Rexx Language Bindings für CORBA 3.0.

Björn Muschall
Eingereicht am xx.12.00

Inhaltsverzeichnis

1.Einleitung!!!
 1.1 Aufgabenbeschreibung
 1.2 Aufbau der Arbeit

2. Die Architektur von Corba
 2.1 Die Bestandteile des Kerns
 2.2 Die Funktionsweise des Datenaustausches

3. Die Interface Definition Language

4. Die offiziellen Language Bindings
 4.1 C

4.2 C++

5. Die inoffiziellen Language Bindings
   5.1.Combat
   5.2.Rapid CorbaServer Development in Tcl
   5.3.weitere Language Bindings

6. Der Ablauf eines dynamischen Aufrufes

7. Language Binding für Rexx
   7.1 Rexx und IDL
   7.2 Das SAA–Api, Statische Umsetzung
   7.3 Pseudoobjekte, Dynamische Umsetzung

1.1 Die Aufgabenbeschreibung

Die amerikanische Standardisierungsorganisation "Object Management Group" (OMG) entwickelt seit Jahren objektorientierte Standards. Die "Common Object Request Architecture" (CORBA) erlaubt es verteilten Objekten über Systemgrenzen hinweg zu kommunizieren. Zentraler Bestandteil ist die Definition von Schnittstellen verteilter Objekte in einer Programmiersprachen übergreifenden Beschreibungssprache (IDL). Eine anschließende Implementierung der Architekur geschieht in einer Programmiersprache. Die OMG hat für eine Reihe von Programmiersprachen "Language Bindings" erarbeitet, die die Umsetzung der IDL–Definitionen und damit die Implementierung definieren (Ada, C, C++, COBOL, Java und Smalltalk)
Für eine Reihe weiterer Programmiersprachen bestehen zur Zeit noch keine offiziellen Language Binding.

Ziel der Aufgabe ist es, für die Sprache Rexx den Entwurf eines Language Binding vorzubereiten, und damit die Implementierung von Corba–Objekten in Rexx zu ermöglichen. Ein erster Schritt dazu ist die Erarbeitung der Architektur von Corba und weiterer Bestandteile, die in Hinblick auf das LanguageBinding zu betrachten sind. Ein weiterer Schritt ist das Auseinandersetzten mit bestehenden Language Bindings. Zuletzt wird der Entwurf eines solchen Language Bindings für Rexx erarbeitet.

1.2 Der Aufbau der Arbeit:

Die Architektur von Corba wird erarbeitet. Corba selber ist eine Infrastruktur zur Realisierung verteilter Systeme. Wie diese Infrastruktur aussieht, wird im ersten Kapitel der Arbeit betrachtet.

Verteilte Systeme kommunizieren mittels Austausch von Nachrichten. Für diesen Nachrichtenaustausch ist eine Schnittstelle für das Senden und Empfangen solcher Nachrichten notwendig. Dafür notwendig ist eine geeignete Datenübertragung und Protokolle. Durch die Definition der Schnittstellen in IDL und der Übersetzung in eine Programmiersprache, die auf einem bestehenden Language Mapping basiert, wird dieser Mechanismus für den Entwickler verteilter Objekte in Corba transparent gehalten. Dieser Mechanismus wird betrachtet. (Der Request, die Protokolle IIOP, GIOP)

Die Metadaten des Interface Repositories, die Typecodes und ihre Bedeutung für die Nachrichtenübermittlung wird genannt. Die Interoperable Objekt Referenz (IOR) ist Teil des Adressierungskonzeptes. Weiter unten werden die inoffiziellen Languagemappings für eine Reihe von Skriptsprachen untersucht. Allen Skriptsprachen fehlen mehr oder weniger die Merkmale einer stark typisierten Sprache wie bei IDL. Die Umsetzung von Methodenaufrufen erfordert Typinformationen. Dieser Abschnitt bereitet die Betrachtung der inoffiziellen Language Bindings für eine Reihe von Skriptsprachen vor.

Die Interface Definition Language wird betrachtet.

Die offiziellen Language Bindings bestehen für eine Reihe von Programmiersprachen. Vorgestellt werden das C,– und C++ Language Binding

Eine Reihe von inoffiziellen Language Mappings wird betrachtet. Die hier betrachteten Sprachen sind wie Rexx ebenfalls Skripsprachen. Im Gegensatz zu den obigen Programmiersprachen, weisen sie einige Besonderheiten auf. Die Ansätze sind teilweise in dynamische und statische zu Unterteilen.

Da einige der vorgestellten Lösungen auf dem dynamischen Generieren von Request basiert, und diese ein immer gleich ablaufendes Schema darstellen, soll hier ein typischer dynamischer Methodenaufruf gezeigt werden.

Die Möglichkeiten ein Language Binding für Rexx zu gestalten werden vorgestellt

2. Die Architektur von Corba

Corba ist eine Infrastruktur zur Realisierung verteilter Systeme. Die Bestandteile eines solchen Systems können über Prozessgrenzen, sogar über Rechnergrenzen hinweg verteilt

werden. Aus Sicht eines Entwicklers, der sich einer Programmiersprache bedient, um diese Bestandteile zu implementieren, läßt sich das System nicht mit den alleinigen Mitteln seiner Programmiersprache, wie Funktionen, Modulen, und Aufrufbeziehungen realisieren. Mittel und Wege über Programmiersprachengrenzen hinweg werden benötigt. Solche Möglichkeiten sind vielfach schon genutzt worden:

- Pipes und Dateien für die Inter–Prozeß– Kommunikation innerhalb eines Rechners
- Sockets und RPC für die Kommunikation über Rechnergrenzen hinweg

Ein Aspekt von Corba ist, daß die Bestandteile seiner Architektur in einheitlicher Sicht in Form von Objekten dargestellt werden. Eine grundlegende Eigenschaft dieser Objekte ist, daß sie über ihre Schnittstellen beschrieben werden, welche eine Abstraktion von ihrer eigentlichen Implementierung darstellt. Die Implementierung dieser Objekte kann dann in der Programmiersprache der Wahl erfolgen. Die Schnittstellenbeschreibung wird in einer einheitlichen Form, mittels einer descriptiven Beschreibungsprache spezifiziert, der Interface Definition Language (IDL). Diese läßt sich dann über die Language Bindings auf die jeweiligen Programmiersprachen abbilden

Ein weiterer Aspekt ist, daß diese Abbildung nicht auf direktem Wege geschieht. Zwar müssen die Konzepte die die IDL beinhaltet (Modellierungskonzepte wie Objekte und Vererbungsbeziehungen, das Typenkonzept einer streng typisierten Sprache, Namensraumstrukturierung etc...) ein entsprechendes Pondon in einer anderen Programmiersprache haben, oder sich zumindest mit anderen Mitteln abbilden lassen, intern wird ein Aufruf einer entfernten Methode aber mittels dem Versenden von Nachrichten realisiert, von denen jedoch abstrahiert wird, um dem Entwickler innerhalb seiner Programmiersprache eine transparente Sicht zu geben. Die Transparenz besteht darin, Methoden entfernter Objekte so aufzurufen, als wären sie lokale Objekte.

2.1 Die Bestandteile des Kerns (Corba Core)

Zum Kern von Corba wird das Objektmodell, der Objekt Request Broker (ORB), die Schnittstellen zum ORB, und die IDL gezählt. Die Schnittstellen unterteilen sich weiter in das ORB–interface, die statischen und dynmischen Interfaces, und den Objekt Adaptern

Objektmodell:
Im Objektmodell werden neben den Eigenschaften von Objekten, dies sind Operationen und Attribute, auch Schnittstellen, und Typen vereinbart. Dabei unterscheidet Corba grundsätzlich zwei verschieden Typengruppen, die einfachen und die konstruirten. Konstruierte sind solche, die sich wiederum aus einfachen zusammensetzten. Zu den Datentypen gehören auch die Objektreferenzen und darüberhinaus Exceptions und Datentypen wie der "Any" oder neuerlich der "Valuetype", die beliebige andere Typen aufnehmen können. Die Objektreferenzen unterteilen sich in domäneninterne und die übergreifenden, externe, interoperable Objektreferenz (IOR), die der eindeutigen Identifikation den Orb übergreifend, und damit weltweit dienen soll.

IDL
Interface Definition Language (kurz IDL), beinhaltet:
Die syntaktische Beschreibung der IDL
Die in IDL geschriebene Spezifikation der elementaren Schnittstellen
Die Language Mappings

Der Object Request Broker (ORB) und die ORB–Domäne
Die Gesamtheit aller Objektimplementierungen und deren Instanzen nennt man ORB–Domäne. Der Orb stellt den Backbone dieser Objekte und eine Sammlung von Vorschriften der Interaktion zwischen den Objekten dar. Es ist weder vorgeschrieben, noch wahrscheinlich, daß der Orb nur aus einer Komponente besteht, oder daß es sich um ein konkretes Objekt handelt, vielmehr ist er eine logische Struktur, die über viele Objekte einer Domäne verteilt ist. Er ist ein abstraktes Konstrukt. Jedoch gibt es in dem meisten Implementationen/ OrbProdukten ein Orbobjekt, in Form eines Pseudoobjektes.

Interfaces
Direkt an den Orb angeordnet und damit auch zum Kern von Corba gehörend, findet man die Interfaces. Diese lassen sich in statische und dynamische und die Orb–Interfaces unterteilen. Zugriffe auf entferne Objekte sollen aus Sicht des aufrufenden Objektes transparent sein, dafür benötigt man lokale Stellvertreter, deren Methoden die gleichen Signatur wie die entfernten Methoden haben, sogenannte Proxy Objekte. Zu jeder Implementierung eines entfernten Objektes muß es ein vorgeschaltenes, also eine korrespondierende Implementierung eines ProxyObjektes geben.

In Corba gibt es den Stub und den Skeleton, welche die statische Variante der Proxyobjekte darstellen. Die dynamische Variante ist mit dem DynamicInvocationInterface (DII) und für die Serverseite mit dem DynamicSkeletonInterface (DSI) vertreten. Das DII stellt die vollständige Funktionalität zur Verfügung, um entfernte Aufrufe dynamisch erzeugen und ausführen zu können. Durch das DSI können Anwendungen eingehende Anfragen dynamisch bearbeiten.

InterfaceRepository (IR),
Das IR ist ebenfalls an den Orb angegliedert und dient der Erfassung von Schnittstelleninformationen, auf die zur Laufzeit entfernt zugegriffen werden kann. Daher liegen diese in maschinenlesbarer, von der IDL–Definition direkt abgeleiteter und damit in genormter Form vor. Das Interface Repository ist wiederum aus CorbaObjekten aufgebaut/ realisisert. Es enthält alle mittels IDL spezifizierten Informationen, bietet auf Definitionen mittels Pseudointerfaces Zugriff. In diesem Interfacerepository werden Operationssignaturen und Vererbungshierarchien der Objekte transparent aufbereitet, aufgelöst.

Objekt Adapter
Der Object Adapter bietet eine Schnittstelle zum Kern des ORB, und damit zu zentralen Dienstleistungen wie das Erzeugen und Interpretieren von Objektreferenzen, das Aktivieren und Deaktivieren von entfernten Objekten. Entfernte Objekte sind meist in Form von Servants realisiert. Ein Server kann mehrere Servants beinhalten. Ein Objekt Adapter dient

der Registrierung von Servants und des Weiterleitens von Aufrufen an diese. Ein Servant ist der reale Stellvertreter eines Objektes. Das Objekt selber ist abstrakt, ein Servant ist die Implementierung eines Objektes in einer Programmiersprache. Ein Sevant muß bekannt sein, eine Identität haben. Diese verschafft ihm der Objekt Adapter.

CommonObjectServices (COS)
Die COS sind nicht mehr Bestandteile des Kerns, sondern auf diesen aufbauende zustätzliche Funktionalität. Sie sind gesammelte und definierte Standarddienste, die für die Entwicklung eine Vereinheitlichung bedeuten, und das Entwickeln erleichtern, da sie nicht ständig neu erfunden werden müssen. Jeder Dienst hat eine genormte Schnittstelle, kann auch mehrere Interfaces haben.
Einer dieser Dienste ist gerade in Hinblick auf den dynamischen Aufruf von Interesse:
Einer der zentralen Dienste, gerade auch im Hinblick auf die dynamischen Methodenaufrufe ist der NamingService, er dient als zentrales Verzeichnis für Objektreferenzen und verwaltet diese und soll das Auflösen von Referenzen eines Objectes auf eine anderes transparent machen.

NamingService
Aufgabe des NamingService ist es, beliebig viele CorbaObjekte unter frei wählbaren Namen zu erfassen. Ähnlich wie in einem Dateisystem ordnet er die Referenzen der Objekte hierarisch in Kontexten an. Kontexte sind dabei vergleichbar mit Verzeichnissen und die Adressen mit Dateien. Somit erlaubt er die Ermittlung aller anderen benötigten Objektreferenzen und stellt damit die erste initale Objektreferenz, die ein Objekt kennen muß, dar. Ein solcher Naming Service kann Kontexte unterschiedlicher Domänen enthalten.

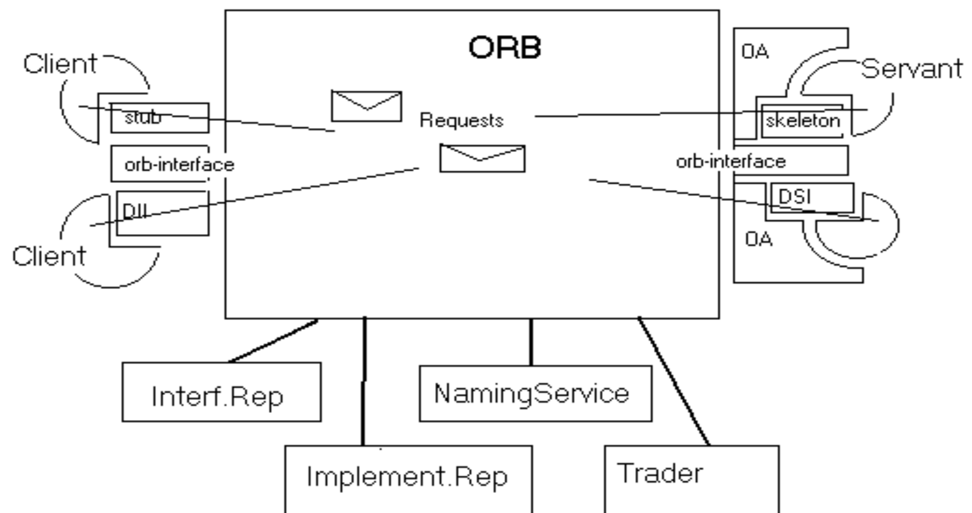

Abbildung1: Die Architektur des Orbs

2.2 Die Funktionsweise des Datenaustausches:

In Abbildung 1 wird die Architektur und Funktionsweise von Corba gezeigt. Auf ihr ist zu sehen, daß der Aufruf von Methoden in das Versenden von Requests resultiert.

Ein Designaspekt von Corba ist, dem Programmierer den Aufruf einer entfernten Operation genauso erscheinen zu lassen, als würde er die Methode eines lokalen Objektes aufrufen. Realisiert wird dies, indem transparent Nachrichten erzeugt werden. Diese nennen sich hier Request. Ein solcher Request wird erzeugt, und kann über System,– und Prozeßgrenzen hinaus verschickt werden. Für diesen Nachrichtenaustausch ist eine Schnittstelle für das Senden und Empfangen solcher Nachrichten notwendig. Ebenfalls notwendig ist eine geeignete Form der Datenübertragung und Protokolle.

Im Falle Corba lassen sich Requests mittels dem Protokoll IIOP bilden. Aus den Spezifikationen der OMG geht hervor, daß die Kommunikation innerhalb einer Domäne relativ frei implementierbar ist, die zwischen den Domänen aber streng definiert. Interaktionen über Domänengrenzen hinaus werden durch das IIOP Protokoll beschrieben. Innerhalb eines ORB kann jedoch ein beliebiges Protokoll implementiert sein, solange gewährleistet ist, daß dieses nach außen über eine sogenannten Bridge in das IIOP Protokoll übersetzt werden kann.

Dies bedeutet, daß der Entwickler irgendwann an den Punkt gelangt, an dem er seine Methodenaufrufe in die Form von IIOP Requests wandeln muß, oder eine Möglichkeit findet, diese Arbeit, die für andere Sprachen bereits geschehen sind, einzubinden. Er muß damit verbunden eine Reihe von Marshalling Routinen selbst bereitstellen, oder durch Integration von Stubs, die diese Logik bereits fest integriert haben, und in einer Sprache wie C++ realisiert wurden, wiederverwenden. Ein später vorgestellter Ansatz nutzt die Möglichkeiten einer Skriptsprache, in C oder C++ realisierte Stubs einzubinden.

Unter dem Marshalling versteht man die Umsetzung eines Methodenaufrufes in eine Nachricht. Eine solche Nachricht benötigt eine Übertragungsform, die unabhängig von unterschiedlichen Rechnerarchitekturen ist. Eingesetzt wird daher eine Umwandlung der Daten in eine netztaugliche Form, im Falle Corba ist dies die Common Data Representation (CDR.). Die Strukturierung der Nachricht, ihre Syntax, wird durch das Protokoll vorgegeben.

TypeCodes

Die Corba Metadatensprache ist IDL, das Corba MetadatenRepository ist das IR. Corba nutzt TypeCodes, um jeden IDL Datentypen mit einem global eindeutigen Identifizierer zu verbinden. Somit werden Daten anhand ihrer eindeutigen Identifizierer selbstbeschreibend. Die Typecodes sind über bestimmte Methoden zu erhalten, und können auch dynamische aus dem IR erhalten werden
Wie bereits erwähnt wurde ist Idl eine streng–typisierte Sprache. Anhand dieser TypeCodes

kann auch die Typüberprüfung auf Seiten des Orbs erst ermöglicht werden. Zudem wird über diese Codes wird bewirkt, daß unterschiedliche Orbs eindeutige Typen verwenden.
Wie weiter unten gezeigt wird, ist ein Nebeneffekt dieser Typecodes, daß man ihre Information für die dynamische Generierung von Requests benötigt. Da ein Request einem Methodenaufruf entspricht, sind wie bei dem Aufruf von Methoden ebenfalls die Typinformationen der Argumente und des Rückgabewertes von Interesse.

Das Internet Interorb Protokol (IIOP)

Die Protokolle Giop und Iiop laufen unter der Fassade der CorbaArchitektur. Die in IDL spezifizierbaren Interfaces bilden aus Programmiersicht die Apis zu diesen Protokollen. Für den direkten Zugang zu diese Programmen exisitieren keine Schnittstellen, diese Protokolle werden automatisch vom Orb verwendet und sind damit transparent für den Entwickler.
Das General Interorb Protokol (GIOP) liegt eine Ebene über dem IIOP. Das IIOP ist die konkrete Ausprägung des GIOP für TCP–IP–basierte Netze. Da dieser Netztyp am weitesten verbreitet ist, ist IIOP der Kommunikationsstandard. Im Rahmen der Interoperabilität, das ist die Fähigkeit eines Clients von ORB A eine in IDL definierte Operation eines Objektes, das sich innerhalb eines ORB B befindet, aufzurufen, verbindet das IIOP Orbs unterschiedlicher Hersteller untereinander. Nach außen hin muß ein ORB zumindest eine Umsetzung des internen Protokolls auf das IIOP leisten.
Da es sich anbietet, als ORB–internes Protokoll gleich IIOP zu wählen, um sich diese Umsetztung zu ersparen, haben einige Orbs als internes Protokoll IIOP gewählt, ein Beispiel ist der Orb Mico.
Die weiter unten beschrieben Anbindung der Sprache TCL an Corba im Falle des ORB "Tcl–Iiop", erfolgt eine Erzeugung dynamischer Requests mit einer direkten Umsetztung des Requests in eine IIOP Nachricht.

Im Gegensatz zu anderen Kodierungsverfahren sind im IIOP keine Angaben zur Datentypisierung eingewebt. Besonders wichtig wäre dies eigentlich für die Argumente und Rückgabewerte der aufgerufenen Methoden. Diese Angaben kann man über die Schnittstellen der Objekte erfragen, oder direkt über das IR, wie im Abschnitt zu den Typecodes schon erwähnt.
Für das Marshalling – das ist die Umsetzung eines Methodenaufrufs in eine Nachricht– ist diese Typinformaton unerläßlich. Das Marshalling hat nicht nur die Aufgabe, dem Protokoll zu entsprechen, sondern auch eine Bestimmte Repräsentation der Bestandteile zu gewährleisten. Diese Repräsentation wird durch die Common Data Representation (CDR) beschrieben.

###Ein weiteres Charakteristikum ist, daß es dem Orb frei gestellt ist, in welcher Reihenfolge der Bitrepräsentation, auch ByteOrder genannt, er seine Daten codiert, er muß dies lediglich kennzeichnen. Die Bestandteile, also dei einzelnen Daten werden, wie die Elemente in Cstrukturen an gewissen Positionen ausgerichtet, und die entstehenden Lücken aufgefüllt.

Die Interoperable Objekt Referenz (IOR)

Sinn einer IOR ist es ein Objekt weltweit zu identifizieren.

Um weltweit eindeutig zu sein benötigt die IOR ihre Bestandteile, einen vom Orb vergebenen Schlüssel sowie für die Unterscheidung des Orbs seine Ip–Adresse und den dazugehörigen Port. Die Ior wird in zwei Teilen spezifiziert, einen netzwerkspezifischen Teil, der durch die IIOP–Spezifikation bestimmt wird, solange es sich um ein TCP–IP–Netz handelt und einen sehr generell gefassten, durch die Giop–Spezifikation gegebenen.
In der IOR ist das IOP, das Interoperability Profile enthalten, das die oben erwähnte Adresse des Orbs enthält, darüberhinaus aber noch eine weitere orbspezifische ObjektReferenz, die zumeist eine Speicheradresse darstellt, die nur der erzeugende Orb versteht. Der Aufbau der Informationen der IOR bestimmt sich nach der CDR, und einer Tatsache, das Alignment, die Anordnung von Datentypen an bestimmten "geraden" Positionen in einem Bytestream

3. Die Interface Definition Language (IDL)

Die Interface Definition Language ist eine Sprache zur einheitlichen, implementierungsunabhängigen Beschreibung der exportierten Attribute und Methoden von entfernten Objekten. Ihr zentraler Bestandteil sind Interfaces, die zu Modulen gruppiert werden können und damit ein Konzept der Namensraum– strukturierung darstellen.
Das sogenannte Interface spiegelt die Schnittstelle eines Objektes wieder. Grundsätzliche Konstrukte sind Operationen, Schnittstellen und Module. Darüberhinaus ist IDL deskriptiv, ihr fehlen operationale Elemente, algorithmische Aspekte, und im Gegensatz zu einigen Skriptsprachen ist sie typisiert.
Omg standardisiert die Sprachbindungen, um diese Konstrukte in andere Sprachen zu übersetzen und löst damit auch eine Reihe von Abbildungsproblemen, da jeder Sprache andere Strukturierungskonzepte unterliegen. Einige Sprachen sind objektorientiert. Unter den objektorientierten Sprachen gibt es unterschiedliche Klassenkonzepte.(zB Ein Java Interface entspricht in C++ einer abstrakten Klasse.)
Idl Spezifikationen ähneln syntaktisch den Klassendeklarationen in C++. Die Entsprechung zu einer C++ Klasse ist dabei eine Schnittstelle, oder IDL–Interface. Innerhalb dieser Schnittstelle werden die vollständigen Signaturen angegeben.
Einen Vererbungsmechanismus bietet IDL auch. Interfaces können von Interfaces erben, dies auch mehrfach. Das Überladen oder Überschreiben von geerbten Methoden ist aber nicht möglich. Dies auch aus Gründen einer Kompatibilität zu Sprachen wie zB, die keine Objektorientiertung unterstützen

Interfaces sind Klassendefinitionen ohne Implementationssektion, also ohne angehängte Methodendefinitionen. Operationen sind Methoden, mit Rückgabewert, Parametern und der Fähigkeit Ausnahmen aufzuwerfen und eine Kontextbeschreibung mitzuliefern.
Datentypen, wobei der Typ any, der jeden anderen Typ aufnehmen kann, besonders interessant ist. Er ermöglicht polimorphe Aufrufsituationen, mit Parametern, die vorher noch nicht feststehen. (??die also auch die strenge Typisierung umgehen??, im gewissen Sinne schon.)

IDL beinhaltet eine Reihe von einfachen Datentypen:

- short, long, char, octet und boolean

- float, double

Desweiteren eine Reihe von konstruirten Typen:

- struct, union, enum, any, sequence, string

Eine Sonderstellung nehmen Typen ein, wie die Exception und die ObjektReferenzen, von denen es zwei unterschiedliche gibt.

Ein weiteres Merkmal, das IDL auszeichnet, ist, daß es den Mechanismus des Exception Handling unterstützt, und dieser in allen Language Bindings auf die Zielsprachen überträgt.

4. Die offiziellen Language Bindings

4.1 C Language Binding

Aufgrund mangelnder objektorientierter Eigenschaften der Sprache C ist eine Konvention getroffen worden, über die entfernte Objekte auseinanderzu halten sind: Operationen, die auf dem Stub aufgerufen werden, erhalten zusätzlich die Zielreferenz des Objektes, sowie eine EnvironmentVariable übergeben.
Unterschiedliche Klassendefinitionen werden daduch auseinnander gehalten, daß die zu je einer Klasse gehörenden Operation über einen zusammengesetzten Namen identifiziert werden in der Form, daß der Interfacename (der Klasse) und der Operationname zusammengebunden werden: *Interface1_operation1()*

siehe an einem Beispiel:

```
interface Bsp1{
        int op1(in int etwas)}
```

wird zu

```
extern Corba_int   Bsp_op1(Bsp1 zielref, Corba_int etwas, Corba_Environment *ev);
```

//die Zielreferenz wird benötigt, da nicht die Form der Dereferenzierung: *Bsp.op1()* wie in C++ möglich ist!!

Über das Mapping wird die Verantwortung für die Speicherallozierung und deallozierereung in Rollen dem Client oder Server zugewiesen. Je nach Parametertyp (in ,inout, out) wird Speicher auf unterschiedlicher Seite alloziiert ode dealloziiert, was dann in getrennten Adreßräumen geschieht. Daher bietet Corba Möglichkeiten dafür, sozusagen Corbaspezifische Objektreferenzen, die nicht automatisch durch die Spracheneigene Laufzeitumgebung gelöscht werden können...

Da die IDLTypen, die einfachen, ein festdefiniertes Speicherlayout haben, müssen die C++Typen die gleiche Form auch haben. Dies ist aber auf unterschiedlichen Plattformen nicht gewährleistet. Anders als bei Java, wo die interne Darstellung der Datentypen plattformübergreifend genormt ist, sind bei der Implementierung von C und C++ Kompilern

die Darstellungen der Datentypen an die Möglichkeiten der unterschiedlichen Plattformen angepaßt worden. Die eine oder andere Architektur hat eine andere Wortlänge, ein anderes ByteOrdering...etc. Ein int ist 32bit groß auf einem PC aber auf einer Workstation eben nicht. Daher wurde ein indirektes Mapping auf Typen mit dem Suffix "Corba_" gewählt:

idl: *long* C: *Corba_long*
idl: *int* C: *Corba_int*
...

Vererbung von Interfaces können in einer OOSprache besser abgebildet werden. Zur Erinnerung, ein Interface in Idl wird auf eine Klasse in C++ abgebildet. So sind die Funktionen die eine abgeleitete Klasse erbt, implizit in ihr enthalten. In C gibt es keine Klassen. Ein Interface wird auf eine Reihe von Funktionen abgebildet, die mit dem Klassennamen beginnen.
zB Interface1_op1.
Eine ererbte Funktion wird daher nochmal explizit mitaufgeführt. Erbt ein Interface2 diese Methode *op1* und definiert nur eine *op2* neu, so werden automatisch erzeugt:

extern Corba_int Interface2_op1 () { }
extern Corba_int Interface2_op () {}
//die Methode Op1 wird also nochmal explizit aufgeführt.

Für das Exeptionhandling gib es einen Struct, der die Exeption abbildet, sowie ein Struct Corba_Environment, der entweder keine oder eine beliebige Exception aufnehmen kann, und den dritten Parameter einer jeden Operation darstellt.

4.2 C++ Language Mapping:

Ein Interface ist eine C++Klasse mit einigen public Definitionen, kann aber nicht selber in C++ abgeleitet werden, dies muss auf Basis der Idl Definition geschehen, wo dann eine abgeleitetes Interface wieder einer eigenen Basisklasse entspricht. (schon wegen des Idl eigenen Polymorphismus durch narrow() bezeichnet, nötig)

##Ein Pointer auf diese Klasse muss entweder als _var oder als _ptr vereinbart werden, der Speicherallloziation zuliebe. Ein Idl Compiler implementiert automatisch Stub, Skeleton etc von der Interfaceklasse, die eine abstrakte Basisklasse sein soll, diese InterfaceKlasse soll nur über den Compiler genutzt werden, aber nicht vom Programmierer implementiert werden....??

Das Interface des Naming Services liefert immer eine Referenz vom Typ Objekt, um nicht auf einen einzelnen Typen beschränkt zu sein. Da aber C++ eine streng typisierte Sprache ist, muss die Refernz auf den eigentlichen Typen mittels der Methode *narrow()* eingeschränkt werden.

Eine Exeption wird auf ein ine C++ Klasse gemappt, die in dem jeweiligen modulxx.h File deklariert wird.. Es gibt eine Basis Exception Klasse von der die User Exceptions und die SystemExceptions abgeleitet werden und davon wieder die eigenen. Also wird die modulxx exception om CorbaModul definiert und im modulxx.h deklariert.

Valuetype sind neu, sie vereinen Interfaceverhalten und Datentypverhalten. Heißt, ein einfacher Typ wird auf einen Typ in C++ gemappt, und hat lediglich einen Zustand. Ein Interface ist lediglich ein Interface, entspricht dann einer Klasse, in der die Methoden vereinbart werden (keine Datentypen in Form von Member/Attributen, da diese ja auf set und get Methoden gemappt werden). Der Valuetype soll beides vereinen.

5. Die inoffiziellen Language Bindings

Die hier vorgestellten inoffiziellen Bindings sind für eine Reihe unterschiedlicher Skriptsprachen erstellt worden. Dabei verfolgten die Entwickler weniger die Absicht ein Language Binding mit allen Aspekten zu realisieren. Die vorgestellten Bindings haben das Ziel, die Vorteile einer gewählten Skriptsprache mit denen von Corba zu verbinden.

Skriptsprachen wie Tcl, Perl oder eben Rexx bieten eine Reihe von Vorteilen gegenüber von Programmier– sprachen wie C, Java, C++:

- Sie sind wesentlich einfacher zu erlernen und flexibler zu handhaben.
- Vielfach werden sie eingesetzt um kleinere Programmieraufgaben, wie das Erstellen eines Prototypen oder das Zusammenstellen einer primitiven Testumgebung schneller zu erreichen. Man muß nicht erst eine Reihe von Variablen deklarieren, und viele Lowlevelfunktionen einbinden. Außerdem ist die Integration von Funktionen und Programmen und der Umgebung besser gewährleistet.
- Modifikationen an einem Skript können direkt ausprobiert werden. Zeitraubende Compilier,– und Link–Vorgänge werden erspart. Dies stellt sich dadurch ein, daß die genannten Skriptsprachen interpretiert sind. Aus diesem Grunde bieten sie zumeist auch eine Shell, in der man interaktiv arbeiten kann, was in einer Programmiersprache wie C++ nicht annähernd so komfortabel gelöst werden kann.

Der Einsatz einer Skriptsprache in Verbindung mit Corba soll folgendes erleichtern:

- Da verteilte Systeme recht komplex sind, ist es eher eine Standardsituation, daß an mehreren Objekten, oder Komponenten (Corba Objekte sind ja per Definition eigentlich Komponenten) gleichzeitig Modifikationen vorgenommen werden sollen.
- Eine Skriptsprache erleichtert dies durch ihre dynamischen Aspekte, und ein viel kürzerer Turnaround kann erreicht werden. Gerade in Hinblick auf solche CorbaObjekte, die CorbaComponents, wie sie in der neuerlichen KomponentenSpezifikation von Corba3 spezifiziert wurden, gibt es aktuell Bestrebungen mit dem "Corba Skripting" für das Konfigurieren von Componenten über Skriptsprachen.

Die hier vorgestellten Integrationsansätze orientieren sich an den Möglichkeiten, wie Aufrufe von Objektmethoden realisiert werden können. Wie oben bereits erwähnt, können Aufrufe

einerseits auf statische, andererseits auf dynamische Weise abgesetzt werden. Der statische Ansatz sieht eine feste Einbindung der Schnittstelleninformationen verteilter Objekte in Proxyobjekten vor. Anhand der IDL wird von einem IDL Kompiler für je eine Klasse von entfernten Objekten jeweils einen zugehöriger Stub und einen Skeleton generiert. Ein Stub enthält die dem IDL Interface entsprechenden Methodensignaturen. Fest integriert in den Stub sind Marshallingroutinen, die transparent einen Request erzeugen. Dies bewirkt, daß die Typinformation der Argumente implizit mit enthalten sind. Sollte sich die IDL –Definition einmal ändern, Beispielsweise die Anzahl und Anordnung der Argumente einer entfernten Methode, so müssen Stub und Skeleton neu kompiliert werden Dieser Ansatz ist der weniger flexiblere, aber der performantere.
Der flexiblere Ansatz, der Dynamische Aufruf erfährt erst zur Laufzeit über Schnittstelleninformationen, die dadurch auch flexibel verändert werden können. Mittels einiger Standardmethoden werden hier die Bestandteile für einen Request zu Laufzeit erfragt und zusammengesetzt. Der Operationsname in Form eines Strings und der Rückgabewert sowie eine Liste von Argumenten mit ihren entsprechenden Typen wird anhand der Informationen aus dem IR zusammengesetzt. Dazu notwendig sind Typinformationen die vom Orb ebenso überprüft werden, nur eben zu Kosten der Laufzeitperformanz.
Änderungen an einer Schnittstelle bewirken lediglich das erneute Übertragen der Schnittstelleninformation.
Im Gegensatz zum statischen Stub und dem Skeleton ist dieses Vorgehen jedoch nicht spezifisch für eine Klasse von Objekten.

5.1.Combat

"Über das Language Mapping hinaus, wird eine Infrastruktur benötigt, die das "Handling von Objekt Referenzen", das "Connection Management", und das "Marshalling in GIOP Messages" übernimmt".
Der hier gewählte Weg der Realisierung eines LanguageBinding führt über den dynamischen Methodenaufruf, wie er oben beschrieben wurde. Anhand der IDL Definition der Schnittstelle wird nicht ein Stub erzeugt, in dem die Typinformationen der Argumente fest integriert sind, sondern der Kompiler generiert die maschinenlesbare Form eines IR–Eintrages. Diese wird dann in das IR geladen, um dynamisch abgefragt werden zu können.

Der Verfasser von Mico bemängelt, daß ein GIOP Request zwar Informationen über das Zielobjekt, den Methodennamen, und die Parameter enthält, aber keinerlei Typinformationen bezüglich der Argumente übertragen werden. Diese sind aber notwendig, damit Sender und Empfänger wissen, wie die eingehenden binären Daten zu interpretieren sind. .
Selbst die IOR, die die Adressierungsinformationen für verschiedene Netzwerke enthalten, enthalten keine Typinformationen, darüberhinaus sind sie teilweise undurchsichtig.
Die einzige Introspektive die Corba bietet, ist daß mittels der definierten Methode corba::type() Typecodes aus dem IR zu beziehen sind, und damit die Typüberprüfung möglich ist.

Bis Tcl 8.0 war die einheitliche Datenrepräsentation der String, mit TCL8.0 wurde ein ein "Typsystem" rückwärtskompatibel hinzugefügt, in dem zwei Räpresentationen nebenherlaufen, die interne und eine "stringifizierte". Um einen eigenen Typ hinzuzufügen, müssen Hin,– und Rück– konvertierungsfunktionen bereitgestellt werden. Strings bleiben der unterste Nenner, bleiben gemeinsame Repräsentation. "whenever a value is requested to

have a certain type the old type updates the string represantation and the integer type for example scans the string.."
Somit hat Tcl die Typen int, boolean, double, string und als komplexen Typ eine Liste für das Languagemapping zur Verfügung.

Die Idl typen werden nach dem CombatMapping übersetzt. Zwei Möglichkeiten: eine langsame, die das Combat nutzt, also die Übersetzung auf Tcl Typen, und damit das Nutzen der Tcl Typen, oder eine Alternative, das Mapping undurchsichtig zu gestalten, mit Zugriffsfunktionen für je einen unterschiedlichen Typen zu gestalten.

In Combat wird zu je einer ObjektReferenz eine Tcl Prozedur – genannt Handle– erstellt, die Typinformationen und die Objektreferenz enthält. Dieses Handel interpretiert das erste Argument als Operationsnamen, und die übrigen als Parameter für diese Operation.
Ein solcher Handle "marshalled" die parameter in eine GIOP Nachricht und läßt sie über den Orb transportieren. Für jedes Handle muß ein existierender Eintrag im IR gefunden werden, damit das Marshalling überhaupt erst umgesetzt werden kann. Auf einen Aufruf der Methode get–interface() auf einem entfernten Objekt wird ein Zeiger auf einen Eintrag in einem entfernten IR returniert, einem "IR record in a remote IR".

Das Vorgehen in Combat im Detail sieht folgendermaßen aus:
Die Clientseite wird generiert, indem ein IDL File für den Compiler genommen wird. Er generiert aus xxx.idl dann xxx.tcl, das innerhalb eines Clientskripts genutzt werden kann. Wird dieses dann "gesourced" , steht danach in der Variablen –ir–xxx das, was in das Ir kommen muß, im Clientskript, bevor irgendwelche Kommandos kommen, wird mit einem Befehl dafür gesorgt, daß dieser Inhalt in ein Ir überschrieben wird.

Ein Handle muß erzeugt werden und wird benötigt, um Operationen in der Form auszuführen:
$handle operationX <parameter> //operation wird ausgeführt
$handle attributX //operation ist "$handle read attributX"
$handle atrributX zusetzenderwert //operation ist "$handle set attributX wert"

Ein solches Handle, ist ein Tcl Kommando das zu einem Serverobjekt korrespondiert. Operationen auf dem Handle bewirken Ausführung auf dem Server. Also ist das Handle ein Proxyobjekt. Mittels corba::string–to–objekt erhält man ein Handle, das man in einer Variablen ablegt.
Da Combat seine Informationen bezüglich der Operationen aus einem IR zieht, muß für jedes Handle zumindest ein Eintrag im InterfaceRepository vorliegen (ein Ir–eintrag korrespondiert zu einem Interface, ein Handle ebenfalls)

Realisiert wurde diese Implementierung in C++, und läßt sich auf jedem Orb installieren, das ein entsprechendes Language Mapping für C++ bietet. Es bildet somit ein "Glue Package", eine eben nicht rein in Tcl implementierte Lösung

5.2. R apid CorbaServer Development in Tcl

Diese Implementierung stammt aus einem zur UsenixKonverenz unterbreitetem Vorschlag "Rapid CorbaServer Development in Tcl" [] und beschreibt eher, wie innerhalb von Tcl ein entferntes Objekt angesprochen werden kann, als eine wirklich saubere Form eines Language Bindings.
"Zukünftige Applikationen werden Dinge wie starke Typisierung und Vererbung nicht als primäre Forderung stellen, sondern die Möglichkeit, Informationen und deren Anwendungen, die in verschiedensten Sprachen realisiert sind, miteinander zu integrieren".
Die Idee des Artikels ist, die "core as scripting language idea", also den Kern einer Anwendung, die eine Menge bestehender Anwendungen kombiniert und anderen integriert, in einer Skriptsprache zu implementieren. Einige der Komponenten einer solchen Anwendung befinden sich lokal, und sind nicht in Tcl realisiert, aber profitieren von TCLs Möglichkeiten der Einbindung, andere sind entfernt und profitieren von der Verteilung durch Corba.

Das Bindeglied zwischen TCL und einem ORB wird unter Nutzung von in C++ kompilierten Stubs und Skeletons gebildet. Bedenkt man jedoch, daß dieser Ansatzt hier im Gegensatzt zu den bereits vorgestellten ein "statischer" ist, sind mit häufig anfallenden Änderungen der IDL–Definitionen von Interfaces auch erneute Kompiliervorgänge verbunden.

Das Design bestimmte weiterhin, daß nicht nur der Client in Tcl geschrieben sein könnte sondern der Server, der ja die entfernten Prozeduren enthält, und damit den Hauptanteil der Applikationslogik, ebenfalls. Dafür muß ein solcher Server einen Tcl Interpreter enthalten, in dem die Idl Methodenaufrufe an gleichlautende Tcl Kommandos weitergeleitet werden.
Im Client, in dem der Aufruf der Methode stattfindet, sollen diese Kommandos wie übliche Kommandos ausehen, die in einem Interpreter, der sogenannten TCL Shell aufgerufen werden. Daher mußte eine Shell für den Clienten so erweitert werden, daß gewährleistet ist, daß die bisherigen Funktionen der Shell unangetastet im lokalen Tcl Kernel umgesetzt werden, die neu hinzukommenden "entfernten", aber transparent "umgeleitet" werden. (In diesem Zusammenhang kann man konzeptionell die Shell und den Kern trennen. In der Client–Shell werden die Kommandos aufgerufen, in dem Kern der Serverseite sind die Kommandos verankert.)

Die zentrale Frage zu Beginn ist, wie man Corba Objekte auf typenlose TclStrings mappen kann.
Eine Möglichkeit ist, jeden Datentyp in Tcl als paarweiser Zuordnung von Wert und Datentypbezeichner zu betrachten, somit die Typinformationen gleichsam mitzuführen, und anhand dessen das Marshalling umzusetzen
Eine andere Möglichkeit ergibt sich, wenn man die mittels eines vom Orb unterstützen C++ Mapping generierten statischen Stubs und Skeletons als Bindeglied benutzt und das damit bereits dort gelöste Marshalling der Methodenaufrufe umgeht.
Dann hat man allerdings das Problem, wie nunmehr die TclStrings in die jeweiligen C++ Typen umgewandelt werden, die innerhalb der C++ Proxy für Marshalling benutzt werden.

Bedenkt man, daß die Typinformation bei einem Methodenaufruf durch die Anordnung der formalen Argumente implizit enthalten ist, braucht man sie nicht explizit mitzuführen. Da die Idl Definition der Methoden maßgeblich für die Erzeugung der entsprechenden C++ Methodensignaturen in den Proxys ist, weiß man immer, daß das erste Argument einer Methode von einem bestimmten Typ ist. Erstellt man desweiteren einmalig eine Reihe von Konvertierungsfunktionen für die Umwandlung der TCL Strings in C++Typen, so ist die zu

lösende Aufgabe, diese Konvertierungsfunktionen in Abhängigkeit von den im IDL– File definierten Typen, mit den generierten Proxyobjekten zu verknüpfen. Jeweils ein Paar für die Kovertierung von Tcl nach C++, sowie von C++ nach Tcl, *Convert::form_tcl* und *Convert::to–tcl()* wird erstellt.
Laut der Beschreibung ist dies eine mittels Skripten leicht zu automatisierende Aufgabe.

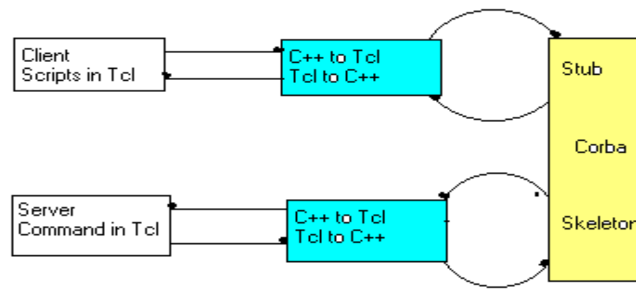

Abbildung2 : Umsetzung des vorgestellten Bindings

5.3.weitere Language Bindings

CorbaScript:
Ist eine eigens für Corbazwecke neu geschriebene Skriptsprache. Sie bietet objektorientierte Eigenschaften in Form von Klassen und erlaubt Mehrfachvererbung und Polymorphismus. Mittels Modulen wird es auch möglich, wiederverwendbare Programmteile in eine Package zusammenzufassen und in einem eigenen Namenraum zu kapseln. Das bestehende Language Mapping basiert auch hier auf dem Dynamischen Aufruf und dem Erhalt der nötigen Typinformationen aus dem IR zu Laufzeit. CorbaSkript selber ist mittels C++ realisiert worden. Die Implementierung bietet sogar eine Schichtenarchitektur, in der die unterste Schicht vom darunterliegenden Orb abstrahiert, mit der Begründung, daß – trotz Standardisierung– Funktionssignaturen und deren Elemente von Orb–Produkt zu Orb–Produkt voneinander abweichen.

Tcl Dii und seine "Neuauflage" TclIop. Beiden ist ebenfalls eine Umsetzung mit Hilfe des Dynamischen Methodenaufrufs gemeinsam. Die Typinformation zur Umwandlung der Argumente resultiert jedoch nicht aus einer Abfrage des Interface Repositories, sondern aus mitgeführen Tags oder Typecodes, die der Programmierer mit den Argumenten mitführen muß. Der dynamische Aufruf geht über eine einzige Methode orb_call(). Dieser werden als Argumente der Bezeichner der aufzurufenden Funktion, gefolgt von einem Platzhalter für den aufzunehmenden Rückgabewert und die einzelnen Argumente paarweise mit Bezeichner

und Typangabe übergeben.
Dies sieht in etwa so aus: *orb_call $result methodname f 1.35*
Weiterhin zu erwähnen bleibt, daß beide Lösungen nur die Clientseite abdecken.

Fnorb mittels Phyton. Dieser Ansatz unterscheidet sich in sofern, daß Requests direkt in Form von IIop Nachrichten verpackt und versendet werden.
Bsp: *orb = corba:orb_init(argv, Corba:orb_id)*
object = orb.string_to_objekt(StringIor)
object.methodname($etwas)

Cope
An dieser Stelle sollte noch kurzt Cope vorgestellt werden. Cope bietet auch eine Integration von Perl und Corba. Wer Cope gerne ausprobieren möchte, muß schnellt feststellen, daß dies eine Implementation ist, die wie die oben vorgestellte Lösung Tcldii (in diesem Falle gemeint die Produkte der Firma Orbix, da die Umsetztung der IDL Information in Tcldii dort mit Hilfe von proprietären Typecodes aus dem IR der Orbix−implementation stattfand.) von einem speziellen Produkt abhängig ist. Der idl2perl Compiler setzt hier das InterfaceRepository des ORBacus voraus. Das Verfahren ist, daß einmalig ServerSkeletons uns Stubs generiert werden unter Zuhilfename des IR. Cope unterstützt auf der Serverseite den Boaansatz, heißt es ist möglich eine Serverimplementierung zu stellen. Leider ist diese Implementierung dann auch auf einen konkreten Orb gemünzt, da jeder Boa eben Orb spezifisch ist.

6. Der Ablauf eines dynamischen Aufruf

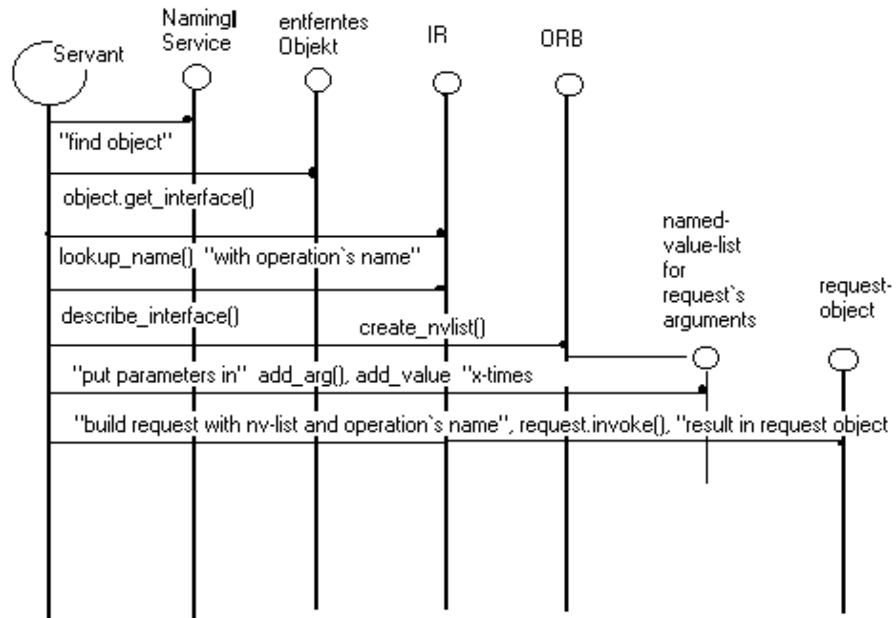

Abbildung3 : Die Ablauf eines dynamischen Methodenaufrufs

In Abbildung 2 ist der Ablauf eines dynamischen Methodenaufrufs dargestellt. Im vorherigem Kapitel wurde erwähnt, daß einige der inoffiziellen Language Bindings Methodenaufrufe dynamisch erzeugen. Zwar benutzen diese LanguageBindings nicht unbedingt die hier vorgestellten Methoden, und mögen das Erzeugen von Request auch in der eigenen Sprache realisieren, eine gemeinsame Problematik wird hier aber deutlich: die der Pseudoobjekte, die innerhalb der jeweiligen Programmiersprache verfügbar sein müssen.

1. Erlangen von Zugang zum Orb
Vorraussetzung um an der Corba Kommunikation teilzunehmen ist, daß ein Client einen Zugang zum seinem Orb erhält. Dazu muß es ein sogenanntes "Orbobjekt" erzeugt und initialisiert werden.
Dieses Orb–Objekt ist ein spezielles Objekt, es ist ein Pseudoobjekt. Pseudoobjekte sind Objekte, die im lokalen Adressraum der verwendeten Programmiersprache liegen, und genauso, wie entfernte CorbaObjekte behandelt werden, was bedeutet, daß aus Sicht des Programmierers kein Unterschied in den Methodenaufrufen zu erkennen ist.
Ist dann der Orb bekannt, muß ein weiteres Pseudoobjekt, der Objektadapter erzeugt werden. Mittels boa_init() wird dieses dann initialisiert.

2. Erlangen einer Objektreferenz eines entfernten Objektes. Dies kann über einen Nameservice oder Traderservice laufen. Über ihre Objektreferenz können entfernte Objekte aufgerufen werden. Diese Objektreferenz ist vergleichbar mit einer Referenz in einer Programmiersprache, nur daß sie eben nicht auf den lokalen Adressraum einer Laufzeitumgebung beschränkt ist.

3. Erlangen einer bestimmten MethodenBeschreibung. Über eine Objektreferenz kann die Methode get–interface() aufgerufen werden, um eine Referenz auf ein InterfaceDefObjekt zu erhalten. Dieses ist ein Bestandteil eines Interface Repositories und ermöglicht Auskunft über ein Interface eines Objektes, sowie

deren Bestandteile zu geben.

4. Eine Argumentliste wird die Parameter, die der entfernten Methode übergeben werden, enthalten. Eine Argumentliste ist eine selbstdefinierende Datenstruktur. Sie wird für den Aufruf erzeugt und ist ebenfalls ein Pseudoobjekt.

5. Der Aufruf wird ausgeführt.

7. Language Binding für Rexx

7.1 Rexx und IDL

Rexx selber weist eine Menge Gemeinsamkeiten zu den hier vorgestellten Skriptsprachen
Hier wird nun versucht, darzustellen, auf welche Rexxeigenschaften sich IDL konzeptionell abbilden läßt.:

Die einfachen Datentypen und Rexx–strings###

Die komplexen Datentypen und evt Stems??##

Für die Operationen, die zu einem IDL–Interface gehören, muß im klassischenRexx dann ein ähnlicher Ansatz herhalten, wie bei C. Werden mangels Objektorientierter Eigenschaften im Klassischen Rexx, die in IDL dort genannten Operationen auf Funktionen, oder in Rexx genannt Subroutinen abgebildet, so muß eine Referenz eines (konzeptionell existierenden Objektes)###immer als erstes Argument mitgeführt werden. Daß die Methode eines bestimmten Objektes eindeutig bestimmt ist, ergibt sich in der Objektorientierung mittels der Quilifizierung durch Dereferenzierung des Objektes *Objekt1.methode1()*, in Form von Funktionen kann diese Qualifizierung jedoch nur über eine Konvention in der Namensgebung erreicht werden: *Objekt1_methode1()*. Eine wirkliche Form der Kapselung ist dies nicht, reicht aber, wenn man diese Namenskonvention nicht verletzt.

Der Namensraum, der durch IDL Module gegeben ist, wie bei den Interfaces, über den Namen??

Für ein ExeptionHandling bietet Rexx die Möglichkeiten diese über Umgebungsvariablen abzubilden. Denkbar wäre auch, wie beim CMapping, die Exception auf einen Struct, abbzubilden, der dann in einheitlicher Weise jeder Methode als ein drittes Argument per Referenz übergeben wird, oder ####auf eine globale Variable (aber dies ist ja mit der Umgebungsvar gegeben) der innerhalb des Servants oder Clients dann Gültigkeit besitzt, und somit global zu den Methoden ist.
(Rexx bietet eine Schachtelung von Umgebungen und deren Umgebungsvariablen....)

Auch für Rexx gibt es eine Objektorientierte Erweiterung, oder besser: Aufbauend auf dem klassischen Rexx ist ObjektRexx entstanden, die eine ähnliche Abwärtskompatibilität zum klassischen Rexx bietet, wie es analog mit C und C++ der Fall ist.

##
Einige Objektorientierte Eigenschaften verwenden zu können wäre von Vorteil: Auf der einen Seite für die Bildung von Servants###, von denen man vielleicht eine größere Anzahl Instanzen bilden muß, zum anderen für die später erwähnten Pseudoobjekte.

7.2 Das SAA–Api, Statische Umsetzung

Der Rexx Interpreter selber liegt in Form einer dynamisch linkbaren Datei vor. Potentiell jede Applikation kann diesen einbinden und verwenden. Er ist voll reentrant und unterstützt, daß im selben Prozess mehrere Rexxprozeduren auf unterschiedlichen Threads laufen können. Über Einbinden einer bereits bestehenden Headerdatei Rexx.h kann man diesen Interpreter innerhalb von C, C++ verwenden.

Teil von Rexx ist ein API genannt SAA. Dieses beschreibt ein Interface, zwischen dem Rexxinterpreter und anderen comipilierten Programmen
Die Spezifikation unterteilt sich in mehrere Unterbereiche:

- Subcommand Handler, welche in der Lage sind, Kommandos für externe Umgebungen einschließen und den Zugriff auf die einzelnen Subcommandos von der umgebenden Applikation sicherstellen
- External Function handler, sollen die Rexx Sprache mit externen Funktionen erweitern. Diese sind für den umgekehrten Weg, dem Zugriff aus dem Interpreter auf die Funktionen der umgebenden Applikation, oder die Funktionen einer dynamischen Bibliothek.
- Ausführen von in Files befindlichen Rexxskripten
- Das Variablen–Interface, erlaubt den Zugriff von außen auf die Variablen im Interpreter
- System exits "which allow to hook into certain key points in the interpreter while it executes a script"

SubcommandHandler
Diese Sektion beschreibt den Subcommand Handler, der es einer Applikation erlaubt, Kommandos, die sich in einem RexxSkript befinden, einzuschließen, und sie selber auszuführen.
Es wird davon ausgegangen, daß eine Applikation den RexxInterpreter gestartet hat, dieser also im gleichen Adressraum läuft wie der Prozeß. Nach Laden eines Rexxskriptes, können die im Skript enthaltenen Unterfunktionen dann vom umgebenden Programm ausgeführt werden. Aus Sicht des Interpreter ist dieses Programm eine externe Umgebung. Ruft der Interpreter den Subkommandohandler auf, und übergibt ihm das Kommando als Parameter so ist das Handle in der Lage das Kommando auszuführen. dieser sieht so aus:

APIRET APIENTRY handler(
PRXSTRING command,
PUSHORT flags,
PRXSTRING returnstring,)

RexxRegisterSubcomExe()
Diese Funktion soll ein Subkommando mit dem Interface registrieren. Dieser SubkommandoHandler, wie oben erwähnt , muß sich innerhalb des Codes des aufrufenden Programmes befinden. Nach dieser Registrierung kann eine Rexx interpreter dieses Subkommando dann ausführen, indem der Subkommandohandle, der sich ja im aufrufenden Programm befindet, aufgerufen wird, und diesem der Name des Subkommandos übergeben wird.

ExternalFunctionHandler
Diese Sektion beschreibt einen externen Funktionen Handler, der die Sprache erweitern kann, indem er externe Funktionen in anderen Sprachen schreiben läßt!!

Betrachtet man die Möglichkeiten des Saa, so sind die Bedingungen für eine Realisierung wie in der vorgestellten Tcl–Lösung "Rapid Corba Server Development" gegeben:

Auf der Serverseite kann die Implementierung in Rexx geschrieben werden, und in einem Interpreter ablaufen der sich innerhalb eines Servers befindet, der in einer anderen Sprache realisiert ist. Genau diesen Vorteil hat sich die vorgestellte Lösung zu Nutze gemacht, daß das C++ Language Binding eine gute Grundlage bilden kann, da es von fast allen ORB schon unterstützt wird. Ein Skript mit den Tcl–Subroutinen ist in einen Rexxinterpreter ladbar.

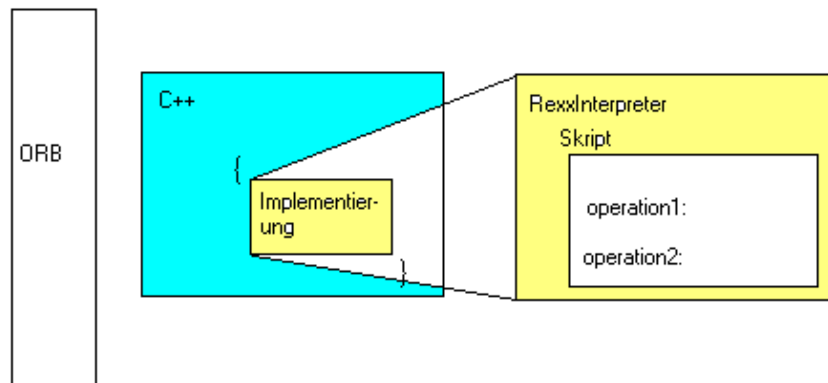

Abbildung4 : Die Realisierung eines statischen Servers

Wie erwähnt wurde, gibt es die Möglichkeit in Rexx, Funktionen im Format einer Dll über RXFuncadd() einzubinden, und zu benutzen als handele sich um ein Rexx eigene Kommandos. Sofern es also möglich ist, aus der in C++ vorliegenden Stubdatei eine dynamische Bibliothek zu erzeugen, könnte man auf die nunmehr

in diesem Stub befindlichen Funktionen aus Rexx beschriebener Weise zugreifen.
Eine andere Alternative, ebenfalls aus dem Saa Packet stammend, scheint der External Function Handler zu sein, mit dem man nach der Registrierung der externen Funktion durch rexxregisterfunctionexe() diese dann auch innerhalb Rexx verwenden kann.

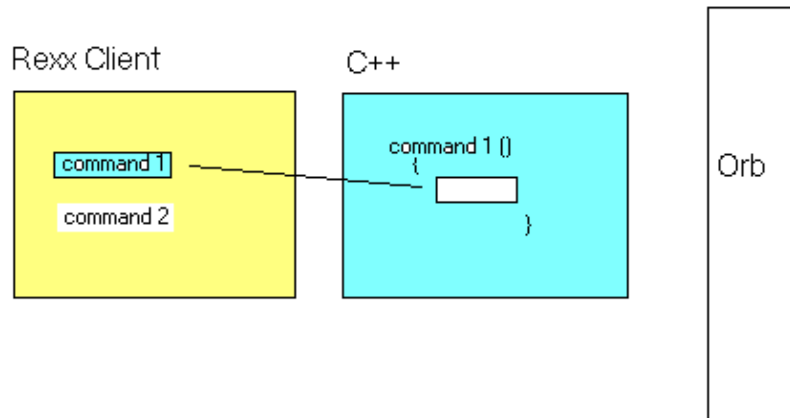

Abbildung5 : Die Realisierung eines statischen Clienten

7.3 Pseudoobjekte

Rekapitulieren wir noch einmal den Vorgang beim dynamischen Methodenaufruf, wie er in Kap6 aufgeführt wurde. Um einen Zugang zu Corba zu bekommen, mußte man zunächst einen Zugang zu einem Orb erlangen. Dazu mußte ein sogenanntes "Orbobjekt" erzeugt und initialisiert werden. Dafür gibt es in einer objekt orientierten Programmiersprache eine Art Klassenmethode, die zu den Pseudoobjekten gehört. .
Eingangs wurde auch erwähnt, daß es sich bei den Objekten Orb und Boa um Pseudoobjekte handelte.
Hier stellt sich die Frage, wie mit Pseudoobjekten umgegangen werden soll.

- "Pseudoobjekte sind Objekte, die im lokalen Adressraum der verwendeten Programmiersprache liegen, und genauso, wie normale CorbaObjekte behandelt werden, was bedeutet, daß aus Sicht des Programmierers kein Unterschied in den Methodenaufrufen entfernter und lokaler Objekte zu erkennen ist."
- "Ein Pseudoobjekt ist ein Objekt, das der Orb direkterweise erzeugt. Der Orb selber ist auch ein Pseudoobjekt.
- "Eine Referenz auf eine Instanz des Pseudoobjektes Orb erhält man, indem man den Corba–Api–Aufruf Orb–init() ausführt"

"Pseudo Objekte sind Sprachenspezifische Objkte, ein solches Orbobjekt ist nicht notwendigerweise entfernt Es ist typischerweise ein Prozess–lokales programmiersprachliches Konstrukt. und ist damit Beschränkungen unterlegen: einerseits, daß es nicht im IR repräsentiert ist, andererseit auf eine offentliche abstrakte Klasse gemappt wird, die keine anderen Klassen erweitert, (oder in Java von Interfaces erbt).

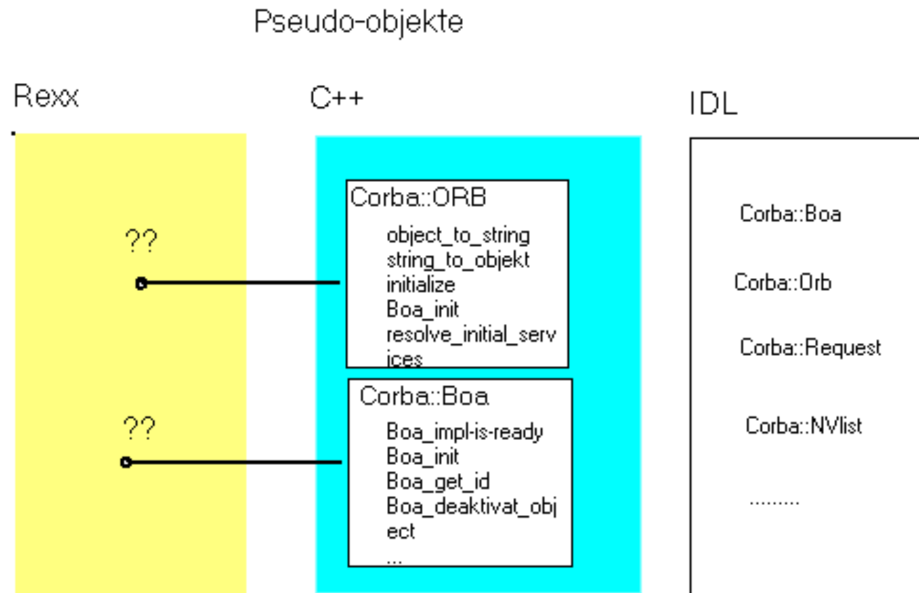

Abbildung6 : Der Zusammenhang mit den Pseudoobjekten

Es stellt sich die Frage, wie man ein solches Pseudoobjekt, das es für C++ gibt, für Rexx verfügbar machen kann??

Für den in Kap 7.2 gezeigten statischen Ansatz, bildet das Language Binding in Rexx eine Schicht, die oberhalb des Stubs liegt. Der Stub enthält konzeptionell die Instanz eines Orbs. Der Orb findet sich in Form von in den Stub miteinkompilierten Funktionen wieder.

Für diesen Ansatz muß lediglich das Problem gelöst werden, wie die Umwandlung der Rexx Argumente in C++ vollzogen wird, und wie man dies automatisieren kann.

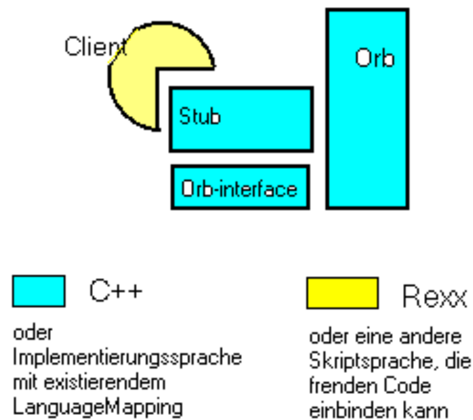

Abbildung7 :

Die Dynamische Seite

Betrachtet man eine Reihe von Orbs, so stellt man fest, daß sie nur einige wenige LanguageBindings unterstützen. Bei manchem ORB scheint die Verbindung der Implementierung des Orbs eng verbunden mit dem angebotenen Language Binding.

Aus der Beschreibung zum Combat–Paket, das das oben genannteTcl–Binding enthält, läßt sich entnehmen, daß es auf theoretisch jedem Orb läuft, der ein C++ Sprachbinding anbietet. "Combat ist ein glue–package, geschrieben in C++, und hat Zugang zu Merkmalen, die nicht verfügbar sind in Tcl.

Auch wenn man die Pseudoobjekte, oder besser die Teile, die sich eng an die Implementierung des ORB anlehnen in ihrer Programmiersprache beläßt so muß das DII nicht unbedingt in C++ geschrieben werden, wenn man einerseits den Orbinternen Protokollmechanismus kennt, sofern dieser abweichend vom IIOP ist, oder wie bei einigen anderen Lösungen beschrieben, die IIOP Requests selber erzeugt. Da dies mit einem für Standardverbindungen in TcpIp tauglichen Socket geschehen kann, könnte man das DII tatsächlich in Rexx schreiben.

neu

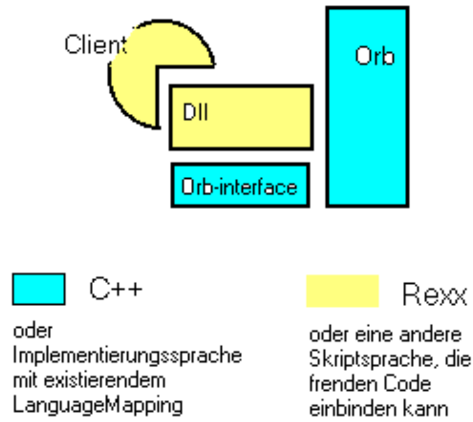

Abbildung8 :