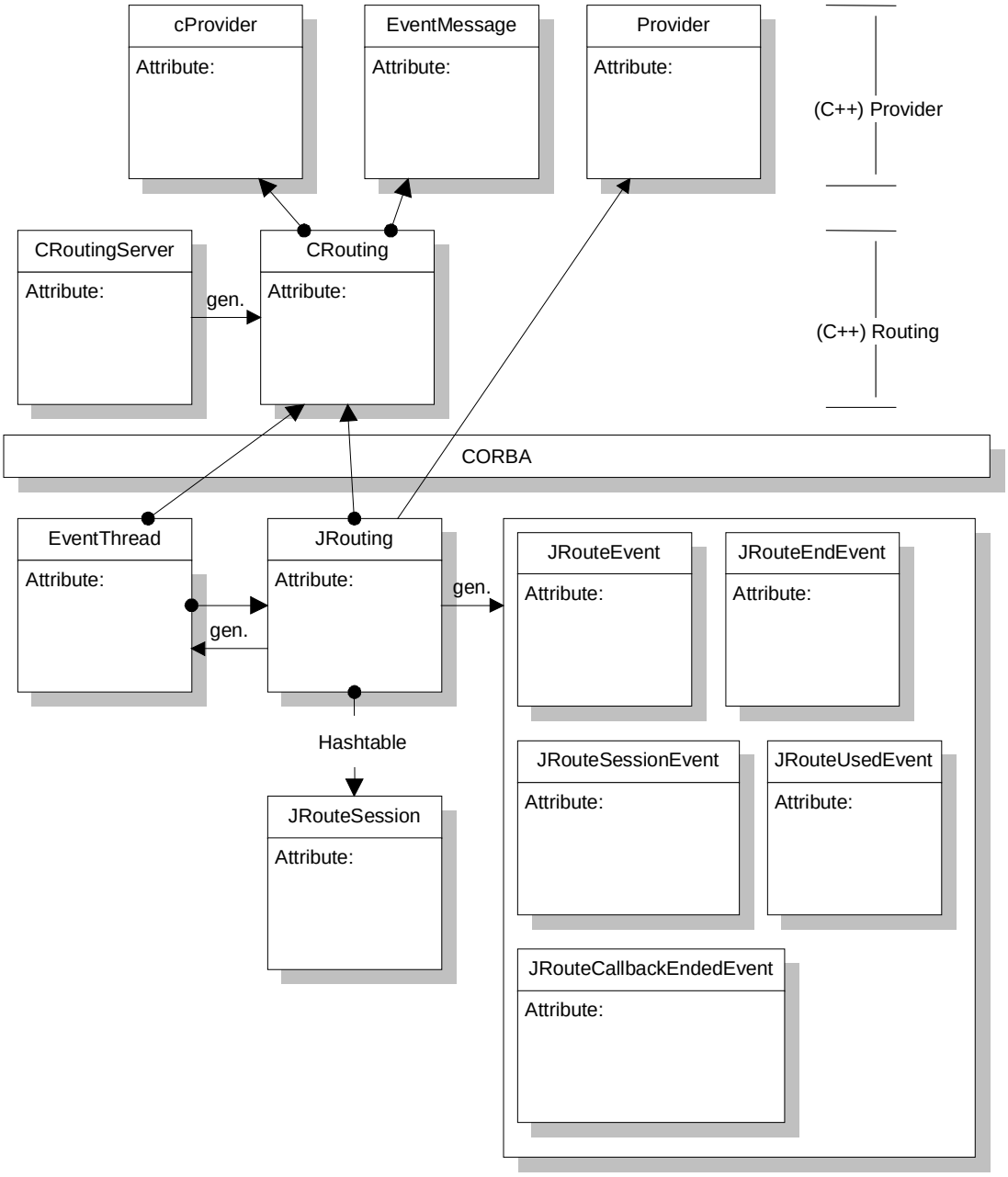


Abbildung 26 : Zusammenspiel von CRouting und JRouting

Bei *handleRouteEndEvent()* wird die Methode *routeEndEvent()* des *Callback*-Objektes mit einer Instanz von *JRouteEndEvent* aufgerufen.

```
public void handleRouteEndEvent(int crossRef){
  JRouteSession s = (JRouteSession)sessions.get(new Integer(crossRef));
  int dialogRef = s.getDialogRef();
  RouteCallback[] callbacks = s.getRouteAddress().getRouteCallback();
  callbacks[0].routeEndEvent(new JRouteEndEvent(s));
}
```

Analog dazu ruft die Methode *handleRouteRegisterAbortAndCancel()* die *Callback*-Methode *routeCallbackEndedEvent()* mit *JRouteCallbackEndedEvent* auf.

```
public void handleRouteRegisterAbortAndCancel(int crossRef){
  JRouteSession s = (JRouteSession)sessions.get(new Integer(crossRef));
  int dialogRef = s.getDialogRef();
  RouteCallback[] callbacks = s.getRouteAddress().getRouteCallback();
  callbacks[0].routeCallbackEndedEvent(new JRouteCallbackEndedEvent(s));
}
```

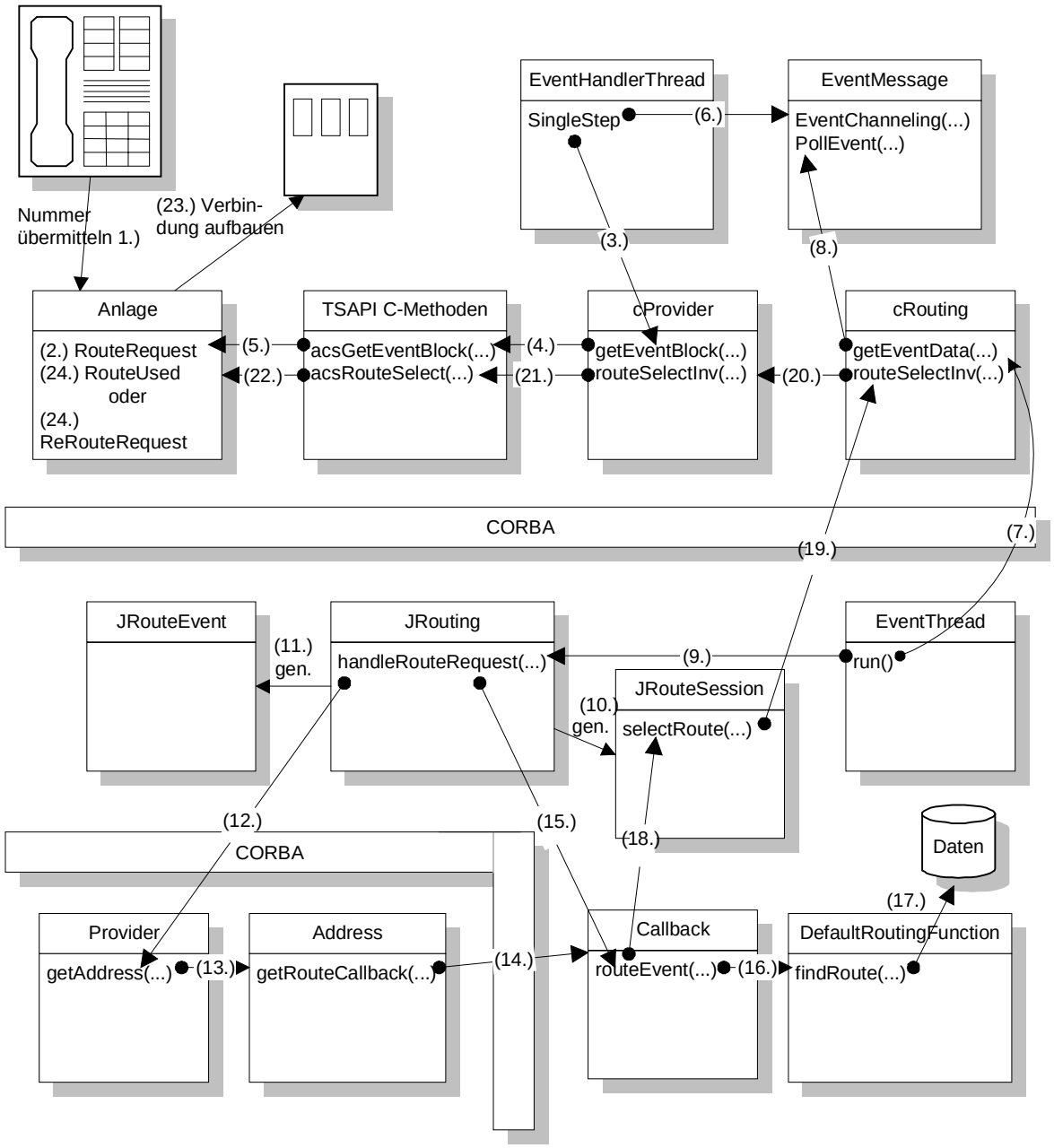


Abbildung 27 : Vollständiger Ablauf einer Routing-Anfrage

Abbildung 26 zeigt die Zusammenhänge zwischen den einzelnen Klassen auf Java- und C++-Seite der Routing-Komponente. In Abbildung 27 wird der gesamte Ablauf beim Aufbau einer Verbindung (ohne Verwendung von ACD) schematisch anhand der beteiligten Klassen und der Methodenaufrufe beschrieben. Nach dem Wählen der Zielnummer (1) löst der Voice-Switch eine Routing-Anfrage (2) aus. Diese wird in den Schritten (3 - 6) vom EventHandlerThread abgeholt und in der Eventliste dem LCR-Modul bereitgestellt. Die LCR-Komponente fragt periodisch das Vorhandensein neuer Requests ab (7 + 8) und übergibt diese Events der *JRouting*-Klasse zur Bearbeitung (9). Anhand des jeweiligen Eventtyps werden *RoutingSession*- und *RouteEvent*-Objekte erzeugt (10 + 11). Über die mitgelieferte Zielnummer werden ein zugehöriges JTAPI-`RouteAddress`-Objekt und ein Verweis auf den entsprechenden Callback (das LCR-Modul), der das JTAPI-Routing-Interface implementiert, ermittelt (12 - 14). *JRouting* leitet nun die Routing-Anfrage an den Callback weiter (15). In den Schritten (16 + 17) wird eine Wegwahl getroffen und über CORBA an die `cRouting`-Implementation weitergereicht (18 + 19). Die Skeleton-Methode übergibt anschließend die Daten über die CSTA-Schnittstelle an die Anlage (20 - 22), woraufhin diese die Verbindung schaltet (23). Weitere Routing-Events (24) werden analog behandelt.

4.8 Entwicklungsumgebung

Für die Entwicklung der Software standen Sun-Rechner (ELC) mit dem Betriebsystem Solaris 2.5 zur Verfügung. Sie wurden hauptsächlich für die Arbeit mit Java eingesetzt. Zusätzlich war ein PC mit Windows NT 4.0 Workstation vorhanden, der den Applikationsserver der TK-Anlage darstellte und auf dem der anlagenspezifische C++-Teil der Aufgabenstellungen implementiert wurde. Folgende Entwicklungswerkzeuge wurden eingesetzt:

- Java Development Kit 1.1.5 [W18],
- IBM Visual Age (Version 1.0),
- MS Visual C++ 5.0.

Weiterhin werden folgende Module anderer Hersteller verwendet:

- JTAPI-Spezifikation (Version 1.1) von Sun [W6]
- TSAPI-Spezifikation (Version 2.1) von Novell [W8]
- TSAPI-Implementation von Bosch Telecom,
- OmniBroker (Version 2.0.2) von Object Oriented Concepts (heißt mittlerweile ORBacus) [W19].

5 Erfahrungen

Die Vervollständigung der Lösungen aus dem Praktikum zu einer funktionierenden Demoversion für die CeBIT erfolgte teilweise im Anschluß an das Praktikum. Thomas Schoch und Marco Gruteser waren maßgeblich an diesen abschließenden Arbeiten beteiligt. Die hier beschriebenen rückblickenden Bemerkungen wurden von deshalb auch aus ihrer Sicht formuliert. Sie fassen aber auch weitgehend die Erfahrungen der anderen Gruppen zusammen.

5.1 Allgemeine Bemerkungen

Wie bereits mehrfach angesprochen, herrschte zu Beginn des Praktikums und der ersten Aufgabe ein Mangel an verständlichen Informationen. Es waren sogar einige Rückfragen nötig, um erst einmal einen groben Überblick über die zu lösende Aufgabe zu bekommen. Dadurch wurden die Arbeiten an der Aufgabe natürlich stark verzögert. Beim Entwurf der Applikation mußten dann noch einige Details geklärt werden.
Die JTAPI-Schnittstelle warf ebenfalls viele Fragen auf, vor allem da keine Implementation vorlag, mit der man experimentieren konnte. Auch nachdem wir uns auf ein Verfahren geeinigt hatten, wie die JTAPI-Routing-Schnittstelle im wesentlichen anzusprechen ist, blieben noch Fragen ungeklärt (z.B. wie man dem Anrufer ein Besetztzeichen zurückgibt).

Wie erwartet, kam es auch im Bezug auf die Arbeit im Team zu kleineren Problemen. Eine optimale Kommunikation war nicht immer gewährleistet, so daß manchmal Schnittstellen verschieden interpretiert wurden. Besonders mühselig gestaltete sich die Integration von verschiedenen Quelltextversionen. Obwohl wir uns zum Ziel setzten, die Quelltextdateien der jeweils anderen Gruppenmitglieder nicht zu verändern, um durch einfaches Zusammenfügen aller Quelltextdateien die aktuelle Version zu erhalten, war dies nicht immer möglich. Das Testen der eigenen Module erforderte ab und zu eine Korrektur oder die Integration einer weiteren Methode in den Dateien der anderen Gruppenmitglieder und dadurch kam es dann zu unterschiedlichen Weiterentwicklungen der gleichen Datei.

5.2 Erfahrungen mit der Sprache Java

Die Sprache Java war zwei Gruppenmitgliedern schon aus den Veranstaltung Grundzüge der Informatik I/II in den letzten beiden Semestern bekannt. Dennoch waren wir von den Fähigkeiten und der Arbeit mit ihr weitgehend positiv überrascht.
Im Vergleich zu C++ fällt auf, daß dem Programmierer viel Arbeit durch die Sprache an sich und durch die Standardklassenbibliothek abgenommen wird. So müssen in C++ immer eine Klassendeklaration und eine Implementation verwaltet werden. Dies geschieht meist in verschiedenen Dateien, welches oft zu Inkonsistenten führt. Weiterhin ist der Programmierer dafür verantwortlich, seine Deklarationen so zu organisieren, daß der Compiler nicht auf unbekannte Symbole trifft. Dadurch kommt es immer wieder zu Fehlern bei der Übersetzung, die einiges an Zeit verschlingen.
Die Sprache Java ist von ihren Möglichkeiten zwar weniger mächtig als C++, vermißt haben wir in der Praxis jedoch höchstens das Überladen von Operatoren. Die Einschränkung auf Referenzen und der integrierte Garbage-Collector vereinfachen die Entwicklung wesentlich. Beim robusten Programmieren in C++ verursacht das Implementieren einer dynamischen Speicherverwaltung und die Vermeidung der Fehler, die dabei auftreten können (nicht freigegebener Speicher, Zeiger auf gelöschte Objekte, etc.), sehr viel Aufwand, der beim Programmieren in Java einfach entfällt.
Überzeugt haben uns auch Objektserialisation und RMI. Die Konfigurationsdaten werden in unserer Architektur in vielen verschiedenen Objekten gespeichert, die sich teilweise gegenseitig referenzieren. Objektserialisation erlaubte es uns, diesen Graphen mit wenigen Zeilen Code zu speichern und wieder zu laden. Mit RMI konnten wir in wenigen Stunden die Kommunikation zwischen GUI-Client und LCR-Server implementieren. Nur einmal verursachte die Zusammenarbeit Probleme. Wir hatten *rmiregistry* aus einem anderen Verzeichnis heraus gestartet. Dadurch waren die Stub- und Skeleton-Dateien nicht mehr über den `CLASSPATH` zugänglich, was zu einer anfangs unverständlichen Exception führte.
Die Dokumentation der Klassenbibliothek kann man jedoch bestenfalls als unzureichend bezeichnen. Oftmals hilft nur Ausprobieren oder das Suchen nach anderen Beschreibungen und Einführungen im Internet oder in Büchern und Zeitschriften.
Ein wirklich großer Nachteil ist die geringe Auswahl von GUI Elementen, die das AWT bietet. Damit ließ sich unser ursprünglicher Entwurf der grafischen Benutzeroberfläche nicht realisieren und zwang uns so zu einer eher spartanischen Variante. Dieses Manko wird aber wohl demnächst mit Swing (Java Foundation Classes) aus JDK 1.2 bereinigt werden.

5.3 Erfahrungen mit der GUI

Mit der Entwicklungsumgebung *IBM VisualAge for Java 1.0* wurden fast durchweg gute Erfahrungen gemacht. Wir benutzten (bei der Entwicklung zu Hause) eine kostenlose Evaluierungsversion, so daß keinerlei Hilfe zur Verfügung stand sowie die Anzahl der selbst geschriebenen Klassen hundert (100) nicht überschreiten durfte. VisualAge ist so intuitiv zu bedienen, daß man die Entwicklungsumgebung nach wenigen Stunden schon fast optimal ausnutzen kann. Gerade die visuelle Programmierung ist sehr gut gelöst. Sogar die Aktionen, die ausgelöst werden, lassen sich visuell durch Zeiger realisieren. So verbindet man z.B. ein `OK`-Button mit der *dispose()*-Methode des Dialoges ohne ein Wort Kode zu tippen. VisualAge kompiliert inkrementell, so daß man zügig arbeiten und Testläufe starten kann, ohne Zeit zu verlieren. Voraussetzung ist aber ein schneller Rechner mit großem Hauptspeicher. Uns diente ein Pentium Pro 200 mit 64 MB RAM. Mit dieser Konfiguration ist das Arbeiten sehr bequem und schnell möglich. Bei geringerer Rechnerausstattung wird das inkrementelle Kompilieren zu langsam, so

daß man nach jeder Änderung an einer Methode etliche Sekunden warten muß, um weiterarbeiten zu können. VisualAge zeigt Packages, Klassen und Methoden nach verschiedenen Sortierungen an, so daß man sich schnell durch das Projekt hangeln kann. Auch die Suchfunktion ist sehr umfangreich. Diese reichhaltige Funktionalität ist nicht mit dem spartanisch ausgestatteten JDK zu vergleichen. Minutenlanges Kompilieren größerer Dateien an den Suns mit *javac* läßt jede Motivation schwinden. Einen großen Nachteil besitzt VisualAge jedoch: versucht man, eine Klasse, die visuell programmiert wurde, nach einem Export wieder zu importieren, so sind die visuellen Informationen verloren gegangen und man steht vor dem Scherbenhaufen, den VisualAge einem hinterlassen hat.
Bei der Ausprogrammierung einer GUI stellt man fest, daß der größte Teil rein schematisch ist und sich algorithmische Probleme eher am Rande stellen. Besonders beim Entfernen von Objekten oder bei der Auswahl eines anderen Objektes muß aufgepaßt werden, daß keine Inkonsistenzen entstehen. Auch die Korrekturfunktionalität via `Cancel`-Button muß bei mehreren Hierarchieebenen genau durchdacht sein. Während der Implementation kam es öfters vor, daß Daten entweder überschrieben oder nicht übernommen wurden. Durch genaues Durchspielen der Methoden kam man den Fehlern schnell auf die Schliche. Allgemein läßt sich feststellen, daß man mit Java und VisualAge relativ schnell größere Programme entwickeln kann, die wenig fehleranfällig sind. Dies liegt zum einem an der objektorientierten Struktur Javas und den Restriktionen, denen es im Vergleich zu C++ unterliegt. Schließlich kann man keinen Zeiger „verbiegen“, den man mit mühevoller Kleinarbeit suchen müßte.
Zum Schluß des Praktikums hin wurde die Arbeit zwar relativ streßig, um die Termine einzuhalten, doch stellte dies eine zusätzliche Herausforderung dar.
Im Rahmen der CeBIT '98 vom 19. bis 25. März wurde die fertiggestellte LCR-Studie am Stand der hessischen Hochschulen und am Stand von Bosch Telecom präsentiert. Die Resonanz darauf war überaus erfreulich. Für Bosch Telecom wurde das Ziel, die Offenheit des neuen ICC-Systems für die Programmierung durch externe Anbieter zu zeigen, voll erreicht. Viele Firmenkunden und Softwarehäuser, die CTI-Applikationen entwickeln, interessierten sich für das gemeinsame Projekt und seine (technischen) Hintergründe.
Als Hochschule war es uns selbst durch die Ausstellung möglich, Kompetenz in modernen Informationstechnologien zu demonstrieren und für Wissenstransfer von der Universität in die Industrie zu interessieren.

6 Ausblick

Abschließend sollen noch einige Ideen zur möglichen Weiterentwicklung und Verbesserung der präsentierten Lösung genannt werden.

6.1 Optimierung der Routing-Algorithmen

Wie bereits in Abschnitt 2.1.1 erläutert, können die TK-Kosten durch Einbeziehen weiterer Parameter in die Ermittlung des Providers weiter reduziert werden. Bezogen auf eine einzelne Verbindung ist hier der Parameter Verbindungsdauer kritisch. Wird ein längerer Zeitraum betrachtet, so ist auch die Zusammensetzung des gesamten TK-Aufkommens interessant, um Grundgebühren, Mindestumsätze und Rabatte der Provider optimal auszunutzen. Da weder die Dauer einer einzelnen Verbindung noch das TK-Aufkommen im voraus bekannt sind, ist die Berücksichtigung dieser Kriterien nicht einfach zu realisieren. Als Lösungsansätze kommen erstens das Abfragen weiterer Informationen vom Benutzer und zweitens der Einsatz von Heuristiken in Frage.
Der erste Ansatz verlagert das Problem der Bestimmung der Verbindungsdauer bzw. des TK-Aufkommens zumindest teilweise zum Benutzer. Dies kann sinnvoll sein, da der Benutzer meist über zusätzliche Informationen verfügt und dadurch bessere Abschätzungen liefern kann. Benutzer können in diesem Fall reale Personen oder aber Programme, die Verbindungen aufbauen, sein.
Werden Verbindungen durch Programme aufgebaut, so besitzen in manchen Fällen auch diese Programme Informationen, die für die LCR-Entscheidung wichtig sind. Soll beispielsweise eine Datei übertragen werden, so kann meistens anhand der Dateilänge und der verfügbaren Bandbreite eine recht genaue Abschätzung der Verbindungsdauer erfolgen. Solchen Programmen könnte eine Schnittstelle bereitgestellt werden, über die zusätzliche Informationen wie die Verbindungsdauer übergeben werden

können. Problematisch ist dabei jedoch, daß die eingesetzten Kommunikationsprogramme speziell an diese Schnittstelle angepaßt werden müßten.
Für die Eingabe einer Abschätzung des TK-Aufkommens kann im GUI-Modul eine Möglichkeit geschaffen werden. Der Administrator müßte dabei ein Gesamtaufkommen und die anteilsmäßige Verteilung auf verschieden Entfernungs- und Zeitzonen sowie eventuell Gesprächsdauern angeben.
Kritisch zu bewerten ist jedoch, daß hier dem Benutzer wieder Zusatzaufwand aufgebürdet wird, den man gerade durch ein transparentes, automatisches Routing vermeiden will.
Beim zweiten Ansatz bestimmt der LCR-Server selbst aus den ihm zur Verfügung stehenden Informationen mittels Heuristiken die wahrscheinliche Dauer für jede Verbindung und die voraussichtliche Zusammensetzung des TK-Aufkommens. Anhand der vom Administrator eingegebenen oder selbst protokollierten Einzelverbindungsnachweise könnte der LCR-Server ein durchschnittliches monatliches TK-Aufkommen und dessen Zusammensetzung bestimmen. Diese Daten wären dann die Grundlage für die Versuche des LCR-Servers Mindestumsätze, Rabatte usw. möglichst gut auszunutzen. Weiterhin kann der LCR-Server die durchschnittlichen Verbindungsdauer für Anrufe, abhängig von der gewählten Zielnummer, dem einzelnen Benutzer und eventuell weiteren Parametern, bestimmen. Sie könnte dann in die Ermittlung des Providers einfließen. Idealerweise sollte die TK-Anlagensoftware automatisch ein Profil vorangegangener Anrufe erstellen und der LCR-Komponente anbieten.
Denkbar sind natürlich auch Kombinationen aus beiden Ansätzen. Beispielsweise könnte der Benutzer Angaben zur Art und zum Inhalt des Gesprächs (geschäftlich / privat, kurze Frage, Besprechung, etc.) machen und der LCR-Server mit entsprechenden Heuristiken die Gesprächsdauer zuordnen.
In jedem Fall jedoch werden die LCR-Module, welche den Provider ermitteln, deutlich komplexer. Je mehr Informationen in die Berechnung einbezogen werden, desto mehr Speicherplatz für Tabellen und Rechenleistung für *on-the-fly*-Berechnungen wird benötigt. Da auch die Entwicklung solcher Module und die Bereitstellung der Informationen für diese Module einigen Aufwand verursacht, sollte zunächst untersucht werden, in welchem Umfang durch diese Erweiterungen zusätzlich Kosten eingespart werden können und ob dieser in einem sinnvollen Verhältnis zum Aufwand steht.

6.2 Vervollständigung der JTAPI-Implementation

Während unserer Arbeiten Ende 1997 war keine JTAPI-Implementation verfügbar, die auf dem ICC aufsetzen konnte. Aus dieser Not heraus fiel der Beschluß, die für LCR unbedingt notwendige Funktionalität selbst auf TSAPI aufbauend zu implementieren. Allgemein wäre sicherlich eine vollständige JTAPI-Implementation für TSAPI interessant, die dann auch für beliebige andere Java-Applikationen eingesetzt werden könnte.
Unsere JTAPI-Implementation besteht aus den Interfaces *Address*, *Call*, *Connection*, *JtapiPeer*, *Provider* und *Terminal* aus dem *Core*-Modul. In diesen sind jedoch nicht alle Methoden vorhanden. Vor allem die Methoden, welche die Verwaltung der *Observer* übernehmen und die *Event*-, *Exception*- und *Capabilities-Interfaces* wurden vernachlässigt.
Im *Call-Center* Modul sind dann nur noch die Interfaces, die für Routing unerläßlich sind, implementiert. Dies sind im einzelnen: *CallCenterProvider*, *RouteAddress*, *RouteCallback* und *RouteSession*. Dazu existieren noch Implementationen für die *Event*-Interfaces *RouteCallbackEndedEvent*, *RouteEndEvent*, *RouteEvent*, *RouteSessionEvent* und *RouteUsedEvent*. Diese Implementationen sind jedoch größtenteils unvollständig.
TSAPI stellt eine C/C++ Programmierschnittstelle zur Verfügung. Für die Umsetzung zwischen den Programmiersprachen Java und C++ nutzt die Implementation CORBA. Dies wurde jedoch hauptsächlich zu Lehrzwecken eingesetzt und ist für eine einfache JTAPI-Implementation unnötig. Sinnvoller wäre der Einsatz des Java Native Interface (JNI), welches einen einfacheren Weg darstellt, um C/C++ Funktionen von Java aus aufzurufen.
Aufgrund diverser Einschränkungen des eingesetzten ORBs im Zusammenhang mit dem Einsatz von mehreren Threads, sind wir vorsichtshalber von der JTAPI-Philosophie, Zustandsänderungen mittels Ereignissen zu melden, etwas abgewichen. Nach außen schickt die Implementierung entsprechend der JTAPI-Spezifikation Routing-Ereignisse an die registrierten Applikationen. Intern jedoch wird ein Polling-Verfahren benutzt, um Zustandsänderungen vom in C++ an den in Java realisierten Teil zu

übermitteln. Hier wäre eine einheitliche Realisierung, die Zustandsänderungen auch intern mittels Ereignissen signalisiert, vorzuziehen.
Die Dokumentation der Version 1.1 der JTAPI-Spezifikation war noch unvollständig. Deshalb mußten wir an manchen Stellen die Implementation nach eigenen Annahmen realisieren. Gegen Ende unserer Arbeiten erschien die neue Version 1.2 der JTAPI-Spezifikation. In diesem Zusammenhang sind wahrscheinlich einige Anpassungen vorzunehmen, um die Implementation auf den neuesten Stand zu bringen.
Die vorliegende JTAPI-Implementation realisiert also nur einen minimalen Anteil der vollen Funktionalität, die für eine komplette Realisierung benötigt wird. Die vorhandenen Basisansätze müssen weiter ausgebaut und insbesondere die Methoden für `Call`-, `Address`-, `Terminal`- und `Connection`-Objekte vervollständigt werden. Die Provider-Schnittstelle ist weitgehend realisiert, jedoch zwang uns der Zeitdruck und ein Implementationsproblem mit OmniBroker dazu, die Providerfunktionalität nicht tatsächlich über CORBA zu erbringen. Statt dessen wurde auf Java-Seite ein Providerstub programmiert, der den echten JTAPI-Provider emuliert und Adressen der TK-Anlage fest codiert verwaltet. Der Aufwand zur Korrektur ist jedoch minimal und bedarf nur eines sorgfältigen Debuggens der Parameterschnittstelle auf CORBA-Seite.
Die bereitgestellte CORBA-JTAPI-Routing-Schnittstelle arbeitet korrekt mit CORBA zusammen und erbringt die spezifizierte Funktionalität über das Netz. Aber auch hier ist noch Ausbaubedarf vorhanden. Da die Provider-Schnittstelle nicht korrekt funktionierte, konnten keine Callback-Methoden über `Provider`- und die `RouteAddress`-Objekte auf C++-Seite eingehängt werden. Die Aufrufbeziehung ist somit invertiert: Nicht die C++-Seite ruft die Callbacks auf, sondern der Router pollt die C++-Implementation über die CORBA-Schnittstelle auf vorliegende Ereignisse. Die korrekte Aufrufreihenfolge ist jedoch bereits beim Entwurf des Provider-Objekts geplant worden, so daß nach Behebung der oben genannten CORBA-Probleme auch eine leichte Integration der spezifizierten Routing-Schnittstelle integriert werden kann.

6.3 Bereitstellung der vollen Routing-Unterstützung durch das ICC

„Echtes“ LCR, d.h. LCR für ausgehende Anrufe über das öffentliche Telefonnetz, scheiterte an der anlageninternen Software des ICC. Die Routing-Funktionalität ist intern mittels *Automatic Call Distribution* (ACD) realisiert. Dies bedeutet, daß Routing-Ereignisse nur generiert werden, wenn bestimmte (interne) vorkonfigurierte ACD-Nummern angewählt werden. Dies ist aber insbesondere für ausgehende Anrufe nicht möglich. Die anlageninterne Software muß also dahingehend erweitert werden, daß für jeden Anruf, der über eine bestimmte TLG nach außen geht, Routing-Ereignisse generiert werden können. Inwieweit Bosch Telecom eine Erweiterung der anlageninternen Software der TK-Anlage plant, ist uns zur Zeit nicht bekannt.
Die Schnittstellen zu JTAPI und zur Routing-Software sind jedoch so ausgelegt, daß voraussichtlich ohne große Änderungen auch externe Verbindungen problemlos geroutet werden können. Die Struktur der Routing-Ereignisbearbeitung wird schließlich von der Generierung der Events nicht betroffen, da der Aufbau der über CSTA erhaltenen Ereignisse für interne und externe Anfragen nahezu identisch ist.
Außerdem gestattet die Integration der fehlenden Software in die Steuerungssoftware des Voice-Switch dann auch ein Routing von Gesprächen, die über ein Endgerät gewählt wurden. Bislang wird die Zielnummer ja nur über die Indirektionsstufe innerhalb der Software ermittelt.
Mit der uns zur Verfügung stehenden Version des ICC war deshalb nur eine Simulation des LCR mittels mehrerer Endgeräte möglich. Der Testaufbau bestand aus drei Telefonen, wobei von einem der Anruf gestartet wurde. Die beiden anderen Telefone repräsentierten virtuell dasselbe Endgerät, das jedoch über unterschiedliche Provider erreicht werden kann.

6.4 Unterscheidung zwischen Besetzt- und Gassenbesetztzeichen

Wegen der Existenz mehrerer Provider ist ein weiteres Besetztzeichen, das Gassenbesetztzeichen, notwendig. Es soll für den Benutzer zu unterscheiden sein, ob der angewählte Anschluß prinzipiell nicht erreichbar ist, oder ob nur die Kapazitäten des benutzten Providers erschöpft sind und der Anschluß über einen anderen Provider erreicht werden könnte. Das Gassenbesetztzeichen signalisiert nun genau den Fall, daß der Verbindungsaufbau am Provider gescheitert ist.

Um ein korrektes Rerouting und Fallback durchzuführen, muß auch der Router zwischen besetzt und gassenbesetzt unterscheiden können. Denn ein Rerouting oder Fallback ist natürlich nur bei einem Gassenbesetztzeichen sinnvoll.
Der LCR-Server führt bei jedem Besetztzeichen ein Rerouting oder Fallback durch, da unklar war, ob und wie ein Gassenbesetztzeichen von der TK-Anlage signalisiert wird. Es muß also zunächst sichergestellt werden, daß die TK-Anlage dieses erkennt, und geklärt werden, in welcher Form dies an der TSAPI-Schnittstelle angezeigt wird. Dann ist die JTAPI-Implementation so anzupassen, daß ReRoute-Ereignisse nur erzeugt werden, wenn wirklich ein Gassenbesetztzeichen vorliegt.

6.5 Skalierbarkeit durch mehrere LCR-Server

Für den Benutzer sind Reaktionszeiten des LCR-Servers von mehreren Sekunden untragbar. Da keine Belastungstests durchgeführt wurden, läßt sich kaum abschätzen, welches TK-Aufkommen der LCR-Server bewältigen kann. Geht man jedoch davon aus, daß ein komplexes Routing-Modul mit voller Funktionalität – wie in Abschnitt 6.1 beschrieben – realisiert ist, so wird ein LCR-Server sicherlich TK-Aufkommen bestimmter Größe nicht mehr bewältigen können.
Neben einer Aufrüstung des Rechners bietet sich dann die Möglichkeit, ein verteiltes LCR-System zu realisieren. Dazu müssen die Routing-Ereignisse auf mehrere LCR-Server auf verschiedenen Rechnern verteilt werden. Zu beachten ist dabei, daß der LCR-Server die zugehörige Routing-Session kennen muß. Es ist also eher eine Verteilung der Sessions anstatt der einzelnen Ereignisse sinnvoll. Dies bedeutet, daß die Routing-Ereignisse der TK-Anlage den verschiedenen Sessions zugeordnet und dann an die entsprechenden Rechner geschickt werden müssen.

6.6 Wartung von Routing-Servern

Die Wartung der Routing-Tabellen und Algorithmen schließlich stellt eine große Herausforderung bezüglich der Aktualisierung und Optimierung dar. Insbesondere sollte eine Anbindung an eine Datenbank zur effizienten Generierung ortsabhängiger Tabellen angedacht werden. Darüber hinaus stellt sich die Fragen nach einem geeigneten Kostenmodell, das Grundgebühren, Zeittakte, Kontingente und Rabatte berücksichtigen kann. Schließlich sollten auch neue Medien wie Voice over IP in diese Modelle mit eingehen.

7 Literatur

[1] Hüskes, R.; Zivadinovic, D.: *Wahlversprechen.* c't-Magazin 7/98, Heise-Verlag, Hannover, S.102 - 107

[2] Endres, J.; Zivadinovic, D.: *Wahlhelfer.* c't-Magazin 7/98, Heise-Verlag, Hannover, S.108 - 115

[3] Hüskes, R: *Richtig wählen*. c't-Magazin 3/98, Heise-Verlag, Hannover, S.62 - 67

[4] European Comupter Manufacturers Association: *Services for Computer Supported Telecommunications Applications (CSTA).* http://www.ecma.ch/ecma-st/e179-pdf.pdf

[5] European Comupter Manufacturers Association: *Protocol for Computer Supported Telecommunications Applications (CSTA.* http://www.ecma.ch/ecma-st/e180-pdf.pdf

[6] Stallings, W.: *SNMP, SNMPv2 and CMIP: The Practical Guide to Network Management Standards.* Addison-Wesley Pub., Massachusetts, 1993

[7] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns*. Addison-Wesley, Reading, MA, 1995, S.151 – 161.

[8] Fleischhauer, C.: *Das große Buch zu Visual C++ 5*, Data Becker, 1997

Informationen im WWW

[W1] Sun Microsystems Corp.: *Java Documentation.* http://www.javasoft.com/docs/index.html

[W2] Sun Microsystems Corp.: *Java Technology in the real World 1997-98.* http://www.javasoft.com/nav/used/index.html

[W3] Sun Microsystems Corp.: *Project Java Hotspot* .
http://www.javasoft.com/javaone/javaone98/keynotes/gosling/hotspot.htm
[W4] Sun Microsystems Corp.: *Secure Computing with Java: Now and the Future.*
http://www.javasoft.com/marketing/collateral/security.html
[W5] Sun Microsystems Corp.:*EmbeddedJava.*
http://java.sun.com/products/embeddedjava/index.html
[W6] Sun Microsystems Corp.: *Java™ Telephony API Home Page.*
http://www.javasoft.com/products/jtapi/index.html
[W7] Sun Microsystems Corp.: *SunXTLTeleservices - Documentation.*
http://www.sun.com/products-n-solutions/sw/SunXTL/SunXTL_docs.html
[W8] Novell Corp.: *NetWare® Telephony Services™ Release 2. Telephony Services Application Programming Interface (TSAPI).* http://webftp.novell.de/pub/updates/napi/ntsdk201/tsapi2.exe
[W9] Microsoft Corp.: *The Microsoft Windows Telephony Platform: Using TAPI 2.0 and Windows to Create the Next Generation of Computer-Telephony Integration.*
http://premium.microsoft.com/msdn/library/conf/html/s9e68.htm
[W10] IBM Corp.: *IBM Callpath and DirectTalk Page.* http://www.networking.ibm.com/callpath/
[W11] Sun Microsystems Corp.: *The Java Telephony API - An Overview.*
http://www.javasoft.com/products/jtapi/jtapi-1.2/Overview.html
[W12] Fünfrocken, S.; Meister, G.: *Praktikum "Telekommunikationsdienste und verteilte Anwendungen mit Java"*. WS 1997/98, TU Darmstadt, Fachgebiet Verteilte Systeme,
http://www.informatik.tu-darmstadt.de/VS/Lehre/WS97-98/JavaTK/Aufgaben.html
[W13] Meister, G.: *Praktikum "Telekommunikationsdienste und verteilte Anwendungen mit Java", Aufgabe 2.* WS 1997/98, TU Darmstadt, Fachgebiet Verteilte Systeme,
http://www.informatik.tu-darmstadt.de/VS/Lehre/WS97-98/JavaTK/A2/aufgabe2.html
[W14] Meister, G.: *Praktikum "Telekommunikationsdienste und verteilte Anwendungen mit Java", Aufgabe 3.* WS 1997/98, TU Darmstadt, Fachgebiet Verteilte Systeme,
http://www.informatik.tu-darmstadt.de/VS/Lehre/WS97-98/JavaTK/A3/aufgabe3.html
[W15] Object Management Group: *The OMG Home Page.* http://www.omg.org/
[W16] International Telecommunication Union: *Specification of Basic Encoding Rules for Abstract Syntax Notation One (ISO 8825, CCITT X.209).*
http://www.itu.int/itudoc/itu-t/rec/x/x200-499/x209_24177.html
[W17] International Telecommunication Union: *Specification of Abstract Syntax Notation One (ISO 8824, CCITT X.208).* http://www.itu.int/itudoc/itu-t/rec/x/x200-499/x208_22887.html
[W18] Sun Microsystems Corp.: *The Java Development Kit*.
http://www.javasoft.com/products/jdk/index.html
[W19] Object Oriented Concepts Inc.: *ORBacus Home Page.* http://www.ooc.com/ob/

Inhaltsverzeichnis

Abbildungen