Friedrich-Alexander-Universität Erlangen-Nürnberg

Konzepte von Betriebssystemkomponenten

CORBA (Überblick, IDL)

Radu Vatav

1. Geschichte

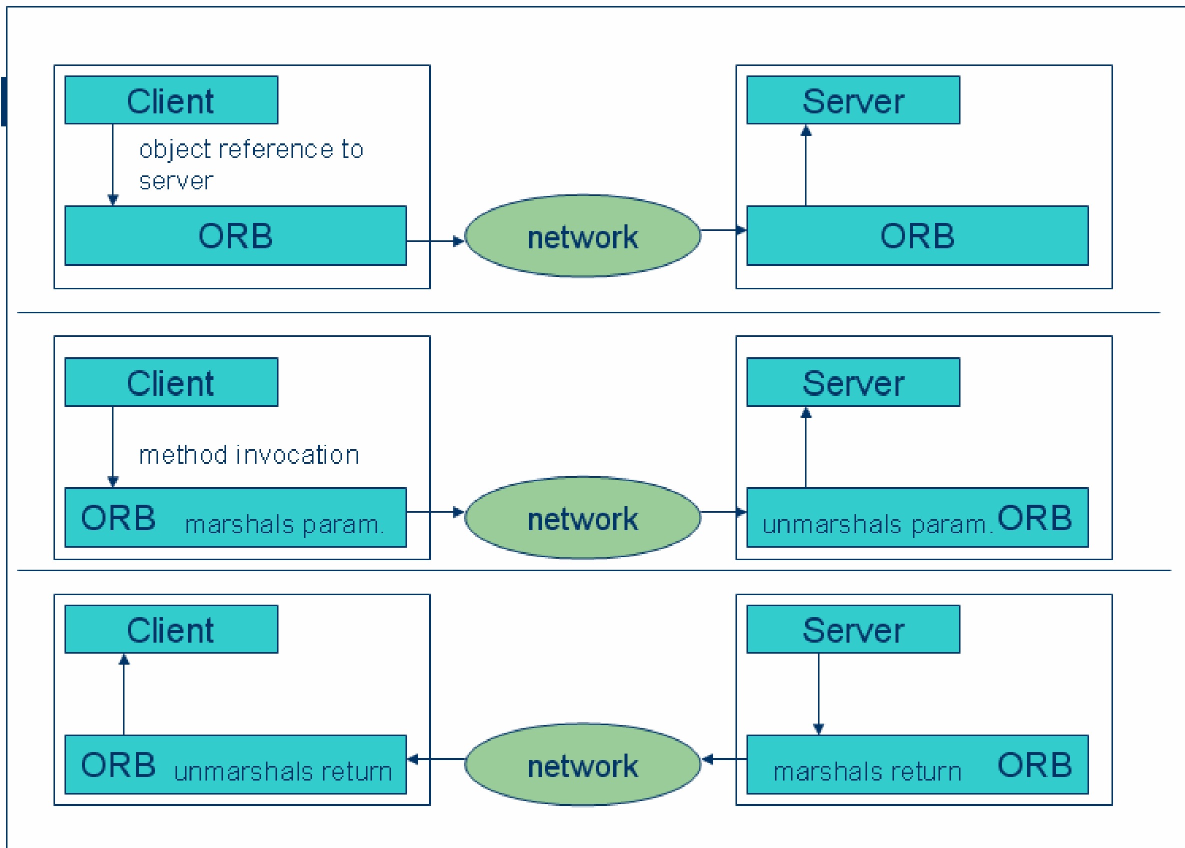
Die Object Management Group (OMG), 1989 gegründet, hatte das Ziel "ein gemeinsames architekturbezogenes Gerüst für objektorientierte Anwendungen zur Verfügung zu stellen, das auf weit verbreiteten Schnittstellenspezifikationen basiert ". Das erreicht die OMG mit der Einführung der OMA (Object Management Architecture), zu deren Bestandteilen CORBA gehört. Diese Standards liefern das architekturbezogene Gerüst, anhand dessen Anwendungen entwickelt werden. OMA besteht aus der Funktion ORB (Object Request Broker), Objektdiensten (CORBAservices), gemeinsamen Einrichtungen (CORBAfacilities), Domänenschnittstellen und Anwendungsobjekten. Die Funktion von CORBA innerhalb der OMA besteht darin, die ORB-Funktion zu implementieren.

Nach der Gründung von OMG wurde CORBA 1.0 im Dezember 1990 eingeführt. Anfang 1991 folgte CORBA1.1, Version in welcher die Interface Definition Language (IDL) und die API für Anwendungen zur Kommunikation mit einem Object Request Broker (ORB) definiert waren. Durch die 1.x Versionen wurde ein erster wichtiger Schritt in Richtung auf Objektinteroperabilität gemacht, denn sie ermöglichen es Objekten auf unterschiedlichen Rechnern, die mit Hilfe von unterschiedlichen Architekturen und in unterschiedlichen Sprachen geschreiben waren, miteinander zu kommunizieren.
1996 folgte CORBA 2.0, die ein Standardprotokoll, über das die ORBs der verschiedenen CORBA-Anbieter miteinander kommunizieren konnten, definierte. Dieses Protokoll heisst Internet Inter-ORB Protocol (IIOP) und muss von allen Anbietern implementiert werden, die ihre Produkte als CORBA 2.0-kompatibel bezeichnen wollen.
Die nächsten 2.x Versionen, die zwischen 1997 und 2001 erschienen, brachten verschiedene Weiterentwicklungen und Spezifikationen, wie Java Language Mapping, C++ Language Mapping, Messaging Specification, Portable Interceptors.

2. Architektur

2.1 Der Object Request Broker (ORB)

Dem ORB liegt folgendes Konzept zugrunde: Wenn eine Anwendungskomponente einen Dienst nutzen möchte, der von einer anderen Komponente zur Verfügung gestellt wird, muß sie zunächst eine Objektreferenz für das Objekt erlangen, welches den Dienst zur Verfügung stellt. Nachdem die Objektreferenz erlangt wurde, kann die Komponente Methoden des Objekts aufrufen und damit auf die vom Objekt zur Verfügung gestellten Dienste zugreifen. Der Entwickler der Client-Komponente weiss zum Zeitpunkt des Kompilierens, welche Methoden eines bestimmten Server-Objekts zur Verfügung stehen.
Die Hauptaufgabe des ORB ist die Auflösung der Anfragen von Objektreferenzen, so dass die Anwendungskomponenten die Konnektivität miteinander herstellen können.



Die Formatübertragung (Marshaling) ist die Umsetzung der Eingabeparameter in ein Format, das über ein Netzwerk zum Remote-Objekt übertragen werden kann. Der ORB macht auch das Umgekehrte (Unmarshaling). Die gesamte Formatübertragung erfolgt ohne jeglichen Eingriff des Programmierers, der Client ruft einfach die gewünschte Remote-Methode auf und bekommt ein Ergebnis zurück, als wäre es eine lokale Methode. Ein anderer Ergebnis von Marshaling besteht darin, dass die Kommunikation zwischen den Komponenten plattformunabhängig stattfindet, weil die Parameter für die Übertragung in ein plattformunabhängiges Format konvertiert werden.

2.2 Das Kommunikationsmodell

Die CORBA-Spezifikation ist hinsichtlich der Netzwerkprotokolle neutral. Der CORBA-Standard legt das GIOP (General Inter-ORB Protokol, allgemeines Protokoll für die Kommunikation zwischen ORBs) fest, das auf einer hohen Ebene einen Standard für die Kommunikation zwischen verschiedenen CORBA-ORBs und -Komponenten definiert.

2.3 Das Objektmodell

Obwohl für die Benutzer die CORBA-Objekte transparent sind, sind diese verteilt, was eine höhere Fehlerwahrscheinlichkeit mit sich bringt (Netzwerkausfälle, Server-Abstürze usw.). CORBA muss also Ausweichmöglichkeiten bieten, die in solchen Fällen greifen. Dies erfolgt durch Bereitstellung von System-Exceptions, die durch jede Remote-Methode ausgelöst werden können.
Noch ein wichtiger Aspekt des Objektmodells von CORBA besteht darin, dass die Objekte durch Referenz weitergegeben werden.

2.4 Object Adapters

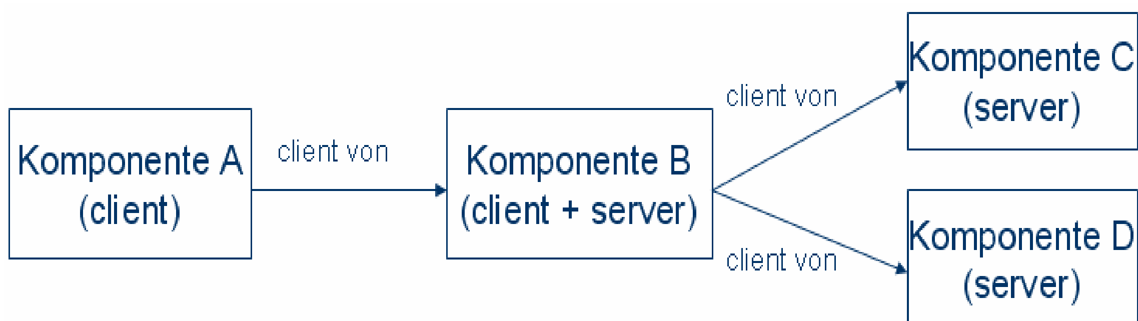
Der CORBA-Standard beschreibt eine Anzahl sogenannter Objektadapter, deren Haptzweck es ist, die Schnittstelle zwischen einer Objektimplementierung und dem zugehörigen ORB zur Verfügung zu stellen. Die OMG stellt drei Gebrauchsmuster für Objektadapter zur Verfügung: den BOA (Basic Objekt Adapter), den Bibliotheksobjektadapter (Library Object Adapter) und den objektorientierten Datenbankadapter) Object-Oriented Database Adapter) Die Beiden letzteren dienen zum Zugriff auf Objekte im dauerhaften Speicher. Der BOA stellt für CORBA eine allgemeine Gruppe von Methoden für den Zugriff auf ORB-Funktionen zur Verfügung. Diese Funktionen reichen von der Benutzeridentifikationsüberprüfung über die Objektaktivierung bis hin zur Objektpersistenz.
Ein wichtiges Merkmal des BOA sind seine Funktionen zum Aktivieren und Deaktivieren von Objekten. Der BOA unterstützt vier verschiedene Aktivierungsstrategien, die angeben, wie die Anwendungskomponenten initialisiert werden sollen. Diese Strategien sind:

- der gemeinsam verwendeten Server (ein einziger Server(Prozess) wird von mehreren Objekten verwendet)
- der nicht gemeinsam verwendeten Server (ein Server enthält immer nur ein Objekt)
- der Server-pro-Methode (ein Server wird automatisch gestartet, wenn eine Objektmethode aufgerufen wird)
- der permanenten Server (der Server wird manuell gestartet)

So wird es dem Anwendungsentwickler ermöglicht, die Art von Verhalten auszuwählen, die für den betreffenden Server am sinnvollsten ist.

2.5 Clients und Server

Wenn eine Komponente ein Objekt erstellt und anderen Komponenten die Sicht auf dieses Objekt ermöglicht, fungiert sie als Server für das Objekt, sie führt also Methoden für andere Komponenten (Clients) aus. In einer CORBA-Anwendung kann eine Komponente sowohl als Client als auch als Server fungieren, wird aberals Client oder Server bezeichnet, nicht als beides. Die Client-Callback-Methode ist eine allgemeine Bezeichnung für eine Methode, die von einem Client implementiert und von einem Server aufgerufen wird. Durch einen Callback wird ein Client zu einer Art beschränktem Server.



2.6 Stubs und Skeletons

(Client-) Stubs und (Server-) Skeletons werden von einem IDL-Compiler aus den vom Entwickler für die Komponentenschnittstelle geschriebenen Definitionen erzeugt. Die Stubs und Skeletons verbinden die sprachenunabhängigen IDL-Schnittstellenspezifikationen mit dem sprachspezifischen Implementierungsquelltext.
Client-Stubs für jede Schnittstelle werden zur Einbindung von Clients zur Verfügung gestellt, die diese Schnittstellen nutzen. Der Client-Stub für eine bestimmte Schnittstelle stellt eine Pseudoimplementierung für jede Methode in der Schnittstelle zur Verfügung. Anstatt Server-Methoden direkt auszuführen, kommunizieren die Methoden des Stubs mit dem ORB, damit für die benötigten Parameter eine Formatübertragung bzw. eine Umgekehrte Formatübertragung durchgeführt wird.
Auf der anderen Seite stehen die Skeletons, die das Gerüst bilden, auf dem der Server erzeugt wird. Für jede Methode einer Schnittstelle generiert der IDL-Compiler eine leere Methode im Server-Skeleton. Der Entwickler stellt dann für jede dieser Methoden die Implementierung zur Verfügung.

3. Interface Definition Language (IDL)

Die IDL dient zur Definition von Schnittstellen zwischen einzelnen Anwendungskomponentnen. Sie ist nicht eine prozedurorientierte Sprache, sondern dient einzig und allein zur Definition von Schnittstellen und nicht von Implementierungen (ähnlich einer C++ header-Datei).
Die IDL-Spezifikation ist dafür verantwortlich, dass die Daten zwischen Anwendungen in unterschiedlichen Sprachen korrekt ausgetauscht werden. Wenn es sich z.B. beim IDL-Typ *long* um eine vorzeichenbehaftete 32 Bit große Zahl handelt, kann diese ein C++ *long* sein (je nach Platform) oder ein Java *int*.

3.1 Grundregeln von IDL

IDL unterscheidet bei einzelnen Bezeichnern die Groß-/Kleinschreibunt, z.B. kann man sich auf einer Schnittstelle *meinObjekt* nicht mit *meinOBJEKT* beziehen. Ausserdem dürfen sich Namen von Bezeichnern im gleichen Gültigkeitsbereich nicht nur in der Schreibweise unterscheiden, z.B. sind gleichzeitig *meineFkt* und *meineFKT* nicht erlaubt.

Ähnlich wie in C, C++ und Java gilt in IDL noch folgendes:

- Definitionen werden durch ein Semikolon (;) abgeschlossen
- für Definitionen, die andere Definitionen beinhalten (z.B. Module u. Schnittstellen), werden geschweifte Klammern ({ }) verwendet
- Kommentare sind sowohl im C- als auch im C++-Stil erlaubt

```
// C++-Stil Kommentar
/*
   C-Stil Kommentar
   ...
*/
```

- C-Präprozessor:
  IDL geht davon aus, dass für die Verarbeitung von Konstrukten, wie z.B. Makro-Definitionen und bedingte Kompilierung, ein C-Präprozessor vorhanden ist.

- Module

Das Konstrukt *module* wird verwendet, um IDL-Definitionen, die einem gemeisamen Zweck dienen, zusammen zu gruppieren.

```
Module Bank {
     Interface Kunde {
     ...
     }
     interface Konto {
     ...
     };
};
```

Einfache Typen, ähnlich wie in C, C++ und Java:

- void
- boolean - true und false
- char - 8 Bit
- wchar - implementationsabhängig, normalerweise 16 Bit
- float - IEEE einfache Genauigkeit
- double - IEEE doppelte Genauigkeit
- long double - IEEE erweiterte doppelte Genauigkeit
- (unsigned) long - 32 Bit Integer
- (unsigned) long long - 64 Bit Integer
- (unsigned) short – 16 Bit
- octet – C und C++: *char* und *unsigned char*, Java: *byte*
- string (feste und variable Zeichenlänge) – C: Zeichenarray, C++: *Cstring*, Java: *String*
- enum (Erzeugung von Typen mit vordefinierte Wertemenge):

```
enum Tage {
     Montag,
     Dienstag,
     ...
  };
```

- struct (Strukturtyp, unterschiedliche Elementwerten, mit call-by-value übergeben):

```
struct Datum {
     short Jahr,
     short Monat,
     short Millisekunde
};
```

- union (Werte verschiedener Typen, Mischung aus C/C++ *union* und *case* –Anweisung):

```
union MeineUnion switch(long) {
     case 1:
          short meinShortWert;
     case 2:
          double meinDoubleWert;
     default:
          string meinString;
};
```

- interface: wie Java- *interface*, beschreibt die von CORBA-Objekten bereitgestellten Dienste (Methoden)

3.2 Methoden und Parameter

Die allgemeine Syntax einer Methodendeklaration ist folgende:

```
[oneway] return_type MethodenName (param1_dir
param1_typ, param2_name, param2_dir param2_typ param2_name, ...);
```

param_typ gibt den Typen des Parameters an
param_dir (*in, out* und *inout*):
  in – Eingabe der Methode
  out – Ausgabe der Methode
  inout – Eingabe und Ausgabe der Methode

  Vor Aufruf der Methode werden *in* und *out* Parameter zum Remote-Objekt übertragen. Nach dem Ausführen der Methode werden alle *in* und *inout* Parameter, zusammen mit dem Rückgabewert (falls vorhanden), zurück zum aufrufenden Objekt übertragen.

oneway
  Im Normalfall ruft das aufrufende Objekt eine Remote-Methode auf und blokiert bis es das Ergebnis zurückbekommt. Wenn der *oneway* Modifizierer verwendet wird, führt das aufrufende Objekt seine Verarbeitung gleich nach der Anfrage fort, während das Remote-Objekt die Remote-Methode ausführt. Diese liefert aber in diesem Fall keinen Wert zurück, deshalb muss die *oneway*-Methode als *void* und alle Parameter als *in* deklariert werden.

3.3 Attribute

```
[readonly] attribute attribut_typ attributName;
```

IDL-Attribute zeigen auf Datenzugriffs- und Datenänderungsmethoden, wenn der IDL-Quelltext kompiliert wird. So bildet die Definition

```
   attribute short meinChannel
```

folgende zwei Methoden ab:

```
   short meinChanel;
   void meinChanel(short value);
```

readonly bedeutet dass der Wert eines Attributes durch einen externen Objekt nicht geändert warden kann.

3.4 Vererbung von Schnittstellen

Eine Schnittstelle kann Methoden und Attribute einer oder mehreren Schnittstellen vererben.

```
Interface A {...};
Interface B {...};
Interface C : A {
   // eigene Attribute und Methoden
};
Interface D : A, B {
```

```
   // eigene Attribute und Methoden
};
```

3.5 Vorwärtsdeklarationen

Eine zirkuläre Abhängigkeit tritt ein, wenn zwei Schnittstellen jeweils über Attribute oder Methoden verfügen, die sich auf die andere Klasse beziehen. Eine Vorwärtsdeklaration teilt dem IDL-Compiler mit, dass der deklarierte Typ später definiert wird.

```
module Zirkulaer {
   interface B;
   interface A {
         void verwendeB(in B einB);
   }
   interface B {
         void verwendeA(in A einB);
   }
};
```

3.6 Exceptions

Wenn eine Exception ausgelöst wird, reicht sie das ORB an das ORB des aufrufenden Objekts, welches dann die Exception an das aufrufende Objekt zurückgibt.
IDL-Exceptions werden mit dem Typen *exception* deklariert. Dieser Typ enthält, ähnlich wie *struct*, verschiedene Datenelemente (jedoch keine Methoden). Vererbung von Exception-Typen wird nicht unterstützt. CORBA stellt auch Standard-Exceptions (System-Exceptions) bereit, die von jedem Remote-Aufruf ausgelöst werden können.

Literatur:
Jeremy Rosenberger, „CORBA in 14 Tagen“