Seminar
Ausgewählte Komponenten von Betriebssystemen:
IDL4 Compiler

Hristo Pentchev

28. Oktober 2004

Inhaltsverzeichnis

Kapitel 1

Einleitung

In großen verteilten Anwendungssystemen müssen häufig bestehende Dienste von heterogenen Systemen integriert werden, die von jedem beliebigen Ort aus genutzt werden können. Die von der Software-Industrie entwickelte Standardtechnologie CORBA kann in diesem Fall die Lösung des Problems darstellen.

Die Common Object Request Broker Architecture (CORBA) wurde im Jahr 1991 von der OMG entwickelt, um ein standardisiertes Verfahren zur Kommunikation von verteilten Systemen anzubieten. Die Object Management Group (OMG) ist ein Konsortium, das im Jahr 1989 von 11 Firmen gegründet wurde und inzwischen über 800 Mitglieder hat. Die Entwicklung von grundlegenden Technologien, die die einfache Herstellung von verteilten objekt-orientierten Anwendungen ermöglichen, ist eines der Ziele der OMG.

Die Schnittstellen von CORBA Objekten werden in einer programmiersprachenunabhängigen Notation, der sogenannten OMG Interface Definition Language (OMG IDL), definiert. Hierdurch wird sichergestellt, daß sämtliche Objekte eines CORBA-Systems auf einheitliche Art angesprochen werden können. Clients rufen Methoden in Server-Objekten auf, ohne beachten zu müssen:

- auf welcher Systemarchitektur diese laufen,
- unter welchem Betriebssystem diese laufen,
- in welcher Programmiersprache diese Objekte implementiert sind.

Die Schnittstellen-Definitionen werden von einem IDL-Compiler verarbeitet. Als Ausgabe liefert der IDL-Compiler Dateien, die die Komunikation zwischen den CORBA-Objekten übernehmen. Mit Hilfe dieser Dateien wird geregelt, wer mit wem und in welcher Form Nachrichten austauschen darf. Diese Dateien sind als Stubs und Skeletons wieter zu treffen. Ein Client-Stub ist ein kleiner Quelltextteil, der es einem Client ermöglicht, auf eine Server-Komponente zuzugreifen. Ein Skeleton ist ein Quelltextteil, der beim Implementieren eines Servers quasi ausgefüllt wird.

Die verschiedenen IDL-Compiler erzeugen Dateien in verschiedenen Programmiersprachen. IDL4 ist ein Code-Generator für L4 Microkernel, der IDL-Definitionen in C-Code transformiert.

Im Kapitel 2 wird die Geschichte von verteilten Systemen kurz zusammengefasst. Darauffolgend werden im Kapitel 3 CORBA und ihre wichtigste, Bestandteile vorgestellt. Im Kapitel 4 wird ein Überblick über IDL gegeben. Anschließend wird im Kapitel 5 ausführlicher IDL4 dargestellt, um dann schließlich seine Anwendungen zu zeigen. Das Ende stellt eine Zusammenfassung der Arbeit dar.

Kapitel 2

Geschichte der verteilten Systeme

2.1 Entwicklung

Die erste wichtige Plattform für Büro-Anwendungen waren die Großrechner. Sie besaßen die Merkmale, daß sie einer Vielzahl von Benutzern ihre Anwendungen über Terminals zur Verfügung stellten. Die Terminals waren nichtprogrammierbar und dienten nur dazu, eine allgemeine Schnittstelle zur Kommunikation mit dem Großrechner anzubieten, auf dem alle Anwendungsprozesse liefen. Die Großrechner wurden zentral verwaltet und waren sowohl in der Anschaffung als auch in der Wartung sehr teuer.

Ein Nachteil der Großrechner war die fehlende Flexibilität. Um diesen Nachteil zu überwinden, wurde die Client-Server Architektur Mitte der 80-er Jahre entwickelt. Bei ihr befand sich die Datenbank auf dem Server und die Benutzeroberfläche auf dem Client. Für Server wurden entweder Großrechner oder die Unix-Rechner eingesetzt. Die Clients waren PCs, die am Anfang (Zweischichtige Client-Server Architektur) die Rechte auf Datenbankzugriffe selber besaßen und verwalteten, welche bei der später entwickelten mehrschichtigen Client-Server Architektur nur von den Servern verwaltet wurden, die alleine für das Prozessmanagement verantwortlich waren. Die PCs waren günstiger und profilierbarer, leider aber auch leistungsschwächer als die Großrechner, was aber für viele Anwendungen keine Rolle spielte.

Der nächste logische Schritt in der Entwicklung von Anwendungsarchitekturen war das Entwickeln der verteilten Systeme.

Ein verteiltes System besteht aus verschiedenen miteinander vernetzten Computern, die miteinander kommunizieren, indem sie Nachrichten austauschen und ihre Aktionen koordinieren. Sie bieten große Flexibilität für Programmierer und Entwickler. Man muss nur Schnittstellen beachten, hat aber die absolute Freiheit

bei der Wahl der Implementierung.

2.2 Die Rolle von CORBA in den verteilten Systemen

Die Rolle von CORBA ist es die Kommunikation von den Rechnern in den verteilten Systemen zu gestalten. CORBA stellt einen Standardmechanismus zum Definieren der Schnittstellen zwischen Komponenten sowie einige "Werkzeugeßur Verfügung, die das Implementieren dieser Schnittstellen in der vom Programmierer gewählten Sprache erleichtern. Schließlich übernimmt CORBA auch den Transfer von Nachrichten, die von den verschiedenen Komponenten einer oder verschiedener Anwendungen verschickt werden.

Die zwei wichtigsten Eigenschaften von CORBA sind die **Sprachen-** und die **Plattformunabhängigkeit**. Dies bedeutet, dass CORBA Objekte, die in verschiedenen Sprachen implementiert sind und die sich auf Rechnern mit verschiedenen Betriebssystemen befinden, miteinander kommunizieren können.

Kapitel 3

Die CORBA-Architektur

Die Geschichte und die Aufgaben von CORBA wurden im vorhergehenden Kapitel dargestellt. Im Folgenden wird ein grober Überblick über ihre Architektur gegeben. CORBA ist die Abkürzung für Common Object Request Broker Architecture und ist ein offener Standard der Open Management Group (OMG). CORBA beschreibt eine Architektur, mit der es in einer verteilten heterogenen Architektur möglich ist, Dienste für andere zur Verfügung zu stellen bzw. Dienste anderer Knoten in Anspruch zu nehmen. Zentraler Bestandteil von CORBA ist der Object Request Broker (ORB). Der ORB stellt alle Dienste zur Verfügung, die zur Kommunikation zwischen Client und Server notwendig sind. Schnittstellen und Datentypen werden in CORBA mit Hilfe einer deklarativen Interface Definition Language (IDL) definiert. Aus der Schnittstellenbeschreibung werden mit Hilfe eines IDL-Compilers Client- und Server-Stubs erzeugt. Der Client-Stub wandelt einen Methodenaufruf inkl. Parameter in einen Datenstrom um (Marshaling) und sendet ihn an den ORB. Dieser sendet den Datenstrom zum entsprechenden Server. Dessen Serverstub entpackt den Datenstrom und führt den eigentlichen Methodenaufruf aus. ORBs können dabei über das von der OMG standardisierte Internet Inter-ORB Protocol (IIOP) miteinander kommunizieren. Im diesen Kapitel werden die CORBA Bestandteile vorgestellt.

3.1 Object Request Broker

ORB ist eine Software-Komponente, die die Kommunikation zwischen Objekten erleichtern soll. Sie hat zwei Aufgaben:

- Das Finden des Remote-Objects durch Objektreferenz.
- Das Übertragen des Formats von Parametern und Rückgabewerten (Marshaling und Unmarshaling).

Der Object Request Broker (ORB) ist der Kommunikationsbus zwischen CORBA-Objekten, der einen Mechanismus bietet, um Anfragen des Clients an die entsprechenden Object Implementations weiterzuleiten, unabhänigig davon, an wel-

chem Ort diese im Netzwerk residieren und unter welchem Betriebssystem sie dort laufen. Der ORB ist verantwortlich für die Übertragung der Aufrufparameter, der plattformabhängigen Formatierung der Werte (z.B. Wertebereich von Integern) und der Erzeugung einer eindeutigen Objektreferenz. Hinter einem einfachen Methodenaufruf des Clients verbirgt sich eine komplexe Netzwerkkommunikation, die vom ORB organisiert wird. Eine Anfrage des Clients, bestehend aus dem Methodenaufruf und den Parametern, wird in einem binären Strom umgesetzt (marshaling) und über das Netzwerk an den Server geschickt. Die Informationen werden auf der Server-Seite wieder decodiert (unmarshaling) und die gewünschte Operation ausgeführt. Rückgabewerte werden auf die gleiche Weise wieder über den ORB an den Client gesendet. Zusammengefasst hat der ORB die Aufgabe, die entsprechende Objektimplementierung zu finden, sie zu aktivieren, falls erforderlich, die Anfrage an das Objekt zu leiten und entsprechende Rückgabewerte wieder an den Client zurückzugeben.

Im Allgemeinen kommuniziert ein Client-Objekt nicht direkt mit dem ORB, wenn es Methoden eines Server-Objektes aufrufen will, sondern über sogenannte Stubs. Stubs sind lokale Repräsentationen von fernen Objekten im lokalen Adressbereich, die aus der IDL-Spezifikation der Serverobjekte generiert werden. Die Kommunikation mit dem ORB geschieht innerhalb der Stubs. Für den Client erscheint dadurch das Serverobjekt wie ein lokales Objekt. Serverseitig geschieht die Anbindung des Server-Objektes an den ORB über sogenannte Skeletons. Diese übermitteln Methodenaufrufe eines Clients an das Serverobjekt.

3.2 Das Transportprotokoll

In einer mit CORBA entwickelten Client/Server-Anwendung existieren zwei ORBs, die miteinander über ein Netzwerk in Verbindung stehen. Mit Hilfe der ORBs werden die entsprechenden Anfragen des Clients an den Server weitergeleitet. Dieser führt den gewünschten Service aus und gibt die Rückantwort wieder über die ORBs an den Client zurück.

Hinter diesem Aufruf steckt eine komplexe Netzwerkkommunikation, die über die von der OMG spezifizierte CORBA-Inter-ORB-Architektur beschrieben ist. Zwei wichtige Teile der Architektur sind das General Inter-ORB Protocol (GIOP) und das Internet Inter-ORB Protocol (IIOP).

- Das General Inter-ORB Protocol (GIOP) spezifiziert eine Menge von Nachrichtenformaten und Datendarstellungen für die Kommunikation zwischen den ORBs. Die in IDL definierten Datentypen werden als Common Data Representation (CDR) in eine einfache Nachrichtendarstellung für Netzwerke abgebildet. GIOP definiert außerdem ein Format für Interoperable Object References (IORs). Ein ORB muss eine IOR von einer Objektreferenz erzeugen, wann immer eine Objektreferenz über ORBs weitergegeben wird.

- Das Internet Inter-ORB Protocol (IIOP) legt fest, wie GIOP-Nachrichten über ein TCP/IP-Netzwerk ausgetauscht werden. Das IIOP ermöglicht

daher, die Busarchitektur und die Technologien des Internets selbst als Basis für eine Inter-ORB-Kommunikation zu verwenden.

3.3 Die Objektreferenzen

Wie jede objektorientierte Architektur weist auch CORBA ein Objektmodell auf, wie die Objekte im System dargestellt werden. Die Kommunikation zwischen CORBA-Objekten erfolgt mit Hilfe von Referenzen (Interoperable Object References IOR).

IORs verbinden eine Sammlung von "tagged profiles"mit Objektreferenzen. Die Profile beschreiben dasselbe Objekt, aber jedes beschreibt, wie mit dem Objekt über einen bestimmten ORB-Mechanismus in Kontakt zu treten ist. Ein Profil stellt selbstbeschreibende Daten zur Verfügung, die die ORB-Domäne identifizieren, der eine Referenz zugeordnet ist, und die Protokolle, die es unterstützt.

Die Objekte bei CORBA sind nur durch die Referenzen zu erreichen und von denen kann keine Kopie auf dem lokalen Rechner gemacht werden.

3.4 Client und Server

Die Begriffe Client und Server sind bei CORBA relativ, denn ein Objekt kann in verschiedene Szenarios seine Dienste anbieten oder Methoden von anderen Objekten aufrufen und damit beide Rollen übernehmen.

Der Client nutzt die vom Server-Programm bereitgestellten Services durch Aufruf der entsprechenden Operationen der Object Implementations. Dabei läuft der Zugriff auf ein entferntes Objekt für den Client vollkommen transparent ab. Der Client hat lediglich die Aufgabe, einen Methodenaufruf abzusetzen, ohne sich mit der Netzwerkkommunikation über den ORB befassen zu müssen.

Die Server-Objekte implementieren die in den IDL-Interfaces spezifizierten Schnittstellenbeschreibungen. Sie können in allen Sprachen realisiert werden, die vom ORB unterstützt werden. Hierunter fallen unter anderem C, C++, Java, Smalltalk und Ada.

3.5 Stubs und Skeletons

Die Stubs und Skeletons sind Teile der Architektur, die bei dem Kompilieren erzeugt werden.

IDL-Stubs stellen die Schnittstellen zu den Objektdiensten bereit. Aus der Sicht des Clients verhält sich der Stub wie ein Stellvertreter für ein entferntes Server-Objekt. Die Dienste werden mit Hilfe von IDL definiert, ihre zugehörigen Client-Stubs vom einem IDL-Compiler generiert. Ein Stub beinhaltet Code für die Umsetzung der Methodenaufrufe mit ihren Parametern in einfache Nachrichtenströme, die an den Server gesendet werden. Der Client fasst die Dienste als einfache Methodenaufrufe. Für den Client bleiben die zugrundeliegenden

Protokolle und auch die Datenanordnungen vollkommen transparent.

IDL Skeletons stellen das Gegenstück zu den Client-Stubs dar. Sie stellen die statischen Schnittstellen für jeden vom Server unterstützten Dienst bereit. Sie beinhalten Code, um eingehende Nachrichten in normale Methodenaufrufe umzusetzen und diese auszuführen. Rückgabewerte werden wieder über die Skeletons in einen binärem Strom umgewandelt und an den Client zurückgeschickt. Wie die Stubs werden die Skeletons vom IDL-Compiler aus IDL-Interfaces generiert.

3.6 Iterface Definition Language

Die Beschreibung des Verhaltens von Objekten erfolgt durch Spezifikation ihrer Schnittstellen mit Hilfe der Interface Definition Language IDL. IDL ermöglicht die plattform- und sprachenunabhängige Spezifikation dieser Schnittstellen. Eine solche IDL-Schnittstelle besteht aus benannten Operationen und den Parametern dieser Operationen. Dadurch versteckt die IDL Implementierungsdetails und veröffentlicht nur, welche Operationen verfügbar sind und wie diese aufgerufen werden. Das Ziel der Sprachenunabhängigkeit wird von der OMG dabei konsequent verfolgt. So werden neben elementaren Datentypen wie *boolean* oder *long* zum Beispiel auch Vererbungsbeziehungen auf der Ebene der IDL beschrieben. Für die Implementierung existieren sogenannte Language Binding-Spezifikationen(Language Mapping), die beschreiben, auf welche Weise IDL-Aufrufstrukturen und Datentypen auf die Strukturen der jeweiligen Programmiersprache abgebildet werden. Diese Spezifikationen existieren für viele wichtigen Programmiersprachen.

Kapitel 4

Interface Definition Language

Die Sprachenunabhängigkeit von IDL wird durch Sprachenabbildung (Language Mapping) sichergestellt. Dies ist eine Spezifikation, mit deren Hilfe IDL-Konstrukte auf die entsprechenden Konstrukte einer Programmiersprache abgebildet werden. Die OMG hat eine Anzahl von Standard-Sprachabbildungen für viele verbreitete Sprachen wie C, C++, Cobol und Java definiert.

CORBA wurde so spezifiziert, dass sie sprach- und betriebssystemunabhängig ist. Dazu ist eine einheitliche Schnittstellenbeschreibung der zu verteilenden CORBA-Objekte erforderlich. Diese Beschreibungssprache ist im CORBA-Standard als IDL (Interface Definition Language) definiert. Operationen und Attribute, die von einem Server der Äußenweltßur Verfügung gestellt werden, sind in einem Interface beschrieben. Aus einem vollständig beschriebenen IDL-Interface werden mit einem IDL-Compiler ein Stub für die Clients und ein Skeleton für den Server in der jeweiligen zu realisierenden Programmiersprache erzeugt. Ein IDL-Interface beschreibt lediglich die Schnittstellen der zu veröffentlichenden Dienste und Komponenten des Servers. Es enthält keine Implementierungsdetails. Dadurch wird es beispielsweise ermöglicht, dass in Java erstellte Clients auf die in C++ realisierte Server-Objekte zugreifen. IDL-Compiler werden mit den ORB-Produkten mitgeliefert. Die strikte Trennung der Schnittstellen von ihrer Implementierung ermöglicht es, bestehende Softwaresysteme in eine CORBA-Umgebung zu binden.

Die unten vorliegende Beschreibung ist ein Überblick über die wesentlichen Merkmale der CORBA-IDL.

4.1 Syntax

Die Syntax der CORBA IDL ist stark an C++ angelehnt. Alle Definitionen werden durch ein Semikolon abgeschlossen. Eine Folge von Definitionen, die in einer anderen Definition enthalten sind, wird mit geschweiften Klammern

gekennzeichnet. Nach der schließenden Klammer erscheint ein Semikolon. Kommentare können, wie bei vielen Programmiersprachen, gesetzt werden. Entweder bis zum Ende der Zeile // oder auf mehrere Zeilen zwischen /* und */.

4.2 C-Präprozessor

Zum Kompilieren von IDL Dateien wird vorausgesetzt, dass ein C-Präprozessor vorhanden ist. Der Präprozessor wird benötigt, um Makros wie *include* zu verarbeiten.

4.3 Allgemeine Konstrukte

IDL bietet die Konstrukte *module* und *exception* zur Verfügung. Sie werden im Folgenden erklärt.

4.3.1 Module

Ein Modul wird durch das Schlüsselwort *module* eingeleitet. Es stellt einen Namensraum bereit, um mehrere IDL-Schnittstellen, die einem gemeinsamen Zweck dienen, zu gruppieren. Dadurch kann eine zusätzliche Ebene in der Hierarchie im IDL-Namensraum eingeführt werden.

Die Verwendung von *module* wird im folgenden Beispiel gezeigt:

```
module Universitaet {
          interface Student {
                .....
           };
          interface Lecture{
        .
          };
          ...
};
module Bank {
          interface Konto {

          };
          ...
};
```

4.3.2 Exceptions

CORBA und IDL unterstützen die Exception-Behandlung über vordefinierte Standard- und benutzerdefinierte Exceptions vollständig. System-Ausnahmen können durch jede mögliche Methode ausgelöst werden (z.B. wenn der IPC ausfällt oder eine Abschaltung geschieht), während benutzerdefinierte Ausnahmen durch die Schnittstellen-Beschreibung ausdrücklich spezifiziert werden müssen. Wie das zu machen ist, wird im folgenden Beispiel gezeigt:

```
exception unbekanterTyp{string meldung};
```

Eine Option, die von IDL4 nicht unterstützt aber von CORBA zugelassen wird, ist, dass Ausnahmen zusätzliche Informationen enthalten (z.B. kann es nützlich sein, eine Nachricht für den Benutzer oder Tipps darüber, wie man die Störung behebt, hinzuzufügen).

4.4 Typen

Datentypen grenzen die möglichen Werte der Parameter, Attribute, Ausnahmen und Rückgabewerte ein, die sie annehmen können. CORBA unterscheidet zwischen einfachen und konstruierten Typen. Die einfachen Typen haben mit wenigen Ausnahmen festgelegte Größen und Wertebereiche. Im nächsten Abschnitt werden die Basistypen und die konstruierten Typen *struct, array, union* und *sequence* beschrieben. Jeder IDL-Datentyp von CORBA wird mit Hilfe von IDL-Compilern auf einen nativen Datentyp abgebildet.

4.4.1 Basis Typen

Zu den einfachen CORBA-Typen zählen *short, long, long long, unsigned short, unsigned long, unsigned long long, float, double, long double, char, wchar, boolean, octet, string, enum, void* und *any*. Sie sind in Tabelle 1 im Anhang beschrieben. Im Folgenden werden nur die interessanteren genauer beschrieben.

- Der Typ *enum* dient zum Erzeugen von Typen, die einen Wert aus einer vordefinierten Wertemenge enthalten können.

```
enum WochenTage{ Sonntag, Montag};
```

- Strings bestehen aus Listen von Characters (ASCII ISO Latin außer ASCII Null). CORBA unterstützt 8-bit und 16-bit Strings mit einer optionalen Längengrenze. Strings werden immer durch den Wert null beendet "terminated". Nachfolgend ein Beispiel:

```
{
typedef string<20> s20_t;
interface foo {
void bar(in string a, inout s20_t b, out wstring<40> c);
};}
```

  Das erste Argument zu *foo::bar* ist ein 8-bit String beliebiger Länge, während das zweite Argument höchstens 20 Buchstaben enthalten kann. Das dritte Argument ist ein 16-bit String von nicht mehr als 40 Elementen. Die ersten zwei Argumente werden durch ein null byte beendet, das dritte endet mit einem 16-bit Nullwert.

- IDL stellt den Typ *any* für Methoden zur Verfügung, die Parameter von allen möglichen Typen akzeptieren müssen. Der Typ*any* wird auch bei Methoden gebraucht, die eien Parameter erwarten, der potentiell von mehreren nicht zusammenhängenden Datentypen sein könnte (keiner der Typen ist von einem der anderen abgeleitet). Der Typ *any* kann jeden möglichen IDL-Datentyp repräsentieren.

```
void foo(in any x) raises (unbekanterTyp);
```

Die Funktion *foo* löst eine Exception aus, wenn sie mit einem Parameter aufgerufen wird, der zu keinem IDL-Typ gehört. Falls die Funktion feststellt, dass der Parameter x von einem bekannten Typ ist, wird die Verarbeitung fortgesetzt.

4.4.2 Konstrukt für Typen

IDL unterstützt durch die Anweisung *typedef* die Erzeugung eines Typnamens. Dieses wird zur besseren Anschaulichkeit von vielen Benutzern bevorzugt.

```
typedef boolean in_der_liste;
```

4.4.3 Zusammengesetzte Typen

- Der Typ *struct* erlaubt, ähnlich einer C-Struktur, das Zusammenpacken einer beliebigen Anzahl von unterschiedlichen Elementwerten in einen neuen Typ.

```
struct DatumStruktur{ short Jahr; short Monat; WochenTage Tag;};
```

- Der Typ *union* stellt Werte dar, die von verschiedenen Typen sind, von denen einer mit Hilfe des Entscheiders (Discriminator) ausgewählt wird.

```
union Multi switch(long){
        case 1:
                short s,
        case 2:
                double d,
        case 3:
        default:
                string str;
};
```

IDL bietet zwei Mglichkeiten für das Erzeugen von ContainerTypen:

- Der *sequence* Typ hat einen Basistyp und als zweites (optionales) Argument die maximale Länge. Dieser ist ein spezieller Typ für das Übertragen von Daten mit variabler Länge.

```
typedef sequence<float, 7> float_sec;
typedef sequence<char> char_sec;
```

Die erste Linie definiert eine Reihe von floats, die nicht maximal sieben Mitglieder enthält. Die zweite Linie definiert eine Buchstaben-Reihe beliebiger Größe. Sequences können nur mit *typedef* eingesetzt werden. Beim Verwenden von Seqences wird vom IDL4 Bufferraum zugeteilt. Es ist also sinnvoll, kleinere Größengrenzen zu benutzen. Das spart Speicher und verbessert die Leistung.

- Die Benutzung von Arrays ist nur unter der Bedingung, dass sie eine feste Grösse haben, erlaubt.

```
typedef string WochenTag[7];
```

Beide Container-Typen dürfen nur Elemente eines Typs oder der davon abgeleiteten Typen enthalten. Sie unterscheiden sich darin, dass Sequences ihre Größe dynamisch verändern können und die Arrays nicht.

4.5 Konstanten

IDL erlaubt die Angabe von konstanten Werten, durch die Anwendung des Modifizierers *const*.

```
{const float pi=3.14;}
```

4.6 Interfaces

In einer IDL-Datei sind ein oder mehrere Interfaces (Schnittstellen) beschreibt . Interfaces enthalten Methoden und Attribute. Eine Schnittstelle ist eine rein deklarative Beschreibung der Dienste, die von einem Client genutzt werden können bzw. die von den CORBA-Objekten bereitgestellt werden. Diese Dienste erscheinen in Form von Methoden und enthalten keine Details über ihre Implementierung. IDL-Schnittstellen unterstützen die Vererbung, deren Konzept man aus C++ oder Java kennt.

4.6.1 Methoden

Die Methoden definieren die Funktionalität eines Objekts. Die IDL definiert die Signatur einer Operation, bestehend aus ihren Parametern und Rückgabewerten. Wenn eine Methode eines Remote-Objekts aufgerufen wird, dann wartet das aufrufende Objekt auf die Ausführung der Methode. Erst wenn die Antwort auf die Anfrage zurückgekommen ist (gegebenenfalls mit den Rückgabewerten), wird die weitere Verarbeitung fortgesetzt. Methoden haben die Option als *oneway* deklariert zu werden. Wurde die Methode als *oneway* deklariert, bedeutet dies, dass das aufrufende Objekt mit der Verarbeitung fortfahren wird, gleich nachdem das ORB das Server-Objekt gefunden hat und die Anfrage geschickt wird. Eine *oneway*-Methode kann keinen Wert zurückliefern, darf nur über *in* Parameter (Begriff wird im Kapitel Parameter eingeführt) verfügen und kann

keine Exception auslösen, also hat man nicht die Möglichkeit festzustellen, ob die Methode erfolgreich ausgeführt ist.

4.6.2 Parameter

Die Methoden können beliebige IDL-Datentypen als Parameter verwenden. Alle Parameter haben ein Orientierungsattribut (*in, out* oder *inout*). Dieses dient dazu, der Inhalt des Parameters zuzuordnen in eine von den folgenden drei Kategorien:

- Input Daten (sie müssen vom Client auf dem Server kopiert werden)
- Output Daten (sie werden vom Server zurückgegeben)
- In-/Output Daten (Kombination von beiden)

Eine Methode, die Rückgabewert des Typs *void* und nur *in* Parameter hat und keine Exceptions auslöst, ist nicht automatisch eine *oneway*-Methode, da nur die erfolgreiche Ausführung dieser Methode oder die Auslösung einer CORBA-System-Exception das Warten auf den Rückgabewert (auch wenn dieser Null ist) beenden kann.

Eine Schnittstelle verfügt neben den Methoden auch über Attribute.

4.6.3 Attribute

Die Attribute einer IDL-Schnittstelle entsprechen den Attributen in einer C++ oder Java-Klasse, nur sind IDL-Attribute immer als *public* zu betrachten.

Beim Kompilieren eines IDL-Quelltextes, wird eine Attribut-Definition, wie im folgenden Beispiel gezeigt, abgebildet:

```
//IDL Deklaration
attribute short meinWert;

//Ausgabe des Compilers
short meinWert;
void meinWert( short wert);
```

Kapitel 5

Der Compiler

Nachdem die IDL-Definitionen für die Objektschnittstellen des Systems erstellt und der Implementierungsansatz gewählt worden sind, ist nun alles für die Kompilierung der IDL-Datei vorbereitet. Es wird nur noch ein IDL-Compiler gebraucht. Es gibt verschiedene IDL-Compiler die für verschiedene Programmiersprachen bestimmt sind. IDL4 ist ein Stub-Code Generator für die L4-Plattform, der Dateien in der Sprache C produziert. Er generiert Kommunikationsstubs von Schnittstellen-Definitionen, die in CORBA-IDL oder in DCE-IDL geschrieben sind. Nachfolgend werden die Eigenschaften von IDL4 beschrieben.

IDL4 unterstützt die Typen *struct, array, union, sequence, short, long, long long, unsigned short, unsigned long, unsigned long long, float, double, long double, char, wchar, boolean, octet, string, wstring, enum, void* und *fpage*. Sie sind im Kapitel 4 oder im Anhang beschrieben und gehören zum OMG Standard. IDL4 unterstützt L4 mapping primitives durch den speziellen Typ *fpage*. *fpage* entspricht dem flexpage Typ vom jeweiligen Kernel Interface und seine Größe ist plattformabhängig.

Beim Einsetzen von typedef ist es interessant zu wissen, dass IDL4 die Typen in einer Schnittstellen-Beschreibung zur Zielsprache abbildet und den header Files, die sie produziert, eine Definition hinzufügt. So können diese Typen auch in dem eigenen Code verwendet werden.

IDL4 wird wie folgt aufgerufen:

idl4 [Optionen] schnittstelledefinition.idl

und hat drei mögliche Ausgaben, die mit der passenden Option bestimmt werden.

- client stubs, die mit jeder Klient Anwendung verbunden werden, d.h. jede Anwendung, die Methoden von der Schnittstelle aufrufen muss
- server stubs, die von jedem Server benutzt werden, der die Schnittstelle implementiert
- server templates, die im Wesentlichen eine "dummyÏmplementierung für

die Schnittstelle enthalten. Sie können als Ausgangspunkt für das Schreiben eines Servers benutzt werden.

Die client und server stubs werden erzeugt in der Form von einer Header-Datei und werden dann inkludiert in den C-Dateien. Das Server-Template ist eine C-Datei, die vom Entwickler mit Code quasi ausgefüllt werden kann. Die Optionen des Compilers werden ausführlich im IDL4 User's Manual[1] beschrieben. Interessanter ist die Anwendung vom IDL4. Sie wird im nächsten Abschnitt kommentiert.

5.1 Beispiele

Wie durch das CORBA C-language mapping spezifiziert, haben client stubs zwei implizite Parameter (d.h. Parameter, die nicht durch die Schnittstelle definiert werden). Nachfolgend ein Beispiel:

Listing 5.1: IDL-Datei

```
Test.idl

module storage{
        interface textfile{
                void readln(inout short pos, out string line);
        };
};
```

Wenn diese Definition kompiliert wird, stellt IDL4 den folgenden client stub her:

Listing 5.2: Client-Stub

```
Test_client.h


static inline void storage_textfile_readln(
                                        storage_textfile _service,
                                        CORBA_short *pos,
                                        CORBA_char **line,
                                        CORBA_Environment *_env){
```

Der erste Parameter (_service) enthält die Thread-ID (Threads sind Unterteilungen innerhalb von Prozessen[3]) des Servers, dem der Antrag gesendet werden soll. Anders als andere CORBA-Compiler stellt IDL4 nicht eine Weise zur Verfügung, dieses automatisch zu erreichen, er nimmt an, dass diese Funktionalität durch Ihr System eingeführt wird. Der letzte Parameter ist ein Zeiger zu einer CORBA_Environment Struktur. Diese Struktur enthält die zusätzlichen Informationen für den Aufruf, wie einen timeout-Wert. Diese Struktur muss initialisiert werden, bevor der Aufruf hervorgerufen wird. Eine Anforderung von readln konnte wie diese aussehen:

Listing 5.3: Implementierungsmöglichkeit

```
Client.c
#include<test_client.h>
void testen(l4_threadid_t server) {
        CORBA_Environment env =idl4_default_environment;
        short pos = 100;
        char *buf;

        storage_textfile_readln(server, &pos, &buf, &env);

        switch (env._major) {
                case CORBA_SYSTEM_EXCEPTION:
                        printf("IPC failed, code %d\n",
                        CORBA_exception_id(&env));
                        CORBA_exception_free(&env);
                        return -1;
                case CORBA_USER_EXCEPTION:
                        printf("User-defined exception");
                        CORBA_exception_free(&env);
                        return -1;
                case CORBA_NO_EXCEPTION:
                        break;
        }

...
```

Das Beispiel zeigt, wie env mit einem IDL4-Default-Wert initialisiert wird. Das bedeutet zum Beispiel, dass der timeout-Wert gleich unendlich wird. Es zeigt auch, wie eine environment-Struktur benutzt werden kann, um festzustellen, ob eine Ausnahme während des Aufrufs auftrat, und welche Art es war.

Der Standard Server-Loop, der in dem Template eingeschlossen ist, besteht hauptsächlich aus zwei Makros:

idl4_reply_and_wait, das Antworten sendet und dann einen ankommenden Antrag empfängt, und

idl4_process_request, das den Antrag analysiert und die passende Funktion aufruft.

Zwischen diesen Makros kann zusätzlicher Code eingesetzt werden. Das zweite Makro benutzt eine Funktions-Tabelle, um zu entscheiden, welche Funktion den Antrag bearbeiten sollte. Es nimmt die Methoden-Nummer als Argument und verwendet sie als Index in der Tabelle. Anträge für die Tabelle, die nicht identifizierte Methoden-Nummern haben, bekommen eine Referenz auf der *discard*-Funktion der Schnittstelle. Die *discard*-Funktion wird nur dann aufgerufen, wenn ein falscher Antrag empfangen wurde.

Das Server-Template enthält auch Funktions-Schablonen für jede Methode in der Schnittstelle. Für die Beispielschnittstelle von oben wird das folgende Template erzeugt:

Listing 5.4: Server-Template

```
Test_template.c

#include "test3_template_server.h"
...
IDL4_INLINE void storage_textfile_readln_implementation(
                                          CORBA_Object _caller,
                                          CORBA_short *pos,
                                          CORBA_char **line,
                                          idl4_server_environment *_env){
  /* implementation of storage::textfile::readln */

  return;
}
...
void storage_textfile_server(void)

{

while (1)
    {
      buffer.size_dope=STORAGE_TEXTFILE_RCV_DOPE;
      buffer.rcv_window = idl4_nilpage;
      partner = idl4_nilthread;
      reply = idl4_nil;
      w0 = w1 = w2 = 0;

      while (1)
        {
          idl4_reply_and_wait(reply, buffer, partner,
                              msgdope, fnr, w0, w1, w2, dummy);

          if (msgdope & 0xF0)
            break;

          idl4_process_request(storage_textfile_vtable,
                               fnr & STORAGE_TEXTFILE_FID_MASK,
                               reply, buffer, partner, w0, w1, w2, dummy);
        }

      enter_kdebug("message_error");
    }
}
...
```

Ähnlich der Klient-Seite, hat die Funktion zwei zusätzliche Parameter. Der erste Parameter (_caller) enthält die Identifikation des Threads, der den Antrag gesendet hat, während der letzte Parameter (_env) auf eine interne Datenstruktur zeigt. Wichtig ist das Makro am Ende der Funktion. Dieses macht die Funktion zugänglich zum Server-Loop. Es ist wichtig, dass dieses Makro nach der Implementierungsfunktion positioniert ist. Anders als die Client-Seite wird Speicher Allozierung auf der Serverseite meistens durch den Stub-Code geregelt, das heißt, dass *CORBA_free* nicht aufzurufen ist, ausgenommen, wenn zusätzliche Puffer zugeteilt wurden. Der Stub-Code teilt einen Puffer für jeden *out*-Wert zu und schickt ein Zeiger zur Implementierungsfunktion im jeweiligen Argument. Wenn es vom Entwickler entschieden wird, keine Antwort zu senden, kann die *idl4_set_no_response* Funktion verwendet werden. In diesem Fall wirft der Stub-Code alle *inout* oder *out* Werte weg und überspringt die Sende-Phase.

Kapitel 6

Zusammenfassung

In dieser Ausarbeitung wurden die Eckpfeiler der Architektur von CORBA vorgestellt. Unter anderem der ORB, der die Kommunikation der CORBA-Objekte untereinander sicherstellt, und die IDL, mit der die Schnittstellen der Anwendungskomponenten definiert werden, von denen ausgehend CORBA-Anwendungen entwickelt werden. Es wurde mit dem Objektmodell von CORBA Bekanntschaft gemacht und einiges über die Protokolle zur Kommunikation zwischen ORBs (insbesondere IIOP) berichtet. Die Verwendung von Objektreferenzen in CORBA wurde ebenfalls vorgetragen. Es ist erklärt worden, wie die Begriffe Client und Server in CORBA zu verstehen sind, dass eine einzige Anwendungskomponente gleichzeitig als Client und als Server fungieren kann. Es wurde außerdem gezeigt, wie über IDL-Definitionen Client-Stubs und Server-Skeletons erzeugt werden, die wiederum zur Implementierung von CORBA-Clients und -Servern dienen.

In Kapitel 4 wurden die grundlegenden Datentypen der CORBA-Programmiersprache Interface Definition Language (IDL), wie numerische Integer- und Gleitkommawerte, boolesche Werte sowie Zeichen und Zeichenketten, vorgestellt. Da viele dieser Datentypen sehr den Datentypen aus Programmiersprachen wie C, C++ und Java ähneln, werden Leser, die mit einer dieser Programmiersprachen vertraut sind, keine Probleme mit den IDL-Typen haben. Einige nützliche IDL-erweiterten Datentypen und -Konstrukte wurden vorgestellt, insbesondere die Typen *sequence* und *array*, die zum Speichern von mehreren Werten eines änlichen Typs verwendet werden. Am wichtigsten ist, dass sich mit dem Schnittstellen-Konstrukt vertraut gemacht wurde. Es ist bekannt worden, wie dieses Konstrukt das Verhalten von CORBA-Objekten definiert. Da *interface* einer der fundamentalen IDL-Datentypen ist, ist ein klares Verstehen dieses Konstrukts für den Entwurf von CORBA-Anwendungen wesentlich. Sie können ohne Schnittstellen keine CORBA-Anwendungen entwerfen oder implementieren. Es wurden außerdem die Exceptions in IDL präsentiert. Abschließend wurde der Typ *any* untersucht, der einen Wert eines beliebigen IDL-Typen enthalten kann. Diese Konstrukte, insbesondere *sequence* und *exception*, sind essenziell für die Erzeugung von stabilen CORBA-Anwendungen.

CORBA hat bereits einen weiten Weg zurückgelegt. Mit einer derzeitigen Mitgliederzahl von über 800, die noch weiter steigt, hat die OMG endgültig eine bedeutende Stellung auf dem Markt erreicht. CORBA wächst und reift als Standard bereits seit mehr als dreizehn Jahren, was im Zeitalter des Computers eine wahre Ewigkeit ist. CORBA-Produkte sind für eine Vielzahl von Plattformen und Betriebssystemen verfügbar, die von Dutzenden von Anbietern stammen. (Zu den von CORBA unterstützten Plattformen gehören AS400, HP-UX, MacOS, MS-DOS, MVS, OpenVMS, OS/2, SunOS und Solaris, Windows 3.x, Windows 95, Windows NT sowie viele Unix-Varianten, die noch nicht genannt wurden.) Es ist nicht zu übersehen, dass sich CORBA aus bescheidenen Anfängen zu einer etablierten, reifen Plattform für die Entwicklung verteilter Anwendungen gemausert hat.

Unter anderem werden, für die Entwicklung CORBA-Anwendungen, IDL-Compiler benötigt. IDL4 ist ein experimenteller IDL-Compiler. Mit seiner Hilfe werden Dateien in C generiert, die dann bei der Entwicklung von CORBA-Objekten eingesetzt werden. IDL-Compiler können für jede Programmiersprache gebaut werden. Bei Sprachen, die nicht zum OMG-Standard gehören, muss demensprechend festgelegt werden, wie die Bestandteile der Sprache, wie Konstrukte und Typen, in IDL abgebildet werden.

Kapitel 7

Anhang

Sheet1

Name	Größe in bits	Typ	Value range
short	16	Signed integer	-(2hoch15).. ..(2hoch15)-1
long	32	Signed integer	-(2hoch31).. ..(2hoch31)-1
long long	64	Signed integer	-(2hoch63).. ..(2hoch63)-1
unsigned short	16	Unsigned integer	0..(2hoch16)-1
unsigned long	32	Unsigned integer	0..(2hoch32)-1
unsigned long long	64	Unsigned integer	0..(2hoch64)-1
float	32	Floating point	3,403*(10hoch38)
double	64	Floating point	1,798*(10hoch308)
Long double	80	Floating point	1,1897*(10hoch4932)
char	8	Charakter	
wchar	16	Charakter	
boolean	1	Boolean value	true, false
octet	8	Unsigned integer	0..255
void	nichtdefiniert		keine
string			

Page 1

Abbildung 7.1: IDL basis Typen

Literaturverzeichnis

[1] A. Haeberlen. *IDL4 User's Manual*. University of Karlsruhe, April 2003.

[2] J.Rosenberg. *CORBA in 14 Tagen*. 1998.

[3] Steffen Knapp. Asynchronous ipc via synchronous ipc.

[4] A. Haeberlen J. Lidke Yoonho Park Lars Reuther Volkmar Uhlig. Stub-code performace is becoming omportant.