Fachhochschule Rosenheim

Fachbereich Informatik
Diplomarbeit

Diplomprüfung WS 97/98

Johannes Peter Antonius Weigend

Thema:

ROSALINDE

Rosenheim application link in a
distributed environment

Erstprüfer : Prof.Dr. J. Siedersleben
Zweitprüfer: Prof.Dr. H. Oechslein

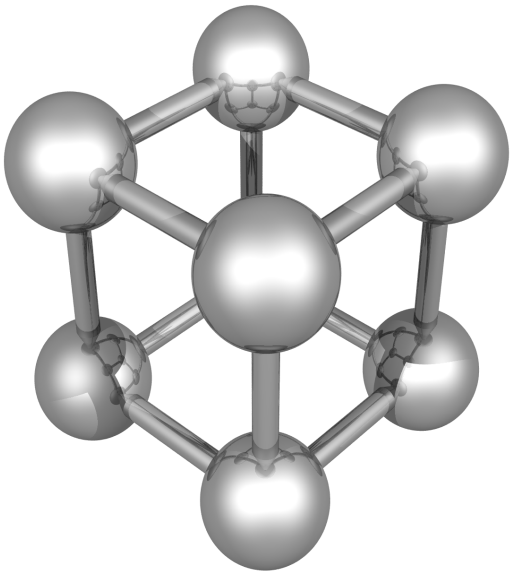
ERKLÄRUNG

Ich versichere, daß ich diese Arbeit selbstständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim, im Januar 1998

Johannes Peter Antonius Weigend

ROSALINDE : ROSENHEIM
APPLICATION LINK IN A DISTRIBUTED
ENVIRONMENT



ABSTRACT

Im ROSALINDE-Projekt wurde eine Beispielanwendung zur Demonstration einer objektorientierten, dreistufigen und beliebig verteilten Client/Server-Architektur auf Basis des ONC-RPCs (SUN-RPC) entwickelt. Diese wurde implementiert unter UNIX (Linux) in der Sprache C und einer grafischen Benutzeroberfläche unter X11 mit der Interpretersprache TCL/TK. Über diese Beispielanwendung hinaus können die verschiedenen Komponenten des ROSALINDE-Projektes als Framework für objektorientierte, verteilte Anwendungen verwendet werden.

Die ROSALINDE-Anwendung bietet „weiche“ Objektorientierung auf der Basis abstrakter Datentypen (ADTs) und realisiert eine komplette Trennung der Netzwerkzugriffsschichten vom Anwendungskern, um die Anwendungsentwicklung vollständig von dieser Ebene abzuschirmen. Es wurden einige der grundlegenden Komponenten und Konzepte von CORBA 2.0, wie Objektbroker und Proxys implementiert, ohne aber den Anspruch zu erheben damit CORBA-konform zu sein. Vielmehr werden grundlegende Eigenschaften und Probleme von dezentralen, verteilten Systemen durchleuchtet. Diese Diplomarbeit erklärt die Komponenten der ROSALINDE-Anwendung, deren Zusammenspiel und vergleicht diese mit kommerziellen Middlewareprodukten. Außerdem wird gezeigt, wie sich diese Konzepte auch auf anderen Plattformen und mit alternativen Middlewarekomponenten einsetzen lassen.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

Nummer Seite

DANKSAGUNGEN/WIDMUNGEN

Ich möchte mich ganz herzlich bei Prof. Dr. Johannes Siedersleben für die fachliche Unterstützung während des gesamten Studiums bedanken. Ebenso bei Dr. Ralf Klotzbücher, der mich wesentlich bei der Entscheidung, Informatik zu studieren bestärkt sowie meinen Einstieg und die Beschäftigung mit objektorientierten Programmiertechniken beeinflußt hat. Meinen lieben Eltern Dr.-Ing. Manfred Weigend und Friederike Weigend, die mir dieses Studium ermöglicht haben und mir in dieser Zeit viel Bestätigung und Rückhalt gaben.

ABKÜRZUNGEN

ACID	atomic consistent independent durable
ADT	abstract datatype
API	application programming interface
AWK	Anwendungskern
CORBA	common object requestbroker architecture
CID	client identifier
DB	database
DCE	distributed computing environment
DCOM	distributed component object model
FIFO	first in – first out (queue)
GUI	graphical user interface
IDL	interface definition language
IPC	interprocess communication
LAN	local area network
ODBC	open database connectivity
OID	object identifier
OLE	object linking and embedding
OLTP	online transaction processing
OTAB	object table
RID	row identifier
ROSALINDE	Rosenheim application link in a distributed environment

RPC	remote procedure call
SID	server identifier
WAN	wide area network

Broker - ein Netzwerkdienst der Clients den „richtigen“ Server für ihre Anfragen vermittelt und deren Netzwerkinformation zentral verwaltet.

Server - ein oder mehrere Prozesse, die ein oder mehrere RPC-Schnittstellen bereitstellen (Software).

Client - ein Nutzer einer RPC-Schnittstelle, die von einem Server bereitgestellt wird (Software).

Fat-Client - wird ein Client in einer Client/Server-Architektur bezeichnet, der neben der Präsentationslogik (GUI) auch die Anwendungslogik beinhaltet und abarbeitet.

Thin-Client - wird ein Client in einer Client/Server-Architektur bezeichnet, der nur die Präsentationslogik (GUI) beinhaltet und keinerlei Anwendungslogik besitzt.

Host - ein Rechner auf dem ein oder mehrere Serverprozesse laufen (Hardware).

Workstation - ein Rechner auf dem ein oder mehrere Clientprozesse laufen (Hardware).

Object-ID (OID) - eine systemweit eindeutige Objektkennung.

Proxy - ein Stellvertreterobjekt welches zumeist nur einen Objektidentifizierer OID enthält mit dem das Anwendungsobjekt am Server angesprochen wird.

Client-ID (CID) - eine systemweite Kennung mit der sich ein Client eindeutig am Server identifiziert.

Client-Tabelle (CTAB) - kapselt am Server den Zugriff auf die Objekte aller Clients und stellt die Zuordnung zur Objekttabelle des Client her.

Objekt-Tabelle (OTAB) - stellt die Verbindung zwischen dem Proxy und dem Anwendungsobjekt her und sichert die Objektidentität der Anwendungsobjekte.

Remote Procedure Call (RPC) - ein entfernter Funktionsaufruf.

Middleware - der Teil der Systemsoftware der eine Kommunikation zwischen Client und Server ermöglicht.

Locator - siehe Broker.

Loopback Device - ein Betriebssystemmechnismus der alle Netzzugriffe transparent auf den eigenen Rechner umleitet.

ONC - das Open Network Computing von Sun Microsystems. Die ONC Produktpalette beinhaltet RPC, XDR, NIS, NLM, rex etc.

Open Database Connectivity (ODBC) - ist eine von Microsoft entworfene Datenbankschnittstelle, die einen Datenbankzugriff über die Sprache C ermöglicht.

Rosenheim application link in a distributed environment

ROSALINDE

Einführung in verteilte Anwendungen

Rosenheim application link in a distributed environment

ROSALINDE

Einführung in verteile Anwendungen

1 Verteilte Anwendungen / Komponentensoftware

1.1 Heute

Die heutzutage bei 90% aller Client/Server-Systeme anzutreffende zweistufige Architektur besteht aus einer Datenbank, auf die mittels ODBC oder sonstiger Middlewaretechnologie (z.B. OCI– Oracle Call Interface) zugegriffen wird und einem Fat-Client. Sie hat neben den hinreichend bekannten Vorteilen den entscheidenden Nachteil, daß die komplette Anwendungslogik auf den Client transferiert wird. Dies entlastet zwar den Host, die Anforderungen an die Clients aber wachsen immens. Die dezentrale Administration solcher Systeme verschlingt schon jetzt Unsummen. Einerseits will man wieder weg vom monolithischen Fat-Client Architekturen, andererseits können die Anforderungen heutiger DV nicht mit Terminals und ebenfalls monolithischen Großrechneranwendungen gelöst werden. Neben der unflexiblen Struktur von traditionellen Anwendungen sind die Performanceanforderungen heutiger und kommender Software immens und können auf dem herkömlichem Weg nur noch unbefriedigend gelöst werden.

1.2 Komponenten

Die Entwicklung geht daher zu dezentralen Systemen, die im Idealfall sogar aus wiederverwendbaren Komponenten bestehen (Komponentensoftware). Diese Komponenten können Skalierbarkeit im Hinblick auf Lastverteilung und Funktionalität bieten. Sie sind ökonomischer, da sie von mehreren Anwendungen gleichzeitig genutzt werden können; sie sind flexibler, da Einzelkomponenten austauschbar sind, und sie können zumeist das der Anwendung zugrundeliegende Problem besser realisieren, weil die meisten Probleme der „realen“ Welt von ihrem Wesen her parallel sind. Genaugenommen bringt erst eine Komponententechnik die Vorteile, welche man sich durch alleinigen Einsatz der Objektorientierung versprach.

Diese Vorteile haben aber ihren Preis: Verteilte Anwendungen haben mehr Kommunikations-aufwand, sind schwerer zu implementieren und zu testen. Das liegt daran, daß ein nicht wesentlicher Teil einer Client/Server-Applikation Kommunikationsaufgaben übernimmt, die bei lokalen Anwendungen normalerweise wegfallen. Die Probleme werden auch deutlich an den vielen gescheiterten Client/Server-Projekten in den letzten Jahren.

Leider steigt der Kommunikationsaufwand mit dem Verteilungsgrad. Da im Extremfall alle Komponenten miteinander kommunizieren müssen, führt $\binom{n}{2}$ dies bei n-Komponenten zu Kommunikationskanälen (alle zweier-Kombinationen einer n-elementigen Menge ohne Beachtung der Reihenfolge).

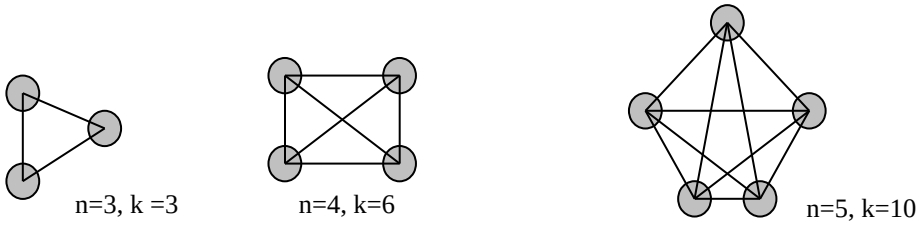


Abbildung 1 Vollständiger Graph

Durch die enorme Anzahl von Kommunikationskanälen müssen Mechanismen gefunden werden, wie verteile Systeme einfach und trotzdem effizient miteinander kommunizieren sollen. Ein Beispiel für einen solchen Objektverteilungsmechanismus ist der CORBA Standard der Object Management Group (OMG). Die OMG ist eine Non-Profit Organisation mit Sitz in den USA, die sich einerseits aus Firmenvertretern und andererseits aus einem Forschungsgremium zusammensetzt und die versucht, eine einheitliche Architektur, die OMA (Object Management Architecture) zu etablieren. Ein wesentlicher Teil dieser visionären Architektur ist der CORBA Standard (common object request broker architecture), der transparente Objektverteilung in LANs und WANs ermöglicht. Produkte die diesen Standard implementieren, sind z.B. Orbix von IONA, DSOM von IBM oder Objekt Broker von Digital Equipment. Auch Microsoft bietet mit DCOM, einer Erweiterung des COM-Objektmodells auf dem alle OLE bzw. ActiveX - Komponenten aufbauen, eine Technik an Objekte[1] über LAN bzw. WAN zu verteilen. Die Firma SUN-Microsystems bietet mit Javabeans ebenfalls ein Komponentenmodell für die Programmiersprache Java an.

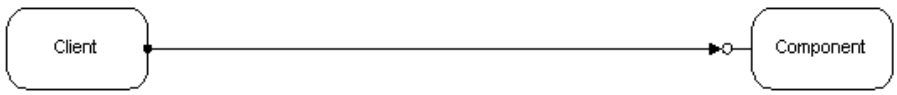


Abbildung 2 Softwarekomponente

1. Microsoft verwendet den Begriff Objekt abweichend von der üblichen Terminologie, so unterstützen DCOM Objekte keine Implementierungsvererbung und keine Polymorphie, bieten aber Schnittstellenaggregation und Delegation.

1.3 Verständnis

Nicht nur der Aufwand der zusätzlichen Kommunikation ist ein Projektrisiko, auch die zusätzliche Komplexität, verursacht durch die Middleware, kann zum Scheitern eines Projektes beitragen. Ein sinnvolles Vorgehen ist, daß einige Spezialisten sauber gekapselte Kommunikationsschichten entwickeln, während die Anwendungsentwicklung wie gewohnt „lokal“ arbeitet. Da aber nie eine vollständige Kapselung erreicht werden kann, sollte jeder Beteilige in einem solchen Projekt über solide Grundlagen der Client/Server-Kommunikation verfügen. Das nötige Rüstzeug ist ein Grundlagenverständnis, welches in Form eines einfachen Durchstichs die Erklärung von Komponenten und ihrer Kommunikation ermöglicht. Die Beispielanwendungen, die mit den jeweiligen Middleware Produkten (Orbix (CORBA 2.0), MS-RPC (DCE)...) mitgeliefert werden, gehen oft nur unzureichend auf ein sinnvolles Anwendungsdesign ein. Auch wird bei diesen Produkten der Benutzer über technische Details häufig im unklaren gelassen. Eine Lösung technischer Probleme ist aber grundsätzlich nur mit ausreichendem Verständnis der darunterliegenden Technik möglich.

1.4 ROSALINDE

Daher war es Ziel des ROSALINDE-Projektes, ein möglichst realitätsnahes und doch einfaches Programm zu entwickeln, das die meisten Probleme, die in verteilten Anwendungen, bzw. mehrstufigen Client/Server-Systemen auftreten können, durchleuchtet. Diese „Durchstichanwendung“ kann mit relativ wenig Aufwand auf andere Plattformen portiert werden und so als erste Grundlage zur Projektentwicklung dienen. Es wurden bewußt Einschränkungen in der Architektur vorgenommen, so wurde auf aktuelle Technik wie C++/DCE/CORBA verzichtet und auf der Basis der in UNIX vorhandenen Werkzeuge sowie der

(Teile wurden allerdings unter Solaris 2.4 / 2.5 entwickelt und mit dem Speicher-Debug-Werkzeug Purify getestet.)

1. Teile wurden allerdings unter Solaris 2.4 und 2.5 entwickelt und mit dem Speicherdebug-Werkzeug „purify“ getestet.

Programmiersprache C programmiert. Die Middleware bildet der SUN-RPC, der von Sunsoft auch ONC-RPC genannt wird. Der ONC-RPC, der unter anderem auch die Grundlage für das Netzwerkdateisystem NFS bildet, gehört zu einer Gruppe von Netzwerkkomponenten, die Sunsoft als Open-Network-Connectivity (ONC) bezeichnet hat. Der ONC-RPC ist aber inzwischen über 10 Jahre alt und entspricht in vielen Merkmalen nicht dem aktuellen Stand der Technik. Da die komplette Entwicklung unter dem Betriebssystem „Linux“ stattfand, fielen keine Kosten für Compiler, Middleware etc. an.[1] Das ROSALINDE-Projekt hat sich als Verständnisgrundlage für „verteilte Anwendungsentwicklung“ bewährt. Fast alle Probleme und Konzepte treten in gleicher oder ähnlicher Form bei heutigen Architekturen wie CORBA, Java Beans, DCOM etc. auf. Das liegt daran, daß die Konzepte verteilter Anwendungen im Grundsatz meist gleich sind, egal welche Werkzeuge und Techniken man verwendet. Verursacht durch den SUN-RPC mußten viele Mechanismen implementiert werden, die bei anderen RPC- oder Middlewareprodukten bereits vorhanden sind. Dieser „Fußgängeransatz“ hat aber den Vorteil, daß eine eigene Implementierung von Konzepten wie Broker oder Proxys zum Gesamtverständnis beiträgt. Überdies ist der SUN-RPC im Quellcode frei verfügbar und somit für eigene Entwicklungen portier- und anpaßbar.

2 Technologischer Überblick

2.1 Client/Server-Technik

2.1.1 Begriffserklärungen

Als Client wird ein Nutzer einer Schnittstelle bezeichnet. Dieser Nutzer kann entweder nur ein Programmodul sein oder aber ein komplettes Programmsystem von dem diese Schnittstelle benutzt wird. Die Bezeichnung „Client“ bezieht sich direkt auf die Software und nicht auf die Hardware. Ganz analog implementiert ein Server eine solche Schnittstelle. Die von einem Server bereitgestellte Funktionalität wird auch als Komponente bezeichnet. Client und Server können in einem Programm zu einer Einheit gebunden werden, aber auch, verbunden durch Middleware, sich in verschiedenen Prozessen oder Rechnern befinden. Als Bezeichnung für die Hardwareseite werden im folgenden die Bezeichnungen „Host“ und „Workstation“ benutzt.

2.2 Message Passing Mechanismen

Bei Client/Server-Systemen stellt sich als ein Hauptproblem der Datenaustausch zwischen den Komponenten dar. Daher ist es sinnvoll, die zur Verfügung stehenden Datenaustauschmechanismen zu klassifizieren. Man unterscheidet zwischen synchronen und asynchronen Mechanismen. Ein Client wartet bei einer synchronen Kommunikation solange, bis der Server das Ergebnis zurückgibt. Im Gegensatz dazu steht das Mailbox-Prinzip, bei dem Nachrichten in den Postkasten des Empfängers gelegt werden und asynchron bearbeitet werden können. Bei asynchronen Verfahren verringert sich die Deadlockwahrscheinlichkeit; allerdings wird der Datenaustausch komplizierter, da eigene Synchronisationsmechanismen gebaut werden müssen.

2.2.1 Pipes

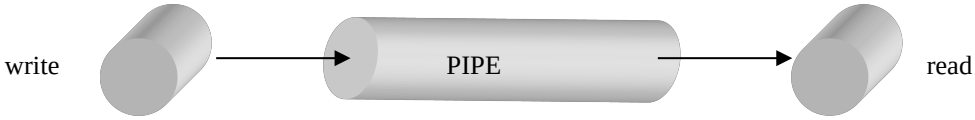


Abbildung 3 Pipe

Interprozeßkommunikation (IPC) innerhalb eines Rechners ist mit sogenannten Pipes möglich. Eine Pipe (Rohr) kann als Verbindung zwischen zwei Prozessen dienen, die darauf lesend bzw. schreibend, ähnlich wie auf eine Datei, zugreifen können. Traditionell sind Pipes immer unidirektional. Seit Unix SVR4 kann auf eine Pipe jedoch gelesen und geschrieben werden. Eine Pipe ist ein FIFO – Puffer aus dem zerstörend gelesen wird. Das Schreiben, wie auch das Lesen findet bei einer Pipe teilweise asynchron statt, da diese begrenzt Daten puffern kann. Falls keine Daten zur Verfügung stehen wird der Leser blockiert. Erst wenn der Schreiber die Pipe schließt (close) empfängt der Leser ein End-Of-File (EOF).

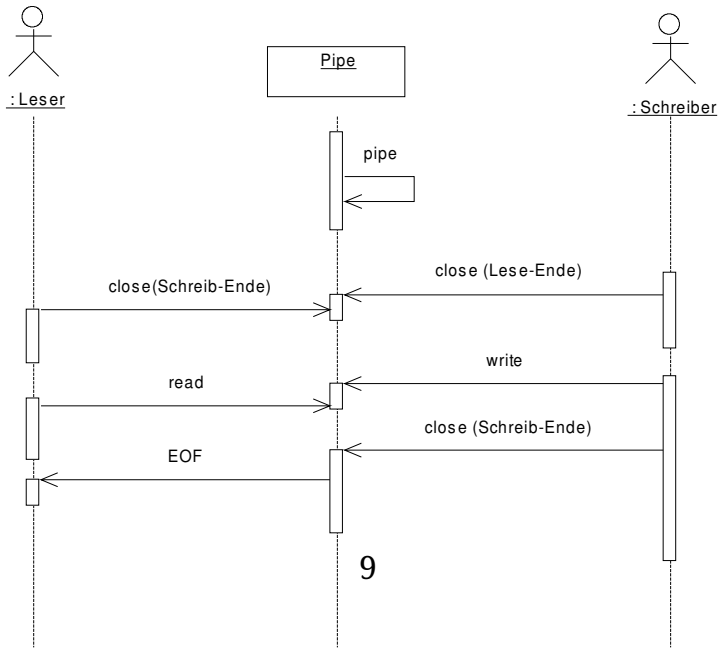


Abbildung 4 Kommunikation über eine Pipe

Dieser ursprünglich in UNIX entwickelte Betriebssystemmechanismus ist heutzutage in allen gängigen Betriebssystemen vorhanden und wird auch verwendet. Windows NT 4.0 verwendet beispielsweise Pipes transparent für Netzwerkzugriffe, die sich auf den eigenen Rechner beziehen.

2.2.2 BSD-Sockets

Sollen Rechnergrenzen überschritten werden, bieten sich BSD Sockets an, die eine IPC im Netzwerk transparent ermöglichen.[1] Wenn man will, kann man Sockets als erweitere Sonderformen von Pipes ansehen, die zusätzliche Mechnismen beinhalten, um netzwerkfähig zu sein. Mit UNIX SVR4 wurde eine neue Schnittstelle mit dem Namen TLI (Transport Layer Interface) entwickelt die in mancher Beziehung moderner ist, sich aber auf breiter Front (noch) nicht durchsetzen konnte [2]. Sockets sind die minimale Ausgangsbasis für jegliche Client/Server-Kommunikation. Die meisten weiterentwickelten Middleware Mechanismen bauen auf der Socket – Schnittstelle auf.

1. Die ursprüngliche Schnittstelle stammt aus dem Release BSD 4.2 des

2.3 Remote Procedure Calls (RPC)

Der Remote Procedure Call ist ein synchroner Kommunikationsmechnismus, welcher Netzwerkzugriffe als „normale“ Funktionsaufrufe kapselt. Somit wird es möglich, transparent Funktionen aufzurufen, die ein entfernter Rechner implementiert. Der Funktionsaufrufer ist immer Client eines RPC-Servers. Da Funktionsparameter auch Zeiger auf beliebige Datenstrukturen sein können und Zeiger nur jeweils innerhalb eines Adressraumes gültig sind, muß bei jeder RPC-Architektur ein Mechanismus vorhanden sein, der Zeiger kopieren kann. Die RPC-Architekturen können teilautomatisch Zeiger behandeln, dabei müssen in einer Protokolldatei die Datenstrukturen angegeben werden auf welche diese Zeiger zeigen. Die RPC-Implementierung muß auch eine Technik zur maschinenunabhängigen Übertragung von eingebauten Datentypen wie int, float, double usw., bereitstellen, da sich die interne Speicherform an verschiedenen Rechnern unterscheiden kann (Little vs. Big Endian). Datenstrukturen, werden rekursiv in einen durchgehenden Übertragungsdatenbereich geschrieben. Dieses „Flachklopfen“ von Übergabeparametern zum Zweck des Netzwerkversendens nennt man Marshalling. Die Marshallingroutinen werden wie schon erwähnt generiert, allerdings können nicht alle Fälle automatisch abgedeckt werden. Die verschiedenen RPC-Architekturen unterscheiden sich hierbei stark von ihrer Leistungsfähigkeit.

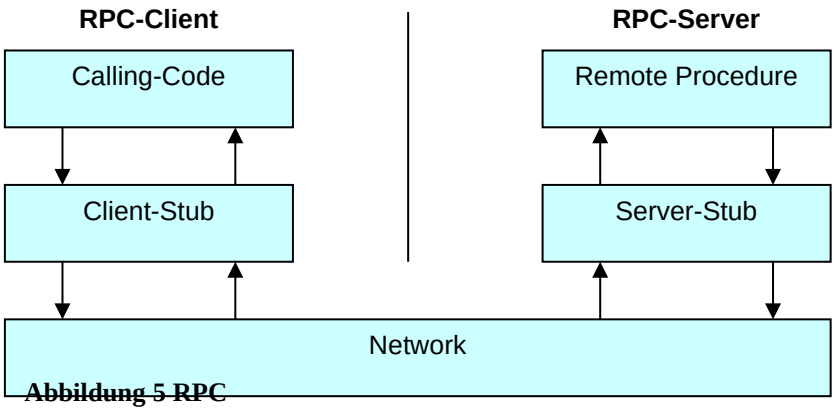


Abbildung 5 RPC

2.3.1 ONC-RPC (SUN-RPC)

Der ONC-RPC wurde von SUN-Soft mit UNIX System V Release 4 im Quellcode mitausgeliefert und verfügt über eine Schnittstellenbeschreibungssprache (RPCL), einen Generator (rpcgen) und außerdem über eine plattformunabhängige Marshallingbibliothek (External Data Representation (XDR)), mit der eine RPC-Kommunikation über unterschiedliche Plattformen gewährleistet ist. Der ONC-RPC hat in der ursprünglichen Version eine wesentliche Einschränkung: Es kann nur jeweils ein Parameter als Übergabe- und Rückgabewert angegeben werden. Neuere Implementierungen des SUN-RPCs haben dieses Problem nicht. Die ROSALINDE-Anwendung benutzt jedoch aus Kompatibilitätsgründen die traditionelle Implementierung des SUN-RPC, da diese auf allen gängigen UNIX-Plattformen verfügbar ist.

2.3.2 DCE-RPC

Der DCE-RPC ist nur ein Teil des „distributed computing environment“, welches durch die OSF (open software foundation) standartisiert wurde. Die DCE-Architektur sieht neben dem RPC, der als technische Grundlage verwendet wird, auch diverse „höhere“ Dienste vor. Diese sind z.B. ein zentraler Zeitdienst, ein Namensdienst zur dynamischen Auflösung von Resourcennamen. (siehe Kaptitel 4 - Der Broker). Wie aus Abbildung 6 ersichtlich setzt der DCE-RPC direkt auf die ebenfalls von der OSF standartisierten Posix-Threads auf.[1]

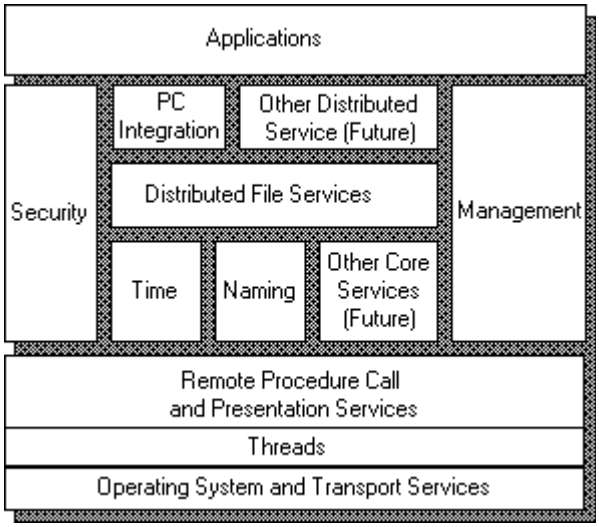


Abbildung 6 Distributed Computing Environment

1. Die POSIX-Schnittstelle war der Versuch eine einheitliche UNIX-API zu schaffen. Es existiert zwar inzwischen ein Minimalstandard, dieser ist jedoch für reale Anwendungen nicht ausreichend.

Die wohl bekannteste Implementierung des DCE-RPCs ist der Microsoft-RPC der für alle Microsoft Betriebssysteme implementiert wurde. Allerdings verwendet Microsoft properitäre Erweiterungen, die inkompatibel zu allen anderen Implementierungen sind.

2.4 Objektorientierte Middleware

2.4.1 CORBA

Sehr schnell wurde erkannt, daß RPC-Schnittstellen eine perfekte Datenkapselung bieten, da der Datenzugriff ausschließlich über Funktionsaufrufe erfolgt. Dieses Kapselungskonzept deckt sich mit den dem der objektorientierten Programmiersprachen. Es war daher naheliegend, den RPC-Mechanismus als rein prozedurale Aufrufschnittstelle um objektorientierte Konzepte zu erweitern. Damit wird es möglich, Objekte auf anderen Computern zu erstellen und deren Methoden dort aufzurufen. Der Zugriff auf diese Objekte erfolgt vollständig transparent, d.h. ein Client braucht nicht zu unterscheiden, ob sich ein Objekt auf dem eigenen oder einem möglicherweise sehr weit entfernten Rechner befindet. Die Schwierigkeit bei der Implementierung einer derartigen Transparenz liegt darin, sicherzustellen, daß ein ausgeführtes Objekt nahtlos mit einem in einem anderen Prozeß ausgeführten Objekt zusammenarbeitet. CORBA wie auch DCOM verfügen über eine objektorientierte Schnittstellenbeschreibungssprache IDL (Interface Definition Language), die es ermöglicht, sprach- und plattformunabhängig Objekte mit ihren Methoden zu definieren. Aus dieser Schnittstellenbeschreibung werden letztendlich Zugriffsschichten (Stubs) generiert, die den jeweils verwendeten RPC-Mechanismus fast vollständig abschirmen. Um eine Netzwerktransparenz zu erreichen, muß ein Client aber wissen wo sich der gesuchte Server befindet, der das Objekt implementiert. Diesen Vorgang heißt Bindung. Bei CORBA existiert ein Dienst der dies dynamisch zur Laufzeit ermöglicht. Dieser Dienst wird als „Broker“ bezeichnet. Daher auch der Name CORBA (Common Object Broker Architekture).

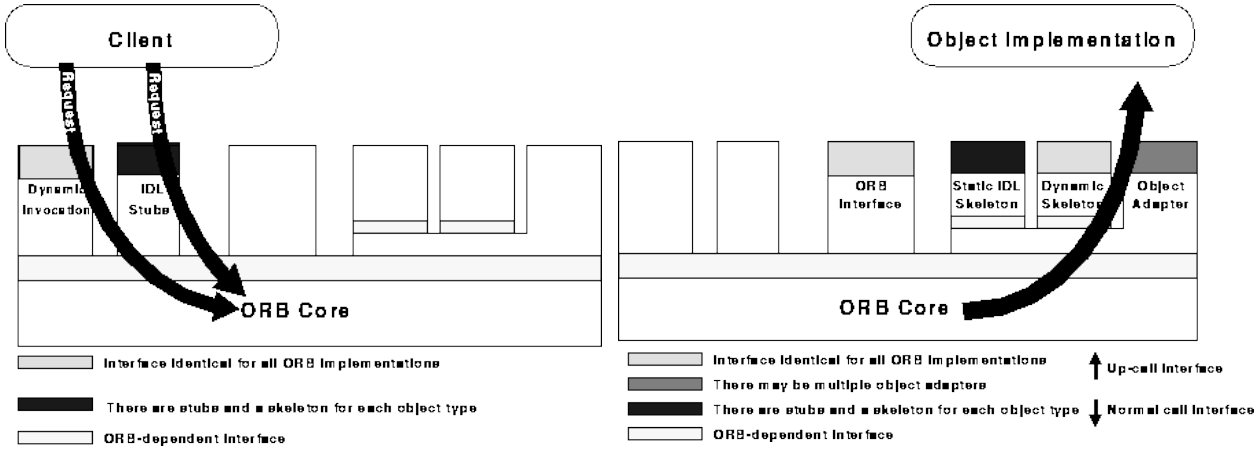


Abbildung 7 CORBA

2.4.2 DCOM

DCOM ist eine Erweiterung des Microsoft COM-Objektmodells, welches aus der Entwicklung von OLE-Verbunddokumenten entstand. COM beschreibt einen Binärstandart für „Objekte“ und ihre Methoden, so daß eine sprachunabhängige Nutzbarkeit von COM-Objekten erreicht wird.[1] Wie schon in der Einführung erwähnt, können COM-Objekte nicht mit CORBA-Objekten verglichen werden, da diese keine Implementierungsvererbung, dafür allerdings Schnittstellenaggregation und –delegation unterstützen. DCOM erweitert COM, setzt direkt auf den Microsoft-RPC auf und bietet einen Zugriff auf entfernte Objekte. Bei DCOM werden die lokalen Funktionsaufrufe durch RPC-Aufrufe ersetzt. Dies ist deshalb möglich, da DCOM einen Binärstandart für die Methodentabelle VMT (virtual method table) definiert. Daher müssen nur die Funktionszeiger innerhalb der VMT durch Zeiger auf RPC-Funktionen ersetzt werden. Für einen Client ist dies (fast) vollständig transparent. Es werden normalerweise Funktionsstummel (Stubs) für alle Methoden eines Objektes generiert. Dies erfolgt mit dem Generatorprogramm „midl“ und der dazugehörigen Schnittstellenbeschreibungssprache IDL (Interface Definition Language). CORBA wie auch DCOM verfügen aber zusätzlich über die Möglichkeit erst zur Laufzeit bekannte Schnittstellen aufzurufen (Dynamic Invocation). Dies ist beispielsweise in sogenannten Verbunddokumenten der Fall, in die Objekte beliebiger Anwendungen eingefügt werden können, welche bis dahin über unbekannte Schnittstellen verfügen.

2.4.3 ROSALINDE

Im ROSALINDE-Projekt wurde ebenfalls ein RPC-Aufsatz entwickelt, der einen transparenten Zugriff auf Objekte über Prozeß- bzw. Rechnergrenzen hinweg

ermöglicht. In den folgenden Kapiteln wird erklärt, was im ROSALINDE-Projekt als Objekt verstanden wird, welche grundlegenden Komponenten dazu nötig sind und welche Probleme dort auftreten.

1. Es gibt bei C++ leider keinen Binärstandart für die Objekte. Dies hat zur Folge, daß Objektcode welcher von verschiedenen Compilern erzeugt wurde inkompatibel ist [11]. Ebenso hängt die exakte Größe von Funktionszeigern direkt von der Plattform ab (Win32 – 32-Bit, Alpha 64-Bit). Damit wird eine sprachunabhängige Nutzbarkeit von Objekten verhindert.

3 ROSALINDE – Systemarchitektur

3.1 Schichtenarchitektur und Verteilung

3.1.1 Übersicht

Die ROSALINDE-Anwendung demonstriert eine mehrstufige Komponentenverteilung und löst Kommunikationsprobleme, die entstehen, wenn Anwendungen aus mehreren Teilkomponenten dezentral aufgebaut sind. Die Kommunikation zwischen den Komponenten erfolgt über entfernte Funktionsaufrufe die mit Hilfe des SUN-RPCs implementiert wurden. Damit können die an der ROSALINDE-Anwendung beteiligten Komponenten, verbunden durch das Internet, weltweit verteilt werden. Mittels eines Proxy-Mechanismus ist sichergestellt, daß der Client nur minimale Kenntnis der Anwendungsobjekte hat und braucht. Der Proxy dient als Stellvertreter des Serverobjektes am Client und delegiert jeden Methodenaufruf an den Server. Dies isoliert einerseits das GUI vom Anwendungskern, andererseits führt diese Strategie zu minimalen Resourcenbedarf am Client. Ein solcher Client wird als „Thin-Client“ bezeichnet.

3.1.2 Schichtenarchitektur

Durch eine konsequente Schichtenarchitektur ist es möglich, einzelne Komponenten innerhalb einer Anwendung auszutauschen. Dies führt zu besseren und zu besser wartbaren Softwaresystemen. Die Schichtenarchitektur ist ein entscheidender Schritt in der Evolution von großen Softwaresystemen. Die klassischen Großrechneranwendungen (Assembler) kannten so etwas nicht. Sie wurden als ein großes monolithisches System gebaut und sind heutzutage

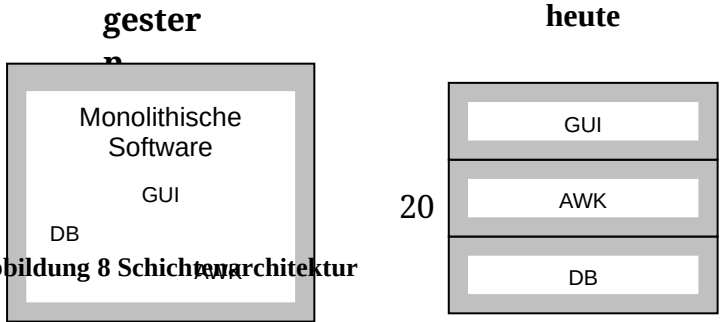


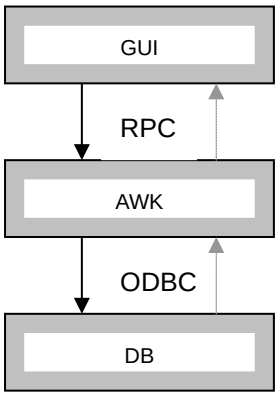
Abbildung 8 Schichtenarchitektur

auch aus diesem Grund kaum mehr wartbar. Daher war ein entscheidender Schritt die Aufteilung in funktionsorientierte Komponenten (Module), die einzeln austausch- und wartbar sind. Die folgende Grafik zeigt diese Evolution.

Die Aufteilung in funktionsorientierte Baugruppen (hier AWK-Anwendungskern, GUI-Dialogoberfläche und DB-Datenbankzugriffsschicht) ermöglicht eine bessere Wartbarkeit und einen, wie wir sehen werden, äußerst wichtigen Austausch von verschiedenartigen Implementierungen.

3.1.3 Verteilung

Wenn Baugruppen nur über Schnittstellen (Funktionen) miteinander kommunizieren, ist es leicht diese Funktionsaufrufe durch Middleware zu ersetzen. Als Middleware bezeichnet man die Systemsoftware zwischen Client und Server, die deren Kommunikation ermöglicht. Damit können Baugruppen über Prozeß- bzw. Rechnergrenzen hinweg kommunizieren.



Bei dem in Abb. 9 gezeigten Beispiel werden zwei leistungsfähige Server benötigt: Ein Datenbankserver

und ein Server der die Anwendungslogik bearbeitet. Der Client kann relativ schlicht ausfallen, da hier nur das Oberflächenprogramm ablaufen muß.

Viele moderne Client/Server-Systeme (Beispielsweise SAP R/3) setzen eine solche verteilte Drei-Schichtenarchitektur bereits erfolgreich ein. Damit ist die Benutzeroberfläche komplett von Anwendungslogik und Datenbankzugriffen entkoppelt.

Abbildung 9 Drei-Schichtenarchitektur

3.1.4 ROSALINDE

Wenn man den Anwendungskern weiter zerlegt kommt man zu einer Komponentensoftware (Componentware). Da der Begriff der Komponente nicht weiter spezifiziert wurde, und eine Komponente so ziemlich alles, angefangen von einer einzelnen Funktion bis hin zu einem umfangreichen Programmsystem, abdeckt, ist eine Erklärung hier notwendig: Jede Komponente übernimmt genau eine spezifizierte Teilaufgabe im Gesamtsystem. Bei der ROSALINDE-Anwendung entspricht jede Komponente exakt einer zugeordneten Objektklasse. So übernimmt eine Komponente die Bearbeitung von Kunden, eine andere die Bearbeitung von Konten. Ebenfalls existieren Komponenten zur persistenten Speicherung dieser.

ROSALINDE : 3-stufige / Komponenten-Architektur

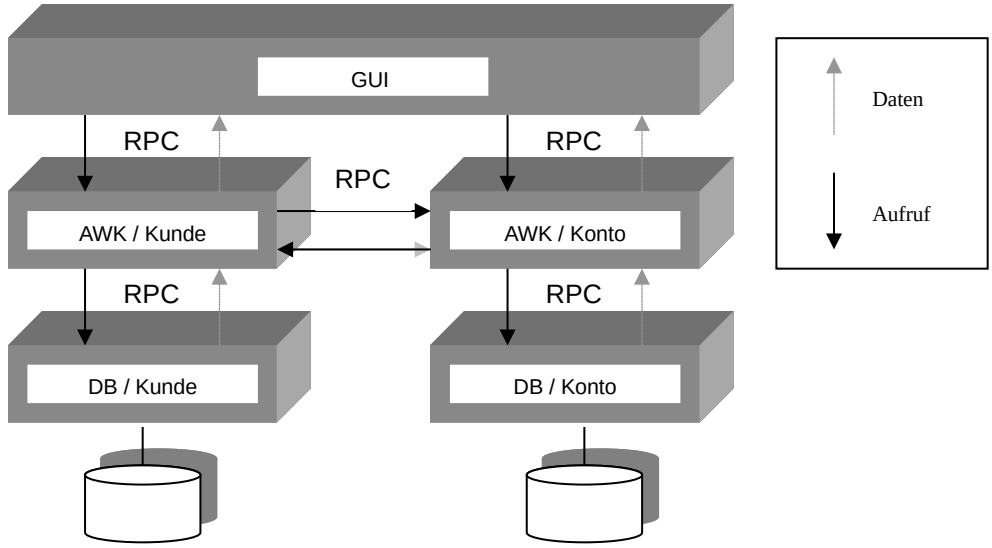


Abbildung 10 ROSALINDE Architekturübersicht

Abbildung 10 zeigt deutlich die beteiligten Komponenten der ROSALINDE-Anwendung. Jeder Kasten von dem ein Aufrufpfeil ausgeht fungiert als Client. Jeder Kasten der einen Aufrufpfeil annimmt, fungiert als RPC-Server der diese Aufrufe implementiert. Der einzige „echte“ Client ist das ROSALINDE-GUI, welches keinerlei Dienste anbietet sondern nur Serverdienste nutzt. Die Datenbankserver für Kunde und Konto sind die einzigen „echten“ Server, da diese nur Dienste bereitstellen aber keinerlei Dienste nutzen.

3.2 Servertypen

Die Server der ROSALINDE-Anwendung lassen sich, mit Ausnahme des Brokers, als Funktionen- und Datenserver klassifizieren.

3.2.1 Funktionenserver

Ein Funktionenserver übernimmt die fachliche Abarbeitung aller Operationen einer Objektklasse. Ein Funktionenserver ist eine Komponente, die zwischen dem Client-GUI und der Datenbank (hier flache Dateien) die Anwendungslogik repräsentiert. Die Funktionenserver sind entscheidend in der Architektur von mehrstufigen Client/Server-Systemen. Die üblichen zweistufigen Client/Server-Anwendungen bearbeiten die komplette Anwendungslogik auf der Clientseite, was unter Umständen zu Performanceproblemen am Client und hoher Netzbelastung führen kann. Die Performanceprobleme können durch die Bereitstellung von Funktionenservern vermieden werden.

- Der Funktionenserver des ADT „Kunde“ umfaßt Operationen wie beispielsweise Kunden anlegen, ändern, speichern, deaktivieren und löschen.

- Der Funktionenserver des ADT „Konto“ bearbeitet die Kontenneuanlage bzw. -kündigung und Geschäftsvorfälle wie „einzahlen“ oder „abheben“.

3.2.2 Datenserver

Der Datenserver speichert alle persistenten Daten eines abstrakten Datentyps. Mehrere Datenserver stellen eine verteilte Datenbank dar. Die dazu nötigen Transaktionsbedingungen: „atomar, konsistent, isoliert und dauerhaft“ (ACID-Eigenschaften) werden von der ROSALINDE-Anwendung nicht erfüllt, da es kein Ziel war eine verteilte Datenbank zu entwickeln. Die Datenhaltung werden in flachen, satzweise gespeicherten Dateien (`KNDE.DAT, KNTO.DAT`) gehalten. Der Zugriff erfolgt über den ADT – BTAB (Basistabelle). Der Datenserver ermöglicht einen Remote-Zugriff auf diese Satzdateien mit einfachen Lese-, Änderungs- und Schreiboperationen. Jeder Funktionenserver hat bei der ROSALINDE-Architektur einen zugehörigen Datenserver der seine Daten persistent speichern kann.

- Der Datenserver des ADT „Kunde“ ermöglicht die Speicherung von Kundendaten in die Datei „knde.dat“ als Zeichenketten.

- Der Datenserver des ADT „Konto“ ermöglicht ebenfalls die persistente Datenspeicherung von Kontendaten in der Datei „knto.dat“

Für die Zukunft ist ein Ersatz der Basistabelle durch eine ODBC-fähige Datenbank geplant.

3.3 Weitere Eigenschaften

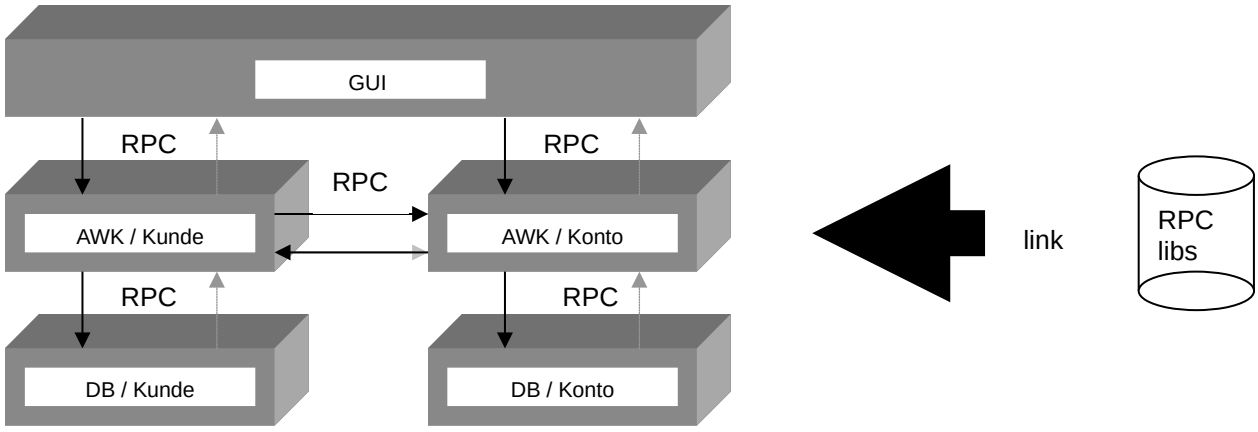
Der Datenaustausch unter den beteiligten Servern findet ausschließlich über entfernte Funktionsaufrufe (RPC) statt. Es gibt keine andere Möglichkeit, Daten untereinander auszutauschen. Das ROSALINDE-Projekt demonstriert eine verteile Anwendung mit einer MIMD – Systemarchitektur (Multiple Instruction – Multiple Data), die typisch ist für Server, die auf seperaten Host-Computern laufen (z.B. Workstation Cluster). Beim High-Perfomance-Parallelcomputing ist eine solche Architektur ebenfalls üblich (z.B. Parsytec CC 8 Prozessoren). Daten können durch die getrennten Prozeß-Adressräume (Distributed Memory) nur über Message Passing Mechnismen ausgetauscht werden (hier RPC). Es existiert kein „Shared Memory“. Daher ist im ROSALINDE-Projekt viel Sorgfalt in das Design der Schnittstellen (Interfaces) investiert worden, um alle Wechselwirkungen zwischen den Prozessen zu berücksichtigen, da Zugriffe auf „fremde“ Daten nicht direkt möglich sind. Diese Eigenschaft deckt sich im übrigen mit dem objektorientierten Grundkonzept der Kapselung. Dieses besagt, daß nur über Schnittstellen auf Objektdaten zugegriffen werden kann. Interne Details sind nach außen unsichtbar. Diese Eigenschaft ermöglicht es der ROSALINDE-Anwendung die lokale Implementierung durch die RPC-Client/Server- Implementierung auszutauschen.

Ein Broker vermittelt die Netzadressen der beteiligten Komponenten untereinander in Form einer dynamischen Tabelle, um so eine Netzwerktransparenz innerhalb der Anwendung zu erreichen. Daher ist die einzige Netzwerkadresse, die beim Programmstart bekannt sein muß, die des Objektbrokers. Wenn im folgenden von Tabellen die Rede ist, sind immer Hashtabellen gemeint die ROSALINDE-Anwendung intensiv nutzt, um eine sequentielle Suche durch Listen zu vermeiden.

Eine ganz grundlegende Eigenschaft des ROSALINDE-Systemdesigns ist, daß alle Komponenten lokal zu einer einzigen monolithischen Anwendung zusammengebaut werden können. Dies wird durch erneutes „linken“ der kompletten Anwendung mit den Implementierungsbibliotheken erreicht. Die Netzwerk-Schichten werden dazu weggelassen.

Die konsequente Kapselung der Anwendungsobjekte macht einen späteren Austausch der lokalen Methodenaufrufe durch RPC-Aufrufe möglich. Da verteilte Anwendungen schwierig zu „debuggen“ sind, macht sich dieser Aufwand schnell bezahlt. Entwickler können fachliche Programmlogik wie gewohnt lokal testen bzw. debuggen und werden von den Netzwerkproblemen abgeschirmt. Außerdem kann die ROSALINDE-Anwendung auch als Beispiel dienen, wie sauber gekapselte, objektorientierte Anwendungen auch zu einem späteren Zeitpunkt verteilt werden können. Es ist weiterhin möglich, bestehende Anwendungen über Schnittstellenkapseln als „Objekte“ im Netzwerk verfügbar zu machen.

ROSALINDE : 3-stufige / verteile Version



ROSALINDE : lokale Version

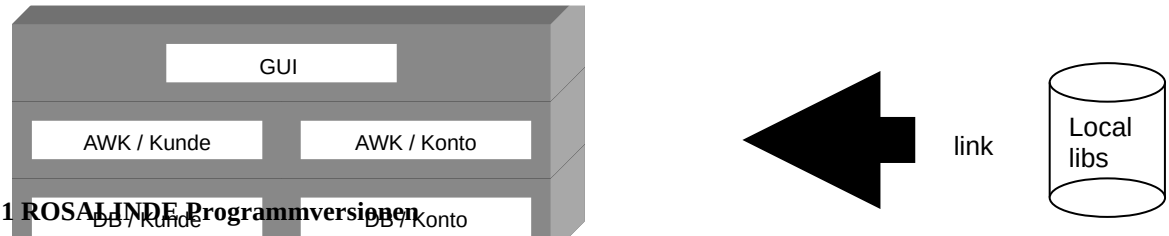


Abbildung 11 ROSALINDE Programmversionen

4 Der Broker

4.1 Funktion

Ein Broker (deutsch: Makler) vermittelt Clients die Informationen, die sie zum Nutzen eines Dienstes benötigen. Die Clients brauchen nichts über den Standort eines Servers bzw. Dienstes wissen. Die Informationen die ein Client normalerweise hartverdrahtet, über Initialisierungsdatei oder Registrierungsdatenbank benötigt, um einen entfernten Dienst per RPC-Call aufzurufen, stellt der Broker dynamisch zur Laufzeit zur Verfügung. Eine Anfrage der Clients am Broker, um die nötigen Informationen zu bekommen, nennt man Bindung. Die zugehörigen Daten bezeichnet man als Bindungsinformationen. Die Anmeldung von Servern am Broker nennt man Registrierung. Die einzige Adresse, die beim Programmstart eines Clients bekannt sein muß, ist die des Brokers. Alle weiteren Serveradressen werden zur Laufzeit des Clients über den Broker geliefert.

Den beteiligten Servern der ROSALINDE Anwendung (Daten- und Funktionenserver) wird beim Programmstart die Brokeradresse als Kommandozeilenargument mitangegeben.

4.2 Das Broker-Interface

Der Broker ist ein Serverdienst welcher ein Broker-Interface per RPC-Schnittstelle bereitstellt. Dieses Interface hat im wesentlichen folgende Methoden:

```
register ();      Fügt einen Serverdienst in die Brokertabelle ein.
unregister ();    Entfernt einen Serverdienst am Broker.
bind ();          Ermittelt die Netzinformationen eines Servers.
```

Während `register()` und `unregister()` von Servern benutzt werden um Informationen an- bzw. abzumelden (registrieren), wird `bind()` von den Clients aufgerufen die den Dienst eines Servers nutzen wollen und dessen Bindungsinformationen benötigen.

4.3 Die Broker-Tabelle

Jeder Server registriert seine Bindungsinformationen mit einem eindeutigen Namen am Broker mittels `register()`. Der Broker erzeugt dann einen neuen Eintrag in der Brokertabelle. Diese Einträge heißen im ROSALINDE-Projekt Broker-Entrys (kurz: bent). Sie sehen folgendermaßen aus:

Tabelle 1 Brokertabelle

Dienstname	Netzadresse	Protokoll	RPC-Programkennung	Versnr
Kunde Funktionen	nexus.fh-rosenheim.de	tcp	0x1234567	1
Konto Funktionen	panoramix.enssat.fr	udp	0x1234568	2
Konto Daten	merlin.enssat.fr	udp	0x1234569	1
...	...	...	...	...

Die Einträge in der Brokertabelle korrespondieren direkt mit den Parametern, die ein Client benötigt um eine RPC-Verbindung zu eröffnen. Über einen solchen Eintrag kann ein Client mit *`clnt_create (char * host, u_long prognr, u_long versnum, char * protocol)`* (ONC-RPC-API)[1] eine RPC-Verbindung zu einem entfernten Server aufbauen. Die Einträge in dieser Form hängen direkt mit der Verwendung des ONC-RPCs zusammen. Bei einer anderen RPC-Architektur (z.B. DCE) müßten die Einträge in der Brokertabelle die benötigten Bindungsinformationen dieser enthalten.

Der Primärschlüssel der Brokertabelle ist der Dienstname, der im gesamten Netzwerk eindeutig sein muß. Aus Gründen der Lesbarkeit sind diese Dienstnamen in der ROSALINDE-Anwendung sprechend. Um die Eindeutigkeit dieser Zeichenketten zu gewährleisten, ist eine maschinengenerierte Kennung sinnvoll. Der DCE-RPC beispielsweise verwendet sogenannte UUIDs (Universal Unique IDentifier) als eindeutige Kennzeichnung der Schnittstellen. Die UUIDs können mit dem Werkzeug (`uuidgen`) generiert werden und verwenden Datum,

Uhrzeit, Rechnername und Hardwareadresse der Netzkarte in verschlüsselter Form. So ein Dienstname könnte z.B. folgendermaßen aussehen:

```
KUNDEN_FUNKTIONEN_SERVER="069961e0-b5ac-11d0-b867-00aa00ae3571".
```

Die dynamische Tabelle des Brokers ist, verglichen mit einer statischen Adressauflösung via Initialisierungsdatei oder Registrierungsdatenbank, wesentlich flexibler, weil Server im laufenden Betrieb ausgetauscht werden können. Das dynamischen Umschalten verschiedener Server, die gleiche Dienste anbieten, ist eine wesentliche Funktionalität des ROSALINDE-GUIs.

Denkbar ist auch eine Lastverteilung, die über den Broker gesteuert wird. Der ROSALINDE Broker kann zwar momentan nur einen Eintrag pro Dienstnamen verwalten, könnte aber mit wenig Aufwand erweitert werden. So könnten zur Laufzeit Aufrufe an verschiedene Server zur Lastverteilung delegiert werden.

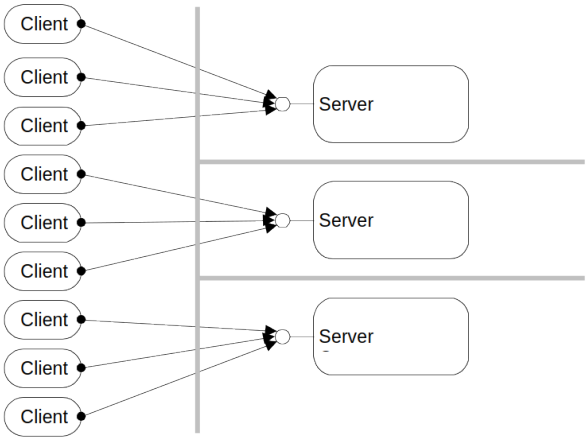


Abbildung 12 Lastverteilung

Falls ein Server heruntergefahren wird, oder sich bei einem Fehlerfall beendet, muß der Eintrag aus der Brokertabelle entfernt werden. In diesem Fall wird am Broker `unregister()` aufgerufen und der Eintrag gelöscht.

4.4 ROSALINDE Broker (r_brok_svc)

Der ROSALINDE Broker (r_brok_svc) ist ein RPC-Server mit den drei Funktionen: „register“, „unregister“ und „bind“. Falls alle Server erfolgreich gestartet wurden, enthält die Brokertabelle Einträge für vier Server. Das sind zum einen die Server für die fachliche Bearbeitung von Kunden bzw. Konten, zum anderen die Datenbanken (hier flache Dateien) für Kundendaten und für Kontendaten. Die Brokertabelle des ROSALINDE Brokers ist nicht persistent. Das bedeutet, daß alle erfolgten Registrierungseinträge beim Beenden des Brokers verloren gehen.

Insgesamt stehen also sechs Prozesse in Kommunikation miteinander: (IPC)

- Broker
- Kunden-Daten-Server
- Kunden-Funktionen-Server
- Konten-Daten-Server
- Konten-Funktionen-Server
- Client (GUI)

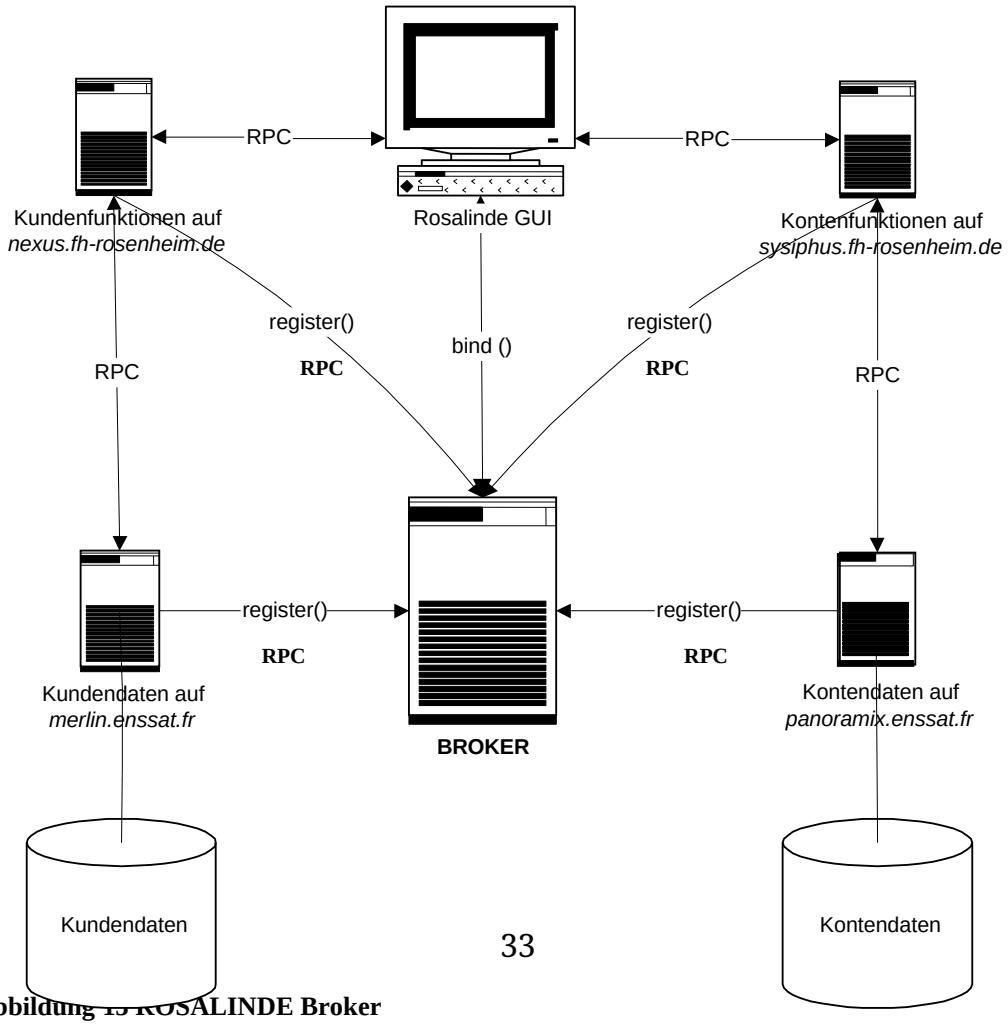
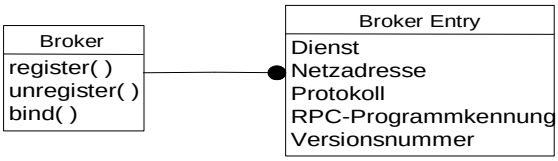


Abbildung 13 ROSALINDE Broker

4.4.1 Schematischer Bindungsablauf

Abbildung 14 zeigt den Ablauf der Adreßauflösung über den Broker: Im ersten Schritt registrieren sich ein oder mehrere Server am Broker (1). Clients, die Server-Dienste benötigen, bekommen mittels *bind*(2) über den Dienstnamen den zugehörigen Eintrag in der Brokertabelle zurückgeliefert (3). Schließlich kann der Client den Dienst der Servers direkt in Anspruch nehmen (4). Die RPC-Aufrufe gehen von da an direkt an den Server. Es entsteht kein weiterer Overhead durch den Broker. Die Information, die ein Client vom Broker bekommen hat, werden sinnvollerweise gespeichert, so daß der „bind“ nur beim allerersten Aufruf stattfindet. Beendet ein Server den Dienst, wird der zugehörige Eintrag mit `unregister()` aus der Brokertabelle entfernt. Falls sich ein Server beendet, (5) der Client aber noch einen Eintrag auf diesen gespeichert hat, wird der nächste RPC-Aufruf fehlschlagen. In diesem Fall wird eine Ausnahme ausgeworfen, die den Client zum erneuten „*bind*“ über den Broker zwingt. Erst wenn kein gültiger Server mehr gefunden wird, beendet sich auch der Client.



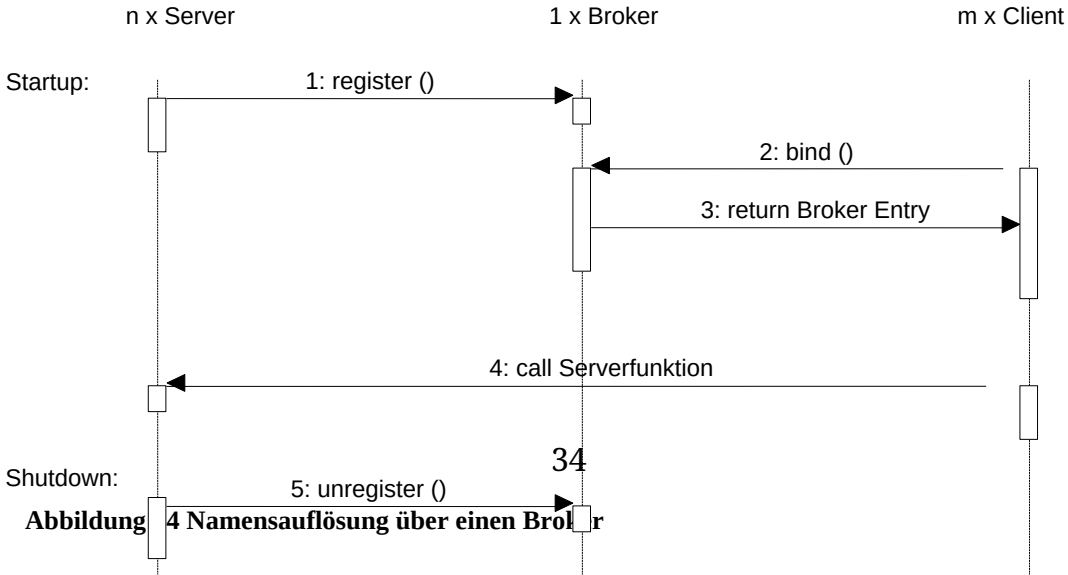


Abbildung 14 Namensauflösung über einen Broker

5 Objektdesign der ROSALINDE-Anwendung

5.1 Abstrakte Datentypen

Ein wesentliches Konzept des ROSALINDE-Systemdesigns ist die Verwendung abstrakter Datentypen (ADTs). Ein abstrakter Datentyp, ist eine Datenkapsel bei der alle Implementierungsdetails verborgen sind. Der Zugriff auf den ADT erfolgt nur über Funktionsaufrufe seiner Schnittstelle. Ein abstrakter Datentyp darf aber nicht mit einem Programmodul bzw. einer Bibliothek verwechselt werden, da ein solcher immer das Verhalten einer „Klasse“ (im Sinn von Gruppe) von Objekten beschreibt. Dies ist ganz analog zu den in den Programmiersprachen eingebauten Typen wie Integer oder String zu sehen. Beispiele für abstrakte Datentypen der ROSALINDE-Anwendung sind: Kunde, Adresse, Konto und Liste. Man kann abstrakte Datentypen durchaus als Objekte bezeichnen, obwohl dies im Sinne der Objektorientierung nicht ganz korrekt ist, da die Kapselung nur ein Aspekt ist und wesentliche Konzepte wie Polymorphie, Vererbung oder Generizität fehlen. Es wird folgende Begriffsanalogie vereinbart:

1. Ein abstrakter Datentyp (ADT) wird auch als Typ oder Klasse bezeichnet.

2. Eine Variable eines abstrakten Datentyps wird auch als Objekt oder Instanz einer Objektklasse bezeichnet.

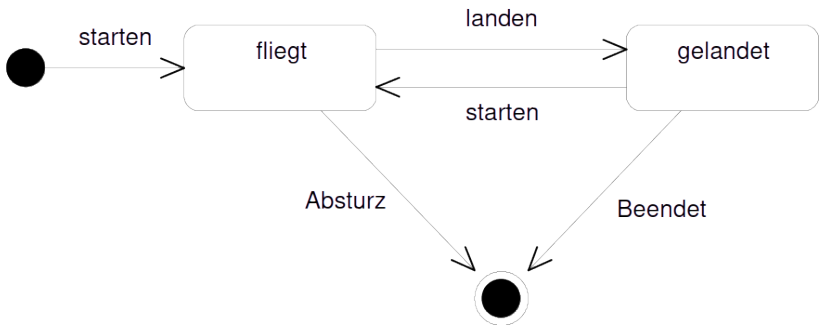
3. Eine Funktion eines abstrakten Datentyps wird auch als Methode bezeichnet.

Abstrakte Datentypen lassen sich per Konvention in jeder Programmiersprache implementieren, wobei sich hierbei die Sprache „C“ hervorragend eignet.

Genaugenommen eignet sich jede Programmiersprache die über einen Gruppierungsmechanismus für Daten und ein „Call by Reference“ verfügt.

5.1.1 Trennung von Schnittstelle und Implementierung

Abstrakte Datentypen ermöglichen eine vollständige Trennung der Schnittstelle (Interface) von der Implementierung. Ein Nutzer einer ADT-Schnittstelle benötigt keinerlei Wissen über den internen Aufbau eines ADT-Objektes. Dies hat den Vorteil, daß sich die Komplexität drastisch reduziert, da ein Nutzer eines ADTs nur die Schnittstelle kennen muß, nicht aber dessen Implementierungsdetails. Da der ADT nur über Schnittstellen verändert werden kann, läßt sich der Lebenszyklus eines solchen ADTs als endlicher Automat darstellen, bei dem die Zustandsübergänge durch Methodenaufrufe verursacht werden.



Das Beispiel zeigt den ADT-„Flugzeug“ mit den zwei Zuständen „fliegt“ und „gelandet“.

Abbildung 15 Zustandsautomat eines ADTs

Diese Automatennotation hat bei komplexen Objekten wie einem Geschäftsvorfall große Vorteile. Beispielsweise werden bei einer Rechnungsstellung in einem Unternehmen diverse Schritte vollzogen, die das

Objekt (Rechnung) in den nächsten Zustand bringen. Viele Zustandsübergänge sind dabei strikt verboten; beispielsweise kann nicht zweimal hintereinander eine erste Mahnung erfolgen. Dies wird bei ADTs durch Vor- und Nachbedingungen innerhalb der Methoden gewährleistet. Desweiteren können Zustandsübergänge nur durch expliziten Methodenaufruf erfolgen, mit der Folge, daß verbotene Übergänge nicht stattfinden können.

Die Trennung von Schnittstelle und Implementierung ermöglicht der ROSALINDE-Anwendung den wie im Kapitel 3.1.4 beschriebenen Austausch der lokalen Funktionsaufrufe durch Remote Procedure Calls.

5.2 Objektidentität, -kopien und Persistenz (lokale Anwendung)

Objektorientierte Systeme müssen die Objektidentität der Anwendungsobjekte sicherstellen. Dies betrifft einerseits die Datenbank in der die Objekte persistent gespeichert werden und andererseits das Anwendungsprogramm bei dem Objekte im Speicher gehalten werden.

5.2.1 Row-ID

Relationale Datenbanken haben gegenüber OO-Datenbanken die Eigenschaft, daß sich Objekte nur durch ihre Attributwerte (Spalteninhalte) identifizieren. In der Praxis ist das ein Nachteil, da es beispielsweise zwei Kunden mit dem Namen „Hans Maier“ geben kann. Die Objektidentität in einer Tabelle die nur den Kundenvornamen und –nachnamen speichert, ist somit nicht automatisch gewährleistet. Daher behilft man sich in relationalen Datenbanksystemen mit Schlüsselattributen, welche innerhalb einer Tabelle eindeutig sein müssen. Ein solches unter Umständen aus einzelnen Spalteninhalten zusammengesetztes Schlüsselattribut heißt Primärschlüssel. Ein Beispiel für einen Primärschlüssel ist die Kundennummer eines Kunden oder die fortlaufende Bestellnummer einer Bestellung. Die Eindeutigkeit aller Datensätze und deren Beziehungen wird bei der ROSALINDE-Anwendung über eine Row-ID Satznummer erreicht. Diese identifiziert innerhalb einer Tabelle einen Datensatz eindeutig und verhindert zusammengesetzte Fremdschlüsselverweise auf andere Tabellen.[1]

Abbildung 16 Beziehungsauflösung über die Row-ID [2]

1. Als ROW-ID kann direkt die Satznummer einer satzbasierten Datei verwendet werden, ohne einen zusätzlichen Schlüssel einzuführen (ROSALINDE).

2. Die Datenhaltung bei der ROSALINDE-Anwendung erfolgt nicht in Microsoft Access, wie oben abgebildet, sondern über physikalische Dateien. Zum besseren Verständnis wurde die Abbildung, in Microsoft Access nachgebildet.

Falls eine physikalische Satznummer als Row-ID verwendet wird, dürfen Sätze nicht gelöscht werden, da sich die Satznummern verschieben und Beziehungen zerstört würden. Dies ist aber kein Problem, da in realen Anwendungen nie physikalisch gelöscht wird, um eine spätere Historisierung zu ermöglichen. Das physikalische Löschen wird, wenn unbedingt nötig, von einem Reorganisationsprogramm übernommen, welches in der Lage sein muß, alle Beziehungen wieder korrekt herzustellen. Das „logische“ Löschen übernimmt bei der ROSALINDE-Anwendung die Basistabelle, die eine Satzmarkierung verwaltet, welche angibt ob der Satz gelöscht wurde oder nicht.

Das Konzept der Row-ID wird in der ROSALINDE-Anwendung über den persistenten Kern (pkrn) abgebildet, der die Row-ID der Anwendungsobjekte speichert:

```
typedef struct {
    long    rid;    eindeutige Row-ID, entspricht der Satznummer in der BTAB
    ...
} pkrn;
```

Alle persistenten Objekte der ROSALINDE-Anwendung (Kunde und Konto) besitzen als Attribut einen aggregierten, persistenten Kern (siehe Kapitel 9.6.1 Persistenter Kern (pkrn)).

```
typedef struct {
   pkrn p;
   ...
} knde;
```

Der persistente Kern führt desweiteren Buch über den internen Zustand des Objektes. Von dem Zustand ist abhängig, ob ein Objekt in der Datenbank neuangelegt, geändert oder als temporäres Objekt ignoriert wird.

Relationale Datenbanksysteme können die Eindeutigkeit von Schlüsselspalten (Row-ID) sicherstellen. Auch bei der Basistabelle der ROSALINDE ist gewährleistet, daß die Row-ID innerhalb dieser eindeutig ist, da wie schon erwähnt, die physikalische Satznummer als Row-ID verwendet wird. Der Nachteil des Row-ID-Konzeptes sei hier nicht verschwiegen: Die Row-ID ist ein künstlicher Schlüssel und ist damit nicht-sprechend. Dies steht im Gegensatz zur Grundidee von relationalen Datenbanken, die eine Identifikation über Indizes gerade eliminieren wollen. Ebenso ist eine Row-ID kein systemweit eindeutiger Schlüssel. Daher ergibt sich ein Zusatzaufwand beim Sicherstellen der Objektidentität im Anwendungsprogramm, da nur die Kombination von Tabellenname und Row-ID eindeutig ist.

5.2.2 Objektkopien

Objektkopien können in Anwendungsprogrammen zu einem katastrophalen Verhalten führen. Folgendes Beispiel der ROSALINDE-Anwendung soll dies illustrieren:

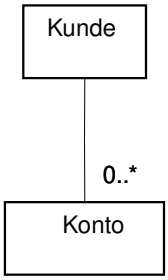


Abbildung 17

Ein Kunde besitzt mehrere Konten: Wenn nun im Programm zwei Konten gelesen werden kann es vorkommen, daß der gleiche Kunde zweimal aus der Datenbank mitgelesen wird. Welcher Kunde soll nun geändert werden?. Was passiert mit dem zweiten Kunden, wenn der erste geändert wird? Um solche Probleme zu eliminieren, muß sichergestellt werden, daß sich nur eine einzige Instanz des Objektes im Hauptspeicher befindet.

Die Objektidentität wird in der lokalen Version der ROSALINDE-Anwendung, wie schon erwähnt über die Row-ID sichergestellt. Momentan werden keine echten (deep) Kopien von persistenten Objekten zugelassen. Es gibt pro Satz in der Basistabelle genau eine Objektinstanz mit einem eindeutigen Schlüssel: Der Row-ID.

Um die Objektidentität zu sichern, werden alle Objektinstanzen von einer zentralen Tabelle verwaltet. Diese Tabelle heißt Objekttabelle.

5.2.3 Die ROSALINDE Objekttabelle

Die Objekttabelle der ROSALINDE-Anwendung dient in der lokalen Version ausschließlich zur Verwaltung von persistenten Objekten. Die Aufgabe der Objekttabelle besteht im Sicherstellen der Objektidentität von persistenten Objekten. Jedes persistente Objekt wird dort eingetragen und kann über dessen

Row-ID gesucht werden. Bei der ROSALINDE Anwendung ist sichergestellt, daß die Row-ID für jedes Objekt systemweit eindeutig ist.[1] Es wird beim Lesen eines Objektes immer zuerst die Objekttabelle durchsucht und erst anschließend auf die Datenbank zugegriffen (siehe Abb. 18).

1. Bei der ROSALINDE-Implementierung kann nur deshalb über die Row-ID in der Objekttabelle gesucht werden, da eigene Row-ID-Offsets für Kunde und Konto vergeben werden. Kunden haben als Start-Row-ID die 0 und Konten haben als Start-Row-ID 10000. Da bei diesem Konzept die möglichen Sätze einer Tabelle begrenzt werden, sollte ein systemweit eindeutiger Schlüssel vergeben werden (syskey).

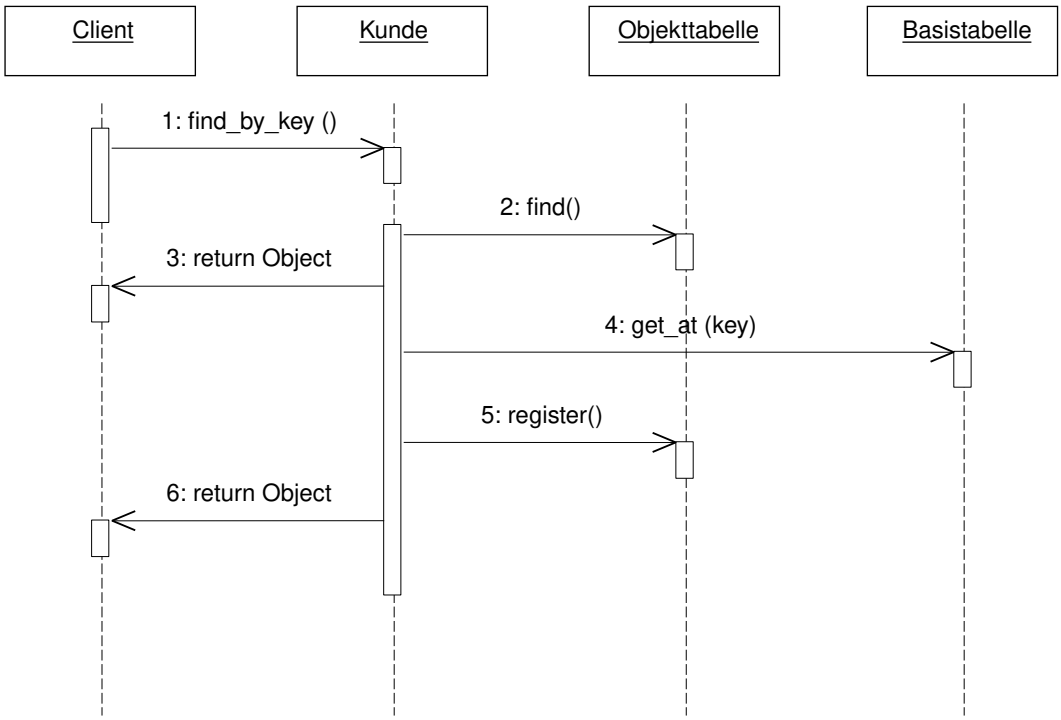


Abbildung 18 Objektsuche über die Objekttabelle

Die Objekttabelle übernimmt aber auch noch eine weitere Aufgabe: Die Zuordnung von Objekt-IDs zu realen Hauptspeicheradressen. Dies ist allerdings nur sinnvoll bei Systemen, die über eine Objekt-ID auf ein Objekt zugreifen müssen (siehe Kapitel: 6.2 Proxys).

6 Verteilte Objekte

Als ein „verteiltes Objekt“ werden im folgenden Objekte bezeichnet, deren Schnittstellen von beliebigen Prozessen oder Rechnern innerhalb eines Netzwerks benutzt werden können. Das Objekt selbst befindet sich nur auf einem einzigen Rechner und wird von einem Serverprozeß erzeugt, bearbeitet und gelöscht. Bei der ROSALINDE-Anwendung kann ein Server ausschließlich Instanzen einer einzigen Objektklasse (z.B. Kunde) erstellen und bearbeiten.

6.1 Anforderungen

Die wichtigste Anforderung bei einer Komponenten-C/S-Architektur ist die Netzwerktransparenz. Für die Nutzer eines Objektes darf es keinen Unterschied geben, ob sich ein Objekt tausende von Kilometern entfernt oder am gleichen Rechner befindet.

Desweiteren muß eine hohe Netzwerkbelastung vermieden werden, da das Netzwerk, nach der Datenbank, der größte „Flaschenhals“ ist.

Die Netzwerktransparenz wird einerseits durch den Broker ermöglicht, andererseits durch Funktionsstummel (Stubs). Letztere tauschen die lokale Implementierung aus und kapseln die gesamte Kommunikation für den Benutzer unsichtbar, was wiederum einen Brokerdienst voraussetzt.

Eine gute Netzwerkperformance wird bei der ROSALINDE-Anwendung durch die Verwendung von Proxys und das Speichern der Verbindungsdaten erreicht.
-> Kapitel 6.2 (Proxys)

6.2 Proxys

Sinnvollerweise werden nicht die kompletten Anwendungsobjekte (möglicherweise sogar Hierarchien) zwischen den Servern transferiert, sondern es wird mit einem Proxymechanismus auf die Objekte zugegriffen. Ein Proxy ist ein Stellvertreterobjekt, welches die gleichen Methoden wie das eigentliche Objekt zur Verfügung stellt, aber – wie bei ROSALINDE - keine oder nur minimale Daten enthält.[1] Die Methoden des Proxys dienen nur zur Delegation der Aufrufe an das eigentliche Objekt. Zur Unterscheidung der Objekte am Server werden diese mit einer eindeutigen Objekt-ID (OID) gekennzeichnet, welche der Proxy speichert. Für den Client ist der Proxy vollständig transparent, kann er doch alle Methoden des Anwendungsobjektes aufrufen.

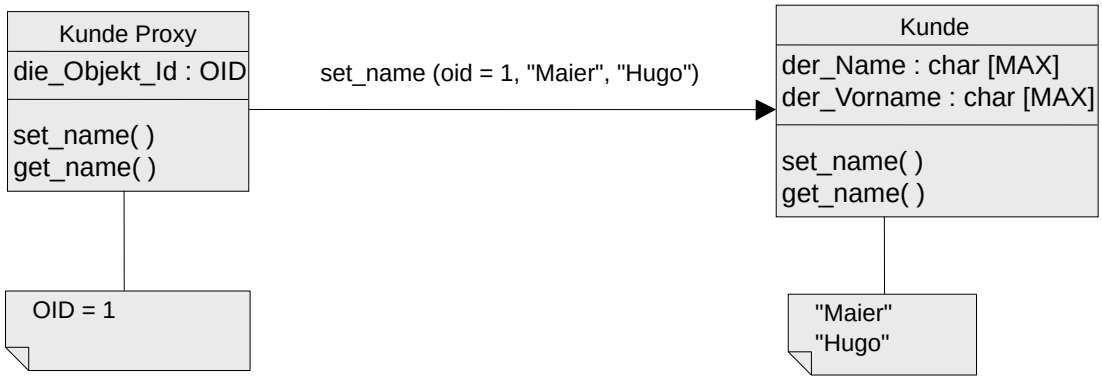


Abbildung 19 Proxy

1. Allerdings ist hier ein effizienter Caching-Mechanismus denkbar, indem die Proxys gewisse Daten speichern können

6.2.1 Vorteile des Proxykonzeptes

6.2.1.1 Kapselung

Proxys bieten perfekte Datenkapselung wie sie nur in wenigen Programmiersprachen überhaupt realisierbar ist. Der Grund dafür ist, daß ein Client, der einen Proxy auf ein Anwendungsobjekt erhält, nur die OID des Anwendungsobjektes kennt und alle weiteren Informationen über RPC-Funktionsaufrufe vom Server bekommt. Die OID kann auch als Objekthandle bezeichnet werden, da über diese auf das Serverobjekt zugegriffen werden kann. Die RPC-Serverschnittstelle ist das „public“ Interface des Objektes. Alle internen Serverfunktionen sind „private“. Diese Kapselung kann nicht, wie beispielsweise in C++ möglich, umgangen werden. Sicherheit, Wiederverwendbarkeit, Wartbarkeit und Zuverlässigkeit werden damit erhöht.

6.2.1.2 Marshalling referenzierter Objekte (Rattenschwanzlesen)

Bei allen RPC-Architekturen werden üblicherweise Marshallingfunktionen generiert, die komplette Objekte am Server „flachklopfen“ und verpacken, über das Netz schicken und im Adreßraum des Clients wieder rekonstruieren. Dieses Vorgehen ist für einfache Anwendungen sinnvoll, versagt aber bei komplexen Objektbeziehungen. Um ohne Proxykonzept das Anwendungsobjekt Kunde, der ROSALINDE-Anwendung, zu ändern, müßten alle Attribute des Kunden zum Client transferiert, dort geändert und anschließend zurückgeschrieben werden. Dies ist schon notwendig, wenn nur ein Bit einer Datenstruktur geändert wird. Proxys vermeiden dies, da die Änderung zusammen mit der zugehörigen OID zum Server gesendet wird und die Datenstruktur dort geändert wird.

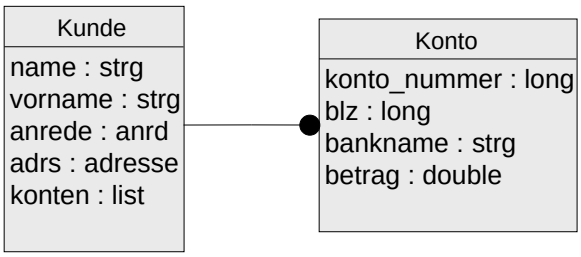


Abbildung 20 Kunde - Konto

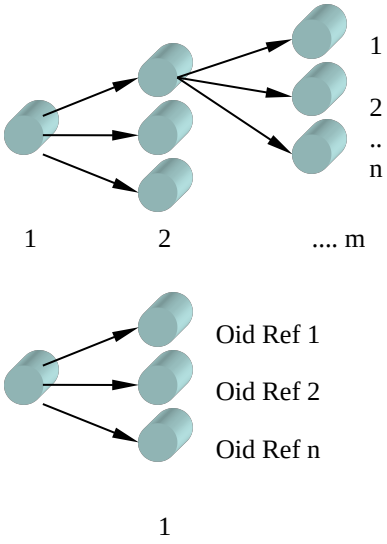
Beispiel: Ein Kunde kann mehrere Konten besitzen. Bei naiver RPC-Programmierung müßte bei einem Umzug (Adreßänderung) das komplette Objekt mit allen Attributen zum Client transferiert, dort geändert und zurück zum Server übertragen werden. In der Praxis sind dies aber nicht nur vier Attribute, die umsonst über das Netz kopiert werden sondern ein Vielfaches mehr. Schlimmer noch: Wenn kein Proxymechanismus verwendet wird, muß der Kunde mit all seinen Konten kopiert werden.

6.2.1.3 Laufzeitverhalten und Komplexitätsbetrachtungen

Bei durchschnittlich *n*-Referenzen und einer Referenztiefe von *m* ergibt sich eine extreme Komplexität der Marshallingroutinen von:

$O(n^{m-1})$ gegenüber $O(n)$ bei der Verwendung von Proxys.

Die folgende Grafik macht dies deutlich:



rekursives „Rattenschwanz“-marshalling

Proxymarshalli

Abbildung 21 Marshalling

Die Anzahl zu bearbeitenden Knoten in einem vollständigen, ausgeglichenem Baum ergibt sich als geometrische Reihe nach der Formel:

$$\sum_{i=1}^{m} n^{i-1} = \frac{n^m - 1}{n - 1}$$

Man erkennt, daß die Anzahl der zu bearbeitenden Knoten bei einem „Rattenschwanz“-marshalling extreme Ausmaße annehmen kann. Es gibt

allerdings auch Fälle bei denen ein „Rattenschwanz“-marshalling Vorteile bietet: Falls alle Knoten eines Baumes bearbeitet werden müssen, erzeugt die Verwendung von Proxys zusätzlichen Mehraufwand, da jeder Proxy einzeln expandiert und anschließend das Objekt bearbeitet werden muß.

Zumeist aber müssen die referenzierten Objekte gar nicht mitbearbeitet werden. In solchen Fällen ist der Verwendung von Proxys unbedingt anzuraten.

Ein anderer großer Vorteil, der sich bei der Verwendung von Proxys ergibt, ist das implizite Aufbrechen von Referenzzyklen innerhalb der Übergabeparameter. In der ROSALINDE-Anwendung hat jeder Kunde eine Liste von zugehörigen Konten. Jedes Konto hat seinerseits einen Zeiger auf den Kunden.

Dieser Umstand ergibt sich direkt aus der Anwendungsmodellierung.

```
typedef struct {
    ...
    list konten;
} kunde;
```

```
typedef struct {
      ...
      knde * der_Kunde;
 } konto;
```

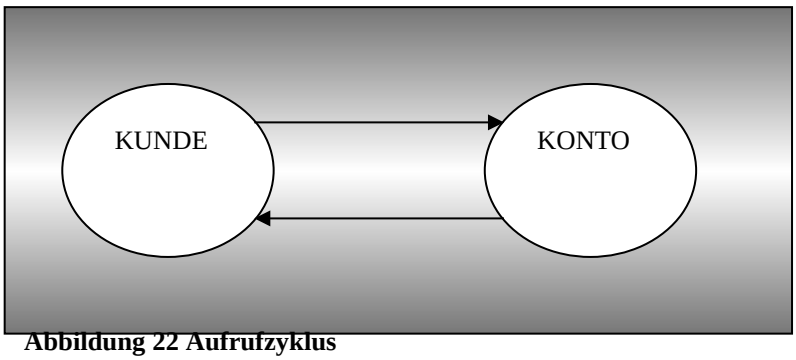


Abbildung 22 Aufrufzyklus

Hier wird deutlich, daß ein einfaches, rekursives Marshalling fehlschlagen muß. Sobald die Marshallingroutinen auf das erste Konto des Kunden in der Liste treffen, wird der Zeiger auf das Konto dereferenziert und anschließend versucht das Konto „flachzuklopfen“. Das Konto enthält aber wiederum einen Zeiger auf

den Kunden, der daraufhin wieder bearbeitet wird. Dies führt zu einer Endlosrekursion ohne Abbruchbedingung. Sind aber die Objektreferenzen nicht als Zeiger, sondern als Proxys implementiert, so findet ein Marshalling nur für die Proxys statt und das rekursive Verfolgen von Zeigern entfällt; der Zyklus ist damit aufgebrochen.

Mittels trickreicher C-Programmierung ist die Verwendung von Proxys im ROSALINDE-Projekt für den Client völlig unsichtbar, da die Zeigervariablen der Objekte im Client-Adressraum zur Speicherung der OID direkt verwendet werden. Die ROSALINDE-Anwendung verwendet untypisierte C-Zeigervariblen (void *) zur Speicherung der OID. Diese Zeiger dürfen natürlich nie dereferenziert werden, was bei untypisiertem Zeiger ohnehin nicht möglich ist.

Beispiel:

```
// Kunde als RPC-Proxy
typedef void knde;
```

```
// Kunde Implementierung
typedef struct {
  ...
} knde;
```

Mehr über die Implementierung findet sich im Kapitel 9 (Implementierung).

6.2.2 Erweiterungen der Objekttabelle

Neben der, im Kaptitel 5.2.3 (Die ROSALINDE Objekttabelle) erläuterten Sicherung der Objektidentität, übernimmt die Objekttabelle in der verteilten Programmversion noch eine weitere Aufgabe: Eine Zuordnung von Objekt-IDs zu realen Objektadressen und damit die Expansion der Proxys am Server. Das ist sinnvoll, weil die Objekttabelle schon eine Liste aller persistenten Objekte verwaltet. Diese muß nur noch um die Speicherung von nicht-persistenten Objekten erweitert werden. Es wäre zwar möglich, sich die Zuordnung der Objekt-IDs in Hauptspeicheradressen zu ersparen und als Objekt-ID direkt die Hauptspeicheradresse eines Anwendungsobjektes am Server zu vergeben. Dem ist aber aus Sicherheitsgründen abzuraten, da über eine manipulierte OID am Server direkt auf fremde Daten zugegriffen werden könnte.

Die Objekttabelle (OTAB) sieht damit folgendermaßen aus:

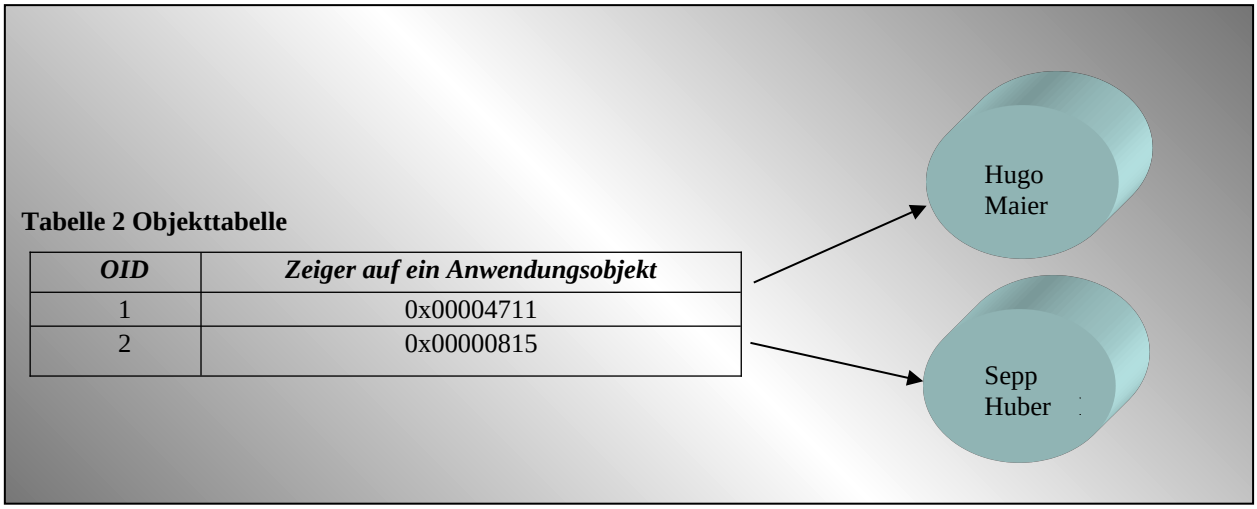


Tabelle 2 Objekttabelle

OID	*Zeiger auf ein Anwendungsobjekt*
1	0x00004711
2	0x00000815

Abbildung 23 Objekttabelle

Es kann über die OID in der Objekttabelle gesucht werden. Falls ein Eintrag gefunden wird, kann der, in der Tabelle gespeicherte Zeiger auf das Anwendungsobjekt zurückgegeben werden. Die Objekttabelle ermöglicht also die Zuordnung der Objekt-IDs eines Clients zu den zugehörigen Anwendungsobjekten.

6.2.3 Clienttabelle

Mit der Objekttabelle können alle Anwendungsobjekte eines Clients unterschieden werden. Bei mehreren Clients muß der Server in der Lage sein, die Objekte der Clients zu unterscheiden. Nicht nur aus Sicherheitsgründen ist es notwendig, daß der Server einen Zugriff über eine OID auf ein Objekt erkennt welches dem Client nicht gehört. Falls ein Client sich beendet ist es eventuell nötig, alle Objekte am Server zurück in die Datenbank zu schreiben und diese aus dem Adreßraum des Servers zu entfernen. Die ROSALINDE-Anwendung verwendet deshalb eine erweiterte Form der OTAB, die Client-Tabelle (CTAB), welche pro Client eine eigene Objekttabelle enthält bzw. erzeugt. Der Zugriff geschieht über eine eindeutige Kennung die der Broker beim Programmstart des Clients vergibt, die sogenannte CID (Client-ID). Damit wird ermöglicht, daß jeder Client seine „eigenen“ Anwendungsobjekte erhält und nicht Objekte fremder Clients verändert.

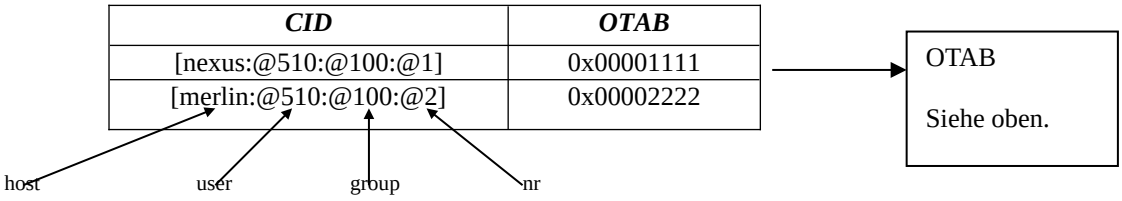
Die Sicherheit dieses Konzeptes hängt direkt von dem verwendeten RPC-Mechanismus ab. Sie ist im Fall des ONC-RPCs nicht gegeben, da der ONC-RPC nur ein minimales Authentifizierungskonzept hat. Den ONC-RPC durch einen „sicheren“ RPC wie DCE in der ROSALINDE-Anwendung auszutauschen ist aber

mit minimalem Aufwand möglich. Eine umfassende Diskussion der Sicherheit des DCE-RPCs findet sich in „Rosenberry, Understanding DCE“. [21]

Die Client-ID wird aus dem Hostnamen, der User-ID des Users welcher den Client gestartet hat, seiner Gruppen-ID und einer fortlaufenden Ziffer generiert.

Tabelle 2 Client-Tabelle

CID	*OTAB*
[nexus:@510:@100:@1]	0x00001111
[merlin:@510:@100:@2]	0x00002222



Eine Objekttabelle pro Client anzulegen bringt auch Vorteile in der Persistenzschicht der ROSALINDE-Anwendung, da ein `commit()` bzw. `rollback()` alle Objekte eines Clients bearbeiten muß und abhängig von ihrem Zustand (*geändert, neu* ...) die Objekte in die Datenbank schreibt. Mehr zur Persistenz, Referenzcounting, OTAB und CTAB findet sich im Kapitel 9 Implementierung.

6.3 Bindung über den „Server-Administrator“

Der Broker liefert wie schon erwähnt, die zur Erzeugung von gültigen RPC-Verbindungshandles nötigen Informationen. Da es wenig Sinn macht, vor jedem RPC-Aufruf den Broker zu kontaktieren, den Verbindungshandle zu erzeugen und nach dem Aufruf wieder zu vernichten, werden diese zwischengespeichert. Diese Aufgaben (Caching von Handles und Bindung über den Broker) übernimmt der „Server-Administrator“, der mittels `sadm_get_serverhandle(„Dienstname“)` in der Lage ist einen gültigen Verbindungshandle zu liefern (falls überhaupt möglich), der als erster

Parameter dem RPC-Aufruf mitgegeben wird. Dies enthält einen Test ob die Verbindung noch besteht und im Fehlerfall ein erneutes Abfragen des Brokers (bind).

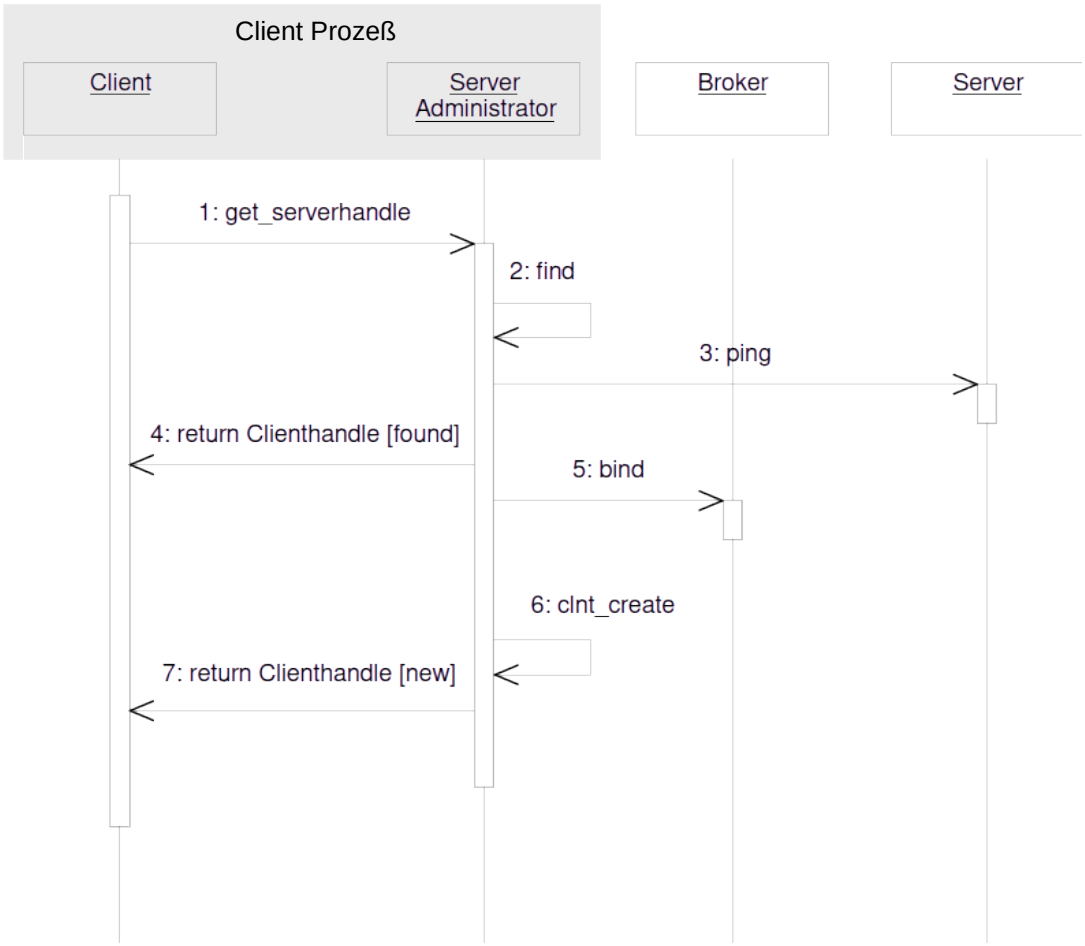


Abbildung 24 Server-Administrator

6.4 Stubs und Marshalling

Der aus dem Bereich der Betriebssysteme übernommene Begriff des Stubs (Stummel) gewinnt in verteilten Systemen erneut an Bedeutung. Während ein Betriebssystem-Stub nur dazu dient, mittels Supervisor-Call eine Betriebssystemfunktion aufzurufen, haben die Stubs in verteilten Anwendungen vielfältige Aufgaben. Ein Stub sollte idealerweise den RPC-Aufruf komplett von der Anwendung abkapseln, d.h. der Stub hat exakt die gleiche Signatur wie der lokale Funktionsaufruf. Dies bedingt beim SUN-RPC aber selbstgeschriebene Stubs, da sich die RPC-Aufrufe deutlich von lokalen Funktionsaufrufen unterscheiden.

6.4.1 Stubs

Die Stubs der ROSALINDE-Anwendung (Client-Stub / Server-Stub) bestehen aus zwei Teilen. Zum einen gibt es generierte Stubs, die mit dem Werkzeug `rpcgen` erzeugt werden. Diese Stubs machen eine prinzipielle Nutzung des RPCs möglich. Sie kümmern sich um das Marshalling der Daten, die über das Netz gesendet werden, entpacken diese an der Serverseite und rufen die eigenen Funktionen auf. Außerdem bestehen die Stubs aus selbstgeschriebenen Funktionen, die am Client eine identische Signatur wie lokale Funktionsaufrufe haben und am Server die Funktionsimplementierung (die Programmlogik) aufrufen. Diese Technik führt zu einer völligen Transparenz des RPC-Mechanismus. Es müssen nur Stubs mit den identischen Funktionssignaturen geschrieben werden, und schon wird für den Client unsichtbar ein entfernter Server aufgerufen. Man kann diese selbstgeschriebenen Stubs in zwei Klassen aufteilen:

- Client Stub-Stub (Client Sandwich)
- Server Stub-Stub (Server Sandwich)

Die Aufgaben des Client Stub-Stub lassen sich kurz umreißen:

1. Holen des RPC-Verbindungshandles vom Server-Administrator.
   - Der Handle ist entweder schon vorhanden oder wird mit Hilfe des Brokers neu erzeugt.
2. Übergabestruktur initialisieren.
   - Dies ist beim SUN-RPC vor allem dann wichtig wenn Strings mitübergeben werden sollen, da NULL-Pointer unzulässig sind. Ein leerer String muß immer ein Zeiger auf einen gültigen Speicherbereich mit dem Inhalt ‚\0' sein.
3. Zuweisen aller Übergabeparameter der Übergabestruktur.
   - Die Objekt-ID wird dem Aufruf mitgegeben, da über diese das eigentliche Objekt am Server identifiziert wird.
4. Client-ID vom Client-Manager (cmgr) abfragen und in die Übergabestruktur kopieren.
   - Die Client-ID ist nötig, da jeder Client eine eigene Objekttabelle mit „seinen" Objekten hält.
5. RPC – Aufruf.
   - Der Aufruf des generierten RPC-Stubs.
6. Entpacken der Rückgabeparameter.
   - Rückgabeparameter setzen und Fehlerbehandlung durchführen.

7. Freigabe des allokierten Speichers.

    - Der SUN-RPC allokiert für jede Rückgabestruktur Speicher der von Nutzer freigegeben werden muß.

8. Funktionsrückgabe

    - wenn nötig.

Das folgende Quellcodebeispiel aus knde_clnt.c soll dies verdeutlichen:

```
void   knde_init (knde * k) {
    rknd r;
    rknd * pr;
    Assert (k);

    cl = sadm_get_serverhandle (KNDE_SVC);      1
    Assert (cl);

    rknd_init (&r);          2

    r.oid = (long)  k;       3
    r.cid = cmgr_get_cid ();     4
    Assert (r.cid);

    pr = knde_init_1 (&r, cl);       5
    Assert (pr && (pr->rc == OK));          6

    xdr_free ((xdrproc_t) xdr_rknd, (char *) pr);     7
}
```

Die Server-Stub-Stubs lassen sich in folgende Aufgaben untergliedern:

1. Rückgabestruktur initialisieren.

2. Client-ID des gerade zu bearbeitenden Client am Server setzen.

   - Da ein Server wiederum Client eines anderen Servers sein kann, muß der Server die Identität des Clients annehmen. Nur so ist eine korrekte Weitergabe von Proxys möglich.

3. Objektzeiger über die Objekt-ID aus der Objekttabelle holen.

   - Über die Client-ID wird die zum Client gehörige Objekttabelle gesucht und ein Zeiger auf das zur OID passende Objekt zurückgegeben.

4. Der „richtige“ Funktionsaufruf.

   - Die lokale Funktionsimplementierung wird aufgerufen.

5. Verpacken der Rückgabewerte und des Returncodes.

6. Funktionsrückgabe der Rückgabestruktur.

Das folgende Beispiel ist aus r_knde.c entnommen:

```
rknd * knde_init_1_svc(rknd *argp, struct svc_req *rqstp) {

    static rknd  result;
    static knde * k;

    Assert (argp);

    rknd_init (&result);
    cmgr_set_cid (argp->cid);

    k = ctab_get_obj (argp->cid, argp->oid);
    if (!k) {
      result.rc = NOK_PROXY_NOT_FOUND;
      return &result;
    }
    knde_init (k);

    result.rc = OK;
    return (&result);
}
```

1
2
3
4
5
6

Das Verpacken aller Parameter in eine eigene Übergabestruktur liegt an einer Einschränkung des SUN-RPCs. Der SUN-RPC kann nur je einen Parameter für Funktionsübergabe und für Funktionsrückgabe verwalten. Ein aus der Schnittstellendatei (*.x) generierter RPC-Aufruf sieht folgendermaßen aus:

```
Rückgabeparameter = xxxx_abcd_1 (Übergabeparameter, Verbindungshandle)
```

Da sich die Übergabeparameter bei jeder Funktion ändern können, es aber nicht sinnvoll ist für jede Funktion eine eigene Übergabestruktur zu definieren, wird eine generische Übergabestruktur verwendet, die die Vereinigungsmenge aller Übergabeparameter der ADT-Funktionen darstellt.

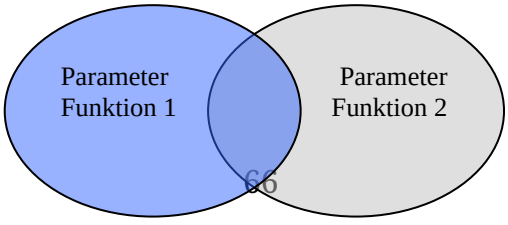


Abbildung 25 Funktionsparameter

Eine solche Übergabestruktur sieht z.B. für den ADT-„Kunde“ folgendermaßen aus:

```
/* r_knde.x (RPCL) */
/* Generische Uebergabestruktur fuer alle Funktionen */

struct rknd  {

    long   oid;                          /* Die OID des Proxys */
    string cid      <MAX_CIDLENGTH>;     /* Die ClientID CID   */

        /* Vereinigungsmenge aller Aufrufparameter */
    string name     <MAX_KUNDENNAME>;
    string vorname  <MAX_KUNDENNAME>;
    string anrede   <MAX_ANREDE>;
    long   liste_der_konten <>;
    radrs  adresse;
    long   Row-ID;
    short  rc;
};
```

Diese Form von Übergabestruktur ist bei der ROSALINDE-Anwendung eine reine Programmierkonvention und hat mit der Funktionalität nichts zu tun. Es könnte auch für jede Funktion eine eigene Übergabestruktur verwendet werden, was aber sicherlich nicht zur Lesbarkeit des Programmcodes beitragen würde.

Die Stub-Stubs der einzelnen Funktionen unterscheiden sich nur in den Parametern und Übergabestrukturen. Daher ist es möglich diese Stub-Stubs zu

generieren. Bei der ROSALINDE-Anwendung wurde zwar kein eigener Generator gebaut, es wird aber deutlich, wie ein solcher funktionieren könnte.

6.4.2 Statische und dynamische Aufrufe (dynamic invocation)

Normalerweise werden die Stubs zur Nutzung von RPC-Mechanismen statisch generiert. Diese statische Generierung ist bei Anwendungen sinnvoll, die komplett realisiert werden. Anders bei Komponentensoftware: Es ist in einer Anwendung oft nicht vorhersehbar welche Schnittstellen eine fremde Komponente unterstützt. Trotzdem ist es oft wünschenswert, Methoden dieser Schnittstelle zu Laufzeit aufrufen zu können. Als Beispiel kann eine Dokumentenanwendung dienen, die beliebige Bestandteile anderer Anwendungen in einem solchen Dokument einbetten kann. Da die Bestandteile bei der Erstellung des Dokumentenprogramms nicht bekannt sind, müssen die Aufgaben, die normalerweise von statischen Stubs übernommen werden, zur Laufzeit durchgeführt werden. Diese Technik nennt man Dynamic Invocation (Dynamische Aufrufe). Um unbekannte Schnittstellen zur Laufzeit korrekt zu bedienen, ist eine Art Schnittstellenbeschreibung nötig. Bei CORBA nennt sich diese Interface-Repository. Bei Microsoft heißt sie schlicht Type-Library (Typ-Bibliothek).

Die ROSALINDE-Anwendung kennt nur statische Stubs. Alle Stubs liegen als Quellcode vor oder übersetzt in Bibliotheken. Daher kann die ROSALINDE-Anwendung keine dynamischen Aufrufe an Server mit zur Compilezeit unbekannten Schnittstellen erzeugen.

Diese Technik ist eine wesentliche Komponente von verteilten Systemen. CORBA wie auch DCOM unterstützen dynamische Aufrufe. Im ROSALINDE-Projektes

waren wir aber nicht gezwungen, solche Mechnismen zu implementieren, da die Schnittstellen von vornherein feststanden.

6.5 Persistenz

Um keine Datenbankmechanismen (z.B. Sperren von Objekten innerhalb des Servers) realisieren zu müssen, ist es sinnvoll, daß jeder Client seine eigenen Objekte am Server hält. Auch wenn zwei Clients den gleichen Kunden aus der Datenbank lesen, sind dies zwei unterschiedliche Objekte im Speicher mit unterschiedlichen OIDs. Erst beim Zurückschreiben in die Datenbank werden eventuelle Inkonsistenzen per Änderungszähler erkannt (nach dem Motto „Der erste gewinnt !“). Diese „optimistische“ Transaktionslogik ist sehr sinnvoll im Hinblick auf lange Transaktionen, da die Datenbanksätze nicht über lange Zeiträume gesperrt werden. Sie hat aber den Nachteil, daß eine Änderung im Konfliktfall zurückgenommen werden muß (rollback). Es hängt von der Anwendung ab, ob dies vertretbar ist. Allgemein kann man jedoch sagen, daß jede GUI-Anwendung, Objekte, die gerade angezeigt werden, nicht sperren darf, da häufig Objekte gesperrt würden, welche gar nicht verändert werden. In Verbindung mit einem „Rattenschwanzlesen“ kommt überhaupt nur eine optimistische Transaktionslogik für GUI-Anwendungen in Frage, da ansonsten neben dem zu bearbeitenden Objekt, auch alle referenzierten Objekte mitgeladen und gesperrt würden.

Das folgende Beispiel aus der Update-Funktion des ADT-Kunde (`knde_update()`) zeigt, daß der Änderungszähler (change_cnt) des aktuellen Kunden mit dem entsprechenden Satz in der Datenbank verglichen wird, womit sichergestellt ist, daß noch kein anderer Client diesen Kunden verändert hat. Die nötige Sperre dieses Satzes umfaßt nur den Bereich vor dem Lesen (reread) bis zum endgültigen Zurückschreiben mit aktualisiertem Änderungszähler.

```
// Ausschnitt aus knde.c
LOCK (k); // noch nicht implementiert
knde_reread (&dbk, key);
if (dbk.persistent.change_cnt > k->persistent.change_cnt) {
      /* ein anderer Client hat den Satz schon veraendert */
      UNLOCK (k);
      throw (DB_CHANGED);
}
k->peristent.change_cnt ++;
/* Hier ist alles OK ... UPDATE() */
UNLOCK (k); // noch nicht implementiert
```

Die Objektidentität der Anwendungsobjekte eines Clients am Server wird über die jeweilige Objekttabelle gewährleistet. So wird bei der Suche nach einem Kunden, zuerst die OTAB und erst anschließend die Datenbank durchsucht. Die ROSALINDE-Anwendung enthält in der aktuellen Version noch keinen Referenzzähler (Referencecounter) für Objekte, was aber in der Weiterentwicklung noch zu beachten ist. So kann es aufgrund der Tatsache, daß kein Referenzzähler vorhanden ist, zur Zeit noch am Client zu folgendem Fehler kommen:

```
k1 = kunde_find_by_key („12345“);
   Der Kunde mit der Nr. „12345“ wird am Server aus der DB gelesen.
k2 = kunde_find_by_key („12345“);
   Der Kunde „12345“ wird in der Server-OTAB gefunden ...
knde_delete (k1);
   loescht über den Proxy (k1) das Serverobjekt.
knde_delete (k2);
```

Damit wird das Objekt am Server nicht mehr gefunden - Der Proxy ist ungültig.

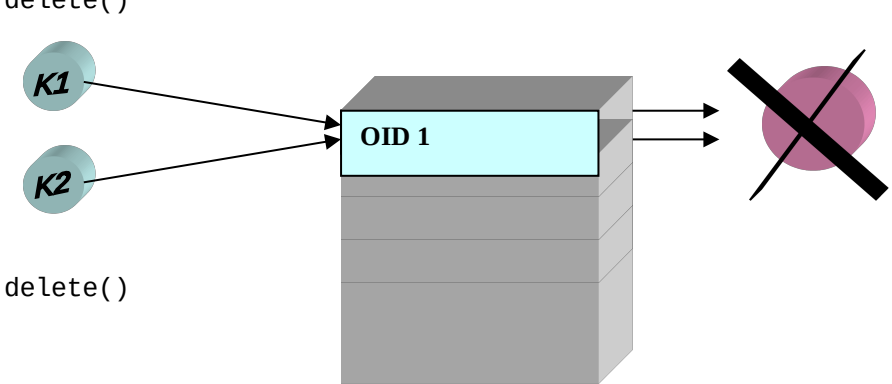


Abbildung 26 Referenzzähler

7 Client/Server-Probleme

7.1 Servertypen und Deadlocks

7.1.1 Quasi-Parallele (single-thread) Server

Beim ONC-RPC wird mit Hilfe des Generatorprogramms „`rpcgen`“ das Serverhauptprogramm generiert. Als der ONC-RPC entwickelt wurde, waren POSIX-Threads und multithread-fähige Laufzeitbibliotheken noch nicht umfassend vorhanden; deshalb entschieden sich die Designer des ONC-RPCs für eine sequentielle Server-Architektur; d.h. ein RPC-Aufruf eines Clients wird vollständig abgearbeitet und erst dann kommt der nächste Client zum Zug. Heutzutage ist es zwar möglich einen multi-thread-fähigen Server mit dem ONC-RPC und Posix-Threads zu bauen, es gibt aber kein Werkzeug um dies zu automatisieren. Sequentielle Server haben den großen Vorteil, daß man keinerlei Sychronisationsmechanismen wie Semaphore etc. benötigt, da jeder Funktionsaufruf vollständig abgearbeitet wird. Allerdings sinkt der Durchsatz, da eine lange Abarbeitungsdauer einer Serverfunktion den Server vollständig lahm legt. Man denke nur an eine Datenbankabfrage die 1Mio Kunden bearbeitet. Das Hauptproblem von sequentiellen Servern ist jedoch das Auftreten von Deadlocks (Verklemmungen). Das klassische Problem der Prozeßverklemmung tritt auf, sobald der Prozeßbelegungsgraph Zyklen enthält. [27]

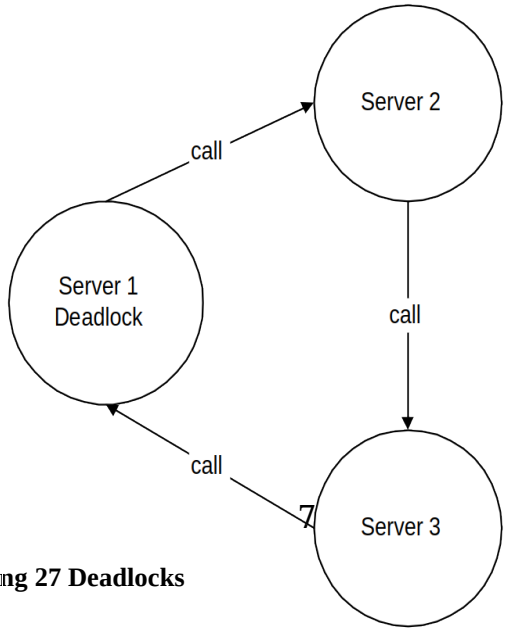


Abbildung 27 Deadlocks

Bei der ROSALINDE-Anwendung können über einen Kunden die zugehörigen Konten gesucht werden, indem am Kundenserver die Funktion `knde_get_konten()` aufgerufen wird. Diese Funktion wiederum läßt sich vom Kontenserver eine Liste der Konten des aktuellen Kunden erstellen. Das Problem hier ist, da der Kontenserver ja nur einen Proxy auf den Kunden am Kundenserver besitzt muß er wiederum Funktionen des Kundenservers aufrufen um auf die gekapselten Attribute wie beispielsweise die Row-ID (rid) des Kunden zuzugreifen. Hier entsteht ein Deadlock. Dieses Problem ist nicht trivial und wird auch von der ROSALINDE-Anwendung nur unbefriedigend gelöst, da dies prinzipiell bei sequentiellen Servern auftreten kann. Die ROSALINDE-Anwendung benützt in den Fällen wo ein Zyklus auftritt, Funktionsaufrufe bei denen alle nötigen Parameter mitübergegeben werden um Rückrufe zu vermeiden. Dies steht natürlich im krassen Widerspruch zum Proxykonzept da Objektdaten nicht am Client gehalten werden sollten. Da es aber keine andere Lösung für dieses Problem gibt hat es dazu geführt, daß bei modernen RPCs bzw. bei CORBA der Server als paralleler, multi-thread-fähiger Server realisiert ist.

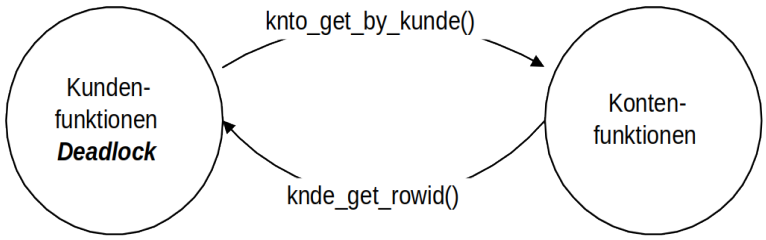


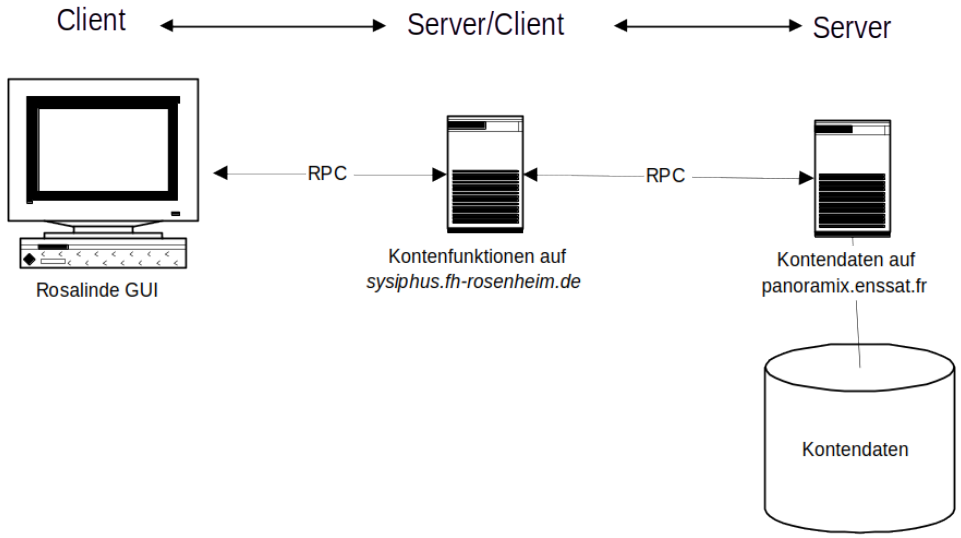
Abbildung 28 Deadlocksituation: Kunde-Konto

7.1.2 Parallele (multi-thread) Server

Mit den meisten modernen Middlewareprodukten (DCE, CORBA) lassen sich multi-thread-fähige Server bauen. Das bedeutet, daß pro Client ein oder mehrere Threads ausgeführt werden. Die Probleme sind aber auch hier nicht gelöst. Da bei einem realen Server Synchronisationsmechanismen (Semaphoren) auf geteilte Resourcen (Dateien u.ä.) gesetzt werden müssen, können auch hier Deadlocks auftreten. Diese Deadlocks sind zumeist viel tückischer und schwerer zu finden als bei einer sequentiellen Server-Architektur, da diese nur in einer genau festgelegten, zeitlichen Abfolge entstehen können.

7.2 Client oder Server ?

Bisher wurde immer selbstverständlich zwischen Client und Server unterschieden. So trivial wie man erst meint ist das leider nicht, das Problem dabei ist, daß diese Rollenverteilung nicht eindeutig ist. Prozesse die Dienste anderer in Anspruch nehmen, können ebenfalls Dienste bereitstellen. Damit agiert der Prozeß als Client und als Server. Die folgende Abbildung macht dies



deutlich.

Der Kontenfunktionenserver ist als Server ebenfalls Client des Kontendatenservers. Dadurch kann es Probleme geben mit der Eindeutigkeit der Zugehörigkeit von Objekten über die Client-ID. Gehören die Objekte am Kontendatenserver dem Kontenfunktionenserver oder dem Client? (ROSALINDE GUI) Die ROSALINDE-Anwendung löst diese Probleme nach der folgenden Regel: Client-IDs werden bei Objekten vom Server weitergereicht, wenn dieses Objekt kein lokales Objekt des Servers ist. Damit ist es möglich über mehrere Server hinweg die korrekte Zuordnung zum richtigen Client (Master) zu erhalten und

Abbildung 29 Client vs. Server

trotzdem lokale Objekte am Server unabhängig vom gerade verbunden Client zu erzeugen. Die folgende Situation bei der ROSALINDE-Anwendung soll dies verdeutlichen:

Kunden können eine Liste von zugehörigen Konten haben. Die Kunden werden vom Kundenfunktionenserver bearbeitet wohingegen die Konten vom Kontenfunktionenserver bearbeitet werden. Da ein Client einen neuen Kunden bzw. Konten anlegen und einem Kunden zuordnen kann, gehören alle beteiligten Objekte dem Client. Somit muß der Kundenserver die Client-ID an den Kontenserver weiterreichen, sonst könnte der Kundenserver nicht auf diese Konten zugreifen. Wenn aber diese Client-ID auch beim Zugriff auf die Basistabelle weitergereicht wird, tritt ein Fehler auf, da das Basistabellenobjekt vom Server angelegt worden ist, und somit nur über die Client-ID des Servers ansprechbar ist (lokales Objekt).

Die ROSALINDE-Anwendung löst diese Problematik leider nur recht unbefriedigend, da neben der Client-ID des verbundenen Clients, zusätzlich eine Server-ID (SID) gespeichert wird, die in der BTAB-Zugriffsschicht anstelle der Client-ID bei BTAB-Aufrufen mitgegeben wird.

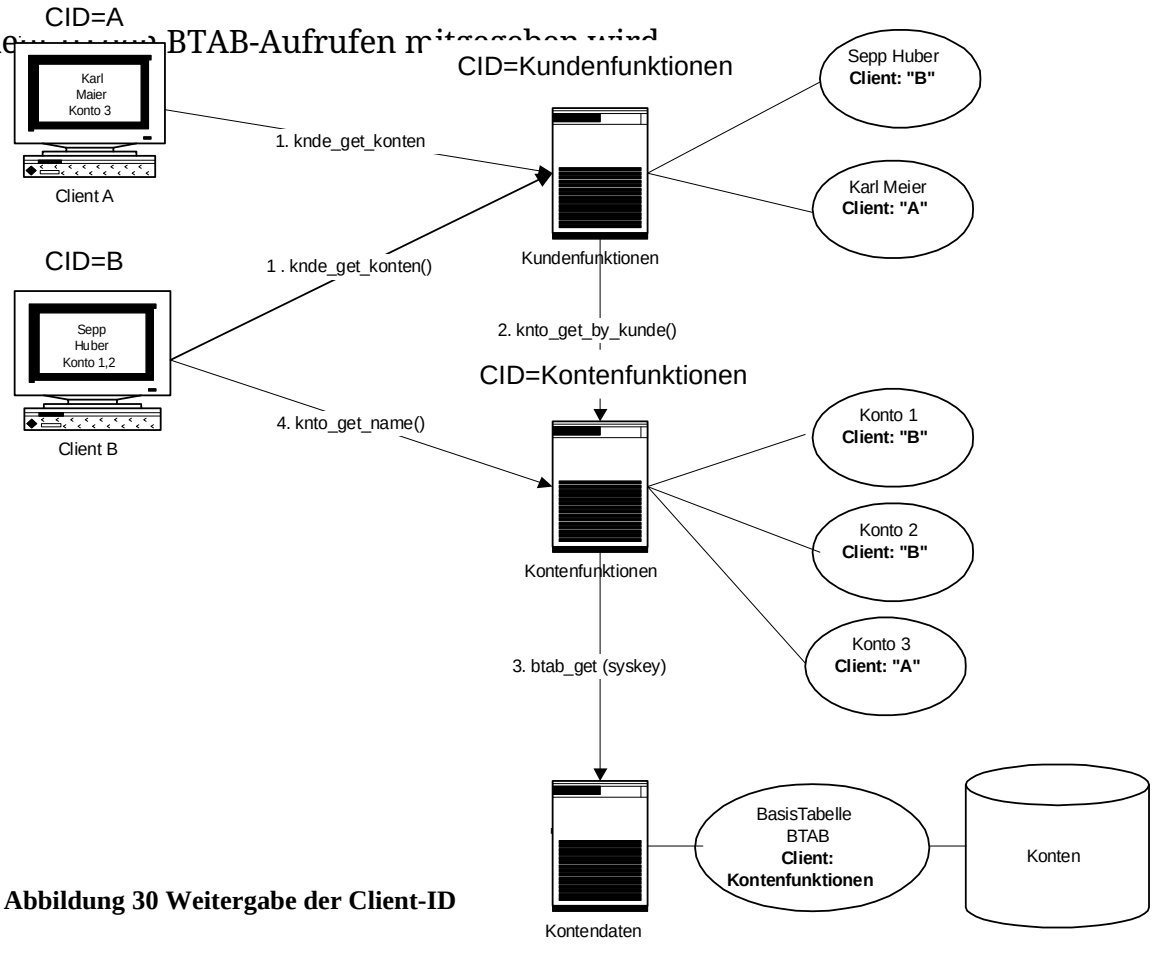


Abbildung 30 Weitergabe der Client-ID

Abbildung 30 verdeutlicht noch einmal die Situation: Ein Client benötigt alle Konten eines Kunden und ruft hierfür die Funktion „`knde_get_konten()`“ am Kundenserver auf. Der Kundenserver muß jedoch erst die Liste alle Konten eines Kunden vom Kontenserver erstellen lassen. Dazu ruft dieser die Funktion „`knto_get_by_kunde()`“ auf, die eine Liste von Kontenproxys zurückliefert. Das Zurückliefern der Proxys setzt voraus, daß die entsprechenden Kontenobjekte am Kontenserver erzeugt und in die Objekttabelle des Clients (Clienttabelle) eingetragen wurden. Der Client muß jetzt in der Lage sein über den Kontenproxy auf die Daten des Kontos zuzugreifen. Dies ist allerdings nur dann möglich, wenn die Konten auch für diesen Client erzeugt wurden. Deshalb muß der Kundenserver die Client-ID für den Aufruf „`knto_get_by_knde()`“ weiterreichen.

Sogar Microsoft kämpft mit dieser Problematik: *„ImpersonateClient (DCOM) ermöglicht es dem Objekt oder einem Thread die Identität eines Clients anzunehmen. Manchmal kann das Objekt sogar im Namen des Client Anforderungen an andere entfernte Objekte senden, was man als delegieren*

bezeichnet. (Das Delegieren wird jedoch in der ersten Version von DCOM noch nicht unterstützt.)“ (David Chappel) [4]

7.3 Verteilte Transaktionen

Im ROSALINDE-Projekt steht man vor der typischen Situation der Transaktionssicherung bei verteilten Datenbanken. Dies kommt daher, da der Kunden-Datenserver und der Konten-Datenserver als zwei getrennte Prozesse realisiert sind, die überdies auf zwei physikalisch unterschiedliche Datenbestände zugreifen. Wenn ein Kunde geändert wird und zugleich ein zugehöriges Konto, dann ist nicht sichergestellt, daß beide commit()-Aufrufe an den Servern korrekt durchgeführt werden. Die Server haben zwar intern die Möglichkeit ihre eigene Transaktion zurückzusetzen (rollback) können dies aber momentan nicht dem anderen mitteilen. Streng genommen verfügt die ROSALINDE-Anwendung über keinen Rollback-Mechanismus, da der Rollback der OTAB nur Objekte die bisher verändert und nicht gespeichert wurden, erneut aus der Datenbank liest. Diese Problematik ist nicht trivial und wird normalerweise von einer Datenbank und einem Transaktionsmonitor erledigt. Produkte wie der Transaction Server von Microsoft versuchen verteilte Transaktionen bei Komponentensystemen korrekt zu behandeln, da dort oftmals auch die Daten verteilt sind.

8 Die Software

8.1 Programmstart

Im ersten Schritt wird die Software lokal gestartet (alle beteiligten Prozesse). Die Kommunikation zwischen den Servern findet grundsätzlich auch am lokalen Rechner mit RPC statt. In diesem Fall wird mittels *loopback device*. (127.0.0.1) kommuniziert. Im Verzeichnis `./Rosalinde/bin` befindet sich ein Unix-Shellscript mit dem Namen `./Rosalinde`, das alle Server und das GUI auf dem lokalen Rechner startet. Dieses Skript startet zuerst den Broker und nacheinander die Anwendungsserver. Die Ausgaben des Brokers sind im Skriptfenster zu sehen. Wie Abbildung 31 zeigt, werden die Server gestartet und registrieren sich nacheinander am Broker. Falls als Brokerhost ein entfernter Rechner angegeben wird, erscheinen die Registrierungsmeldungen im Skriptfenster des entfernten Rechners, da sich die Server an dessen Broker registrieren. Das ROSALINDE-Skript kann auf mehreren Rechnern gleichzeitig gestartet werden, es ist nur wichtig, daß der Brokerhost für alle Server gleich ist; d.h. immer der gleiche Rechnername bzw. dessen IP-Adresse angegeben wird.

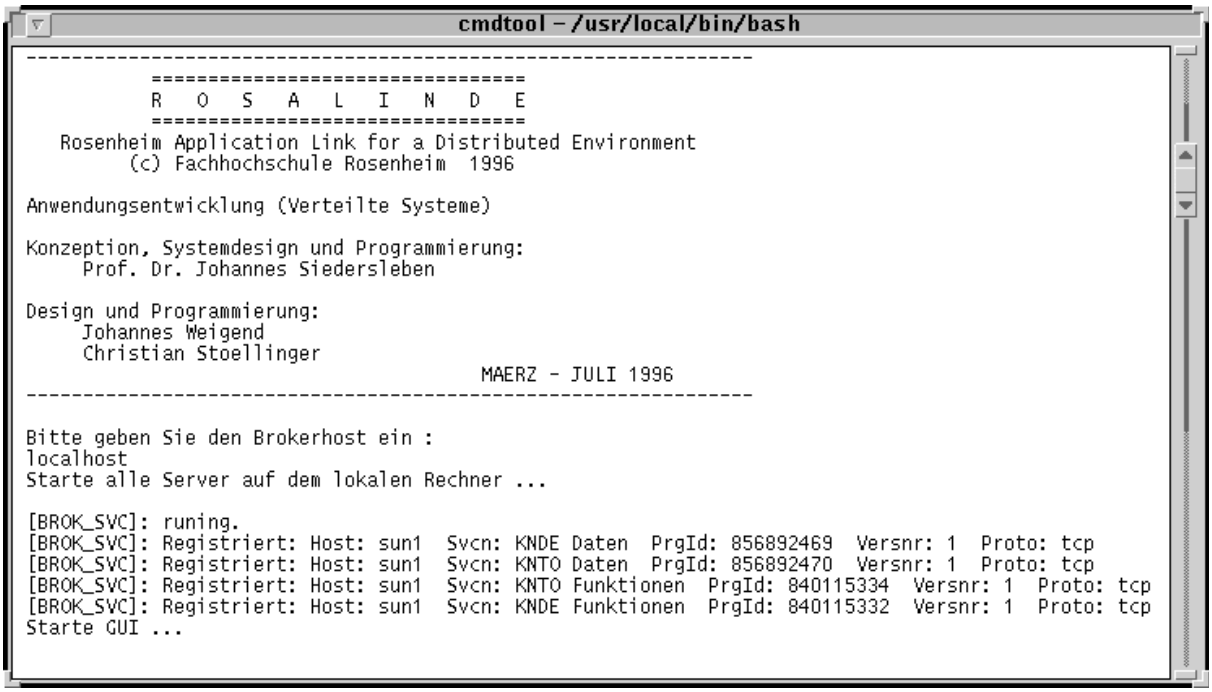


Abbildung 31 ROSALINDE Programmstart

Die Protokollausgaben der verschiedenen Server werden in dem Skript in Logdateien (*.log) umgeleitet. Zu Debug- und Demonstrationszwecken ist es aber oftmals sinnvoll, die Server in verschiedenen Fenstern per Hand zu starten. Im Verzeichnis ./ROSALINDE/exe befinden sich folgende Programme die auch jeweils einzeln gestartet werden können:

Tabelle 3 Programmdateien

Dateiname	Funktion
r_brok_svc	Broker
r_knde_svc	Kunden-Funktionen-Server
r_knto_svc	Konten-Funktionen-Server
r_btab_knde_svc	Kunden-Daten-Server
r_btab_knto_svc	Konten-Daten-Server
../src/rosa/rosa_rpc.tk	Client (GUI)

Beim Starten der Server per Hand ist zu beachten, daß grundsätzlich der Broker als erster Server im Netzwerk verfügbar sein muß, da sich alle anderen Server über diesen verbinden. Außerdem benötigen alle Server, mit Ausnahme des Brokers, als ersten Parameter die Angabe des Brokerhosts:

Beispiel:

```
$r_knde_svc nexus.inf.fh-rosenheim.de
```

8.2 ROSALINDE-GUI

Nach Starten aller Server erscheint schließlich das ROSALINDE-GUI Hauptfenster, indem sich über die Menüleiste verschiedene Funktionen per Maus auswählen lassen.

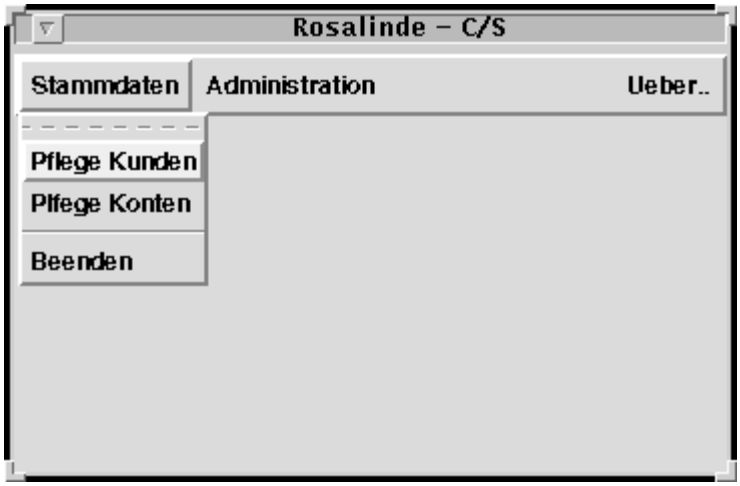


Abbildung 32 ROSALINDE GUI

Über den Menüpunkt „Stammdaten“ können verschiedene Funktionen des Anwendungsbeispiels ausgeführt werden. „Pflege Kunden“ umfaßt die Neuanlage und Bearbeitung von Kunden bzw. die Suche von Kundendaten. Ebenso können einem Kunden ein oder mehrere Konten mit diesem Dialog zugeordnet werden. Mit „Pflege Konten“ können neue Konten angelegt, vorhandene gesucht und auch geändert werden.

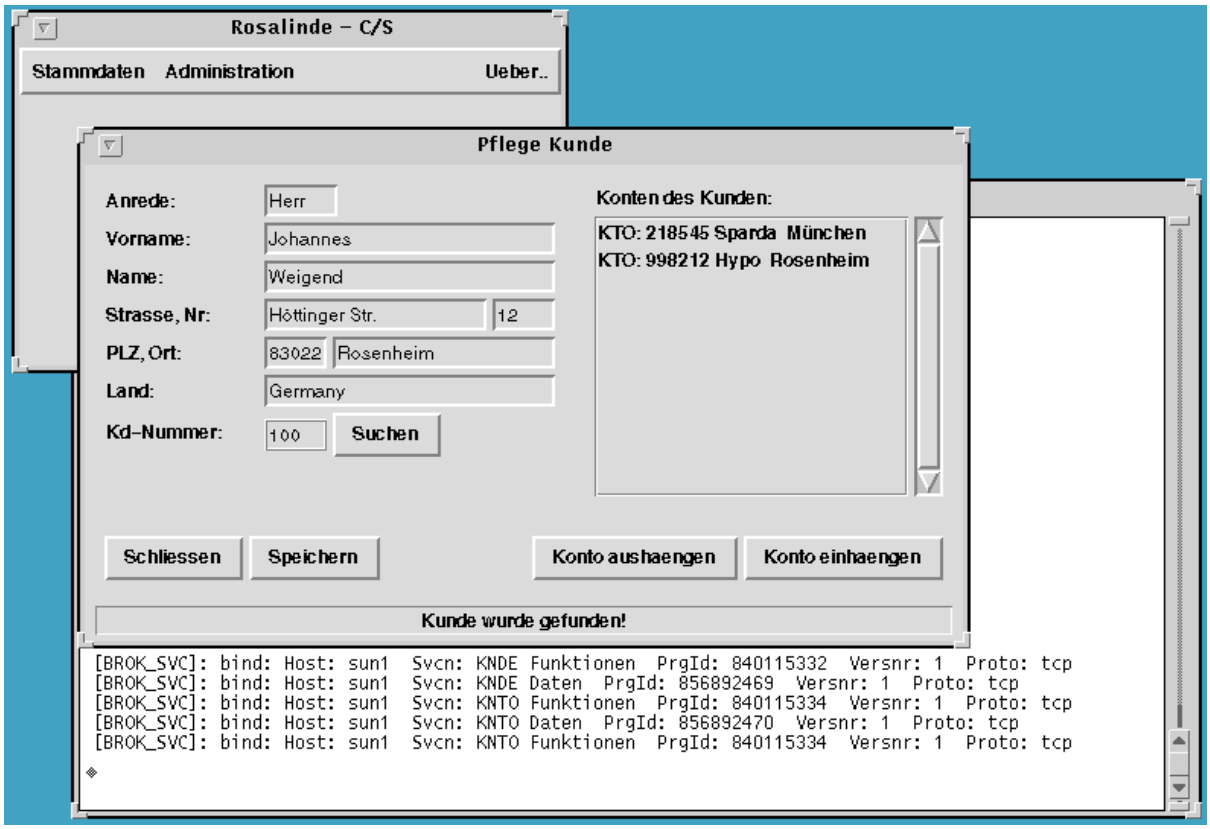


Abbildung 33 Kundensuche

Abbildung 33 zeigt den „Pflege Kunde“-Dialog nach erfolgreicher Suche nach einem Kunden mit der Kundennummer 100. Das Logfenster im Hintergrund zeigt, daß fünf mal ein „bind“ am Broker stattgefunden hat. Dies verursacht zum einen der Client, der Kunden und Kontenfunktionen benötigt und deshalb den Broker kontaktiert. Der Kunden-Funktionen-Server benötigt eine Verbindung zum Kunden-Daten-Server um persistente Objekte aus der Datenbank (BTAB) zu lesen und zu speichern, ebenso der Konten-Funktionen-Server. Der letzte bind() auf den Konten-Funktionen-Server wird schließlich vom Kunden-Funktionen-Server ausgeführt, da der Kunden-Funktionen-Server die Liste der Konten benötigt. (`knde_get_konten()`). Da die ROSALINDE-Zugriffsschicht („Server-Administrator“) diese RPC-Bindinghandles zwischenspeichert finden normalerweise keine weiteren bind() – Aufrufe statt.

Abbildung 34 Suche nach unzugeordneten Konten eines Kunden

Durch einen Mausklick auf den Button „Konto einhängen“ erscheint eine Liste aller noch keinem Kunden zugeordneten Konten. Jetzt kann ein Konto selektiert werden, und mit „Uebernehmen“ dem Kunden zugeordnet werden. Dannach

schließt das „Suche Konto“ Dialogfenster und das Konto erscheint in der Liste der Konten des aktuellen Kunden

Ebenso können Konten neuangelegt, gesucht und geändert werden. Dieses passiert mit dem „Pflege Konto“ Dialog über den Menüpunkt „Pflege Konto“. Konten können über die eindeutige Konto-ID gesucht und anschließend geändert werden.

Abbildung 35 Kontensuche

Nach einer erfolgreichen Suche erscheinen die Kontendaten im Dialog. Abbildung 34 zeigt die Suche nach einem Konto mit der ID 10001 (Konto-IDs sind in der ROSALINDE-Anwendung immer größer gleich 10000).

Die bisher betrachteten Dialoge sollen ein einfaches Anwendungsbeispiel auf fachlicher Ebene abdecken. Die Kernfunktionalität der ROSALINDE-Anwendung (transparenter Netzwerkzugriff über einen Broker) kann im folgenden Dialog getestet werden: Im Dialog „Server Administration“ können zur Laufzeit Server am Broker registriert bzw. deregistriert werden. Voraussetzung ist, daß diese Server bereits gestartet sind. Durch Änderung von „Host:“ kann für einen Dienst (z.B. Kunde Funktionen) der Eintrag am Broker geändert werden. Sollte sich an dem neuangegebenen Rechner kein gültiger Server befinden lehnt die Dialogoberfläche die Änderung mit der Meldung „[sadm]: Ping Failed“ ab. Hier sind verschiedene Vorführszenarien denkbar. Wir haben für die 10-Jahresfeier des Fachbereiches Informatik an der FH-Rosenheim die Server auf vier verschiedenen Rechnern in Rosenheim und Frankreich laufen lassen. Mit dem Dialog konnte der Standort des Datenservers ausgewählt werden. Damit wurde festgelegt ob französische oder deutsche Kunden bzw. Konten in den Dialogen erscheinen.

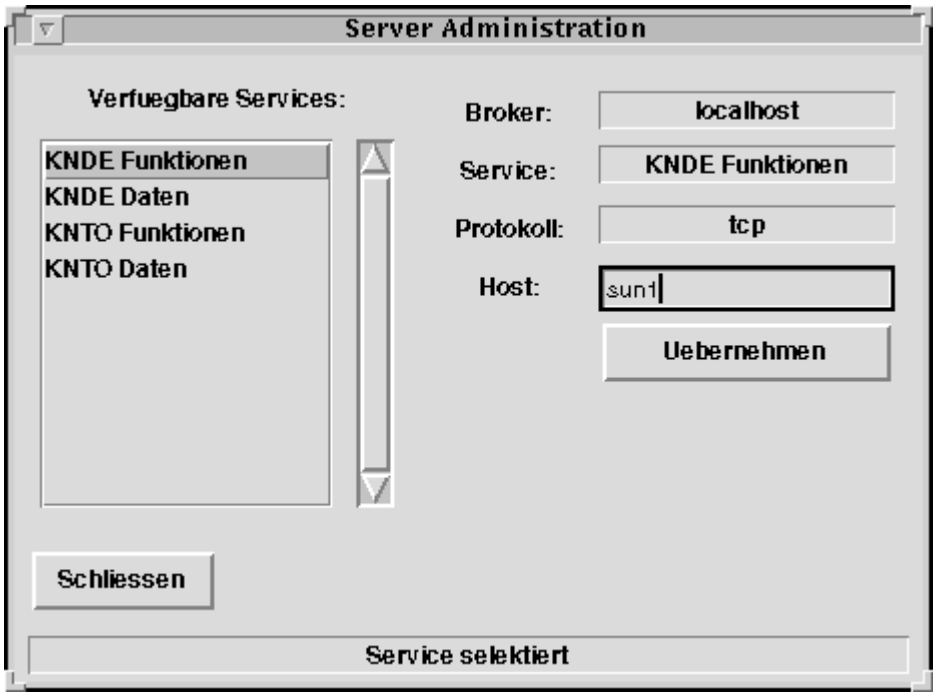


Abbildung 36 Änderung der Bindungsinformation

Nach dem Ändern des Hosteintrages und Betätigen von „Uebernehmen“ wird am Broker der vorhandene Eintrag auf einen neuen Host geändert und die Server initialisieren ihre RPC-Verbindungshandles neu. Dies kann sehr gut in Abbildung 37 nachvollzogen werden.

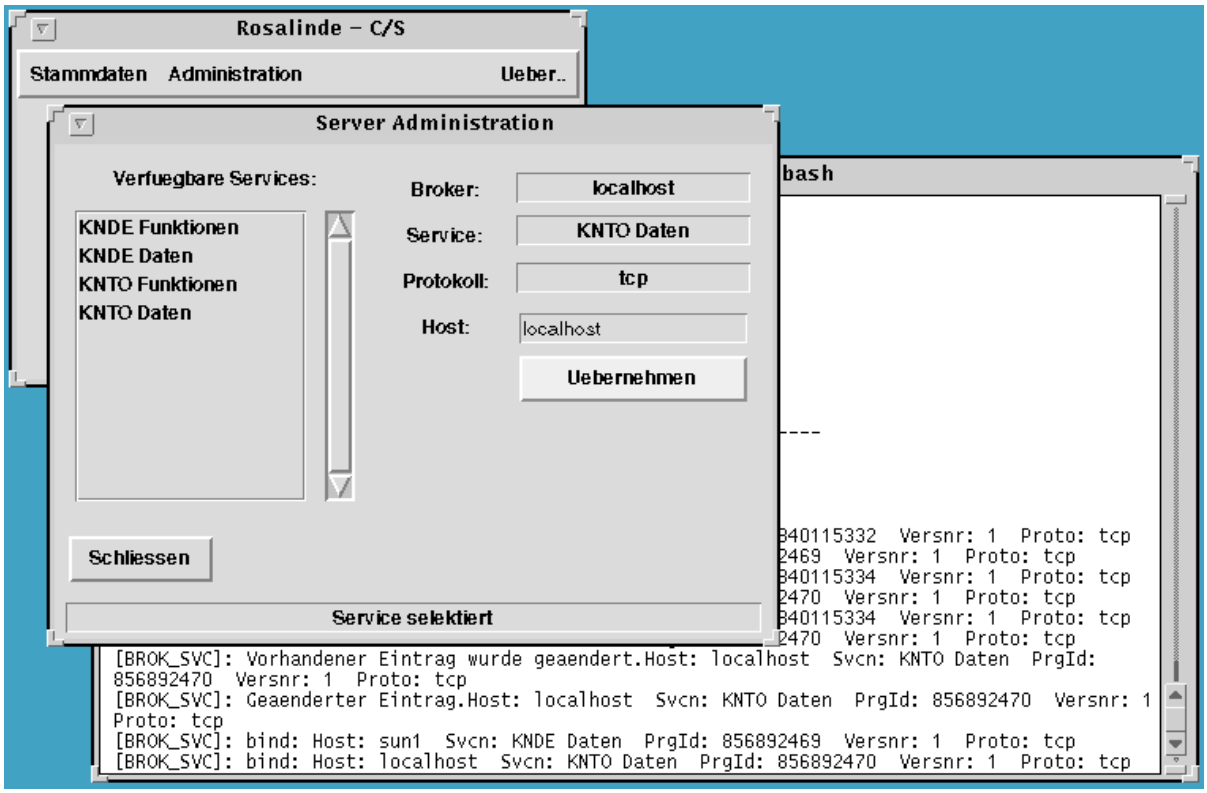


Abbildung 37 Änderung eines Servers

Das Logfenster im Hintergrund zeigt die durchgeführten Aktionen bei einer erfolgreichen Änderung des Hosts für einen Dienst. Zuerst wird der Eintrag am Broker geändert. Da automatisch nach dem Start der ROSALINDE-Anwendung alle Dienste am Broker registriert werden, wird der vorhandene geändert. Die beiden nächsten Meldungen stammen von den Funktionen-Servern, die wie schon besprochen, ebenfalls als Client agieren und über einen Broadcast die Meldung bekommen, daß ihre Datenbankverbindungsproxys ungültig sind und neu aufgebaut werden müssen.

Um ein Gefühl für die Abläufe zu bekommen, ist es auf jeden Fall empfehlenswert, alle Server per Hand in eigenen Fenstern zu starten. So kann

man anhand der reichlich erzeugten Protokoll-Information die genauen Abläufe studieren.

9 Implementierung

9.1 DV-Konzept

Wie in jedem größeren Projekt war das Vorhandensein eines DV-Konzeptes für uns eine große Erleichterung. Ohne ein solches wäre das Projekt nicht realisierbar gewesen. Das DV-Konzept ist im wesentlichen eine Erweiterung des von Prof. Dr. Siedersleben in Anwendungsentwicklung I und III eingeführten DV-Konzeptes. Es werden hier nur kurz die wichtigsten Merkmale genannt.

Im folgenden wird für Übergabeparameter von Funktionen eine, an die gängigen IDL-Sprachen angelehnte, Syntax verwendet:

[IN]: Parameter dient nur als Eingabeparameter und wird von der Funktion nicht verändert. Der Parameter kann ein Wert sein oder aber auch ein Zeiger (CALL-BY-VALUE).

[OUT]: Parameter wird von der Funktion als Ausgabeparameter gesetzt. Dieser Parameter muß ein Zeiger sein um ein CALL-BY-REFERENCE zu ermöglichen.

[INOUT]: Parameter wird sowohl von der Funktion gelesen als auch über die Referenz geschrieben. Der Parameter muß ebenfalls ein Zeiger sein um ein Rückschreiben zu ermöglichen.

Obwohl die ROSALINDE-Headerdateien diese Makros nicht verwenden, wäre eine folgende Definition sinnvoll, da sich über die optionalen Angaben von [IN], [OUT], [INOUT] die in C üblichen Schnittstellenmißverständnisse beseitigen lassen.

```
#define IN      /* nichts */
#define OUT     /* nichts */
#define INOUT   /* nichts */
```

Da der C-Präprozessor leider nicht in der Lage ist, Namen welche eckige Klammern enthalten, zu ersetzen, kann dies mit einem einfachen UNIX-sed Kommando erreicht werden und vor der Compilierung erfolgen:

```
sed s/\[IN\]//p    input.c > output1.c
sed s/\[OUT\]//p   output1.c > output2.c
sed s/\[INOUT\]//p output2.c > output3.c
```

9.1.1 Abstrakt Datentypen

Abstrakte Datentypen werden grundsätzlich als Strukturen realisiert, die mittels typedef Anweisung wie eingebaute Typen verwendet werden können. Die Strukturdeklaration erfolgt in der C-Headerdatei (xxxx.h).

```
typedef struct xxxx_tag {
      yyyy * y;               Zeiger auf yyyy (knde, knto ...)
      zzzz z;                 Wert vom Typ zzzz (float, int, ...)
      ...
} xxxx;                       ADT xxxx
```

Parameterübergabe von ADTs findet nur per Referenz (Zeiger auf ADT) statt. Der erste Parameter einer Funktion (Methode) ist immer ein Zeiger auf das angesprochene Objekt. Die einzige Ausnahme ist die Methode `xxxx_new()` die ein Objekt erzeugt und den Zeiger auf das neue Objekt als Funktionsrückgabewert übergibt.

Alle abstrakten Datentypen werden durch 4-stellige Kürzel in Kleinbuchstaben abgekürzt z.B. Kunde = knde, Konto = knto. Jeder abstrakte Datentyp verfügt über folgende Methoden:

```
xxxx * xxxx_new ()                    Speicherallokation+Initialisierung
void   xxxx_init ([INOUT] xxxx *)     Initialisierung
void   xxxx_clear ([INOUT] xxxx *)    Deallokation von Referenzen innerhalb
void   xxxx_delete ([INOUT] xxxx *)   Deallokation Struktur + Referenzen
```

wobei `new()` ein `init()` impliziert und `delete()` ein `clear()`. Dieses Vorgehen ist identisch mit dem Konstruktor / Destruktormechanismus in C++. Wobei `new()` und `delete()` vom Compiler generiert werden, und `init()` bzw. `clear()` dem

selbstgeschriebenen Konstruktor bzw. Destruktor entsprechen. Daraus folgt, daß Objekte die mit `xxxx_new()` erzeugt wurden, immer automatisch initialisiert sind. Nur bei Stackobjekten die automatisch erzeugt werden ist ein manueller Aufruf von `xxxx_init()` nötig. Eine Ausnahme bilden sogenannte Singleton Objekte [6]. Dies sind Objekte von denen nur genau eine Instanz vorliegen darf. Als Beispiel kann hier die Clienttabelle (ctab) eines Servers dienen, da es pro Server nur genau eine Clienttabelle geben darf. Bei der ROSALINDE-Anwendung werden Singleton Objekte als statische Variablen realisiert (von außen unsichtbar) die implizit von den Methoden referenziert werden. Daher verfügen Singleton Objekte nicht über `new()`und `delete()`, außerdem wird der erste Parameter der das Objekt referenziert weggelassen.


```
void   xxxx_init ()          Initialisierung
void   xxxx_clear ()         Deallokation von Referenzen innerhalb
```

Eine weitere Möglichkeit Singleton Objekte in C zu erzeugen findet sich im Kapitel 9.11.2. Beispiele für Singleton Objekte finden sich in Kapitel 9.7.4.1 (ctab), Kaptitel 9.7.3 (sadm), Kapitel 9.7.2.1 (brok).

Alle in Hashtabellen speicherbaren Datentypen müssen folgende Methoden besitzen:

```
int     xxxx_cmp ([IN] const void *, [IN] const void *);
size_t  xxxx_hash_fnk ([IN] const void *);
```

Zusätzlich verfügen viele abstrakte Datentypen über eine Kopierfunktion:

```
void    xxxx_copy ([IN] xxxx * source, [OUT] xxxx * target);
```

Hier ist zu beachten, daß man zwischen „shallow“ und „deep“ Kopierfunktionen unterscheiden muß. „Shallow“ Kopien sind einfache Strukturkopien die durch Zuweisungen entstehen können. „Deep“ Kopien verdoppeln auch alle ihre referenzierten Objekte. Die Kopierfunktion erzeugt im Normalfall eine „deep“ Kopie ihres Objektes. Die Parameterreihenfolge bei der Kopierfunktion ist genau invers ist zur C üblichen (Vgl. `strcpy()`). Würde man die Parameterreihenfolge den ANSI-C-Funktionen (strcpy, memcpy...) angleichen, hätte man eine Inkonsistenz zur Regel: „Der erste Parameter ist immer der Zeiger auf das Objekt.“ Die ANSI-C Laufzeit Bibliothek ist leider in vielen Punkten inkonsistent, so daß eine eigene Kapselung aufgrund eines durchgängigen DV-Konzeptes sinnvoll erscheint. Als Beispiel sollen hier die Funktionen fprintf und fputc genannt werden, die den stream-Zeiger an unterschiedlicher Stelle erwarten:

```
int fprintf (FILE * stream, char * s, ...);
int fputc (int c, FILE * stream);
```

Dieses Beispiel zeigt sehr deutlich, daß ein akribisches Festhalten an von außen vorgegebenen Konventionen für eigene Softwareprojekte nicht sinnvoll ist, auch wenn diese von einer hohen Institution wie dem ANSI-Konsortium stammen. [10]

Alle persistente Datentypen haben zusätzlich noch folgende Methoden:

```
void   xxxx_db_in ([IN] xxxx *, [INOUT] strg *)
aus Datenbankstring füllen

void   xxxx_db_out ([INOUT] xxxx *, [INOUT] strg *)
Datenbankstring erzeugen
```

Das DV-Konzept geht davon aus, daß in der Datenbank (BTAB) nur Strings gespeichert werden. Daher benötigt jedes Attribut eines ADTs die DB-

Funktionen zum rekursiven Aufbau des Datenbankstrings (db_in) bzw. rekursiven Aufbau der internen Attribute aus einem String (db_out).

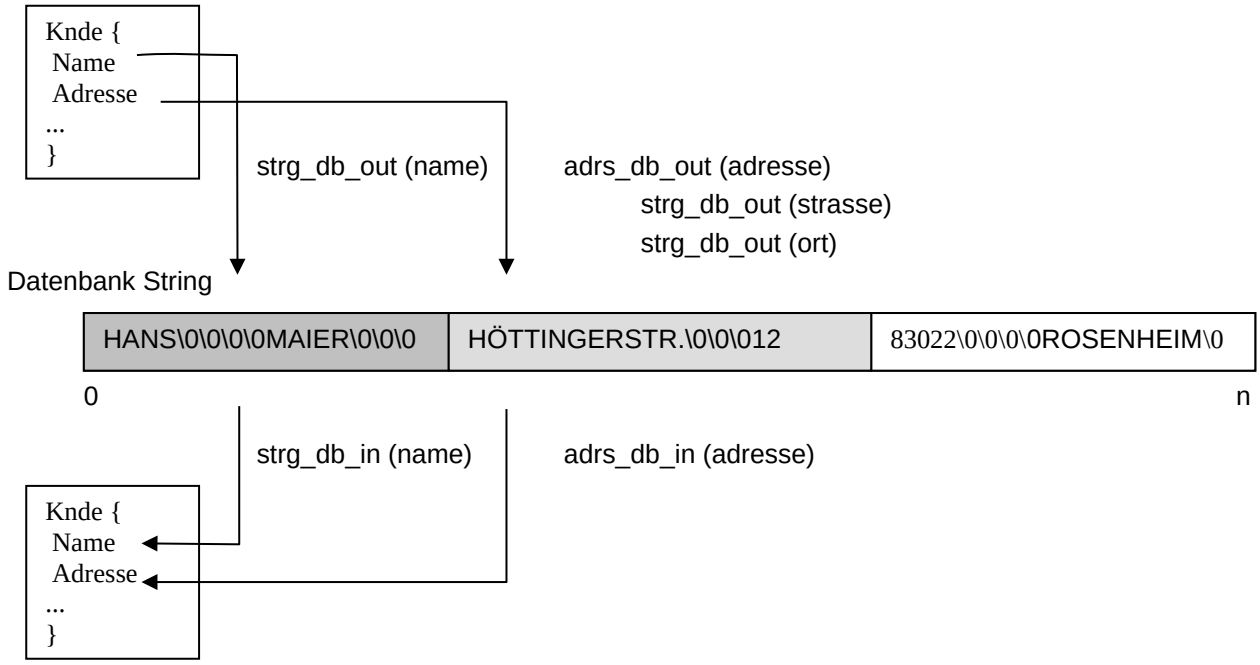


Abbildung 38 Erzeugung von Datenbankstrings (db_in/db_out)

Zusätzlich haben persistente, autonome Objekttypen folgende Datenbank-Methoden:

```
void   xxxx_insert ([IN] xxxx *, [OUT] long * Row-ID)
void   xxxx_update ([IN] xxxx *, [IN] long Row-ID)
void   xxxx_reread ([IN] xxxx *, [IN] long Row-ID)
xxxx * xxxx_find_by_key ([IN] xxxx *, [OUT] RC * rc, [IN] long Row-ID)
```

„Insert“ fügt einen neuen Satz am Dateiende der Basistabelle an. Dieser Satz ist die Stringdarstellung des ADTs xxxx (Vgl. db_in()). „Update“ aktualisiert einen Satz über die zugehörige Row-ID. „Reread“ liest ein bereits geändertes Objekt erneut aus der Basistabelle während des Rollbacks der Objekttabelle (otab_rollback). „FindByKey“ ermöglicht die Suche über eine Row-ID.

Um flüchtige, autonome Objekte in persistente Objekte umzuwandeln, gibt es für alle persistenten Objekte folgende Registrierungs-Methoden (Dies darf nicht verwechselt werden mit der Registrierung eines Clients am Broker !):

```
void xxxx_register ([INOUT] xxxx * x, [OUT] OID * oid)
Registriert xxxx in der OTAB, intialisiert den persistenten Kern von x und
gibt die OID des Objektes zurück.

void xxxx_unregister ([INOUT] xxxx * x)
Löscht xxxx aus der OTAB.
```

Es werden zum Attributlesen bzw. –schreiben grundsätzlich `get()` und `set()` Methoden benutzt:

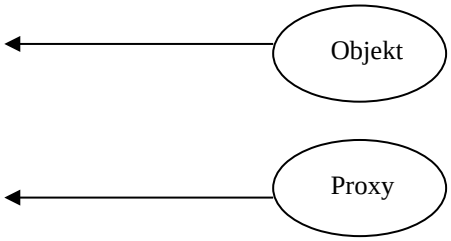
```
YYYY   xxxx_get_YYYY ([IN] xxxx *)                  Attribut YYYY lesen
void   xxxx_set_YYYY ([INOUT] xxxx , [IN] YYYY(*))  Attribut YYYY schreiben
```

Direkte Zugriffe sind verboten, da die Kapselung verletzt, und eine anschließende Verteilung verhindert würde:

```
knde * k;
k = knde_new ();
k->name = „Hugo“              Möglich, aber per Konvention verboten !
knde_set_name (k, „Hugo“);    So ist es richtig.
```

Alle Objekte die verteilt werden, und über einen Proxy ansprechbar sein sollen, definieren den Proxy vom Typ void (long in der aktuellen Version):

```
#ifndef XXXX_PROXY
typedef struct xxxx_tag {
    yyyy y;
    ...
} xxxx;
#else
typedef void xxxx;
#endif
```



Da nur Zeiger auf xxxx-Strukturen verwendet werden, können diese Zeiger zur Speicherung der OID am Client verwendet werden.

Es werden per Konvention Pre- und Postconditions in Funktionen ausgeführt wo z.B. alle übergebenen Zeiger auf Gültigkeit überprüft werden (Precondition) und beim Verlassen die interne Konsistenz der Datenstruktur gecheckt wird (Postcondition). Es wird zwischen Debug- und Releaseversion mittels Makro #define DEBUG unterschieden.

Alle ADTs besitzen für Ausgabezwecke und zur Fehlersuche eine print-Methode um ihre internen Daten an STDOUT oder in eine externe Datei zu schreiben.

```
void   xxxx_print ([IN] xxxx *)          xxxx für Tracezwecke ausgeben
```

Alle „print“-Methoden verwenden den LOG-Mechnismus (LOG) um die Attribute formatiert auszugeben (siehe Kapitel 9.4.4 Tracemechanismus (log)).

9.2 Namenskonventionen für Dateien

Im ROSALINDE-Projekt gibt es strikte Namenskonventionen für Dateien. Das ist zwingend notwendig, da zu einer lokalen Implementierungsdatei, etliche Dateien hinzukommen. Folgende Namenskonventionen werden verwendet:

9.2.1 Testtreiber

Jeder ADT verfügt über ein Testprogramm, einen sogenannten Testtreiber, der die korrekte Funktion nachweist. Alle Funktionen eines ADTs sollten in einem solchen Testtreiber mindestens einmal aufgerufen werden. Testtreiberdateien beginnen mit dem Kürzel tt_ (tt_knde, tt_knto). Wird beim Testtreiber zwischen lokaler und verteilter Version unterschieden, ergibt sich folgende Namenskonvention: tt_r_ (verteilte Version), tt_ (lokale Version).

9.2.2 RPC Programmdateien

Die RPC-Schnittstellenbeschreibungsdateien mit der Erweiterung „.x“ haben alle den Präfix „r_“. Es existieren in der verteilten Version eines ADTs folgende Dateien:

```
ADT Name = xxxx

r_xxxx.x          // RPC Interfacedefinitionsdatei (selbstgeschrieben)
r_xxxx.h          // SUN-RPC Headerdatei für RPC-Schnittstelle (r_xxxx)
r_xxxx_clnt.c     // SUN-RPC Client-Stubs
r_xxxx_svc.c      // SUN-RPC Server-Stubs
r_xxxx_xdr.c      // SUN-RPC XDR-Marshallingfunktionen
```

```
r_xxxx.c            // Server Stub-Stub
r_xxxx_svc_main.c // Server Hauptprogramm (Initialisierung)
xxxx_clnt.c         // Client Stub-Stub
xxxx.c              // lokale Programmlogik
```

selbstgeschrieben

9.2.3 Bibliotheken

Alle unter UNIX verwendeten Programmbibliotheken beginnen mit dem Präfix „lib“, und enden mit der Erweiterung „.a“ (Shared Librarys (.so)). Daher verwendet das ROSALINDE-Projekt ebenfalls diese Namenskonvention. Überdies gibt es folgende Erweiterungen:

- xxxx_rpc_server (libxxxx_rpc_server.a) bestehend aus:

```
1. r_xxxx_svc.c         // Server-Dispatcher
2. r_xxxx_xdr.c         // XDR-Marshallingroutinen
3. r_xxxx.c             // Server-Stub-Stub
4. r_xxxx_svc_main.c    // Server-Hauptprogramm
5. xxxx.c               // Die Programmlogik
```

Serverversion der Programmbibliothek. Diese darf nur am Server verwendet werden.

- xxxx_rpc_clnt (libxxxx_rpc_clnt.a) bestehend aus:

```
1. xxxx_clnt.c          // Client-Stub-Stub
2. r_xxxx_clnt.c        // generierter Client-Stub
3. r_xxxx_xdr.c         // XDR-Marshallingroutinen
```

Clientversion der Programmbibliothek. Diese darf nur am Client verwendet werden.

- xxxx_local (libxxxx_local.a):

```
1. xxxx.c               // Die Programmlogik
```

Programmbibliothek zum Erzeugen einer lokalen Programmversion (Kein RPC).

9.3 Container

9.3.1 Liste (list)

Der ADT „list“, der intern als dynamisches Zeigerarray realisiert ist, stellt eine lineare Liste mit den üblichen Zugriffsfunktionen (rewind, get_next) dar. Der ADT „list“ wurde von Prof. Dr. Siedersleben für die Vorlesung Anwendungsentwicklung I (AE1) entwickelt. Die Liste kann Zeiger auf beliebige Typen speichern (void*). Die Liste ist nicht zustandsfrei und es existiert kein Itteratorkonzept. All diese Verbesserungen sind inzwischen in die Vorlesung AEI eingeflossen, die Liste wurde in Vektor (vect) umbenannt, und die Zugriffsfunktionen in eine eigene Itteratorstruktur gekapselt (itvc). Für die ROSALINDE-Anwendung spielt das jedoch keine Rolle, da im wesentlichen nur Hashtabellen verwendet werden, die ihrerseits die Liste kapseln. Für eine Weiterentwicklung ist jedoch eine Anpassung der Hashtabelle an den ADT-Vektor (vect) nötig.

9.3.2 Hashtabelle (hash)

Die Hashtabelle ist von grundlegender Bedeutung in der ROSALINDE-Anwendung. Alle Listen bzw. Tabellen werden in dieser Form verwaltet um effizienten Zugriff über Schlüsselattribute, zumeist Strings, zu ermöglichen. Die Hashtabelle besteht aus n-Listen (Buckets). Es wird „einfaches“ Hashing verwendet. Unter „einfach“ versteht man die Eigenschaft, daß genau einmal eine Position errechnet wird. Anschließend erfolgt die Suche sequentiell innerhalb der jeweiligen Liste. Jeder zu speichernde Datentyp muß über eine eigene Hashfunktion verfügen, damit per Callback die Hashtabelle die interne Position ermitteln kann. Ebenso wird eine Vergleichsfunktion für die zu speichernden Datentypen benötigt, um suchen zu können. Diese Art von

Hashing wird oft als „offenes“ Hashing bezeichnet, wobei „offen“ auf die Eigenschaft hinweist, daß es keine Grenze der Zahl der möglichen Einträge gibt (solange noch Speicher vorhanden ist).

Diese Methode gibt einem die Möglichkeit, *e* Anfragen auf *n* Namen in einer Zeit $\frac{n \cdot (n+e)}{n}$ proportional zu für eine gewählte Konstante *m*, auszuführen. Weil *m* beliebig groß werden kann, bis zu *n*, ist diese Methode generell effizienter als die Suche über lineare Listen.

Die Hashtabelle hat im wesentlichen folgende Funktionen:

```
void   hash_insert ([INOUT] hash * hp, [IN] void * obj, HFNK hfnk);
```

hash_insert() – speichert einen Zeigers auf ein beliebiges Objekt mit der objektspezifischen Hashfunktion hfnk in der Hashtabelle. Daraus folgt, daß Typen die in der Hashtabelle gespeichert werden sollen, über eine eigene Hashfunktion verfügen müssen. Der Aufrufer ist verantwortlich für die Allokation von obj.

```
void * hash_get ([IN] hash * hp, [IN] void * key, HFNK hfnk, COMP_FCT cf);
```

hash_get() - sucht nach einem Objekt über den Schlüssel (key), einer Hashfunktion (hfnk) und einer zum Suchen benötigten Vergleichsfunktion (cf). Die Funktion gibt NULL zurück wenn das Objekt nicht in der Hashtabelle gefunden wurde.

```
void * hash_remove ([INOUT] hash * hp, [IN] void * key, HFNK hfnk, COMP_FCT
      cf);
```

hash_remove() - entfernt ein Objekt aus der Hashtabelle und gibt einen Zeiger auf dieses zurück. Die Funktion liefert ebenfalls NULL zurück wenn das Objekt mit dem Schlüssel (key) nicht in der Hashtabelle gefunden wurde.

```
void  hash_itterate ([IN] hash * h, [IN] HASH_CALLBACK_FNK callbackfnk, [IN?
      OUT?] void * params);
```

hash_itterate() - itteriert über alle Objekte in der Hashtabelle und ruft die übergebene Callbackfunktion auf, die als ersten Parameter den Objektzeiger erwartet. Es können auch weitere Funktionsparameter mitübergeben werden. Diese Funktion ist sehr wichtig für ein einfaches Handling der Hashtabelle, da oftmals für alle Objekte gleiche Funktionen durchgeführt werden.

(Vgl. Kapitel 9.6.2 Objekttabelle (OTAB))

Ein Beispiel für eine sehr gute Hashfunktion für Strings, die eine gleichmäßige Verteilung in die einzelnen Buckets ermöglicht und in der ROSALINDE-Anwendung mehrmals verwendet wird, ist die Funktion hashpjw von P.J. Weinberger:

```
int hashpjw (char * s)
{
      char * p;
      unsigned h=0, g;
      for (p = s; *p != '\0'; p++) {
              h = (h << 4) + (*p);
              if ( g = h & 0xf0000000 ) {
                  h = h ^ (g >> 24);
                  h = h ^ g;
              }
          }
      return h % PRIME
}
```

Diese Hashfunktion ermöglicht eine sehr gleichmäßige Verteilung von Einträgen auf die zur Verfügung stehenden Listen. Dies funktioniert insbesondere für maschinell generierte Zeichenketten wie z.B. Kunde1, Kunde2 ... KundeN.

9.4 Allgemeine Funktionen

9.4.1 Aufzählungstypen (enum)

Die ROSALINDE-Anwendung verwendet gekapselte Aufzählungstypen die jeweils eine Kurz- und eine Langform des Enumerationstypes als String darstellen können. Der Aufzählungstyp wird nur in der Anrede verwendet, die einerseits in Kurzform („Hr“) und anderseits auch in einer Langform („Herr“) vorliegen kann. Die Zusammenfassung der verschieden Darstellungsformen eines Enumerationstypen ist sehr sinnvoll. Bei großen Softwaresystemen kann die Anzahl der benötigten externen Darstellungen eines Enumerationstypen leicht in die Dutzende gehen. Ohne einen solchen Enumerationsmechnismus müßten für alle diese Darstellungen zusätzliche Zugriffsfunktionen geschrieben werden, was extrem fehleranfällig ist und die Softwarewartung erschwert:

Beispiel:

```
char * anrd_dlg_out (a);            // Ausgabe für Dialoge
char * anrd_btx_out (a);            // Ausgabe für BTX
char * anrd_fax_out (a);            // Ausgabe für FAX
char * anrd_as400_out (a);          // Ausgabe für AS400 Filetransfer
char * anrd_nachbarsystem_out (a);  // Ausgabe für Nachbarsystem
...
```

Besser sind, wie in der ROSALINDE-Anwendung verwendet, generische Enumerationstypen, die verschiede externe Darstellungen über Tabellen realisieren:

```
char * anrd_out (a, KURZ);
char * anrd_out (a, LANG);
char * anrd_out (a, BTX);
...
```

Um diese generischen Enums zu verwenden muß nur die Deklaration im Headerfile über ein Makro erzeugt werden:

```
GEN_ENUM (anrd);
```

Anschließend muß noch die Tabelle geliefert werden. (Die ROSALINDE-Anwendung unterstützt nur zwei Repräsentationen. Allerdings ist eine Erweiterung trivial.

```
#define anrd_MAX  4
#define anrd_TAB\
            {{ "?",     "Hr",   "Fr",   "Fa" },\
            { "undef", "Herr", "Frau", "Firma" }}
GEN_ENUM_IMP (anrd);
```

9.4.2 Serialisierung von Standard C-Typen (misc)

Wie bei den ADT-Funktionen schon erwähnt werden die Datenbankstrings rekursiv aus den Attributen erzeugt (db_in(), db_out()). Am Ende einer solchen Rekursion stehen immer Standart-C Typen für die ebenfalls Marshallingroutinen geschrieben werden müssen (z.B. int, float ...). Diese Funktionen stehen in der Datei „`misc.c`“.

```
void long_db_in(long *l, strg *t);
```

long_db_in() - füllt eine Longinteger Zahl (l) aus einem String t. Der String wird sich dabei um die für l reservierte Länge verkürzen.

```
void long_db_out(long *l, strg *t);
```

long_db_out() - erzeugt die Stringdarstellung einer Longinteger Zahl (l) und hängt diese an t an.

Die Funktionen für int, float, double... sind analog.

9.4.3 Strings (strg)

Die ROSALINDE-Anwendung verwendet einen eigenen Stringtyp der Zeichenketten fester und dynamischer Länge verwalten kann. Die Unterscheidung zwischen statischem und dynamischem String erfolgt beim Erzeugen bzw. Initialisieren, da dort die Maximallänge übergeben wird. 0 bedeutet beliebige Länge (dynamisch), jede andere Zahl bedeutet feste Länge (statisch). Übliche Fehler der Stringmanipulation können so verhindert werden. Die üblichen C-Bearbeitungsfunktionen stehen für diese Strings (strg) zur Verfügung.

```
void strg_cat ([INOUT] strg * s,  [IN] strg * t);
void strg_cat_str ([INOUT] strg * s,  [IN] char * c);
```

strg_cat() - hängt den String t an den String s an. Dabei wird, falls nötig, der Rest abgeschnitten (statischer String). Ist s dynamisch, so wird dieser um die entsprechende Länge aufgeweitet.

```
char * strg_chr ([IN] strg * s, char c);
```

strg_chr () - siehe strchr() ANSI-C-Laufzeitbibliothek.

```
int strg_len ([IN] strg * s);
```

strg_len() - siehe strlen() ANSI-C-Laufzeitbibliothek.

```
void strg_sprintf ([INOUT] strg * dest,  [IN] strg * s,...);
void strg_sprintf_str ([INOUT] strg * s,  [IN] char * c,...);
```

strg_sprintf() - siehe sprintf() ANSI-C-Laufzeitbibliothek.

```
void strg_cut ([INOUT] strg * s, int s_offset);
```

strg_cut() - Schneidet s um s_offset Zeichen vorne ab. Dynamische Strings werden physikalisch um s_offset Zeichen verkleinert. Statische Strings werden nur entsprechend umkopiert.

9.4.4 Tracemechanismus (log)

Wie jedes größere Programm kann auch die ROSALINDE-Anwendung Traceinformation erzeugen. Die Ausgabe geschieht wahlweise an die Standardfehlerausgabe (STDERR) oder in eine, im Programm angegebene, Datei. Das Traceobjekt ist ein Singleton Objekt, daher entfällt die Objektreferenz in der Schnittstelle. Zu verwenden ist der Logmechanismus wie printf() in Ansi-C. Das heißt beliebige Formatierungen können angegeben und eine variable Zahl von Parametern übergeben werden. Beispiel:

```
LOG_OPEN („broker.log“, „w+“);
LOG („%x %lf %i“, zeiger, double_wert, integer_wert);
....
LOG_CLOSE ();
```

9.4.5 Speicherverwaltung (mmgr)

Nachdem in der Entwicklung mehrfach Abstürze wegen Speicherfehlern aufgetreten sind, erschien eine eigene gekapselte Speicherverwaltung die alle Speicherzugriffe (malloc(), free ()) protokolliert und auf Konsistenz überprüft als sinnvoll. Die Idee dazu ist dem hervorragendem Buch „Writing Solid Code (Nie wieder Bugs)“ von Steve Maguire entnommen[10]. Diese Speicherverwaltung wird in der Realease-Version komplett ausgeschaltet. Allerdings wurden wir von der Technik überholt und konnten die endgültigen Programmversionen mit dem Testwerkzeug Purify (Speicherwerkzeug von Rational Software) für Solaris 2.5 prüfen. Die jetzige Version der ROSALINDE-Anwendung ist, was Speicherfehler betrifft, zu 99.x% fehlerfrei. Allerdings liefert Purify immer noch Warnungen, die auf die SUN-RPC Laufzeitbibliotheken zurückzuführen sind. Die Dateien des Speicherverwalters werden in der aktuellen Version nicht mehr

verwendet. Übrigens: Die MFC (Microsoft Foundation Class) verwendet ebenfalls eine eigene Prüflogik zum Aufspüren von Speicherfehlern in der Debug-Version.

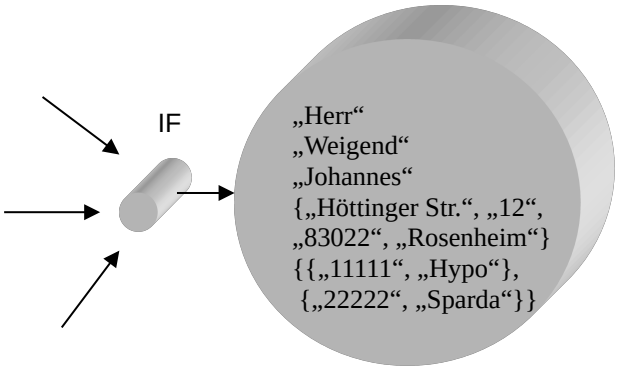
9.5 Fachliche Anwendungslogik

9.5.1 Autonome Objekttypen

Als autonomer Objekttyp bezeichnet man Objekte die als einzelne Instanzen vorkommen und nicht nur als Attribute anderer Objekte. Die autonomen Objekte stellen die Anwendungsobjekte der jeweiligen Umgebung dar. Sie modellieren direkt das der Anwendung zugrunde liegende fachliche Problem. Beispiele hierzu sind Kunde, Konto, Rechnung oder Bestellung.

9.5.1.1 Kunde

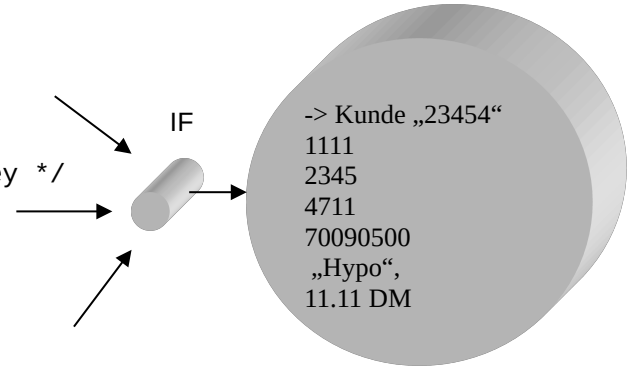
```
typedef struct knde_tag {
    pkrn persistent;
    anrd anrede;
    strg name;
    strg vorname;
    adrs adresse;
    list konten;
} knde;
```



Ein Kunde hat eine Anrede, einen Namen und Vornamen, eine Adresse und kann beliebig viele Konten besitzen. Der Kunde ist ein persistentes Objekt und hat daher als Attribut einen persistenten Kern (pkrn).

9.5.1.2 Konto

```
typedef struct knto_tag {
      pkrn persistent;
      knde *kunde;
      long konto_id; /* ID = Key */
      long kunde_rid;
      long konto_nummer;
      long blz;
      strg bankname;
```



```
      double betrag;
}
knto;
```

Eine Konteninstanz steht in direkter Beziehung zu einem Kunden. Hierzu wird einerseits ein Zeiger auf den jeweiligen Kunden verwendet und andererseits, da solche Zeiger nicht persistent sein können, ein Fremdschlüssel (kunde_rid) benutzt um die Beziehung beim Lesen eines vorhandenen Kontos herzustellen.

9.5.2 Nichtautonome Datentypen

Nichtautonome Datentypen werden in Anwendungen nie alleine benutzt. Sie sind entweder Attribute von autonomen Objekttypen oder Attribute eines anderen nichtautonomen Datentyps. Bei der ROSALINDE-Anwendung kommt als einziger nichtautonomer Datentyp die Adresse des Kunden vor.

9.5.2.1 Adresse

Der ADT-Adresse der ROSALINDE-Anwendung stellt die Adreßangaben eines Kunden dar. Die Adresse ist eine Struktur die Straße, Hausnr, PLZ und Ort kapselt und set- bzw. get-Methoden für diese bereitstellt. Als interne Speicherform für diese Attribute werden ROSALINDE-Strings (strg) verwendet um die typischen, in Verbindung mit der ANSI-C Stringbehandlung auftretenden Fehler zu verhindern. Die Implementierung ist weitestgehend trivial und benötigt keine weiteren Erklärungen.

9.6 Technische Querschnittsfunktionen

9.6.1 Persistenter Kern (pkrn)

Jedes persistente Objekt besitzt als Attribut einen persistenten Kern (pkrn). Der persistente Kern führt Buch über den Zustand des Objektes, speichert die Funktionszeiger für die Datenbankzugriffsfunktionen und die Row-ID. Folgendes, gegenüber der Praxis vereinfachtes, Zustandsmodell wird für persistente Objekte in der ROSALINDE-Anwendung verwendet:

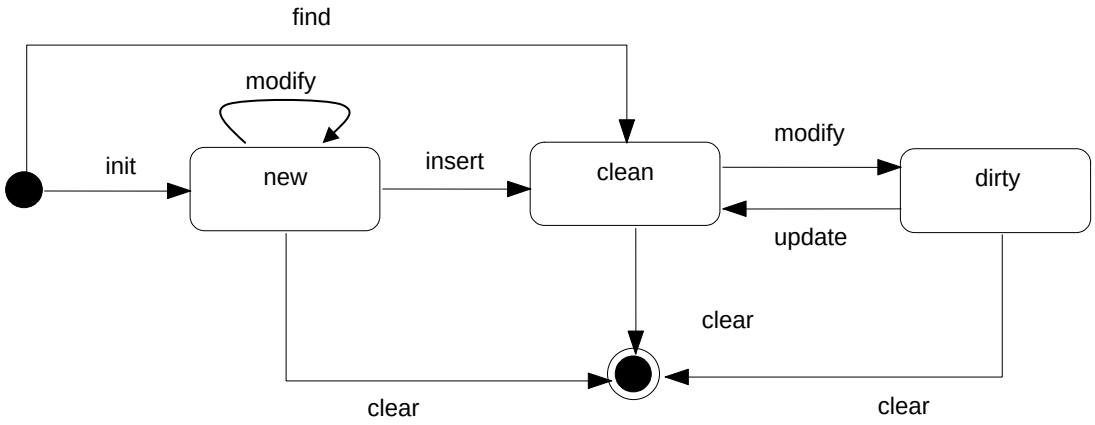


Abbildung 39 Persistenter Kern

Jedes neuangelegte, persistente Objekt beginnt seinen Lebenszyklus mit dem Zustand *new*. Neuangelegte Objekte verlassen diesen Zustand nicht, auch wenn sie modifiziert werden. Erst beim ersten Schreiben auf die Datenbank (commit) wird erkannt, daß es sich um ein neuangelegtes Objekt handelt, und somit dessen Insert-Methode aufgerufen. Dieser "Insert" überführt das Objekt in den neuen Zustand *clean*. Dieser Zustand bedeutet, daß sich ein Objekt in der Datenbank befindet und nicht geändert wurde. Die set-Methoden des Objektes rufen bei Änderungen an Attributen ein `pkrn_modify()` auf, welches das Objekt

von *clean* in den Zustand *dirty* überführt. Ein „commit“ auf ein Objekt im Zustand *dirty*, ruft dessen Update-Methode auf, welche die Datenbank aktualisiert und das Objekt wieder in den Zustand *clean* überführt. Beim Löschen eines Objektes wird mit clear() das Zustandsmodell verlassen. Sinnvollerweise werden Objekte nie physikalisch aus der Datenbank gelöscht, sondern nur markiert, um eine Historisierung zu ermöglichen. Die ROSALINDE-Anwendung sieht außerdem kein Löschen von Objekten (Kunde, Konto) vor.

9.6.2 Objekttabelle (OTAB)

In einer „normalen“ Objekttabelle befinden sich alle persistenten und geladenen Objekte einer Anwendung. Dies dient zur Konsistenzsicherung und Wahrung der Objektidentität (5.2). Anders bei der ROSALINDE-Anwendung: Da die Objekttabelle auch zur Proxyexpansion an den Servern verwendet wird (OID -> obj) müssen auch nichtpersistente Objekte dort gespeichert werden können. Daher unterstützt die Objekttabelle Schnittstellen für persistente und für nichtpersistente Objekte. An dieser Stelle unterscheiden sich lokale Programmversion und C/S – Programmversion eklatant, da die lokale Version nur eine Objekttabelle für persistente Objekte benötigt.

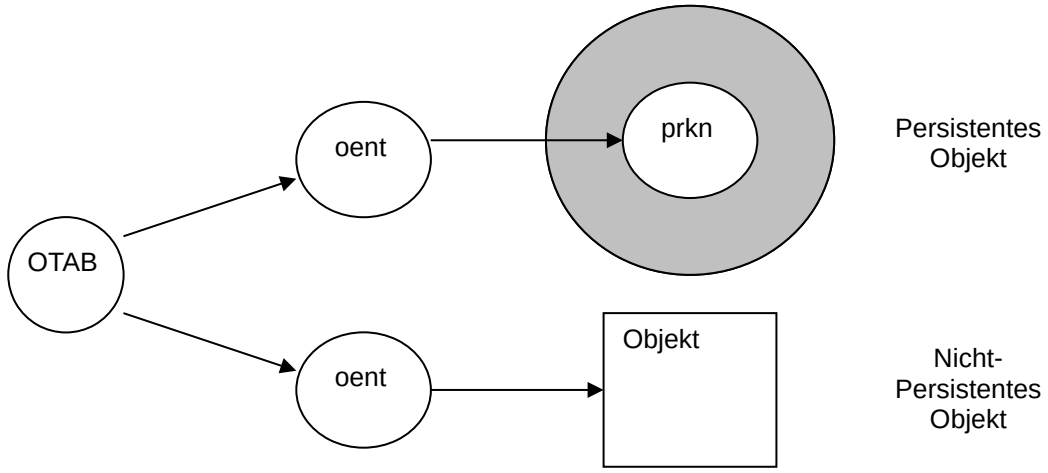


Abbildung 40 „Polymophie“ bei OTAB-Objekt-Einträgen

Wie schon im Kapitel 6.2.2 (Erweiterungen der Objekttabelle) ausgeführt speichert die Objekttabelle der ROSALINDE-Anwendung Einträge für die nichtpersistenten Objekte um von außen über die OID auf diese zugreifen zu

können. Die Objekttabelle ist eine Hashtabelle die OTAB-Entrys speichert (oent). Diese OTAB-Entrys verfügen neben der OID über zwei Attribute:

1. Zeiger auf den persistenten Kern wenn es sich um ein persistentes Objekt handelt (Kunde, Konto).

2. Zeiger direkt auf das Objekt wenn es sich um ein nicht-persistentes Objekt handelt. (Basistabelle)

Es darf nur einer der beiden Zeiger gültig sein. Der jeweils andere muß NULL sein.

Die Objekttabelle kann auch für alle Objekte ein „commit()“ bzw. „rollback()“ durchführen, indem für diese Objekte, abhängig vom Zustand, die jeweilige Insert- bzw. Updatefunktion aufgerufen wird. Nichtpersistente Objekte können sich mit otab_append() bzw. otab_remove() in der Objekttabelle ein- oder austragen. Persistente Objekte verwenden die Methoden otab_register() und otab_unregister(). Diese Funktionen existieren analog in der Clienttabelle, da eine Objekttabelle immer zu einem spezifischen Client gehört (siehe 9.7.4.1 Client Tabelle (ctab)).

```
RC   otab_get    ([IN] otab * o, [IN] long oid, [OUT] oent ** oe)
```

otab_get() – liefert einen Zeiger auf den Tabelleneintrag (oe) mit der passenden Objekt-ID (oid).

```
RC otab_find_by_rid ([IN] otab * o, [IN] long rid, [OUT] oent ** oe);
```

otab_ find_by_rid () – liefert einen Zeiger auf den Tabelleneintrag (oe) mit der passenden Row-ID (rid).

```
void otab_commit ([IN] otab * o)
```

otab_commit() – aktualisiert alle persistenten Objekte in der Datenbank.

```
void otab_rollback ([IN] otab * o)
```

otab_rollback() – liest alle persistenen Objekte der Objekttabelle erneut aus der Datenbank.

```
void otab_register ([INOUT] otab * o, [OUT] long * oid, [IN] pkrn * p, [IN]
void * obj, [IN] UPD_FCT update_fct, [IN] INS_FCT insert_fct, [IN] REA_FCT
read_fct)
```

otab_register() – trägt ein Objekt in die Objekttabelle ein und gibt dessen OID zurück (Bei nicht-persistenten Objekten müssen die Funktionszeiger und der Zeiger auf den persistenten Kern NULL sein).

```
void otab_unregister ([INOUT] OTAB * o, [IN] long oid);
```

otab_unregister() – löscht den Eintrag eines Objektes mit der passenden OID aus der Objekttabelle.

9.6.3 Basistabelle (BTAB)

Die Basistabelle ist ein ADT der satzweisen Zugriff auf eine Datei ermöglicht. Diese Sätze sind Zeichenketten der Länge n. Innerhalb einer geöffneten Basistabelle, ist die Satzlänge fest, und kann nicht variiert werden. Die Basistabelle dient der ROSALINDE-Anwendung als Datenbankersatz und ist im wesentlichen mit der Unix-Dateisystemdatenbank „db“ (gdb) zu vergleichen.

```
void btab_open ([INOUT] btab * b, [IN] char * path, int n, long base);
```

btab_open() - öffnet die Datei (path) und legt die Satzgröße auf n-Bytes fest. „base“ gibt die erste Satznummer an. Falls die angegebene Datei nicht vorhanden ist, wird eine neue, leere Datei angelegt.

```
void btab_close ([INOUT] btab * b);
```

btab_close() - schließt die geöffnete Datei.

```
void btab_rewind ([INOUT] btab * b);
```

btab_rewind () - positioniert auf den ersten Satz in der Datei. (aktuelle Satznummer = base).

```
RC btab_get_next ([INOUT] btab * b, [OUT] char * record);
```

btab_get_next() - liefert den folgenden Satz zurück. Nach „btab_rewind()“ wird mit „btab_get_next()“ der erste Satz geliefert. Der Aufrufer ist für die ausreichende Speicherplatzgröße des Satzpuffers (record) verantwortlich.

```
RC btab_get_previous ([INOUT] btab * b, [OUT] char * record);
```

btab_rewind () - liefert den vorhergehenden Satz zurück.

```
RC btab_get_at ([INOUT] btab * b, long rid, [OUT] char * record);
```

btab_get_at () - ermöglicht einen wahlfreien Zugriff über die Satznummer (Row-ID).

```
void btab_put_at ([INOUT] btab * b, [IN] char * record);
```

btab_put_at () - ermöglicht ein wahlfreies Schreiben über die Satznummer (Row-ID).

```
void btab_append ([INOUT] btab * b, [IN] char * record, [OUT] long * rid);
```

btab_append () - fügt einen neuen Satz an das Dateiende an und gibt dessen Satznummer (Row-ID) zurück.

9.7 Client/Server-Funktionalität

9.7.1 SUN-RPC

Die Verwendung eines RPC-Mechnismus, wie dem SUN-RPC, ist mit einer grundsätzlichen Explosion der Dateizahl verbunden. Wie die folgende Skizze zeigt, werden aus einer einzigen Schnittstellen-Definionsdatei vier weitere Dateien generiert. Zusätzlich dazu kommen noch die selbstgeschriebenen Funktionen, welche die generierten zusätzlich umhüllen und die RPC-Aufrufe wie lokale Funktionsaufrufe aussehen lassen. Aus einem abstrakten Datentyp mit zwei Dateien (`xxxx.c, xxxx.h`) wird in der verteilten Version plötzlich ein Gebilde aus zehn! Dateien. Davon sind nur vier Dateien generiert. Allerdings ist es möglich einen Generator zu bauen, der die Anzahl der selbst zu schreibenden Dateien, von 6 auf 3 reduziert.

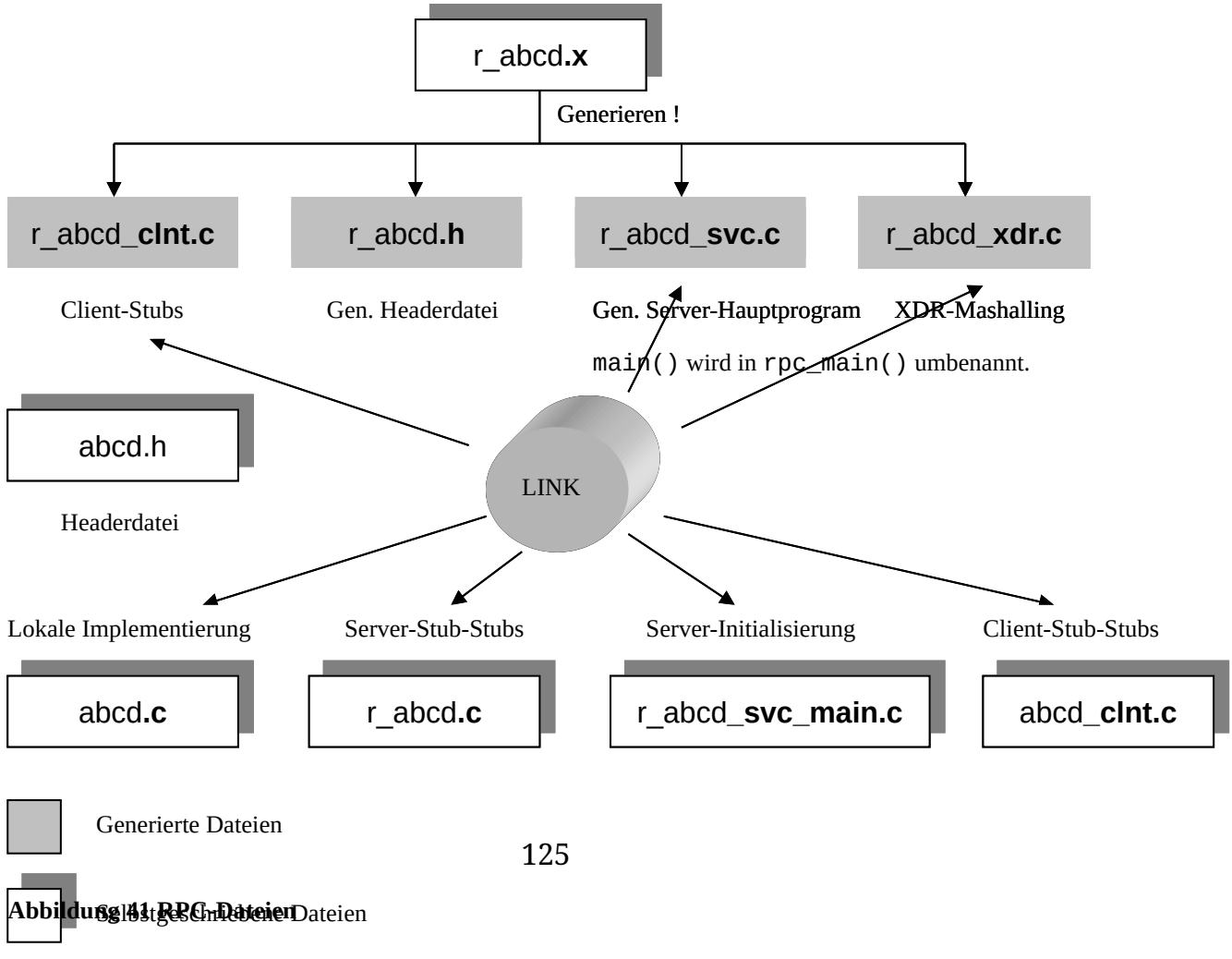


Abbildung 41: RPC-Dateien

9.7.2 Nutzungskonzept für RPC Anwendungen

Durch die enorme Zahl von zusätzlichen Dateien, ist ein Nutzungskonzept, das ein generisches Vorgehen möglich macht, notwendig. Es werden hier die wichtigsten Punkte genannt:

1. Die Implementierung eines ADTs liegt grundsätzlich in einer rein lokalen Version vor. Diese Datei hat keinerlei Wissen über die zugrundeliegende RPC-Architektur. Normalerweise wird ein ADT in einer C-Datei (.c) implementiert und besitzt eine Headerdatei. (.h). Die Anzahl der zur Implementierung nötigen Dateien kann allerdings beliebig sein.

2. Die Schnittstellendefinitionsdatei (.x) hat als Namenskonvention einen Präfix (r_) und anschließend das vierstellige Kürzel des jeweiligen ADTs (z.B. r_knde.x). Die Schnittstellendefinitionsdatei darf nicht den Namen eines vorhandenen Headerfiles haben, da ein gleichnamiges generiert wird.

3. Der Nutzer eines verteilten, abstrakten Datentyps (z.B. tt_abcd) darf keinerlei Wissen über die zugrundeliegende RPC-Architektur haben. Es muß also möglich sein. Nutzer und lokale Implementierung als eigenständige, lokale Anwendung zusammenzubauen.

4. Punkt 1 und 3 werden über Zwischenschichten realisiert: Den Stub-Stubs. (Server-Stub-Stub, Client-Stub-Stub). Die Stub-Stubs (r_xxxx.c, xxxx_clnt.c)

werden selbstgeschrieben und kapseln die internen Schichten der ROSALINDE-Anwendung.

5. Das vom Werkzeug (rpcgen) generierte Server-Hauptprogramm (main) muß in rpcgen_main() umbenannt werden. Das selbstgeschriebene Hauptprogramm (r_xxxx_svc_main.c) erledigt die nötige Programminitialisierung, initialisiert den Exceptionhandlingmechnismus und startet das generierte Server-Hauptprogramm (rpcgen_main ()).

Die folgenden Abbildungen (42 u. 43) zeigen den Ablauf der RPC-Kommunikation bei der ROSALINDE-Anwendung mit den einzelnen Aufgaben innerhalb der Dateien.

Als Beispiel wurde die Funktion „`knde_init()`“ ausgewählt:

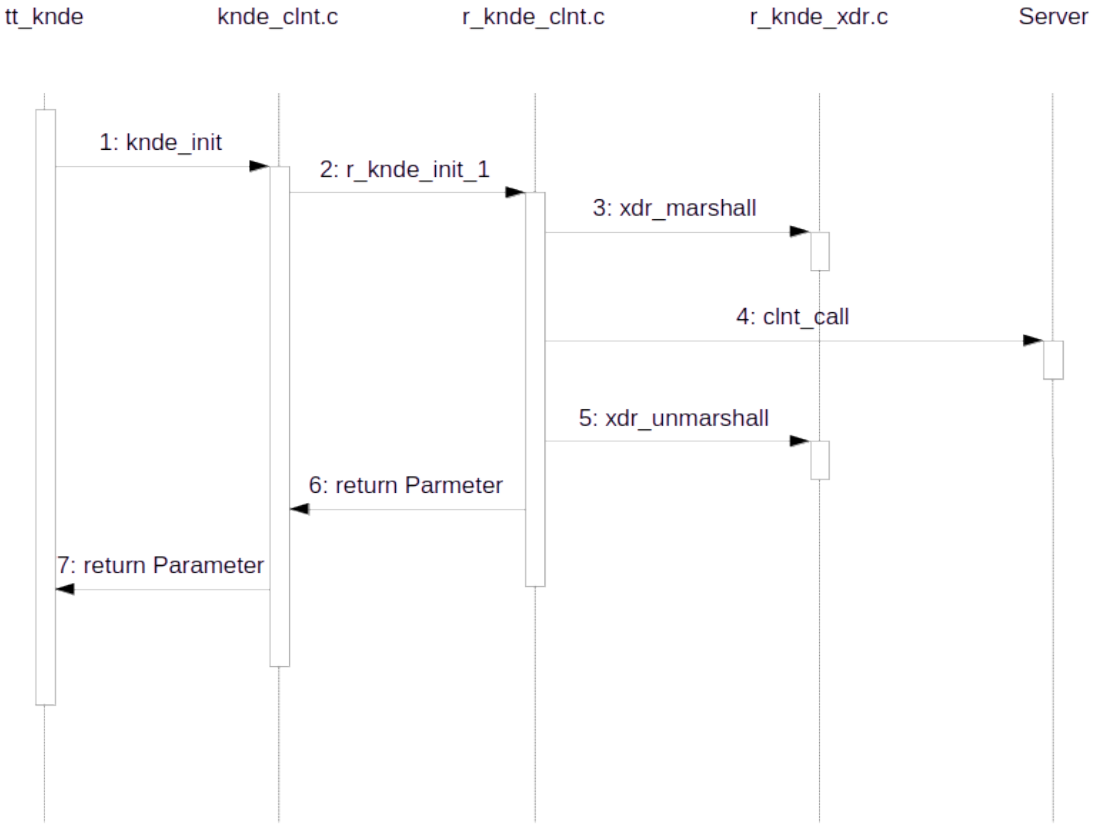


Abbildung 42 RPC-Ablauf am Client

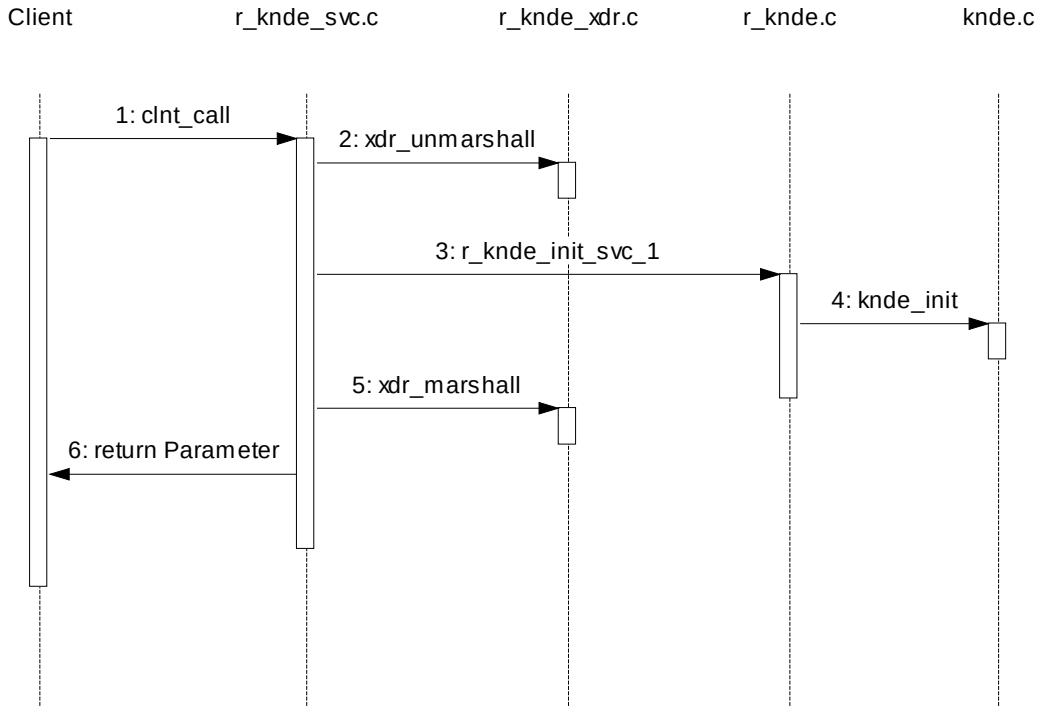


Abbildung 43 RPC-Ablauf am Server

9.7.2.1 Broker (brok)

Der Broker ist als Hashtabelle realisiert und ist der einzige RPC-Server der ROSALINDE-Anwendung dessen Objekte (Einträge der Brokertabelle) nicht über Proxys verwaltet werden. Neben den Brokerfunktionen `bind()`, `register()` und `unregister()` vergibt der Broker die nötigen Client-IDs mittels `brok_get_cid()`. Der Zugriff des Clients auf diese Funktionen wird jedoch über den Server Administrator (sadm) und den Client-Manager (cmgr) gekapselt.

9.7.3 Clientseite

9.7.3.1 Server Administrator (sadm)

Alle Clients müssen über einen vollständig transparenten Zugriff auf die Server verfügen. Das Wissen wo welcher Server einen Dienst bereitstellt, liefert der Broker mittels bind(). Es macht allerdings wenig Sinn vor jedem RPC-Aufruf den Broker zu kontaktieren. Daher werden die Verbindungen von einem „Server Administrator“ zwischengespeichert. Der Name ist leider irreführend, da der Server Administrator nur am Client benötigt wird. Da die Rollen Server/Client auch wie schon erwähnt, wechseln können, findet sich der „Server Administrator“ allerdings auch bei den Funktionen-Servern (`r_knde_svc`, `r_knto_svc`) der ROSALINDE-Anwendung.

Der Serveradministrator ist ein typisches Singleton-Objekt; d.h. es gibt genau eine Instanz dieses ADTs am Client. Daher verfügt dieses Objekt über keine new () – Methode um neue Instanzen zu erzeugen. Die einzige gültige Instanz ist eine statische Variable in der Datei sadm.c. Diese Instanz muß beim Programmstart mit dem Hostnamen des Brokers initialisiert werden (`sadm_init(„Hostname“)`). Dabei baut der Serveradministrator eine RPC-Verbindung zum Broker auf und speichert den erzeugten Verbindungshandle (CLIENT*). Wenn nun ein Client

einen Verbindungshandle zu einem Anwendungsserver benötigt wird zuerst die interne Hashtabelle mit dem eindeutigen Namen des Dienstes durchsucht und wenn kein Handle gefunden wurde ein bind() am Broker auf diesen Dienst durchgeführt; der Clienthandle erzeugt, in die Hashtabelle eingetragen und dem Aufrufer zurückgegeben. Der Serveradministrator kümmert sich auch, daß alle von ihm vergebenen Handles korrekt sind, da dieser in der Lage ist jeden Handle durch eine PING-Funktion zu überprüfen. Ein Client wird nie den Broker direkt nach einem Anwendungsserver fragen, sondern über den sadm einen Verbindungshandle anfordern der als erster Parameter im eigentlichen RPC-Funktionaufruf dient.

```
typedef struct _server_administrator {
    hash cltab;                                 // Verbindungshandles
    char brokerhost [MAX_HOSTNAME];             // Broker Hostname
    CLIENT * brokerhandle;                      // Verbindungshandle Broker

} sadm;

void sadm_init ([IN] char * brokerhost);    Initialisierung der Hashtabelle
```

sadm_init() - muß beim Programmstart von jedem Client, der einen entfernten Dienst in Anspruch nehmen will, aufgerufen werden. Alle weiteren nötigen Initialisierungen werden automatisch vorgenommen.

```
CLIENT * sadm_get_serverhandle  ([IN] char * svcn);
Gibt einen initialisierten Serverhandle zurück
```

sadm_get_serverhandle() - wird in den Client-Stub-Stubs verwendet um den Handle (CLIENT *) zum RPC-Funktionsaufruf zu erhalten. Die Handles werden in einer Hashtabelle gespeichert und im Fehlerfall mit Hilfe des Broker neu erzeugt.

9.7.3.2 Client Manager (cmgr)

Der Client Manager verwaltet Client- und Server-IDs. Die Client-ID dient zur Identifikation eines Clients am Server, daher wird diese jedem RPC-Aufruf als Parameter mitgegeben. Die Client-ID wird automatisch vergeben und vom Client nicht mehr verändert. Die Server-ID wird beim Programmstart eines Servers vergeben und auch dort nicht mehr geändert. Diese wird als Parameter weitergegeben, falls der Server Client eines anderen Servers ist und es sich um ein vom Server verwaltetes Objekt handelt. Am Server ist die Client-ID immer die Client-ID des gerade zu bearbeitenden Clients.

```
void   cmgr_set_cid  (char * cid);   Setzt Client-ID / Server

char * cmgr_get_cid();               Gibt Client-ID zurueck / Client, Server

char * cmgr_get_sid ();              Gibt Server-ID zurueck / Server
```

9.7.4 Serverseite

Jeder Server steht vor der Aufgabe bei RPC-Aufrufen die Proxy-OID in die eigentlichen Anwendungsobjekte zu expandieren. Diese Expansion findet, wie schon erwähnt, mithilfe der Objekttabelle (OTAB) statt. Allerdings steht am Server für jeden Client eine eigene Objekttabelle zur Verfügung. Das Objekt, das die Objekttabellen aller Clients kapselt heißt Client Tabelle (ctab).

9.7.4.1 Client Tabelle (ctab)

Die Client Tabelle (ctab) ist ebenfalls wie der Serveradministrator (sadm) ein Singleton Objekt. Daher muß für die einzige Instanz die `init()`-Methode beim Programmstart aufgerufen werden. Wird ein neues Objekt am Server erzeugt, kann dieses mittels `ctab_register()` in eine Tabelle eingetragen werden. Dieser Eintrag ist für persistente wie auch nichtpersistente Objekte zwingend notwendig, da sonst keine Möglichkeit besteht über die OID des Proxys an das Objekt zu kommen. Daher findet der Aufruf von `ctab_register()` immer im Server-Stub-Stub der `new()`–Methode statt. Bei persistenten Objekten wird diese Funktion zweimal aufgerufen. Beim ersten Mal wird die OID für ein neues Objekt eines Clients vergeben. Alle anderen Parameter sind NULL. Nach diesem Aufruf handelt es sich bei dem Objekt immer noch um ein nichtpersistentes Objekt. Beim zweiten Aufruf (z.B. `knde_register()`) wird dieses Objekt persistent, und die fehlenden Informationen (pkrn, insert_fct, update_fct, read_fct) geliefert. Dieses zweistufige Vorgehen ist insofern sinnvoll, da mit temporären Objekten gearbeitet werden kann, die dann aber nicht in der Datenbank gespeichert werden.

```
void ctab_register ([IN] char * cid, [INOUT] long * oid, [IN] pkrn * p, [IN]
void * obj, [IN] UPD_FCT update_fct, [IN] INS_FCT insert_fct, [IN] REA_FCT
read_fct);
```

ctab_register() - übernimmt im wesentlichen zwei Aufgaben:

1. Registrierung eines Objektes in der zugehörigen OTAB für diesen Client. Ist noch keine OTAB vorhanden wird eine neue angelegt. Das Ergebnis ist eine Vergabe einer eindeutigen OID für das Objekt.

2. Der Persistenzmechanismus wird für dieses Objekt aktiviert. Dazu muß der persistente Kern des Objektes mitübergeben werden, ebenso die Zeiger auf die Datenbank-Lese und –Schreiboperationen. Nach dem Aufruf dieser Funktion wird das Objekt beim nächsten `commit()` in die Datenbank geschrieben.

```
      /* Objekt mit der Objekt-ID oid wird aus der Objekttabelle entfernt */
void ctab_unregister ([IN] char * cid, long oid);
```

ctab_unregister() - sorgt für die Entfernung des Eintrages aus der Clienttabelle bzw. zugehörigen Objekttabelle. Nach Aufruf dieser Funktion ist die OID des Objektes ungültig.

```
      /* Commit fuer die Objekte eines Clients */
void  ctab_commit ([IN] char * cid);
```

ctab_commit() - führt einen „Commit“, für alle Objekte die sich in der Clienttabelle persistent registriert haben, durch. Es wird die Objekttabelle des Clients gesucht und anschließend ein `otab_commit()` aufgerufen (siehe Kapitel 9.6.2 Objekttabelle).

```
      /* Rollback für alle Objekte eines Clients */
void  ctab_rollback ([IN] char * cid);
```

ctab_rollback() - führt einen Rollback für alle Objekte die sich in der Clienttabelle persistent registriert haben durch. Es wird die Objekttabelle des Clients gesucht und anschließend `otab_rollback()` aufgerufen. (siehe Kapitel 9.6.2 Objekttabelle).

```
      /* Expandiert den Proxy (oid) zum eigentlichen Objekt (void *) */
void * ctab_get_obj ([IN] char * cid, [IN] long oid);
```

ctab_get_obj() - Neben `ctab_register()` ist diese Funktion die wichtigste der Clienttabelle. Hier kann über die OID (Proxy) das damit verbundene Objekt angefordert werden. Diese Funktion wird in den Methoden aller Server-Stubs benötigt um das Objekt anzufordern und anschließend die nötigen Methoden dessen aufzurufen. Voraussetzung ist ein vorheriges Registrieren des Objektes mittels `ctab_register()`.

9.8 Grafische Benutzeroberfläche (GUI)

9.8.1 Tcl/Tk

Die grafische Benutzeroberfläche wurde mit Tcl/Tk realisiert. Entwickelt wurde Tcl/Tk von John K. Ousterhout, Professor für Electrical Engineering und Computer Science an der University of California, Berkeley. Tcl ist eine C-ähnliche Interpretersprache. Zusammen mit Tk, dem Toolkit für X-Windows (X11), können sehr einfach Oberflächenprogramme geschrieben werden. Es ist ebenfalls möglich C-Funktionen mit-einzubinden. Die Programmierung des GUIs und die Anbindungskonzeption wurde von Christian Stöllinger erarbeitet. Hier

möchte ich nur noch auf die Diplomarbeit: „Dialogerstellung mit Tcl/Tk und C“ von Christian Stöllinger verweisen (Fachhochschule Rosenheim 1996).

9.9 Fehlerbehandlung

9.9.1 Fehlerarten

Es können vielschichtige Fehler bei dem Betrieb und in der Entwicklung von Client/Server-Softwaresystemen auftreten. Gerade bei der ROSALINDE-Anwendung, wo Server weltweit verteilt sein können, treten unter Umständen Schwierigkeiten auf: Netzkomponenten können ausfallen, Server können ausgeschaltet werden und die Software selbst kann fehlerhaft sein. Im folgenden werden die Fehler innerhalb der Software betrachtet und das Verhalten derselben, wenn Fehler in Nachbarsystemen (z.B. Netzwerk) auftreten. Da ein größeres Softwareprojekt auch nie aus 100% fehlerfreiem Code besteht, ist ein sinnvolles reagieren im Fehlerfall unbedingt notwendig.

Im ROSALINDE-Projekt werden folgende Fehlerarten unterschieden:

- fachliche Fehler
- technische Fehler
- Ausnahmen (Exceptions)

9.9.2 Fachliche Fehler

sind Fehler die auftreten, falls fachliche Konsistenzbedingungen im Programm durch Eingaben oder Nachbarsysteme verletzt werden. Auf Softwareebene werden fachliche Fehler über Returncodes realisiert:

Beispiel:

```
anrd_in (anrede, &rc, LANG, s);
if (rc != OK) {
        MessageBox („Es existiert keine Anrede der Form %s.\n“, s);
        // try again
}
```

Returncodes müssen nicht, wie in C üblich, nur einfache Integerwerte sein, sondern können Zeiger auf Fehlerstacks sein, die eine Rückverfolgung der aufgetretenen Fehler möglich machen. Auch ein hierarchischer Aufbau von Fehlercodes ist möglich. Bei der ROSALINDE-Anwendung gibt es keine Fehlerhierarchie sondern nur einfache Integerwerte. Folgende Fehlerbehandlungstrategie wird im ROSALINDE-Projekt angewandt:

1. Returncodes sind immer vom Typ RC.

```
typedef long RC;  // rc.h
```

2. Returncodes werden, wenn möglich, als Rückgabewert der Funktion übergeben. Eine Ausnahme stellt der ADT anrd (Anrede) dar.

```
/*  registriert einen Service:
    RC: OK                   wenn alles klappt
        DUPLICATE_SVC_ENTRY  fuer doppelte Registrierung */
RC brok_register (bent * be);
```

3. Falls Zeiger als Rückgabewert einer Funktion verwendet werden, kennzeichnet der Wert NULL einen gültigen Rückgabewert. Wenn Fehler auftreten, werden diese über den zweiten Funktionsparameter (als Referenz) zurückgeliefert.

```
knto * knto_find_by_key (long key, RC * rc);
```

Da nicht automatisch erzwungen werden kann, daß jeder Nutzer einer Funktion (Client) den Rückgabewert prüft und richtig interpretiert, ist es wichtig, daß jedes Objekt sich selbst auf Konsistenz überprüfen kann. Solche Konsistenzprüfungen sollten abhängig von der Entwicklungsversion (Debugversion) bzw. Endversion ein- und ausschaltbar sein. So könnten in der Entwicklungsversion vor dem Durchführen von Transaktionen, alle Objekte auf Konsistenz überprüft werden.

9.9.3 Technische Fehler

Technische Fehler treten im Zusammenhang mit technischen Komponenten auf. Solche Komponenten können Beispielsweise Datenbank-, ODBC-Schnittstellen, API-Funktionen und RPC-Verbindung sein. Die Abgrenzung zu den Ausnahmen ist oftmals schwierig und hängt vom konkreten Anwendungsfall ab. Wie auch beim Win32 API werden technische Fehler bei der ROSALINDE-Anwendung mit Returncodes oder mit Ausnahmen behandelt.

9.9.4 Ausnahmen

Ausnahmen sind Fehlerzustände die niemals unter „normalen“ Umständen auftreten sollten. Beispiele für Ausnahmen sind:

- Kein Freispeicher mehr (Out of memory).
- Nullzeiger wurde an Funktionen übergeben.
- Objektreferenz ist NULL.
- Inkonsistenzen in Basisobjekten (Listen, Tabellen ...).

Wichtig ist jedoch, daß in jedem Fall noch eine Möglichkeit bestehen muß auf diese Ausnahmen sinnvoll zu reagieren und kein einfacher `exit()` ausgeführt wird.

Beispielsweise könnte eine Speicherverwaltungsroutine versuchen eine Speicherbereinigung via Garbage Collection durchzuführen um so Speicher frei zu machen. Ebenso müssen auch im Katastrophenfall offene Datenbankverbindungen geschlossen und Objektereferenzen in verteilten Systemen freigegeben werden. Falls Prozesse in anderen Adressräumen gestartet wurden ist ebenfalls eine explizite Beendigung derer nötig.

Die Server der Rosalinde-Anwendung versuchen sich im Ausnahmefall am Broker abzumelden (unregister()) und die Datenbankverbindungen zu schließen.

Als günstiger Mechanismus für Ausnahmen die selten Auftreten und dann eine zentrale Bearbeitung verlangen, hat sich das C++ Exception Handling etabliert. Mithilfe der Schlüsselwörter try, throw und catch wird eine Ausnahmebehandlung realisiert. Dieses Exceptionhandling wurde bei der

ROSALINDE-Anwendung über Makros nachgebaut, mit der Einschränkung, daß keine Klasseninstanzen übergeben werden können, sondern nur einfache Returncodes.

Exception Handling ist ein Steuerungsmechanismus für horizontale Programmflüsse, während Returncodebehandlung einen Steuerungsmechanismus für vertikale Programmflüsse darstellt:

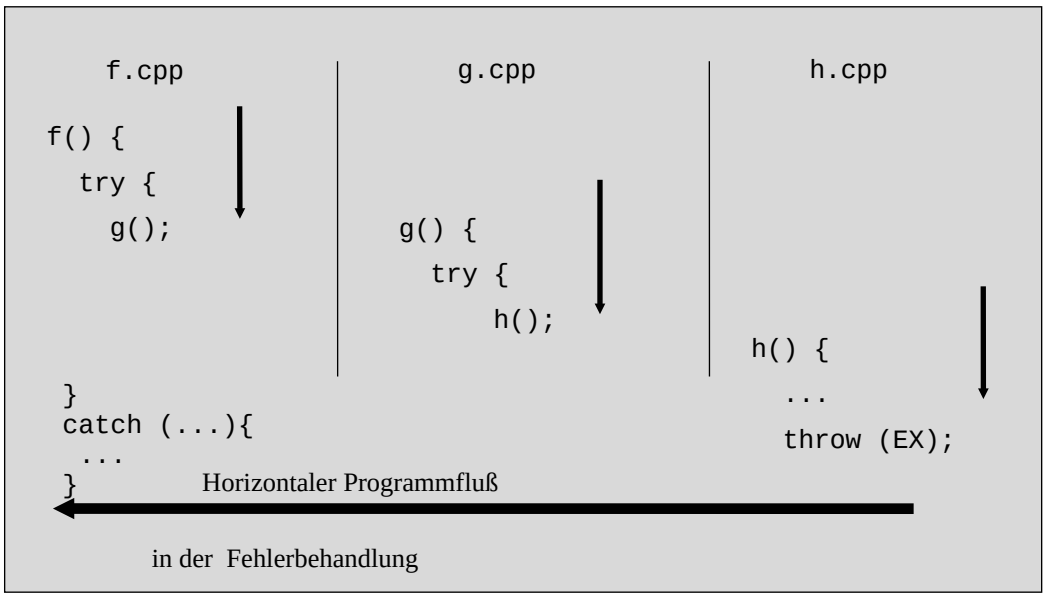


Abbildung 44 Horizontaler Programmfluß

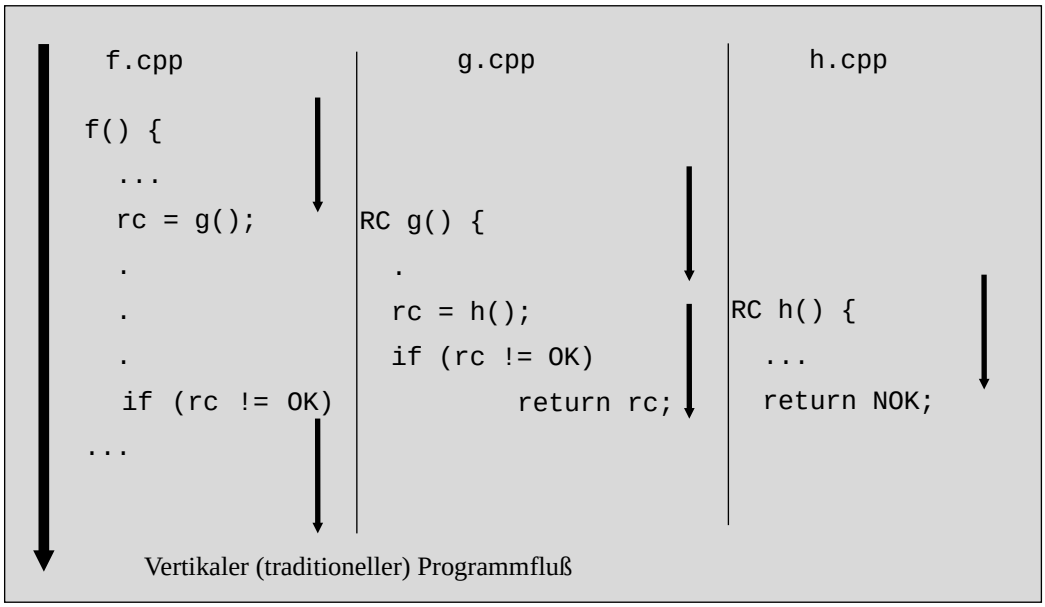


Abbildung 45 Vertikaler Programmfluß

```
/* Ausschnitt aus r_knde_main.c */
try {
        try
            // Exception wenn Fehler beim Registrieren auftritt !
            brok_register (&bentry);
        }
        catch (FAT_EXCEPT) {
            LOG ("\n[KNDE_SVC]: cant connect to broker.\n");
            exit (-1);
        }
        end_try;

        LOG ("\n[KNDE_SVC]: running.\n");
        // Hauptprogramm
        rpcgen_main ();
    }
    catch (FAT_EXCEPT) {
            brok_unregister (bent_get_svcn(&bentry)); // Am Broker abmelden
            try {
                dbms_close (); // versuche Datenbanken zu schließen
            }
            catch (FAT_EXCEPT) {
                LOG ("\n[KNDE_SVC]: cant shut down DB. (Sorry).");
            }
            end_try;
    }
    end_try;
```

Dieser Ausschnitt aus dem Kunden-Server-Hauptprogramm zeigt sehr deutlich den Nutzen des Exceptionhandlings: Wenn in beliebigen Modulen des Programms fatale Fehler auftreten, ist trotzdem eine Abmeldung am Broker und ein Schließen der Datenbank gewährleistet. Dieses Verhalten kann zwar auch über Returncodes realisiert werden, setzt aber voraus, daß jede Funktion die Returncodes der aufgerufenen Funktionen korrekt behandelt bzw. weiterreicht. Dieses „weiterreichen“ führt zu einem „codebloat“ und damit zu schwer

lesbaren Programmen, da nach jedem Funktionsaufruf, auch wenn keine fachlichen Fehler auftreten können, Code für das Weiterreichen von Ausnahmen geschrieben werden muß. Die Praxis zeigt, daß diese Forderung illusorisch ist, da ein Codeteil der keine korrekte Fehler/Exception Behandlung durchführt, die Kette unterbrechen kann. Überdies kann ein Exceptionhandling bei üblichen Stackfehlern zumeist das Programm noch korrekt beenden, da der Stack bis zum letzten try-Block abgeräumt wird. Typische Stackfehler wie die +-1 Fehlerklasse können so oft noch erkannt werden.

```
char s[3];
sprintf ( s, „ABC“); // Puffer s ist zu klein („ABC\0“ sind 4 Zeichen)
f(s);    // Hier wird eine inkosistenz erkannt -> Stack wird abgeräumt
```

9.10 Makefilekonzept

Selbst in der Zeit der visuellen GUI-Programmierumgebungen sind immer noch Makefiles der professionellste Ansatz große Projekte zu verwalten. Auch die Firma Microsoft, die mit ihren visuellen Entwicklungswerkzeugen den Anwender von der lästigen Tätigkeit Makefiles zu schreiben fernhalten will, verwendet intern einem Makefilemechanismus. So sind alle Beispiele des Windows (NT) SDK (Software Development Kit) nicht als Projektdateien vorhanden, sondern als Makefiles. Der Grund liegt klar auf der Hand: Kann doch mit Hilfe von Makefiles komplett nachvollzogen werden, wie Programme compiliert und gelinkt werden. Bei komplexeren Programmen mit vielen Bibliotheken, wo möglicherweise die Reihenfolge in der Linkeranweisung eine Rolle spielt, sind Makefiles immer noch erste Wahl. Im ROSALINDE-Projekt kam für uns sowieso keine andere Möglichkeit in Frage, da es für Linux keine professionellen, visuellen Entwicklungswerkzeuge gibt.

Es gibt im ROSALINDE-Projekt ein Haupt-Makefile („Makefile“) das alle anderen mit einbindet (inkludiert). Mit einem „`make all`“ kann das gesamte Projekt übersetzt werden. Eine Plattformunabhängigkeit ist in beschränktem Maße gewährleistet durch die Environment – Datei *env.mak*. In *env.mak* stehen die Sufix-Rules für implizites Übersetzten. Sufix-Rules sind Regeln wie eine gesuchte Datei mithilfe eines Kommandos aus einer anderen Datei erzeugt werden kann. Die Dateierkennung geschieht über die Dateierweiterungen, den Extensions (z.B. .o, .c, .x). Um die ROSALINDE-Anwendung beispielsweise auch auf Solaris oder HP-UX übersetzen zu können, ist das Linken zusätzlicher Laufzeitbibliotheken nötig (*libnls.a*). Die dazu nötigen Anpassungen werden ebenfalls über die Environment-Datei *env.mak* gesteuert.

Eine grundlegende Eigenschaft der ROSALINDE-Anwendung - die Möglichkeit eine komplett lokale Programmversion zu erzeugen - spiegelt sich direkt in den Makefiles wieder (*lib.mak*).

```
#
# Ausschnitt aus lib.mak
#
BROK_RPC_CLNT_LIB =                                     \
           $(ROSA)/src/brok/bent.o                      \ Brokerentry
           $(ROSA)/src/brok/brok_clnt.o                 \ Client Stubs
           $(ROSA)/src/brok/r_brok_clnt.o               \ generierte Stubs
           $(ROSA)/src/brok/r_brok_xdr.o                  XDR Marshalling

BROK_RPC_SERVER_LIB =                                   \
           $(ROSA)/src/brok/brok.o                      \ Brokerfunktionalität
           $(ROSA)/src/brok/bent.o                      \ Brokerentry
           $(ROSA)/src/brok/r_brok.o                    \ Server Stubs
           $(ROSA)/src/brok/r_brok_xdr.o                \ XDR Marshalling
           $(ROSA)/src/brok/r_brok_svc.o                \ generierter Server
          $(ROSA)/src/brok/r_brok_svc_main.o            \ Server Hauptprogramm

BROK_LOCAL_LIB =                                        \
           $(ROSA)/src/brok/brok.o                      \ Brokerfunktionalität
           $(ROSA)/src/brok/bent.o                      \ Brokerentry
```

Mit der lokalen Bibliothek (BROK_LOCAL_LIB) ist es möglich die gesamte Brokerfunktionalität in einem lokalen Programm zu nutzen; hier finden keine RPC-Aufrufe statt. Durch Austausch der lokalen Bibliothek durch die verteilte Client-Bibliothek (BROK_RPC_CLNT_LIB) ist es schließlich möglich, durch einfaches neulinken, eine Programmversion zu erzeugen, die per RPC an einem entfernten Broker Funktionen aufruft. Dies ist im eigentlichen Programm unsichtbar. Allerdings muß in diesem Fall ein RPC-Server vorhanden sein, der über die Bibliothek BROK_RPC_SERVER_LIB bereitgestellt wird.

9.11 Verbesserungsmöglichkeiten

9.11.1 Forward-Deklarationen

Momentan wird in der Schnittstellenheaderdatei über ein Makro zwischen Implementierung und Client-Schnittstelle umgeschalten:

```
#ifndef KUNDE_PROXY
typedef struct {
      ...
}
#else
typedef void knde;
#endif
```

Ein wesentlich besseres Konzept zur Definition von Proxys ist jedoch die komplette Entnahme der Strukturdeklaration aus der Headerdatei. Da bei abstrakten Datentypen sowieso nur mit Zeigern auf Strukturen gearbeitet wird ist eine reine Forwärtsdeklaration in der Headerdatei möglich. Dies hat zur Folge, daß auch nur Zeiger auf diese Struktur verwendet werden können und die Unterscheidung zwischen Proxy und Implementierung komplett aus der Headerdatei verbannt wird, da dieses Wissen in der jeweiligen Implementierung steckt:

Dies kann folgendermaßen aussehen:

knde.h (Schnittstelle)	knde.c (Implementierung)

`typedef struct knde_tag knde;`
 	
`knde * knde_new ();`
	`typedef struct knde_tag {`
	
	`char sName[256]; ...`

```
    void knde_delete (knde *);
```

```
    } knde;
```

Jetzt führt der direkte Datenzugriff von außen automatisch zu einem Compilerfehler.

knto.c

```
#include „knde.h“

knde * k = knde_new ();

strcpy (k->sName, „Kunde1“);
```

Außerdem kann für den Nutzer des ADTs die Implementierung unbemerkt geändert werden. So kann die Implementierung durch einen Proxy ersetzt werden:

knde.h

```
typedef struct knde_tag knde;

knde * knde_new ();

void knde_delete (knde *);
```

r_knde_clnt.c (Client-Stub)

```
typedef struct knde_tag {

    long proxy_oid; ...

} knde;
```

Mit diesem Konzept ist man sogar der Programmiersprache C++ überlegen, da es zwar möglich ist Forward-Deklarationen für Klassen anzugeben, die Methoden der Klassen aber dann ebenfalls für den Benutzer unsichtbar sind. Damit ist dieses Konzept in C++ unbrauchbar.

Durch den einfachen Trick der Forward-Deklaration löst man auf einen Schlag mehrere Probleme:

☺ Eine optimale Datenkapselung wird erreicht.

☺ Es können keine uninitialisierten Stackobjekte angelegt werden.

☺ Die Austauschbarkeit der Implementierung ist sichergestellt (Client-Stub -> Server-Implementierung).

☺ Es können keine fehlerhaften Stukturdefinitionen, verursacht durch fehlendes Proxymakro, auftreten.

Allerdings hat das Konzept der Forward-Deklaration auch Nachteile:

☹ Alle Objekte liegen auf dem Heap. Dies kann bei Programmen mit vielen „kleinen“ Objekten zu schweren Performanceproblemen führen.

☹ Verursacht durch die strikte Kapselung, müssen für jeden Zugriff auf die internen Daten eines Objektes, Methoden geschrieben werden. Diese Funktionsaufrufe kosten ebenfalls zusätzliche Rechenzeit und können in extremen Fällen einen hohen Prozentsatz des Quellcodes ausmachen was nicht gerade zur Lesbarkeit und Wartbarkeit des Programmcodes beiträgt.

9.11.2 Singleton Objekte

Als Singleton Objekte werden Objekte bezeichnet, die systemweit genau einmal vorkommen. Beispielsweise handelt es sich bei der Client-Tabelle (CTAB) am Server um ein Singleton Objekt, da genau eine Client-Tabelle pro Server existiert.

Die derzeitige Implementierung stellt bei Singleton Objekten zwar sicher, daß nur eine einzige Instanz angelegt wird. Die Schnittstelle unterscheidet sich allerdings von der „normaler“ ADTs (siehe 9.1.1). Die Objekttypen wie OTAB, CTAB sind zwar Singleton Objekte, die Realisierung entspricht aber nicht dem von Gamma, Helm, Johnson und Vlissides im Buch Design Patterns spezifizierten Baumuster.[6] Dieses beschreibt Singleton-Objekte in der Sprache C++. Es ist jedoch leicht möglich dieses in C zu implementieren:

```
/* ston.c */
typedef struct ston
{
    ...
};
```

Bei diesem Beispiel handelt sich es um einen ADT mit dem Namen „ston“. Die Strukturdefinition erfolgt in der C-Quelldatei (ston.c). Damit Zeiger auf diese Struktur von „außen“ verwendet werden können, wird mittels Forward-Deklaration der Name bekannt gegeben (siehe 9.11.1).

```
/* ston.h */
typedef struct singleton ston;
```

Damit nicht beliebig viele Objekte dieses Typs über `ston_new()` angelegt werden können, werden die Funktionen `ston_new()` und `ston_delete()` als statisch (`static`) implementiert. In der Headerdatei sind diese Funktionen **nicht** deklariert und daher von außen unsichtbar (internal linkage).

```
/*
 * ston_new() ist nicht von außen aufrufbar.
 */
static ston * ston_new ()
{
    /* wie gewohnt */
}

static void ston_delete(ston *)
{
    /* Wie gewohnt */
}
```

Damit trotzdem das Singleton-Objekt erzeugt werden kann, gibt es eine Funktion `ston_create()` welche die einzige Instanz anlegt und bei mehrmaligem Aufruf einen Zeiger auf dieses Objekt zurückgibt:

```
/*
 * Die statische Variable pSingletonObject speichert den Zeiger auf die
 * einzige Objektinstanz.
 */
static ston * pSingletonObject = NULL;

/*
 * ston_create () - erzeugt die einzige Instanz beim allerersten Aufruf.
 */
ston * ston_create ()
{
    if (! pSingletonObject) {
        pSingletonObject = ston_new ();
        ASSERT(pSingletonObject);
    }
```

```
    return pSingletonObject;
}
```

Durch die Forward-Deklaration des ADTs „`ston`“ wird sichergestellt, daß Objektinstanzen nur über `ston_create()` erzeugt werden können. Es können keine Stackvariablen des Typs „`ston`“ außerhalb von `ston.c` erzeugt werden. Der große Vorteil gegenüber der jetzigen Implementierung ist, daß sich ein solcher Singleton-ADT nach der Erzeugung nicht von der Verwendung eines „normalen“ ADTs unterscheidet.

9.12 Ein einfaches Beispiel

Im folgenden soll mit einem einfachen Beispiel die Realisierung eines abstrakten Datentyp gezeigt werden, der das ROSALINDE-Framework nutzt. Zuerst wird der abstrakte Datentyp vollständig lokal implementiert:

```
#ifndef _ADT_H_
#define _ADT_H_

/* Mit diesem Namen registriert sich der Server am Broker */
#define ADT_SVC "MY_ADT"

#ifndef ADT_PROXY

/* Die Implementierung des Servers */
typedef struct _der_ADT {
   char der_Name[255];
   /* ... */
} adt;

#else

typedef void adt;  /* Der Proxy des Clients */

#endif

    /* Allokation und Initialisierung */
adt * adt_new ();

    /* Loeschen */
adt * adt_delete (adt * this);

    /* set Methoden */
void   adt_set_name  (adt * this, char * name);

    /* get Methoden */
char * adt_get_name    (adt * this);

    /* lokal am Bildschirm drucken */
void adt_print (adt * this);

#endif /* _ADT_H_ */
```

Die Unterschiede zu einer lokalen Version dieses ADTs sind hervorgehoben. Die dazugehörige Implementierung ist trivial. Um nun diesen ADT verteilen zu können, muß eine Protokolldatei „`r_adt.x`“ geschrieben werden.

```
/* Generische Uebergabestruktur für alle Funktionen */

struct r_adt  {
    long oid;                           /* Die OID des Proxys */
    string cid      <MAX_CIDLENGTH>;    /* Die ClientID CID   */

        /* Vereinigungsmenge aller Aufrufparameter (hier nur der Name)*/
    string name     <MAX_KUNDENNAME>;
};

program R_ADT {
    version R_ADT_VERSION {
      radt ADT_NEW (r_adt)            = 1;
      radt ADT_DELETE (r_adt)         = 2;
      radt ADT_SET_NAME (r_adt)       = 3;
      radt ADT_GET_NAME (r_adt)       = 4;
} = 1;           /* Versionsnummer */
} = 0x32132484;  /* eindeutige Nummer */
```

Aus der Protokolldatei werden nun mit dem Werkzeug „`rpcgen`“ die nötigen Stubs für den SUN-RPC generiert. „`rpcgen`“ generiert außerdem das Serverhauptprogramm mit Dispatcher und die nötigen Marshallingfunktionen.

Die folgenden Server-Funktionen müssen zusätzlich implementiert werden (`r_adt.c`):

```
r_adt * adt_new_1_svc (r_adt *argp, struct svc_req *rqstp);
r_adt * adt_delete_1_svc (r_adt *argp, struct svc_req *rqstp);
r_adt * adt_get_name_1_svc (r_adt *argp, struct svc_req *rqstp);
r_adt * adt_set_name_1_svc (r_adt *argp, struct svc_req *rqstp);
```

Diese Funktionsrümpfe dieser Server-Stub-Stubs lassen sich allerdings mit der Option „-s“ des rpcgen-Generatorprogramms mitgenerieren.

```
// Server-Stub
r_adt * adt_new_1_svc(r_adt *argp, struct svc_req *rqstp) {
    static r_adt  result;
    adt * p;

    p = adt_new ();  // Erzeugung des Objekts (lokale Implementierung)
    Assert (p);
    // Registrierung des Objektes in der Objekttabelle des Clients
    ctab_register (argp->cid, &result.oid, NULL, k, NULL, NULL, NULL);

    return (&result);
}
```

Wie schon im Kapitel 6.4.1 beschrieben nimmt die new() – Methode eine Sonderstellung ein. Nach der Erzeugung des Objektes muß dieses in die Objekttabelle eingetragen werden, das geschieht mit `ctab_register()`. Dort wird die OID vergeben, die nachher als Rückgabeparameter zum Client zurückgesendet wird. Alle anderen Funktionen unterscheiden sich nur insofern, daß sie die erhaltene OID wieder zum Objekt expandieren (`ctab_get_obj()`) und anschließend die eigentliche Implementierung aufrufen. Bei einem Löschen des Objektes muß dieses natürlich ebenfalls aus der Objekttabelle entfernt werden (`ctab_unregister()`).

```
// Server-Stub
r_adt * adt_delete_1_svc(rknd *argp, struct svc_req *rqstp) {

    static r_adt result;
    static  * k;
    // Objekt über die OID holen
    k = ctab_get_obj (argp->cid, argp->oid);
    // TODO Fehlerbehandlung

    ctab_unregister (argp->cid, argp->oid);
```

```
    adt_delete (k);

    return &result;
}
```

Als weiterer Schritt müssen noch die Client-Stub-Stubs geschrieben werden, die den eigentlichen RPC-Aufruf vor dem Anwendungsprogramm verbergen. Die Client-Stub-Stubs haben die identische Signatur wie die Implementierung am Server:

```
// Client-Stub
adt * adt_new () {

    r_adt * pr;
    r_adt r;
    CLIENT * cl;

    cl = sadm_get_serverhandle (ADT_SVC);
    Assert (cl);
    // client ID mitübergeben
    r.cid = cmgr_get_cid ();

    pr = adt_new_1 (&r, cl); // Der generierte RPC Aufruf

    return (adt *) pr->oid;  // Der „ROSALINDE CAST“
}
```

RPC ⟶

Über den Server-Administrator kann der Client einen gültigen RPC-Verbindungshandle anfordern. Dieser dient dann als zweiter Parameter im RPC-Aufruf. Der erste Parameter ist die generische Übergabestruktur in der alle Funktionsparameter „verpackt“ werden. Zusätzlich wird die OID und die CID mitgegeben. Der Zeiger auf den ADT ist tatsächlich nur eine Variable in der die

OID des Serverobjektes gehalten wird. Da am Client nie direkt auf diesen Zeiger zugegriffen wird, ist das möglich.

Jetzt fehlt noch das Hauptprogramm des Servers damit sich dieser am Broker registrieren kann. Dazu wird das generierte Serverhauptprogramm main() in rpcgen_main() umbenannt und von einem selbstgeschriebenen Hauptprogramm (main()) aufgerufen:

```
// SERVER
ENABLE_ERROR_HANDLING;

main()
{
    bent bentry = { "localhost", ADT_SVC, "tcp", R_ADT, R_ADT_VERSION };

    try {
        ctab_init ();

        /* Am Broker registrieren */
        brok_register (&bentry);
        rpcgen_main (); // Das generierte Hauptprogramm
    }
    catch (FAT_EXCEPT) {
        brok_unregister (bent_get_svcn(&bentry));
        ctab_clear ();
    }
    end_try;
}
```

Es ist äußerst wichtig, daß der Server seine Clienttabelle vor dem ersten Zugriff mit ctab_init() initialisiert. Ebenso muß sich der Server am Broker registrieren. Erst anschließend darf das generierte Serverhauptprogramm aufgerufen werden.

Am Client muß der Server-Administrator mit dem Hostnamen oder der TCP-IP Netzwerkadresse des Brokers initialisiert werden:

```
// CLIENT

ENABLE_ERROR_HANDLING;

main ()
{
    adt * p;
    sadm_init (brokerhost);

    p = adt_new ();
    adt_set_name (p, „test“);
    ...
    adt_delete(p);
}
```

Jetzt kann der ADT von einem entfernten Client vollständig transparent genutzt werden. Der Aufwand erscheint einem im ersten Moment als außerordentlich hoch, ist aber durchaus mit dem anderer Middleware wie beispielsweise DCOM zu vergleichen. Man muß bedenken, daß sich ein großer Teil dieses Quellcodes auch über Generatoren automatisch erstellen läßt.

10 Ausblick: Verteilte Anwendungen in C++

10.1 Vorteile von C++ / Nachteile von C

10.1.1 Stackobjekte

In prozeduralen Programmiersprachen wie C ist es nicht möglich Konstruktoren und Destruktoren zu spezifizieren, daher können verteilte Objekte nur über explizite Funktionsaufrufe wie knde_new() bzw. knde_delete() erzeugt und gelöscht werden. Oftmals sind aber auch lokale Stackobjekte sinnvoll, die automatisch beim Verlassen des Gültigkeitsbereichs gelöscht werden. In C ist es aber nicht möglich diese Objekte automatisch zu initialisieren. So kann es vorkommen, daß ein Objektproxy uninitialisiert bleibt. Die ROSALINDE-Anwendung erzwingt ein explizites Erzeugen von Objekten durch folgende Konstruktion:

```
typedef void knde;
```

Was zuerst völlig unsinnig aussieht hat durchaus seinen Sinn. Durch „`typedef void`“ kann erreicht werden, daß nur Zeiger auf Kundenobjekte verwendet werden. Stackvariable können dann nicht mehr angelegt werden.

```
knde k;                       // NOK: Fehler beim Compilieren
knde * k = knde_new ();       // OK
```

Dieser Trick hat aber seinen Preis. Alle Zeiger auf Anwendungsklassen sind `void` – Pointer (void *) und damit werden alle Typüberprüfungen seitens des Compilers ausgeschaltet. Dieses Vorgehen ist extrem gefährlich, da folgende Konstrukte möglich werden:

```
knde * ku = knde_new ();      // OK
knto * ko = ku;               // OK
knto_find (ko, 4711);         // OK
```

Dieses Beispiel zeigt, daß gefährliche Typecasts nicht erkannt werden und die Vorteile der statischen Bindung dahin sind.

Wie im Kapitel 9.11.1 schon ausgeführt, können diese Probleme mit einer Forward-Deklaration gelöst werden. Damit wird das Konstruktor/Destruktorproblem für Stackobjekte zwar verhindert, allerdings liegen dann alle Objekte am Heap, die Performance verschlechtert sich und die Speicherverwaltung verkompliziert sich. Daher ist es sinnvoll C++ zu verwenden und in den Konstruktoren bzw. Destruktoren der Anwendungsklassen bzw. Basisklassen die Erzeugung bzw. Destruktion der Proxys bzw. Objekte vorzunehmen.

```
// C++
{
   Kunde ks;                        // OK -> Proxy wird initialisiert !
   Kunde * k  = new Kunde ();       // OK -> Proxy wird initialisiert !
   Konto * ko = k;                  // NOK -> Fehler wird erkannt !
   delete k;                        // OK -> Objekt wird am Server gelöscht
}  // ks wird automatisch beim Verlassen des Blocks gelöscht.
```

10.1.2 Vererbung

In vielen Bereichen der Anwendungsentwicklung wird Vererbung als Design- und Implementierungskonzept eingesetzt. Daher ist es wünschenswert wenn diese Vererbungs-beziehungen bei verteilten Objekten erhalten bleiben und transparent für den Client sind.

Die Architektur der Rosalinde-Anwendung sieht vor, daß jeder Server genau einen ADT implementiert. Falls Vererbung eingesetzt wird, ist diese Einschränkung nicht mehr sinnvoll, da beispielsweise ein Kundenserver, Auslands- und Inlandskunden bearbeiten muß. Da der SUN-RPC nicht in der Lage ist, Interfaces mehrerer ADTs zu unterstützen, wird in den folgenden Beispielen der Microsoft (DCE) RPC verwendet.

Bei einer Vererbungshierarchie muß die Erstellung des Client-Proxys in einer gemeinsamen Basisklasse realisiert werden. Von dieser müssen alle verteilten Objekte abgeleitet werden. Dies hat den Grund, daß bei einer Vererbungssituation die Konstrukttoren aller geerbten Klassen implizit aufgerufen werden. Falls die Erzeugung im Konstuktor der jeweiligen Klasse vorgenommen würde, hätte dies zur Folge, daß mehrfach Objekte am Server angelegt werden. Der Konstruktor der Basisklasse übernimmt die Konstruktion eines Objektes über seine Klassen-ID am Server. Diese Basisklasse wird bei CORBA als BOA (Basic-Object-Adapter) bezeichnet. Der Basic-Object-Adapter verwendet folgende RPC-Schnittstelle:

```
interface BasicObjectAdapter
{
      typedef long OID;

         Erzeugt Objektinstanz am Server über eine Klassen-ID
         und liefert die zugehörige OID zurück.
```

```
      OID   BOA_create  ([in, string] char * Klassen-ID);

         Löscht Objektinstanz am Server über die Objekt-ID
      void  BOA_destroy ([in] OID o);
}
```

create() – erzeugt ein Serverobjekt über eine eindeutige Klassen-ID in Stringform und gibt dessen OID zurück.

destroy() – löscht das Serverobjekt über dessen Objekt-ID.

Am Client wird der BOA benutzt um das Serverobjekt zu erzeugen und die Objekt-ID zu speichern (Proxy).

```
//------------------------------------------------------------------
// CLIENT: Der Basisklassenkonstruktor erzeugt das Objekt am Server
// und speichert die OID (Proxy)
//------------------------------------------------------------------
BasicObjectAdaptor::BasicObjectAdaptor (ClassId & clsid)
{
    oid = BOA_create (clsid.AsString());  // --> RPC
}
```

Die Serverimplementierung von `BOA_create` erzeugt das Serverobjekt über eine Factory:

```
//------------------------------------------------------------------
// Server: Die Server-Implementierung der RPC-Funktion erzeugt das Objekt
//------------------------------------------------------------------
OID BOA_create (unsigned char * die_Klassen_ID)
{
      Abstract_Factory * paf;

      paf = FindFactory(die_Klassen_ID);

      BOA * p = paf->create ();

      return (OID) p;
}
```

Bei der obigen Implementierung wird die Hauptspeicheradresse des Serverobjektes direkt als OID verwendet. Die Nachteile dieses Konzeptes wurden bereits im Kapitel besprochen.

Wie wird jetzt aus der Klassen-ID ein Objekt? In C++ sind Konstrukte wie new(„Kunde“) nicht möglich. Daher verwendet „FindFactory“ eine Tabelle in der zur Laufzeit neue Factorys eingetragen und über die zugehörige Klassen-ID gesucht werden können. Jede Factory hat eine „create()“-Methode, die ein Objekt erzeugt und einen Zeiger auf dieses zurückgibt. Alle Factorys erben von der zentralen Basisklasse „Abstract_Factory“.

```
class Abstract_Factory
{
public:
    Abstract_Factory (char * clsid);
        Trägt neue Fabrik über die clsid in die globale  Liste ein

    virtual BOA * create () = 0;     virtueller Konstruktor
};
```

Auch die jeweilige Fabrik kann generisch erzeugt werden:

```
template <class T>
class Concrete_Factory : public Abstract_Factory
{
public:
      // Erzeuge Fabrik und registriere die Fabrik unter clsid
      Concrete_Factory (char * clsid) : Abstract_Factory (clsid) {}

      // Erzeuge Objekt
      BOA * create () { return new T; }
};
```

Mit dieser Technik muß nur eine Instanz für die jeweilige Klasse als statische Variable erzeugt werden:

```
static Concrete_Factory<Kunde>(CLSID_KUNDE);
```

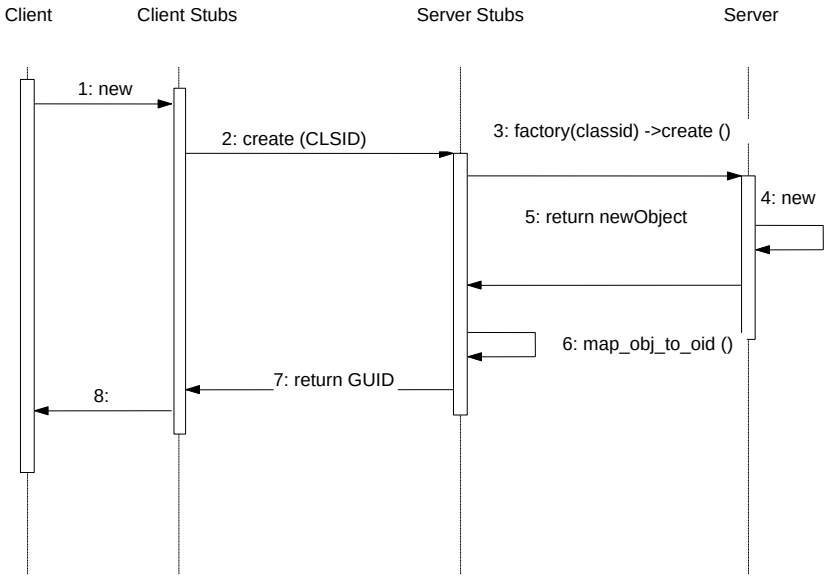


Abbildung 46 C++ Factory

Abbildung 46 zeigt noch einmal den Ablauf der Objekterzeugung . Der Client erzeugt eine neue Instanz eines verteilten Objektes (1). Der Basisklassenkonstruktor wird implizit aufgerufen, da alle verteilten Objekte von diesem erben. Dort wird der generische „create“-Aufruf durchgeführt. Dieser benötigt die Klassen-ID als Parameter, um ein geeignetes Objekt am Server zu erzeugen (2). Jetzt wird am Server der Serverstub ausgeführt und die geeignete Factory für übergebene Klassen-ID gesucht. Über diese Factory kann schließlich das Serverobjekt am Heap angelegt und in der Objekttabelle eingetragen werden (6). Anschließend wird die OID (GUID: general unique identifier) zurückgegeben (7). Der Client speichert diese OID in der Basisklasse später auf das Serverobjekt zugreifen zu können (Proxy).

Mit diesen Konzepten ist es zum einen möglich Objekte zu erzeugen, die zum Übersetzungszeitpunkt dieses Codes noch nicht bekannt sind. Ebenso kann ein

Server mehrere Klassen implementieren, was bei Vererbung zwingend notwendig ist.

10.1.3 Polymorphie

Als Polymorphie (dt. Vielgestaltigkeit) bezeichnet man die Eigenschaft, daß ein abgeleiteter Typ direkt durch die Oberklasse (Supertyp) ersetzt werden kann. Daher kann ein Zeiger auf einen Kunden auch auf einen Auslands- oder Inlandskunden zeigen. Man unterscheidet in typorientierten Programmiersprachen zwischen statischem und dynamischen Typ. Der statische Typ ist zur Compilezeit bereits bekannt. Der dynamische Typ ergibt sich erst zur Laufzeit. So kann ein Zeiger auf einen Kunden grundsätzlich auf jede davon abgeleitete Klasse zeigen, was sich erst durch den Programmablauf ergeben kann.

Folgendes Beispiel soll dies illustrieren:

Eine Kundenklasse implementiert eine Methode (z.B. print()), die von einer abgeleiteten Klasse (z.B. Auslandskunde) überschrieben wird. Hier muß sichergestellt werden, daß die jeweils richtige „print“-Methode aufgerufen wird:

```
knde * k = GetAuslandKunde();
k->print (); // k ist ein Auslandskunde -> ((Auslandskunde *) k)->print();
```

Der genaue Typ kann jedoch nur zu Laufzeit bestimmt werden. Dies wird in C++ über eine VMT (virtual method table) realisiert. Wichtig ist, daß die Polymophieeigenschaft ebenfalls bei einem verteilten Objekt erhalten bleibt. Daher findet sich am Client wie auch am Server die gleiche Vererbungshierarchie. Nur die jeweiligen Implementierungen bzw. Stubs unterscheiden sich.

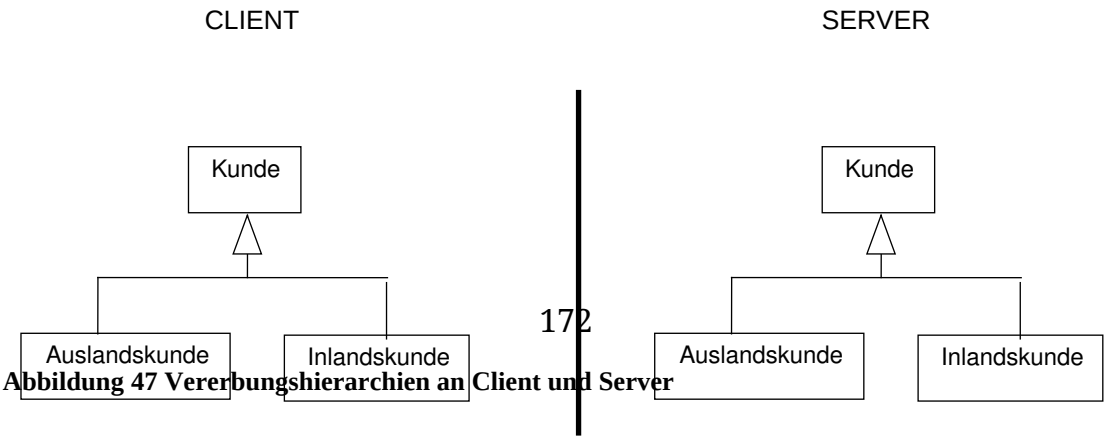


Abbildung 47 Vererbungshierarchien an Client und Server

10.2 Weitergehende Überlegungen

Wie gezeigt wurde, ist eine Objektverteilung auch in C++ transparent möglich. Die Ideen zu den erläuterten Beispielen sind im wesentlichen der CORBA-Spezifikation entnommen. Interessant wäre eine Referenzimplementierung des Rosalinde-Projektes mit C++ und dem DCE-RPC. Es läßt sich zwar abschätzen, daß fast alle im Rosalinde-Projekt realisierten Konzepte in ähnlicher Weise gebaut werden müßten, trotzdem besteht eine gewisse Notwendigkeit dafür: Der UNIX-Markt wird in den nächsten Jahren weiter abnehmen und der beim ROSALINDE-Projekt verwendete SUN-RPC wird ebenfalls vom DCE-RPC verdrängt. Daher wäre eine Weiterentwicklung des Rosalinde-Frameworks für den DCE-RPC wünschenswert. Vielleicht ergeben sich weitere Diplomarbeiten zu diesem Thema.

Anmerkung

Die Erfahrungen welche durch die Arbeit am ROSALINDE-Projekt gewonnen wurden sind in gewisser Weise unbezahlbar. Durch den genauen Fertigstellungstermin (Juli 1996) war dem gesamten Projekt ein zeitlich begrenzter Rahmen abgesteckt. Mit viel Wochenendarbeit schafften wir die Implementierung anläßlich der 10-Jahresfeier des Fachbereichs Informatik an der FH-Rosenheim fertigzustellen. Die Vorführungen waren ein großer Erfolg. Trotz des modellhaften Charakters des Anwendungsbeispiels, waren wir mit den wesentlichen Problemen konfrontiert, die eine Client/Server-Softwareentwicklung im großen Stil mit sich bringt. Diese Diplomarbeit ist letztendlich eine Dokumentation dieser Erfahrungen.

Johannes Weigend

„nothing can survive in a vacuum“ (rush – hold your fire)

Tabellarischer Vergleich

Tabelle 4 Tabellarischer Vergleich

	CORBA	DCOM	ROSALINDE
Generierte Stubs	Ja	Ja	teils
Dynamic Invocation	Ja	Ja	Nein
Proxys	Ja	Ja	Ja
Vererbung	Voll	Nur Schnittstelle	Nein
Kapselung	Ja	Ja	Ja
Polymorphie	Ja	Nein	Nein
Broker	Ja	Nein / Nur innerhalb der eigenen Domäne	Ja

Mehrfache Interfaces	Ja	Ja	Nein
RPC	UNO	DCE-RPC	ONC-RPC
IDL	IDL	DCE-IDL	ONC-RPCL

Literaturverzeichnis

[1] Bloomer, John. Power Programming with RPC. O´Reilly: 1992

[2] Brown, Chris. Programmieren veteilter UNIX-Anwendungen. Prentice Hall: 1994

[3] Curry, David A. Using C on the UNIX System. O´Reilly: 1991

[4] Chappell, David. ActiveX und OLE verstehen. Microsoft Press: 1996

[5] Coplien, James O. Advanced C++. Addison-Wesley: 1994

[6] Gamma, Helm, Johnson, Vlissides. Design Patterns. Addison-Wesley: 1994

[7] Goodheart, Cox. Unix System V Release 4. Prentice Hall 1994

[8] Gosling, Arnold: The Java Programming Language. O´Reilly: 1996

[9] Gilly, Daniel. UNIX in a Nutshell. O´Reilly: 1994

[10] Maguire, Steve. Nie wieder Bugs !. Microsoft Press: 1993

[11] Mayers, Scott. More Effective C++. Addison-Wesley: 1996

[12] Murray, Robert B.. C++ Strategies and Tactics. Addison-Wesley: 1993

[13] Musser, Saini. STL Tutorial and Reference Guide. Addison-Wesley: 1996

[14] Mittendorfer, Josef. Objektorientierte Programmierung mit Smalltalk/V für Windows..

[15] Niemeyer, Peck. Exploring Java. O´Reilly: 1996

[16] Oram, Talbott. Managing Projects with make. O´Reilly: 1993

[17] Peek, O´Reilly, Loukides. UNIX POWER TOOLS. O´Reilly: 1993

[18] Petkovic, Dusan. Informix 4.0/5.0. Addison-Wesley: 1993

[19] Petkovic, Dusan. SQL-die Datenbanksprache. Mc Graw Hill: 1990

[20] Rumbaugh, Blaha, Premerlani, Lorensen. Objektorientiertes Modellieren und Entwerfen. Hanser: 1992

[21] Rosenberry, Kenny, Fisher. Understanding DCE. O´Reilly: 1993

[22] Sedgewick, Robert. Algorithmen in C++. Addison-Wesley: 1992

[23] Shirley, Hu, Magid. Guide to Write DCE Applications. O´Reilly: 1994

[24] Shirley, Rosenberry. Microsoft RPC Programming Guide. O´Reilly: 1995

[25] Stroustrup, Bjarne: Design und Entwicklung von C++. Addison-Wesley: 1994

[26] Stroustrup, Bjarne. Die C++ Programmiersprache. Addison-Wesley: 1992

[27] Tannenbaum, Andrew S. Moderne Betriebssysteme. Hanser: 1992

Index