Ein Object Request Broker für Java*

Gerald Brose und Boris Bokowski

Institut für Informatik, Freie Universität Berlin
Takustrasse 9, D-14195 Berlin
{brose,bokowski}@inf.fu-berlin.de

Wir beschreiben die Verteilungsplattform JacORB, einen Object Request Broker nach dem CORBA-Standard der OMG, bei dessen Entwicklung versucht wurde, einen hohen Grad an Verteilungstransparenz zu erreichen. JacORB wurde vollständig in Java implementiert und erlaubt sowohl die Integration von Java–Objekten in heterogene CORBA–Anwendungen als auch die Verteilung reiner Java–Anwendungen.

1 Einführung

Wir beschreiben die Architektur und die Entwicklung von JacORB, einem Object Request Broker, der die Entwicklung verteilter Anwendungen in Java[Gosling et al. 96] und ihre Integration in CORBA–Umgebungen [OMG 95] erlaubt, sowie dabei gewonnene Erfahrungen mit Java. Aufgrund der Nähe der Objektmodelle von CORBA und Java liess sich bei der Erweiterung des Java–Objektmodells um CORBA–Konzepte ein hoher Grad an Verteilungstransparenz erreichen. Das bedeutet, dass die Entwicklung von CORBA–Applikationen mit JacORB sich nur wenig von der herkömmlichen Entwicklung in Java unterscheidet.

2 CORBA

Mit CORBA [OMG 95] hat die *Object Management Group* (OMG) einen Standard für die Infrastruktur ihrer *Object Management Architecture* (OMA) definiert, der Objektfernaufrufe in heterogenen und verteilten Umgebungen erlaubt. Zu diesem Zweck legt CORBA ein eigenes Objektmodell fest, das sich in der *Interface Definition Language* (IDL) von CORBA manifestiert.

Die Schnittstellen von CORBA–Objekten werden in IDL definiert. Ihre Implementierung kann in jeder Programmiersprache erfolgen, für die ein *IDL Language Mapping*, eine Abbildung der IDL–Konzepte in diese Sprache, vorliegt. Die OMG hat in [OMG 95] Sprachabbildungen für C, C++ und Smalltalk definiert und arbeitet zur Zeit an einer Abbildung für Java, deren Standardisierung jedoch noch nicht abgeschlossen ist.[1]

1. Für diesen zu verabschiedenden Standard gibt es jeweils einen Vorschlag von IONA und Xerox sowie einen gemeinsam erarbeiteten der Firmen IBM, Netscape, VisiGenic, HP, Oracle, Novell, Sun, Digital und Expersoft.

* in: *Informatik/Informatique*, Zeitschrift der schweizerischen Informatikorganisationen, No. 3/97, 27-30.

CORBA–Objekte sind durch einen *Object Request Broker* (ORB) verbunden, der die verteilungstransparente Kommunikation zwischen Objekten ermöglicht. Die Implementierung des Kommunikationsmechanismus ist dabei nicht festgelegt, jedoch gibt es ein vorgeschriebenes Protokoll zur Kommunikation zwischen verschiedenen ORBs, das Objektaufrufe auch über ORB–Grenzen hinweg ermöglicht. Dieses Protokoll heisst *Inter–ORB Protocol* (IOP) und definiert Nachrichtenformate sowie eine Trans-fersyntax für Datentypen. IOP kann auf verschiedenen Transportsystemen realisiert werden, obligatorisch für jede ORB–Implementierung ist jedoch eine auf TCP basierende Variante, das *Internet Inter–ORB Protocol* (IIOP).

Der Objektaufruf kann in CORBA–Umgebungen auf zweierlei Weise erfolgen. Zum einen kann aus der IDL–Definition eines dienstanbietenden Objekts mit Hilfe eines IDL–Compilers automatisch Code für ein Vertreterobjekt generiert werden, dessen Schnittstelle mit der des dienstanbietenden Objekts übereinstimmt. Der Klient des dienstanbietenden Objekts ruft dessen Operationen auf diesem Vertreter, dem sogenannten *Stub*, auf, der für die Weiterleitung des Objektaufrufs an den ORB zuständig ist. Auf der Anbieterseite sorgt ein entsprechendes Halterobjekt (*Skeleton*), dessen Code ebenfalls vom IDL–Compiler erzeugt wurde, für den Operationsaufruf beim Anbieter. Argumente und Resultate von Operationen werden transparent übertragen.

CORBA bietet ferner die Möglichkeit, Operationen beliebiger Signatur über das *Dynamic Invocation Interface* (DII) zur Laufzeit aufzurufen, ohne dass der Typ des Dienstanbieters dem Klienten schon zur Übersetzungszeit bekannt sein muss. Eine weitere wichtige Komponente der CORBA–Architektur ist der *Object Adapter*, der für die Aktivierung und Identifizierung der Anbieter–Objekte zuständig ist. CORBA verlangt keinen spezifischen Objektadapter sondern erlaubt vielmehr die Definition massgeschneiderter Adapter für verschiedene Anwendungsgebiete. Allerdings wird ein *Basic Object Adapter* (BOA) spezifiziert, den ein ORB mindestens zur Verfügung stellen muss. Um zur Laufzeit Typinformation bereitzustellen, definiert CORBA ferner eine Typmanagement–Komponente, das *Interface Repository*.

3 JacORB

JacORB [Brose 97] ist eine an der Freien Universität Berlin entwickelte Teilimplementierung des CORBA–Standards und ist im WWW unter http://www.inf.fu-berlin.de/~brose/jacorb/ verfügbar. Wir beschreiben im folgenden zunächst die Architektur und die Entwicklung von JacORB. Anschliessend gehen wir auf die Implikationen der Wahl von Java als Zielsprache für einen ORB ein.

3.1 JacORB–Architektur

JacORB besteht aus einem IDL–Compiler, einem Vertretergenerator, einem Laufzeitsystem und einem als CORBA ObjectService [OMG 94] implementierten Namensdienst. Intern verwendet JacORB IIOP und ist daher mit anderen CORBA–Implementierungen interoperabel, wie wir an einem Beispiel mit IONAs Orbix bestätigen konnten. Der Einfachheit halber wurden DII/DSI, Repository–Funktionen sowie der BOA bisher nicht implementiert, was jedoch den Nutzen von JacORB nicht wesentlich beschränkt. Die ORB–Funktionalität wird vollständig durch die generierten Vertreter– und Halterklassen (Stubs und Skeletons) realisiert und bereitgestellt. Es ist keine spezielle Konfiguration von Netzwerkknoten nötig, auf denen JacORB–Anwendungen oder –Server laufen sollen, und auch Daemon–Prozesse werden nicht benötigt, so dass die Installation von JacORB einfach zu bewerkstelligen ist.

Die einzige Voraussetzung ist der Zugriff auf einen HTTP–Server, der zum Auffinden des JacORB–Namensdienstes benötigt wird, denn die vom Generator erzeugten Stubklassen verwenden generell den Namensdienst, um den Dienstanbieter zu lokalisieren, den sie beim Klienten vertreten. Ein explizites Adressieren von Objekten geschieht nur über Namen, nicht über Netzwerkadressen.

3.2 Entwicklung und Implementierung

JacORB entstand aus einer Java–Klassenbibliothek, die zunächst lediglich Java–Programme um die Möglichkeit transparenter Fernaufrufe erweitern sollte. Dieser Ansatz erfordert eine Entkopplung der Definition von Objekttypen (Interfaces) von ihrer Implementierung in Klassen, denn ein generiertes Vertreterobjekt, das typkompatibel zu dem von ihm vertretenen Objekt sein muss, hat eine andere Implementierung als dieses. Das in Java vorhandene Interface–Konzept ist gut geeignet, um aus der Schnittstellenbeschreibung Vertreter– und Halterklassen für den so beschriebenen Objekttyp zu erzeugen. Mit JacORB lassen sich daher solche Programme transparent verteilen, in denen alle Parameter– und Rückgabetypen von Operationen Interfaces sind.

Bei der Entwicklung des Vertretergenerators konnte auf einen eigenen Parser für Java–Quelltexte verzichtet werden, da das Format bereits mit dem Java–Compiler übersetzter Klassen, das `.class`–File, kanonisiert und gut dokumentiert ist. Unser Generator konnte auf diese Weise einfacher gehalten werden, als dies andernfalls möglich gewesen wäre. Der Generator parsiert den Byte–Code einer Java–Klasse oder eines Interfaces und generiert daraus entsprechenden Quelltext für Vertreter– und Halterklassen. Die Objekte dieser Klassen sind in der Lage, Operationsargumente und Resultate zu serialisieren und zu übertragen, so dass ein transparenter Aufruf von Methoden entfernter Objekte möglich ist.

Die Verteilung von Anwendungen führt im allgemeinen Fall zur Einführung von Nebenläufigkeit in ein System, selbst wenn die Implementierung von Dienstanbietern und Klienten strikt sequentiell ist. Java erlaubt darüberhinaus den einfachen Einsatz von Nebenläufigkeit durch Threading, so dass es zwei mögliche Quellen für Nebenläufigkeit gibt: Zum einen durch den gleichzeitigen Zugriff mehrerer Klienten eines Anbieters in einer verteilten Umgebung, zum anderen schon durch den nichtsequentiellen Zugriff eines Klienten.

Bei der Entwicklung des JacORB–Kommunikationssytems wurde darauf geachtet, keine der beiden Quellen von Nebenläufigkeit zu unterbinden, um sowohl konzeptionell nebenläufige Klienten zu erlauben, wie auch Effizienzverluste durch unnötig serialisierte Zugriffe auf Dienstanbieter zu vermeiden.

Vergleicht man das von uns um Verteilung erweiterte Java–Objektmodell mit dem durch die CORBA–IDL definierten, so zeigen sich viele Ähnlichkeiten. Um die Integration von in Java geschriebenen Applikationen mit CORBA zu ermöglichen, wurde JacORB um die Konzepte der CORBA–IDL erweitert, wobei sich herausstellte, dass dies mit erstaunlich wenigen Einschränkungen der Verteilungstransparenz möglich war.

Um die Sprachabbildung zu realisieren, wurde ein IDL–Compiler geschrieben, der eine IDL–nach–Java

Übersetzung vornimmt. Zur Erstellung des Compilers wurde ein LALR Parser–Generator [Hudson 97] verwendet, der sowohl Java–Code erzeugt als auch selbst in Java geschrieben ist. Auf eine Integration des IDL–Compilers mit dem Vertretergenerator wurde verzichtet, um weiterhin verteilte Java–Programmierung auch ohne IDL–Spezifikationen zu erlauben.

Abb. 1 stellt den Prozess der Generierung von Vertretern dar. Wird ein verteiltes Java–Programm entwickelt, so kann mit in Java spezifizierten Schnittstellen begonnen werden. Ist auch die Integration in CORBA–Umgebungen angestrebt, so muss zuvor eine IDL–Spezifikation der Schnittstellen erstellt werden, aus der der IDL–Compiler dann die benötigten Java–Klassen generiert.

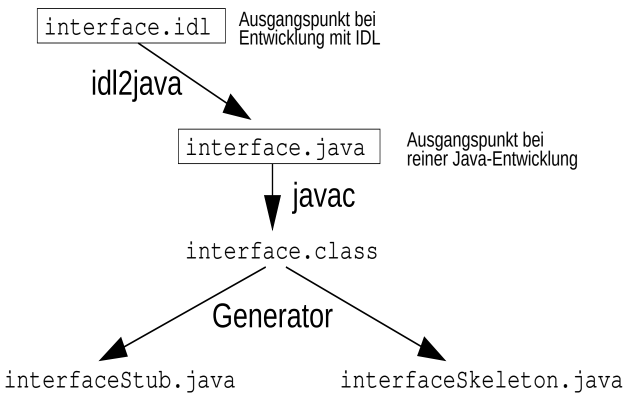

Abb. 1: Vertretergenerierung mit JacORB

3.3 Java als Zielsprache für einen ORB

Für den Fall, dass die Anwendung mit CORBA in heterogene Umgebungen integriert werden soll, setzt dies eine Abbildung von IDL nach Java voraus, die in Abb. 1 durch die Komponente *idl2java* vorgenommen wird. Es zeigt sich, dass CORBA–IDL in vielen Fällen als eine Erweiterung des Java–Objektmodells verstanden werden kann. An einigen Stellen führt die Einführung von IDL–Konzepten jedoch auch zu Transparenzeinschränkungen. Wir umreissen im folgenden zunächst die Sprachabbildung und gehen anschliessend auf die Punkte ein, die für die angestrebte Transparenz besondere Bedeutung haben. Für eine ausführlichere Darstellung der IDL–Abbildung siehe [Brose 97].

Die Java Sprachabbildung

Da, wie eingangs bereits erwähnt, noch keine von der OMG standardisierte Abbildung von IDL nach Java existiert, musste eine eigene Abbildung definiert werden. Solche Abbildungen existieren bereits bei Sun und anderen [OMG 96] und [IONA 96] und haben die von uns verwendete beeinflusst.

Allgemein sind die meisten Punkte einer solchen Sprachabbildung naheliegend, da sowohl Syntax wie Semantik von Java–Interfaces den Spezifikationen in IDL ähneln. So werden die Basisdatentypen von IDL auf die Basistypen von Java abgebildet, ebenso IDL Interfaces auf Java Interfaces. Bei den Basistypen ist allerdings zu beachten, dass Java keine vorzeichenlosen Ganzzahltypen kennt, so dass es zu Bereichsüberschreitungen kommen kann, wenn solche Werte vorzeichenloser IDL–Typen verwendet werden.

Etwas schwieriger ist die Nachbildung solcher IDL Konzepte, für die es keine Java–Entsprechungen gibt, z. B. längenbegrenzte Arrays und `sequences`, `typedef`, `oneway`, `in`, `out` und `inout`–Parameter, `structs` und bestimmte IDL–Sichtbarkeitsbereiche.

Da IDL–Arrays stets mit festen Grenzen deklariert werden, Java–Arrays dies jedoch nicht erlauben, müssen IDL Array–Deklarationen auf spezielle Klassendefinitionen in Java abgebildet werden, die ein fixes Längenattribut haben. IDL–`sequences` ohne Längenbegrenzung können einfach auf Java–Arrays abgebildet werden.

Als Beispiel für die Array–Abbildung dient das folgende IDL Interface `server`:

```
module idltest{
  interface server
  {
    typedef short nums[43];
    oneway void notify(in nums n);
  };
};
```

Der IDL–nach–Java Übersetzer erzeugt daraus die folgende Klassendefinition in einer getrennten Datei:

```
// Automatically generated by
// idl2java from array type numbers
package idltest;
public class servershortArray43
implements jacorb.Orb.Serializable
{
  public short values[] = null;
  public static final int size = 43;

  public servershortArray43()
  {
      values = new short[size];
  }
```

```
  public servershortArray43(
      short _array[]) throws
      ArraySizeMismatchException
  {
      if( _array.length != size )
            throw new
ArraySizeMismatchException();
      values = _array;
  }
    ...
}
```

Das erzeugte Java–Interface sieht dann wie folgt aus:

```
// Automatically generated by
// idl2java from interface server
package idltest;
public interface server
      extends CORBA.CORBject
{
   void notify(servershortArray43 n);
   boolean notify$oneway = true;
}
```

Das IDL–Schlüsselwort `oneway`, das für eine Operation eine asynchrone Aufrufsemantik vereinbart, wird vom IDL–Compiler in Konstanten der erzeugten Java–Interfaces umgesetzt, an der das Laufzeitsystem erkennen kann, welche Operationen asynchron aufzurufen sind. Im Beispiel ist dies das Attribut `notify$oneway`.[2]

Eine weitere bereits angedeutete Schwierigkeit stellen die unterschiedlichen Regeln für die Definition von Sichtbarkeitsbereichen in den zwei Sprachen dar. In IDL werden sowohl durch Module wie auch durch Interfaces und Strukturen neue Namensräume definiert, in Java hingegen werden nur in Packages neue Sichtbarkeitsbereiche eröffnet, d.h. die Sichtbarkeit von Klassennamen kann innerhalb eines Packages nicht weiter eingeschränkt werden. Um bei der Umsetzung qualifizierter Namen von IDL nach Java mögliche Namenskonflikte zu vermeiden, werden in unserer Abbildung allen Namen die Namen sie umgebender IDL Sichtbarkeitsbereiche vorangestellt.

Ein Beispiel dafür ist der Name des Parametertyps der Operation `notify` in dem oben aufgeführten Interface `server`, der nicht nur `shortArray43` lautet, sondern zusätzlich noch als Präfix den Namen des Interfaces `server` trägt, da der betreffende Typ in der IDL Spezifikation lokal zum umgebenden Interface `server` definiert wurde.

2. In Java sind Attribute von Interfaces implizit `public static final`, d. h. "konstante Klassenvariablen".

Ein wesentliche Lücke zwischen IDL und Java, die es in der Sprachabbildung zu schliessen gilt, besteht in der Art der Parameterübergabe für Operationen. CORBA spezifiziert *pass–by–value* Semantik für alle Werte von Basis– und strukturierten Typen, jedoch *pass-by-reference* Semantik für Objektwerte. Da in unserer Sprachabbildung jedoch auch strukturierte Typen auf Java–Klassen abgebildet werden, deren Werte ebenfalls Objekte in Java sind, muss die Sprachabbildung es dem Laufzeitsystem ermöglichen, zwischen echten CORBA–Objekten und solchen Objekten zu unterscheiden, die Werte strukturierter IDL–Typen repräsentieren. Letztere müssen *by value* übergeben werden, erstere *by reference*.

Diese Überlegungen führen dazu, dass der IDL–Compiler erzeugte Java–Klassen, deren Objekte als Wert übergeben werden müssen, als serialisierbar kennzeichnet und automatisch Code zur Serialisierung einfügt. Das Laufzeitsystem kann dann feststellen, ob ein Objekt ein Exemplar einer serialisierbaren Klasse ist, und gegebenenfalls den entsprechenden Code ausführen.

CORBA erlaubt ferner *pass-by-result* Semantik, d.h. die Kennzeichnung von Parametern als `out` oder `inout`, was bedeutet, dass nach Operationsausführung in diesen Parametern Werte erwartet werden. Da es eine solche Parameterübergabe in Java nicht gibt, muss sie durch das Einsetzen von Behälter–Klassen, sogenannten *Holdern*, nachgebildet werden. Das Laufzeitsystem kann dann erkennen, ob ein Parameter ein *Holder* ist und daher nach Operationsausführung rückübertragen werden muss.

Ein weiterer wichtiger Punkt ist die Frage, wie Objektreferenzen verteilungstransparent behandelt werden können. Objekte können sowohl als Argumente an Operationen übergeben wie auch als Resultate von Operationen empfangen werden. In solchen Fällen werden vom Laufzeitsystem transparent Stubs und Skeletons als Exemplare zuvor generierter Halter– und Vertreterklassen erzeugt, so dass ein Programmierer nicht mit CORBA–Objektreferenzen in Berührung kommt, sondern stets Java–Objekte sieht.

Einschränkungen der Verteilungstransparenz

Das CORBA–Objektmodell ist keine reine Erweiterung gegenüber Java, es gibt auch Einschränkungen. Die aus der Sicht des Entwicklers schwerwiegendste solche Einschränkung ist die Tatsache, dass IDL kein Überladen von Operationsnamen erlaubt, was in Java gestattet ist. Durch systematisches Umbenennen von Operationsnamen lassen sich jedoch Programme, die überladene Operationen verwenden, in Programme ohne überladene Operationen verwandeln.

Eine weitere Einschränkung ergibt sich aus der Tatsache, dass `strings` und Arrays in Java Objekte, in CORBA–IDL aber Werte sind. Dadurch kann z. B. der Unterschied zwischen einem leeren String und einem `null`–String, der in "reinem" Java besteht, in einem verteilten Programm nicht mehr ausgedrückt werden, denn Null–Referenzen dürfen in diesen Fällen nicht verwendet werden. Diese Fälle, ebenso wie die bereits erwähnten Wertebereichsüberschreitungen, schränken aus der Sicht eines Java–Entwicklers die Verteilungstransparenz ein, da sie für ausschliesslich lokale Operationsaufrufe andere Regeln als für entfernte Aufrufe induzieren.

4 Zusammenfassung

Das Ziel der Entwicklung von JacORB war es, eine Plattform für die verteilungstransparente Entwicklung verteilter Anwendungen in Java und deren Integration in CORBA bereitzustellen. Durch die Nähe des CORBA–Objektmodells zum Objektmodell von Java bietet sich die Entwicklung einer CORBA–Plattform für Java an. Wir haben die Entwicklung eines Object Request Brokers in Java beschrieben. Es zeigte sich, dass die Abbildung des CORBA–Objektmodells in das Java–Objektmodell verhältnismässig gut machbar ist. Entsprechend erfordert die Entwicklung von CORBA–Anwendungen mit JacORB kein grosses konzeptionelles Umdenken gegenüber der Entwicklung verteilter Anwendungen in einer homogenen Java–Umgebung, denn fast alle Java–Interfaces lassen sich in CORBA–IDL ausdrücken — mit den angesprochenen Einschränkungen.

Neben dem Erreichen grösstmöglicher Verteilungstransparenz war ein weiteres Prinzip bei der Entwicklung von JacORB, auf mögliche Effizienzverbesserungen durch *native code*, also in C geschriebene Bibliotheken, zu verzichten, um eine einfache Portierbarkeit und einen möglichst geradlinigen Entwurf zu ermöglichen. Die im Vergleich zu C++ strikter objektorientierten Konzepte von Java unterstützten unseren Ansatz, einen klare Struktur gegenüber hoher Effizienz zu priorisieren.

Die gemeinhin mit der Programmierung in Java assoziierten Vorteile wie automatische Speicherverwaltung, Ausnahmebehandlungskonzepte, Feldgrenzen–Kontrolle, Verbot expliziter Zeiger-manipulation und Integration von Multi–Threading in die Sprache haben sich auch in unserem Fall als Erleichterung der Implementierung herausgestellt. Die Erfahrungen mit der Implementierung von JacORB haben bestätigt, dass sich Java nicht nur für die Programmierung kleinerer Applets, sondern auch als Implementierungssprache eines mittelgrossen Softwaresystems wie JacORB eignet, dessen Kern gegenwärtig ca. 130 Klassen und 10.500 Zeilen Java–Code umfasst.

Die Erfahrungen aus dem Einsatz von JacORB in einer Lehrveranstaltung des Hauptstudiums haben gezeigt, dass es auch Studenten mit wenig oder keinen Java–Kenntnissen gelang, sich in ca. vier Monaten sowohl die Sprache Java als auch die benötigten Verteilungskonzepte anzueignen und dann verteilte Anwendungen mit JacORB zu realisieren.

Literatur

[Brose 97] Brose, Gerald, *JacORB — A Java Object Request Broker*, Technical Report B 97-2, Institut für Informatik, Freie Universität Berlin, April 1997.

[Gosling et al. 97] Gosling, J. and B. Joy, G. Steele, *The Java Language Specification*, Addison–Wesley 1996.

[Hudson 97] Hudson, Scott, *Constructor of Useful Parsers — CUP*, Version 0.9, `http://www.cc.gatech.edu/gvu/people/Faculty/hudson/java_cup/home.html`.

[IONA 96] IONA Technologies Ltd., *IDL Java Language Mapping*, OMG RFP Submission, OMG TC document orbos/96-12-11, December 1996.

[OMG 96] DEC, Expersoft, HP, IBM, Netscape, Oracle, Novell, Sun, Visigenic, *IDL/Java Language Mapping*, OMG RFP Submission, OMG TC document orbos/96-12-08, December 1996.

[OMG 94] OMG, *Common Object Services Specification, Volume I*, Revision 1.0, March 1994, OMG Document 94–1–1.

[OMG 95] OMG, *The Common Object Request Broker: Architecture and Specification*, 2.0, July 1995.

[Siegel 96] Siegel, Jon (ed.), *CORBA Fundamentals and Programming*, Wiley 1996.