Java und .NET verbinden

Brücken bauen

Java und .NET sind sich in ihrer grundsätzlichen Architektur ähnlich. Im Detail gibt es aber viele Unterschiede. Alle Software-Entwickler, für die es kein Entweder-oder gibt, müssen sich mit Kopplungstechniken auseinander setzen. dotnetpro stellt vier entsprechende Produkte vor. Auch abseits von XML Web Services gibt es Möglichkeiten zur Interoperabilität.

Während die Java-Entwicklergemeinde beharrlich Javas Plattformunabhängigkeit betont, schwört die .NET-Entwicklergemeinde auf Sprach-Interoperabilität und bessere Performance. In vielen Projekten muss eine eindeutige Entscheidung entweder für schwarz oder für weiß getroffen werden. Doch was ist, wenn einmal *grau* erforderlich ist und man um eine Kombination aus Java und .NET nicht herumkommt?

Freuen kann sich derjenige, der noch nie in diesem Dilemma gesteckt hat, denn er hat sich einige Kopfschmerzen erspart. Wer allerdings eine Integration von Java und .NET nicht vermeiden kann, sei es, weil er aus .NET auf einen J2EE-basierten Applikations-Server zugreifen muss, weil er seine bereits vorhandene Software-Komponente auch der jeweils anderen Entwicklergemeinde zugänglich machen möchte oder weil ein Kunde plötzlich ein .NET-Frontend für seine Java-basierte Server-Applikation fordert, der sieht sich mittlerweile einer Fülle von Lösungsmöglichkeiten für dieses Problem gegenüber.

Die zur Verfügung stehenden Ansätze reichen von

- einer sehr engen Kopplung mittels direktem In-Process-Zugriff auf die kompilierten Klassen der jeweils anderen Seite über
- die lockerere Kopplung mittels Remoting-Mechanismen wie Java RMI, .NET Remoting oder IIOP,
- selbst entwickelten Kommunikationslösungen mittels TCP/IP-basierter Socket-Verbindungen bis hin zur
- sehr losen Kopplung mittels XML Web Services.

Je nachdem, welchen Umfang die zu erstellende Applikation besitzt, ob nur lokale oder auch entfernte Kommunikation vonnöten ist, ob Open-Source- oder kommerzielle Software zum Einsatz kommen soll oder welche Zusatzdienste wie etwa Lastverteilung und Transaktionsunterstützung noch gebraucht werden, ist für einen speziellen Fall eher der eine oder der andere Ansatz geeignet.

Dieser Artikel beschränkt sich auf die zwei zuerst genannten Lösungen, die auch unter dem Stichwort Component Bridging bekannt sind. Component Bridges versprechen mehr Performanz und Funktionalität als die noch sehr im Entwicklungsfluss verharrende Web-Service-Welt und mehr Entwicklungskomfort als die TCP/IP-basierte Socket-Kommunikation.

Auf einen Blick

Autor



Dr. Holger Schwichtenberg unterrichtet Wirtschaftsinformatik an der Fachhochschule für Oekonomie und Management (FOM) in Essen.

Sven Panko ist selbstständiger Entwickler mit langjähriger Erfahrung im Bereich Java und C++. Er arbeitet momentan an seiner Diplomarbeit über künstliche Intelligenz und evolutionäre Algorithmen.

Björn Röber ist selbstständiger Entwickler mit langjähriger Erfahrung in den Bereichen Microsoft .NET und Borland Delphi. Zuvor war er Leiter der Software-Entwicklung bei einem mittelständischen Systemhaus im Bereich ÖPNV.

dotnetpro.code
A0410JavaBridge

Sprachen Java, C#

Technik Component Bridges

Voraussetzungen .NET Framework 1.0/1.1, Java SDK ab 1.3, Ja.NET, IIOP.NET, JuggerNET, Janeva, J#-Runtime (für Janeva)

Ja.NET

Ja.NET der Firma Intrinsyc [1] liegt mittlerweile in der Version 1.5 vor und verspricht eine einfache und schnelle Möglichkeit, Java und .NET in beide Richtungen miteinander zu verbinden. Ja.NET ermöglicht dies, indem es zu Java .NET-Remoting-Fähigkeiten hinzufügt. Dazu liefert der Hersteller eine Java-Bibliothek, die Aufrufe aus Java heraus entgegennimmt und den .NET-Remoting-Aufruf durchführt. Der umgekehrte Weg, also von .NET nach Java, ist darüber hinaus ebenfalls möglich. Zur Laufzeit ist die oben genannte Java-Bibliothek die einzige benötigte Komponente, was das Deployment sehr einfach macht.

Die Vorgehensweise von Ja.NET beim Erstellen einer heterogenen Anwendung ist überschaubar und bewusst einfach gehalten:

- Erstellen der Klassen, auf die zugegriffen werden soll (entweder in Java oder .NET),
- Erzeugen von Stellvertreterklassen für die jeweils andere Umgebung,
- Kompilieren der Anwendungen auf beiden Seiten und Ausführung.

Im Falle einer Kommunikation von Java nach .NET haben die Stellvertreterklassen die Aufgabe, Aufrufe von Java an .NET entgegenzunehmen, an die Laufzeitbibliothek weiterzugeben und eine mögliche Antwort zurückzusenden. Der Zugriff auf Java-Objekte von .NET aus läuft ebenfalls über Stellvertreterklassen, diesmal jedoch auf der .NET-Seite: Die

Stellvertreterklassen leiten Aufrufe an die Laufzeitbibliothek weiter, die ihrerseits mit der Anwendung kommuniziert.

Der Entwickler steht beim Erstellen seiner Anwendung vor der Wahl, welches Protokoll auf der Transportschicht beziehungsweise für den Datenaustausch verwendet werden soll. Ja.NET unterstützt TCP/IP mit einem binären Datenformat und HTTP mit SOAP als Datenformat. Sollte HTTP als Protokoll gewählt werden, dann muss auf der .NET-Seite der Redmonder Web-Application-Server „Internet Information Services" (IIS) verwendet werden. Auf der Java-Seite kann jeder J2EE-fähige Webserver genutzt werden. Soll eine .NET-Verbindung mit einer Enterprise Java Bean (EJB) hergestellt werden, ist zusätzlich ein EJB-fähiger Applikationsserver nötig.

Installation

Die Installation von Ja.NET ist einfach. Das Paket kommt in Form einer Anwendung, die ein Installationsprogramm enthält. Nach der Installation finden sich auf der Festplatte die folgenden sechs Ja.NET-Komponenten:

- die Laufzeitbibliothek, auch Ja.NET-Runtime genannt (janet.jar),
- ein Tool zur Erstellung einer applikationsspezifischen XML-Datei, die das Verhalten der Ja.NET-Runtime steuert (Janetor),
- ein Tool, um aus Java-Klassen .NET-Stellvertreterklassen zu erstellen (GenNET),
- ein Tool, um aus .NET-Klassen Java-Stellvertreterklassen zu erstellen (GenJava),
- ein Windows-Dienst, der beim Erstellen der Stellvertreterklassen benötigt wird (GenService),
- Tutorials und Beispielanwendungen sowie ein Online-Handbuch.

Der Windows-Dienst wird standardmäßig nicht in der Diensteverwaltung registriert. In der Programmgruppe findet sich dazu jedoch ein Eintrag, mit dem dies nachgeholt werden kann. Mit dem Dienst kann man auf .NET-Assemblies zugreifen. Er wird nur von den Tools GenNET zum Schreiben von Stellvertreterklassen beziehungsweise GenJava zum Lesen von Typinformationen per Reflection benötigt. Beim Deployment wird er später nicht mehr gebraucht.

Der Einsatz eines Dienstes für diese Aufgabe ist dabei nicht unproblematisch,

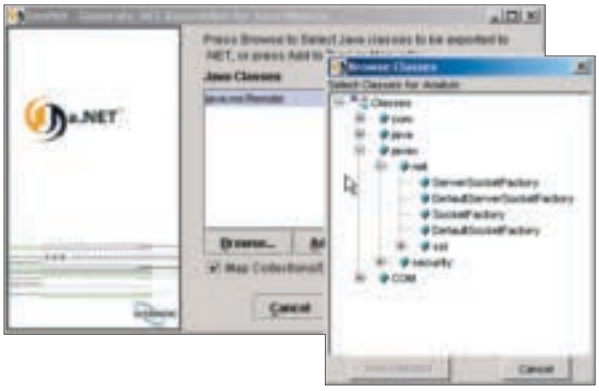

Abbildung 1 Ja.NETs Werkzeug GenNET generiert aus Java-Klassen .NET-Stellvertreterklassen.

da dieser nur von Administratoren installiert werden kann – nicht jeder Entwickler besitzt aber gleichzeitig diesen Status. Auf Nachfrage teilte Intrinsyc mit, dass dieser Nachteil bekannt sei, jedoch der Vorteil eines Dienstes überwiege: Der Dienst müsse auf nur einem System im Netzwerk installiert sein und alle Entwickler könnten ihn nutzen. Dies vereinfache die Installation und senke somit auch Kosten.

Die Tutorials sind bei der Einarbeitung in Ja.NET nicht sehr hilfreich, da sie keine allgemeine Anleitung für die verschiedenen Integrationsmöglichkeiten geben, sondern nur das Erstellen der mitgelieferten Beispielanwendungen erläutern. In den Tutorials wird dann darauf verwiesen, sich deren Quelltexte näher anzusehen. Gut gelungen ist hingegen das Online-Handbuch, das sämtliche Möglichkeiten, die Ja.NET bietet, erklärt.

Bedienung und Features

Ja.NET integriert sich nicht in Entwicklungsumgebungen, bietet aber mit Janetor, GenNET und GenJava drei Komponenten mit einer Oberfläche an, die die Bedienung erheblich vereinfachen. Besonders das Konfigurieren der Laufzeitumgebung mit Janetor wird dadurch zum Kinderspiel. Auch die Bedienung von GenNET – siehe Abbildung 1 – und GenJava ist im Großen und Ganzen gefällig.

Ja.NET stellt über seine Laufzeitumgebung sicher, dass alle für den entfernten Zugriff konfigurierten Klassen verfügbar sind. Die Programmierung eines eigenen Servers, der Objekte instanzieren und bereitstellen muss, entfällt somit. Dabei unterstützt die Laufzeitumgebung verschiedene Arten, wie die entfernten Objekte bereitgestellt werden sollen: als Client-Activated-Object, wobei die Kontrolle über die Lebensdauer beim Client liegt, oder als Server-Activated-Object, bei dem der Server diese Kontrolle innehat. Im letzteren Fall unterscheidet Ja.NET weiter zwischen Objekten, die nur einmalig erzeugt werden und dann allen Clients gemeinsam zur Verfügung stehen (Singletons), und pro Methodenaufruf generierten Objekten.

Die von .NET-Remoting verwendete Methode zur Kontrolle des Lebenszyklus von verteilten Objekten (Leasing) haben die Ja.NET-Entwickler auch für Java-seitig bereitgestellte Objekte hinzugefügt. Somit braucht sich der Entwickler darüber keine weiteren Gedanken mehr zu machen. Er konfiguriert einfach in Janetor einmalig die Art des Zugriffs und die Laufzeitumgebung erledigt den Rest.

Darüber hinaus lassen sich mit Janetor auch diverse Einstellungen für die Transportprotokolle und die Kompatibilität zu .NET 1.0 und 1.1 vornehmen. Interessant sind vor allem die Möglichkeiten der Sicherheitseinstellungen. Ja.NET unterstützt dabei Basic und Digest HTTP-Security, allerdings nur im Zusammenspiel mit dem IIS. Die Unterstützung für Verschlüsselung befindet sich momentan laut Website im Betastadium.

Das Deployment von Ja.NET gestaltet sich ebenfalls sehr einfach: Laufzeitumgebung konfigurieren, alle benötigten Klassen auf das Server-System kopieren, evtl. einige Einstellungen an Applikations-Servern vornehmen und schließlich noch die Laufzeitumgebung starten – fertig.

Insgesamt geht die Arbeit mit Ja.NET sehr flott, größere Schwierigkeiten gab es keine. Ein einfaches Beispielprogramm, unabhängig von den mitgelieferten Beispielen, lässt sich in weniger als 45 Minuten erstellen, wobei die meiste Zeit für die erstmalige Einarbeitung benötigt wird. Wer öfter mit Ja.NET arbeitet, wird vor allem das Tool Janetor schätzen, das einem die meiste Arbeit beim Erstellen der Konfiguration für die Laufzeitbibliothek abnimmt.

JuggerNET

Einen grundlegend anderen Ansatz verfolgt JuggerNET der Firma Codemesh [8]. Das Produkt kommt als einzige Lösung im Test ohne zusätzliche Kommunikations-Middleware wie CORBA, RMI oder .NET Remoting aus und stellt eine lokale In-Process-Kopplung von .NET und Java über PInvoke und das Java Native Interface (JNI) her. Die momentan verfügbare Version 1.1 ermöglicht leider bisher nur den Zugriff von .NET auf Java-Klassen. Der umgekehrte Weg soll mit Version 3.0 möglich werden, die für Mitte 2005 geplant ist.

Allerdings bietet JuggerNET ein Workaround für die noch nicht vorhandene Zugriffsmöglichkeit von Java auf .NET-Klassen. Mithilfe von asynchronen Callbacks auf der Java-Seite ist es möglich, in .NET geschriebene Ereignisbehandlungsroutinen auf Java-Ereignisse zu registrieren und dadurch indirekt auch Aufrufe von Java nach .NET durchzuführen. Auch der Java Message Service (JMS) kann über diesen Weg genutzt werden.

Zugriff auf Java-Objekte

Um von einer beliebigen .NET-Sprache aus auf Java-Objekte zugreifen zu können, wird zunächst eine Java-VM in den CLR-Prozess geladen und per Konfigurationsdatei oder Quellcode konfiguriert. Dabei werden unter anderem der *Pfad* zur zu benutzenden JVM sowie der *ClassPath* zu den zu benutzenden Java-Klassen festgelegt. Die Kommunikation mit den Java-Klassen erfolgt über C#-Proxy-Klassen, die aus den ursprünglichen Java-Klassen erzeugt wurden. Diese Proxy-Klassen leiten die Aufrufe an eine .NET-Assembly (*netrt11.dll*) weiter, die den .NET-basierten Teil der Laufzeitbibliothek darstellt. Um die Lücke zwischen den beiden Plattformen .NET und Java zu überbrücken, leitet diese Assembly die Aufrufe im Anschluss per PInvoke an eine in C geschriebene DLL (*netrt11supp.dll*) weiter, die den nativen Teil der Laufzeitbibliothek darstellt. Diese DLL kommuniziert über das JNI mit den eigentlichen Java-Klassen in der JVM. Beim Zugriff auf die durch JuggerNET generierten .NET-Proxies besteht generell kein Unterschied zum Zugriff auf .NET-eigene Typen. Allerdings gibt es einige Besonderheiten:

- *Primitive Java-Datentypen* werden 1:1 auf die entsprechenden .NET-Datentypen abgebildet.
- *Arrays von primitiven Java-Datentypen* können sowohl in Form von normalen .NET-Arrays der entsprechenden Datentypen als auch in Form von .NET-Proxy-Arrays verwendet werden, die durch JuggerNET bereitgestellt werden. Anstelle von *byte[]* kann beispielsweise die Klasse *Codemesh.JuggerNET.byteArray* verwendet werden. Der Vorteil der Benutzung der Proxy-Array-Klassen besteht darin, dass die einzelnen Elemente nicht wie bei der Benutzung der .NET-Arrays vor der Übergabe an die JVM noch einmal komplett kopiert werden müssen.
- *Java-Referenz-Typen* – also alle Klassen und Interfaces – werden mit Ausnahme der Klassen *java.lang.Object* und *java.lang.String* generell auf .NET-Proxy-Klassen abgebildet. Diese beiden Klassen werden gesondert behandelt, da auf .NET-Seite mit den Klassen *System.Object* und *System.String* bereits nahezu äquivalente Klassen existieren. Überall dort, wo ein *java.lang.String* erwartet wird, kann ein *System.String* verwendet werden. Bei *java.lang.Object* können dagegen nur *System.String* und alle generierten Proxy-Klassen verwendet werden. Die Übergabe generischer .NET-Objekte soll mit Version 2.0 von JuggerNET ermöglicht werden.
- *Java-Interface-Typen* weisen auch eine kleine Besonderheit auf: Im Gegensatz zu .NET ist es in Java erlaubt, statische Members wie Konstanten in Interfaces zu definieren. Aus diesem Grund wird ein Java-Interface, das statische Member enthält, auf ein .NET-Proxy-Interface und zusätzlich auf eine .NET-Proxy-Klasse abgebildet. Das Proxy-Interface beinhaltet dann die entsprechenden Methoden-Deklarationen. Die Proxy-Klasse implementiert das Proxy-Interface und enthält zusätzlich alle statischen Member der Ursprungsklasse. Da die Proxy-Klasse nicht den gleichen Namen besitzen kann wie das Proxy-Interface, wird der Klassenname noch durch ein *Impl* ergänzt, zum Beispiel *IServer* und *IServerImpl*.

Da es in Java im Gegensatz zu .NET keine komplexen Werttypen gibt, besteht nicht die Notwendigkeit, komplexe Objekte per Call-by-Value zu übertragen.

Kompatibilität

JuggerNET ist kompatibel mit der Sun JVM ab Version 1.1.7, der IBM JVM ab Version 1.1.8 und allen anderen JVMs, die das vollständige Java Native Interface implementieren.

Installation

Auch JuggerNET verfügt über ein Installationsprogramm, das dem Benutzer die Komponenten

- JuggerNET Code-Generator,
- Example Files,
- Help Files und
- .NET Integration Libraries

zur Auswahl stellt. Im Verlauf der Installation wird der Benutzer gebeten, die zu benutzende JVM auszuwählen. Neben den bereits installierten JVMs steht auch eine eigens mitgelieferte JVM zur Auswahl. Des Weiteren kann durch Angabe einer E-Mail-Adresse die automatische Fehlerübermittlung eingeschaltet werden.

Im Anschluss an die Installation kann man direkt ohne zusätzliche Vorarbeiten mit der Arbeit beginnen.

Eine Integration in Entwicklungsumgebungen bietet JuggerNET nicht, dafür aber, wie in Abbildung 2 zu sehen, einen Code-Generator mit einer eigenen Benutzeroberfläche, über den sämtliche Konfigurations- und Generierungsarbeiten vorgenommen und Visual-Studio-.NET-Projektdateien erzeugt werden können. Optional lassen sich alle Aktionen auch über die Kommandozeile durchführen und automatisieren.

Erste Schritte

Die sehr gelungene und detaillierte Dokumentation, die allerdings nur auf Englisch vorliegt, bietet einen tiefen Einblick in die Funktionsweise von JuggerNET und ist ein guter Ausgangspunkt, um mit der Arbeit zu beginnen. Der Ablauf der Anwendungsentwicklung ist sehr einfach:

- Zunächst wird der Code-Generator gestartet.
- Anschließend wird ein Java-Modell durch Importieren der zu benutzenden Java-Klassen erstellt und bei Bedarf weiter angepasst. Konfigurierbar ist beispielsweise, welche Klassen, Felder, Konstruktoren und Methoden in .NET verfügbar sein sollen und welche Namenskonventionen (Java-Stil oder .NET-Stil) für das Generieren der Proxy-Klassen verwendet werden sollen.
- Aus diesem Java-Modell werden daraufhin .NET-Proxy-Klassen generiert

und mit in die zu erstellende Anwendung eingebunden.

Deployment

Auch das Deployment von JuggerNET-basierten Anwendungen ist denkbar einfach: Soweit programmintern nur relative Pfade verwendet werden und die zu benutzende JVM direkt mit ausgeliefert wird, ist ein vollständiges XCopy-Deployment möglich. Neben dem Microsoft .NET Framework Redistributable Package wird lediglich die JuggerNET-Laufzeitumgebung in Form der beiden Dateien *netrt11.dll* und *netrt11supp.dll* benötigt.

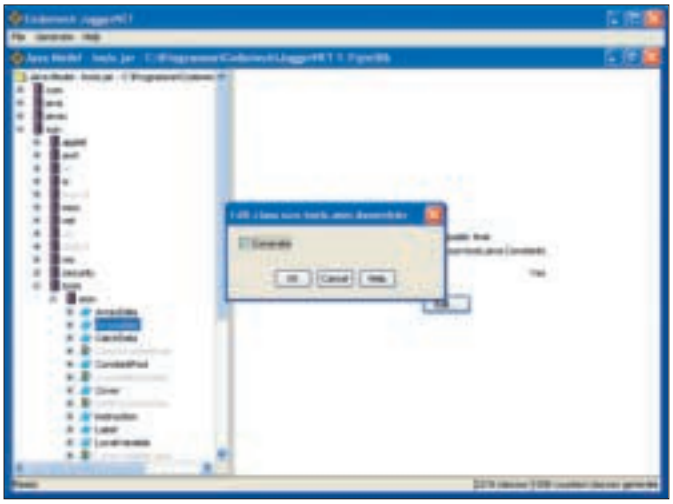

Abbildung 2 JuggerNET Code-Generator.

IIOP.NET

IIOP.NET ist eine Open-Source-Lösung, die im Rahmen einer Diplomarbeit von Dominic Ullmann an der ETH Zürich entwickelt wurde. Das Projekt wird mittlerweile offiziell von Ullmanns neuem Arbeitgeber, der Elca Informatique SA, betreut und unter der Lesser General Public Licence (LGPL) bereit gestellt sowie laufend aktualisiert [4]. Durch die LGPL ist es ausdrücklich erlaubt, IIOP.NET in kommerziellen Produkten einzusetzen, ohne das Produkt selbst unter die GPL stellen zu müssen.

Um Java und .NET in beide Richtungen miteinander zu verbinden, nutzt IIOP.NET, wie der Name bereits andeutet, das Internet Inter-Orb Protocol (IIOP). Näheres zu IIOP findet sich im Kasten „CORBA, IIOP und Co.".

Unterstützung für IIOP ist von Haus aus in Java in Form von RMI/IIOP vorhanden, einer Sammlung von Klassen, die Javas Remote Method Invocation (RMI) auf Basis von IIOP implementieren. Damit können auch solche Anwendungen auf Java-RMI-Objekte zugreifen, die selbst nicht in Java implementiert sind. Was zu einer Anbindung an .NET-Anwendungen jetzt noch fehlt, ist eine IIOP-Implementierung für das .NET Framework. Microsofts .NET-Remoting unterstützt von Haus aus lediglich eine binäre und eine auf SOAP basierende Anbindung an andere Anwendungen. Diese Lücke füllt IIOP.NET, indem es IIOP-Unterstützung zu .NET hinzufügt.

Dadurch ist es .NET übrigens nicht nur möglich, auf Java-RMI-Objekte zuzugreifen, sondern auch mit CORBA-Anwendungen in anderen Programmiersprachen zu kommunizieren, was IIOP.NET zu einer sehr mächtigen Software-Lösung macht. Einziger Wermutstropfen ist, dass dazu ein CORBA-fähiger Applikations-Server benötigt wird. Die IIOP.NET-Website [4] dokumentiert den Einsatz von IIOP.NET mit einigen weit verbreiteten Application Servern, unter anderem BEA WebLogic und IBMs WebSphere. Für die Integration dieser Server mit .NET stehen Tutorials mit Beispielcode bereit. Darüber hinaus besteht auch die Möglichkeit, von .NET aus auf Enterprise Java Beans (EJB) zuzugreifen, wobei in diesem Fall ebenfalls ein Applikations-Server benötigt wird.

Damit über IIOP auf verteilte Java- und .NET-Objekte zugegriffen werden kann, benötigen die Compiler zur Entwicklungszeit Informationen über die Methoden und Attribute der Objekte, auf die überhaupt zugegriffen werden kann. Dieses Wissen ist für CORBA-Anwendungen laut Spezifikation der OMG zwar auch dynamisch, das heißt zur Laufzeit abrufbar, jedoch unterstützt IIOP.NET dies nicht. Dafür unterstützt es den Entwickler dabei, diese Informationen zu erhalten und in Form automatisch generierter Klassen in die Projekte einzubinden.

Installation

Um IIOP.NET zu verwenden, muss es zunächst installiert werden. Die aktuell verfügbare Version trägt die Versionsnummer 1.6.1 und steht auf der IIOP.NET-Website zum Download bereit. Das Paket enthält momentan die Quelltexte ohne ein Installationsprogramm. Das gilt auch dann, wenn man bei der Download-Auswahl das „Binary Package" gewählt hat. Deshalb muss die Software nach dem Entpacken zunächst kompiliert werden.

Dabei tauchten einige kleinere Hürden auf, deren Umgehung im Kasten „IIOP.NET-Installation" beschrieben ist. Die Software besteht aus der Datei *IIOPChannel.dll*, einer .NET-Assembly, die das .NET-Remoting um IIOP erweitert, sowie den Tools *CLSIDLGenerator* zum Erzeugen von IDL-Dateien aus .NET-Klassen und *IDLToCLSCompiler* zum Erzeugen von .NET-Klassen aus IDL-Dateien. Momentan generiert das letztgenannte Tool nur C#-Klassen, die in kompilierter Form jedoch auch von allen anderen .NET-Sprachen verwendet werden können. Darüber hinaus enthält das Paket eine HTML-Dokumentation mit Tutorials und Beispielprojekten. Letztere sind sehr gut ausgearbeitet. Die Dokumentation geht vor allem auf die Umwandlung von CLS zu IDL ein.

Bedienung

Nach der Installation kann IIOP.NET sofort verwendet werden. Die Software integriert sich nicht in eine IDE, sondern wird ausschließlich über die Kommandozeile verwendet.

Um mit IIOP.NET eine Java- mit einer .NET-Anwendung zu verknüpfen, sind folgende Schritte notwendig:

- Erzeugen der Klassen, auf die zugegriffen werden soll,
- Erstellen einer Server-Anwendung, die diese Klassen für die Verwendung mittels entfernter Aufrufe registriert,
- Erstellen von IDL-Dateien aus diesen Klassen,
- Erstellen von Stellvertreter-Klassen aus den IDL-Dateien für die jeweils andere Umgebung,
- Einbinden der Stellvertreter-Klassen in ein Projekt und Kompilierung,
- Start der Server-Anwendung,
- Start der Client-Anwendung.

Beispielhaft wird nun gezeigt, wie man eine einfache Anwendung erstellt, bei der ein Java-Client auf .NET-Objekte zugreift. In diesem Beispiel wird auf der .NET-Seite die Programmiersprache C# verwendet. Auf der .NET-Seite existiert eine Klasse *Datenspeicher*, die in einer *Hashtable* Einträge vom Typ *Eintrag* verwaltet. Dazu existieren Methoden zum Speichern und Laden von Einträgen sowie zum Entfernen. Die *Eintrag*-Klasse enthält zwei private Members: *name* und *wert*. Zudem existiert für jedes der Members eine Methode zum Lesen und Schreiben.

Die .NET-Seite

Um eine Klasse über IIOP.NET nach außen hin zugänglich zu machen, steht man vor der Wahl, diese entweder von *MarshalByRefObject* abzuleiten oder die Klasse durch das Attribut [*Serializable*] als serialisierbar zu kennzeichnen. Im ersten Fall werden Objekte stets als Referenz übergeben, das heißt, mehrere Anwendungen „teilen" sich eine Objektinstanz. Im zweiten Fall wird das Objekt als Wert übergeben, die Java-Seite erhält also eine Kopie der Daten, deren Veränderung nicht auf der .NET-Seite reflektiert wird.

Auf die Klasse *Datenspeicher* soll in unserem Beispiel als Referenz zugegriffen werden. Hier ist ein Szenario denkbar, bei dem mehrere Anwendungen auf den gleichen Speicher zugreifen müssen. Die Einträge hingegen sollen als Werte übergeben werden, damit jede Anwendung eine Kopie der Daten erhalten und modifizieren kann. Zusätzlich wird noch eine Server-Anwendung benötigt, die ein Objekt der Klasse *Datenspeicher* erzeugt und veröffentlicht, sodass per IIOP.NET darauf zugegriffen werden kann. Im Beispiel erfolgt der Zugriff über den TCP-Port 3456. Listing 1 auf der Heft-CD enthält den Quelltext der drei Klassen.

Um die Assembly zu kompilieren, muss die Bibliothek *IIOPChannel.dll* von IIOP.NET entweder über die benutzte Entwicklungsumgebung als Verweis oder an der Kommandozeile des C#-Compilers über den Parameter „/r" hinzugefügt werden. Bei der Ausführung der Server-Anwendung muss diese DLL ebenfalls in einem Verzeichnis liegen, das Windows auf der Suche nach Bibliotheken einschließt. Typischerweise ist dies das *system32*-Verzeichnis oder das Anwendungsverzeichnis.

Konvertierung

Damit aus Java heraus auf die *Datenspeicher*- und *Eintrag*-Klassen zugegriffen werden kann, müssen nun Stellvertreterklassen erzeugt werden. Dabei unterstützt IIOP.NET den Entwickler mit dem *CLSToIDLGenerator*, einem Tool, das aus .NET-Klassen IDL-Dateien erzeugt. Diese werden benötigt, um anschließend mithilfe des beim Java-SDK mitgelieferten *idlj*-Tools die Java-Stellvertreterklassen zu erzeugen.

Falls von .NET aus auf Java-Objekte zugegriffen werden soll, muss an dieser Stelle ein anderes Vorgehen gewählt werden: Zunächst wird mithilfe des mit dem Java-SDK mitgelieferten RMI-Compilers (*rmic.exe*) aus der Java-Klasse die IDL-Datei erzeugt. Diese wird von dem Tool *IDLToCLSCompiler* benötigt, um eine .NET-Stellvertreterklasse zu erstellen. Danach ist das weitere Vorgehen wieder analog.

Im vorliegenden Fall müssen folgende Befehle aus dem Ausgabeverzeichnis des Projekts auf der Kommandozeile abgesetzt werden, wobei der Pfad zu *CLSIDLGenerator.exe* hierbei weggelassen wurde:

```
CLSIDLGenerator IiopExample.Datenspeicher
  Server.exe
```

Der erste Parameter gibt dabei an, welcher Typ in eine IDL-Datei konvertiert werden soll, der zweite Parameter spezifiziert die Assembly, die den Typ enthält. Verschachtelte Typen löst der *CLSIDLGenerator.exe* übrigens automatisch auf und erzeugt separate IDL-Dateien. Deswegen ist eine separate Konvertierung des Typs *Eintrag* nicht nötig. Für die IDL-Dateien hat *CLSIDLGenerator* ein Unterverzeichnis mit dem Namen des Namespaces angelegt.

Der nächste Befehl dient dem Erzeugen der Java-Stellvertreterklassen aus den IDL-Dateien, wobei auch hier der Pfad zu *idlj.exe* weggelassen wurde:

```
idlj -i <Pfad zum java-sdk\lib-Verzeichnis>
<Pfad zur IDL-Datei>
```

CORBA, IIOP und Co.

Wer sich mit der Integration von Java und .NET auseinander setzt, wird zunächst einmal stutzen, wenn in diesem Zusammenhang der Begriff „CORBA" fällt. Den meisten Entwicklern ist CORBA zwar ein Begriff, jedoch verbinden viele eine Integration mittels dieser Architektur mit der berühmten Redewendung „mit Kanonen auf Spatzen schießen". Zwei der von uns getesteten Produkte, Janeva und IIOP.NET, verwenden jedoch eben diesen Ansatz und setzen beim Benutzer zumindest rudimentäre CORBA-Kenntnisse voraus.

CORBA steht für „Common Object Request Broker Architecture" und ist eine von der Object Management Group (OMG) entwickelte und standardisierte Architektur, um verteilte Systeme zu realisieren. Im Mittelpunkt steht dabei der Object Request Broker (ORB), eine Software-Komponente, die Aufrufe von und zu verteilten Objekten vermittelt. Für die Kommunikation wurde bei der OMG das General Inter-Orb Protocol (GIOP) standardisiert, das mit dem Internet IOP (IIOP) eine auf TCP/IP basierende Implementierung erhalten hat.

Damit ein ORB Aufrufe zu Objekten vermitteln kann, benötigt er zunächst das Wissen um den Aufbau dieser Objekte. Für diesen Zweck hat die OMG die Interface Definition Language (IDL) erschaffen, eine von Programmiersprachen unabhängige Darstellung von Objekten und ihren Methoden und Attributen. Eine in IDL erzeugte Darstellung eines Objekts kann man sich als Vertrag vorstellen, zu dessen Einhaltung das Objekt verpflichtet ist. Basierend auf den IDL-Informationen erfolgt dann vom ORB aus die Weiterleitung und Durchführung von Aufrufen an die verteilten Objekte.

Damit verteilte Objekte in den eingesetzten Programmiersprachen verwendet werden können, benötigen die Compiler bereits zur Entwicklungszeit Informationen über den Aufbau der Objekte. Zu diesem Zweck werden lokale Stellvertreter für die entfernten Objekte erzeugt, die Aufrufe entgegennehmen und die Kommunikation mit dem entfernten Objekt (über den ORB) durchführen können. Für das Erzeugen der Stellvertreterklassen bieten die meisten Programmiersprachen entsprechende Tools an; die von uns getesteten Produkte brachten diese bereits mit.

Näheres zu CORBA und den verwendeten Konzepten findet sich in den FAQ der OMG unter www.omg.org/gettingstarted/corbafaq.htm.

Wichtig ist hierbei der Parameter „-i": Dieser gibt an, in welchem Verzeichnis *idlj* nach der Datei *ORB.idl* suchen soll, die in den IDL-Dateien referenziert ist. Diese Datei befindet sich im Unterverzeichnis *\lib* der Java-SDK-Installation. Da *idlj* keine Verschachtelungen auflöst, muss der Befehl für jede IDL-Datei ausgeführt werden. Zusätzlich muss *idlj* noch die Datei *Predef.idl* verarbeiten. Diese Datei wird von *CLSIDLGenerator* standardmäßig miterzeugt und enthält nur einen Datentyp. Das dazu erzeugte Unterverzeichnis spiegelt die Package-Struktur *Ch.Elca.Iiop* wider.

Die Java-Seite

Sämtliche von *idlj* erzeugten Java-Klassen müssen nun in die Java-Anwendung eingebunden werden. Dazu werden sie am besten in ein Unterverzeichnis des Java-Projekts kopiert, wobei auf die Package-Beschreibung zu achten ist. Abschließend bleiben noch zwei Aufgaben: Zum einen muss die eigentliche Client-Anwendung erzeugt werden. Sie fordert ein Datenspeicher-Objekt per IIOP an. Sobald die Referenz vorhanden ist, fügt sie Einträge zu der Hashtable hinzu und liest sie aus.

Zum anderen erfordert die Klasse *Eintrag* auf der Java-Seite noch ein wenig mehr Aufmerksamkeit: Da die Objekte dieses Typs als Werte übergeben werden, hat *idlj* eine abstrakte Klasse erzeugt. Deshalb wird noch eine Implementierung von *Eintrag* mit dem Namen *EintragImpl* erstellt.

Im vorliegenden Beispiel sind dabei nur die Methoden *Name* und *Wert* mit Leben zu füllen. Die eigentliche Client-Anwendung stellt über IIOP die Verbindung zum .NET-Server her und holt sich eine Referenz auf das Datenspeicher-Objekt. Danach findet ein kleiner Test mit anschließender Ausgabe der Ergebnisse statt. Die Quellen der beiden Klassen finden sich auf der Heft-CD in Listing 2 .

Um die fertige Anwendung zu testen, muss zunächst der .NET-Server über den Befehl *Server.exe* im Ausgabeverzeichnis des C#-Projekts gestartet werden. Danach wird die Java-Anwendung gestartet:

```
java -Djava.naming.factory.initial=
com.sun.jndi.cosnaming.CNCtxFactory
-Djava.naming.provider.url=
iiop://localhost:3456 -cp
<Erforderlicher Classpath>
Iiopexample.Client
```

Den ersten zusätzlichen Parameter benötigt Java für die RMI-Unterstützung. Der zweite Parameter spiegelt den URI des Servers wider, der im Beispiel zusammen mit der Port-Information verwendet wurde. Als Ausgabe der Anwendung erhält man erwartungsgemäß „Holger Schwichtenberg".

IIOP.NET-Installation

Die Installation von IIOP.NET gestaltet sich für GUI-Liebhaber etwas unangenehm. Vor der Benutzung von IIOP.NET setzt der Entwickler die Kompilierung der Quelltexte voraus, die jedoch ohne einige Vorbereitungen kläglich scheitert. Dem Paket liegt aber ein Makefile bei, das alle für die Kompilierung benötigten Informationen bereitstellt und von einem make-Tool interpretiert werden kann. Ein make-Tool bringen viele IDEs mit, bei Visual Studio .NET trägt es zum Beispiel den Namen *nmake*. Im Weiteren werden wir dieses Tool verwenden, der Vorgang ist aber auch auf andere make-Tools übertragbar. Zusätzlich wird noch der mit dem .NET Framework ausgelieferte C#-Compiler benötigt.

Damit *nmake* verwendet werden kann, sollte man zunächst den Pfad zu *nmake.exe* (liegt im *VS.NET\SDK\v1.1\bin*-Verzeichnis) in der Umgebungsvariablen *PATH* setzen. Gleichzeitig sollte dort auch der Pfad zu *csc.exe* (dem C#-Compiler) gesetzt werden, der unterhalb des Windows-Verzeichnisses in *Microsoft.NET\Framework\v1.1.4322* zu finden ist. Nachdem dies geschehen ist, wechselt man in das Verzeichnis *IIOPNet.src.1.6.1.final* des entpackten IIOP.NET-Paketes und führt den Befehl

```
nmake build-base
```

an der Kommandozeile aus. Daraufhin werden alle von IIOP.NET mitgelieferten Tools kompiliert und liegen zur Benutzung bereit. Der Befehl erstellt lediglich die Tools, jedoch nicht die Tutorial-Beispiele. Das erreicht man, indem der Befehl *nmake* ohne einen Parameter ausgeführt wird. Jedoch müssen vorher noch die Dateien *orb.idl* und *ir.idl* aus dem *Java\lib-Verzeichnis* in das Unterverzeichnis *IDL* kopiert werden. Darüber hinaus werden einige Umgebungsvariablen abhängig von den gewünschten Tutorials benötigt; eine genaue Auflistung findet sich in der Datei *readme.txt* des IIOP.NET-Paketes.

Janeva

Einen ähnlichen Ansatz wie IIOP.NET verfolgt auch *Borlands Janeva* [3]. Diese Software ist keine Open-Source-Lösung. Sie kann aber kostenfrei als Vollversion heruntergeladen und – soweit sie nur für die Kommunikation auf dem lokalen Computer eingesetzt wird – auch kostenfrei genutzt werden. Janeva besteht im Kern aus einer Java-basierten CORBA-Implementierung, die mittels J# nach .NET portiert wurde und einen vollwertigen ORB bereitstellt. Auch hier erfolgt jegliche Kommunikation über IIOP.

Zugriff auf Java-Objekte

Neben dem Zugriff auf CORBA-Objekte ist mit Janeva auch der Zugriff auf EJB-Komponenten aus .NET-Anwendungen heraus möglich. Dazu stehen für den Entwickler drei verschiedene Wege bereit:

- Der erste Weg besteht darin, über .NET Remoting mithilfe eines IIOP-Channels auf CORBA-Objekte zuzugreifen, wie es Listing 3 auf der Heft-CD zeigt. Hier werden mit dem mitgelieferten CORBA-IDL-to-C#-Compiler *idl2cs.exe* aus vorhandenen IDL-Dateien C#-Proxy-Klassen generiert, die anschließend in die Anwendung mit eingebunden werden. Dies entspricht dem Vorgehen bei IIOP.NET. Da die Arbeit mit .NET Remoting am wenigsten Aufwand bedeutet, ist dies wohl der angenehmste und einfachste, aber auch der unflexibelste Weg.
- Alternativ dazu hat der erfahrene CORBA-Entwickler die Möglichkeit, im CORBA-Stil zu arbeiten, wie es Listing 4 auf der Heft-CD zeigt. Dazu kann er aus vorhandenen IDL-Dateien ebenfalls mit dem mitgelieferten CORBA-IDL-to-C#-Compiler C#-basierte CORBA-Stubs generieren und diese mit in die Anwendung einbinden. Die Vorgehensweise hier ist zwar der ge-

wohnten Vorgehensweise in Java sehr ähnlich, allerdings setzt sie einiges an CORBA-Grundwissen voraus.

- Der dritte Weg besteht darin, mithilfe des Java-to-C#-Compilers *java2cs* aus J2EE-Archivdateien, also EAR- oder JAR-Archiven, Proxys zu generieren und analog zur Vorgehensweise in Java auf EJB-Komponenten zuzugreifen. So zeigt es Listing 5 auf der Heft-CD.

Die bei Janeva mitgelieferten Tools erzeugen zwar allesamt nur C#-Code. Weil aber ausschließlich CLS-konforme Typen verwendet werden, ist nach der Kompilierung der Dateien auch der Zugriff von jeder anderen .NET-Sprache aus möglich.

Zugriff auf .NET-Objekte

Seit dem Erscheinen der Janeva-Version 6 ist nun auch die Bereitstellung von .NET-basierten Server-Objekten über .NET Remoting oder direkte CORBA-Anbindung möglich. Der Zugriff auf diese Objekte aus Java erfolgt regulär wie der Zugriff auf ein normales CORBA-Objekt. Da Janeva ursprünglich mit dem Gedanken entwickelt wurde, bestehende CORBA- oder J2EE-Anwendungen um .NET-Clients zu erweitern, wird kein Werkzeug mitgeliefert, um die von Java und CORBA benötigten IDL-Definitionen aus .NET-Klassen zu erzeugen. Ausgangspunkt sind bei Janeva also immer IDL-Definitionen oder J2EE-Archive, aus denen im Anschluss entsprechende Proxy-Klassen generiert werden. In diesem Punkt ist IIOP.NET komfortabler.

Funktionsumfang

Janeva kann nicht nur alle Java-Datentypen automatisch in ihre äquivalenten .NET-Datentypen konvertieren, sondern auch Plattform-spezifische Container-Objekte entsprechend umwandeln. Neben dem obligatorischen *per-reference*-Zugriff auf entfernte Objekte ermöglicht Janeva beim Zugriff auf EJBs allerdings auch die Konvertierung benutzerdefinierter Werttypen. Besitzt ein Objekt nur öffentliche Variablen, so kann die Konvertierung weiterhin automatisch erfolgen. Sollen aber auch private Variablen sowie Methoden genutzt werden, ist ein manueller Eingriff des Entwicklers in die Konvertierung nötig.

Da das Transferieren von Werttypen zwischen Java und .NET aufgrund der unterschiedlichen Objektmodelle nicht einfach mit Serialisierung und Deserialisierung getan ist, kommen hier so genannte ValueFactory-Klassen zum Einsatz. Sie sind für das Erzeugen und Initialisieren der Objekte in .NET sowie den korrekten Transfer des Objekt-Inhalts zwischen .NET und Java zuständig. Für jede Java-Klasse, die *per-value* übertragen wird, existiert eine eigene ValueFactory-Klasse, die das Bindeglied zwischen der jeweiligen .NET- und Java-Klasse darstellt.

Die Erzeugung eines Objekts in .NET und der Transfer des Objektinhalts in dieses Objekt werden von der ValueFactory-Klasse in zwei getrennten Schritten vollzogen. Dies gibt dem Entwickler die Möglichkeit, durch die Angabe so genannter Hints beim Generieren der C#-Proxy-Klassen Einfluss auf diesen Vorgang zu nehmen. Da Janeva die Konvertierung von Methoden beim *per-value*-Zugriff nicht unterstützt, weil dafür ein Reengineering der Java-Klassen notwendig wäre, hat der Entwickler an dieser Stelle die Möglichkeit, eine eigens erstellte C#-Klasse zu spezifizieren. Sie wird anstelle der automatisch generierten Klasse zum Instanzieren des .NET-Objekts benutzt und stellt auch unter .NET die entsprechenden Methoden zur Verfügung. Auch die Behandlung von privaten Variablen kann an dieser Stelle beeinflusst werden. Ausführliche Informationen zur Vorgehensweise liefert die Janeva-Dokumentation.

Neben der Typkonvertierung unterstützt Janeva als einziges Produkt in dieser Testreihe auch zusätzliche Funktionen wie Lastverteilung, Transaktionen, Fehlertoleranz und Sicherheitsfunktionen wie Autorisierung und Authentifizierung, soweit sie durch die entsprechenden CORBA- oder J2EE-Server bereitgestellt werden. Hilfestellung leistet auch hier die Dokumentation.

Kompatibilität

Janeva ist kompatibel mit den OMG-GIOP- und -IIOP-1.2-Spezifikationen. Die Janeva-CORBA-Unterstützung basiert auf der CORBA-2.4.2-Spezifikation und ist theoretisch mit allen CORBA 2.x unterstützenden Produkten interoperabel. Des Weiteren unterstützt Janeva alle auf der J2EE-Version 1.3 basierenden Applikations-Server.

Installation

Im Gegensatz zu IIOP.NET wird Janeva als vollständige Installationsversion ausgeliefert. Der Benutzer hat dabei die Auswahl zwischen den folgenden Komponenten:

- Janeva Runtime,
- Janeva Tools,
- Dokumente und Beispiele und
- Visual-Studio-.NET-Add-In/C#-Builder-Add-In, soweit die entsprechende Entwicklungsumgebung installiert ist.

Die Installation an sich verläuft problemlos, allerdings hat der Benutzer für ein korrektes Funktionieren noch dafür zu sorgen, dass der Pfad zum zu benutzenden .NET Framework in den Umgebungsvariablen korrekt gesetzt und die J#-Klassenbibliothek *vjslib* installiert ist. Um Versions-Konflikte zu vermeiden, sollte für jede installierte Version des .NET Frameworks auch die entsprechende Version der J#-Laufzeitumgebung installiert sein. Sollte dies nicht akzeptabel sein, besteht auch die Möglichkeit, den Compilern *idl2cs* und *java2cs* über die .NET-Konfigurationsdatei manuell mitzuteilen, welche Version des Frameworks sie benutzen sollen. Allerdings ist die in dieser Version relativ gute und umfangreiche Dokumentation bisher nur auf Englisch und auf der Borland-Website zusätzlich auf Japanisch vorhanden. Außerdem verursacht das Visual-Studio-.NET-Add-In in der Version 2003 beim Start der Entwicklungsumgebung grundsätzlich eine Fehlermeldung. Gut gelungen ist Borland die Integration in die Entwicklungsumgebung C# Builder. Durch die Erweiterung des *Verweise*-Dialogs wird das Hinzufügen von Java-Klassen und CORBA-Objekten so einfach wie die Referenzierung von .NET-Klassen selbst.

Erste Schritte

Die mitgelieferten Codebeispiele sind nach der Installation noch unvollständig und müssen zunächst durch Erzeugung der Proxy-Klassen aus den vorhandenen IDL-Dateien erstellt werden. Dafür steht dem Entwickler eine Reihe von Batch-Dateien zur Verfügung, die die Generierung automatisieren sollen. Dies erweist sich allerdings als problematisch, da neben den C#-Quellcode-Dateien auch Java-Quellcode-Dateien erzeugt werden sollen. Für diese werden die dafür erforderlichen Compiler aber nicht mitgeliefert. Die fehlenden Compiler sind Bestandteil von Borlands Visibroker.

Verzichtet man auf die Erzeugung der Java-Dateien und hangelt sich anhand der Dokumentation durch die Beispiel-Projekte, so kommt man relativ schnell zu ersten Erfolgserlebnissen. Die ersten eigenen Gehversuche erweisen sich den-

noch als recht knifflig und erfordern ein wenig Einarbeitungszeit.

Deployment

Das Deployment von Janeva-basierten Anwendungen ist relativ einfach und erfordert keine Änderungen an bestehenden CORBA- oder J2EE-basierten Systemen, falls diese durch .NET-basierte Programme erweitert werden sollen. Neben dem obligatorischen Microsoft .NET Framework Redistributable Package werden folgende Komponenten benötigt:

- das Microsoft Visual J# Redistributable Package,
- die Janeva-Laufzeit-Bibliotheken, die in Form von .NET-Assemblies vorliegen, und
- ein Janeva Deployment License Key, der beispielsweise als eingebettete Ressource weitergegeben werden kann.

Fazit

In diesem Artikel wurden vier der bekanntesten Component Bridges beleuchtet, die Java und .NET auf unterschiedliche Art und Weise miteinander verbinden. Die Frage nach dem besten Ansatz kann jedoch nicht eindeutig beantwortet werden, denn sie hängt in starkem Maße damit zusammen, welches Ziel mit der Integration der beiden Welten verfolgt wird.

Ja.NET verspricht, die Plattformen in beide Richtungen auf einfache und schnelle Weise zu verknüpfen. Das funktioniert auch verteilt über ein oder mehrere Netzwerke. Besonders gut gefallen hier die Funktionsvielfalt der leicht zu bedienenden Laufzeitumgebung sowie die Möglichkeit, auch auf J2EE-Applikations-Server zugreifen zu können.

Wer nur lokal von .NET aus auf Java-Klassen zugreifen und eine sehr enge Kopplung herstellen möchte, findet mit JuggerNET eine sehr schlanke, intuitive und vor allem performante Lösung vor, mit der sich Java-Code leicht in .NET-Applikationen einbinden lässt. Dadurch, dass die JVM direkt in den CLR-Prozess geladen wird, treten kaum Geschwindigkeitsnachteile durch die Überbrückung der beiden Plattformen auf. Insgesamt geht die Arbeit mit JuggerNET sehr gut von der Hand. Allerdings ist mit Jugger.NET noch keine bidirektionale Kopplung möglich. Mithilfe von asynchronen Callbacks auf der Java-Seite lässt sich jedoch je nach Anwendungsfall auch diese Einschränkung umgehen. Die direkte lokale Kopplung von Java und .NET ermöglicht des Weiteren auch die Nutzung sämtlicher Java-Remoting-Mechanismen innerhalb von .NET-Anwendungen.

Beachtung verdient das Projekt IIOP.NET vor allem hinsichtlich der Tatsache, dass es kostenlos verfügbar ist und dennoch vom Funktionsumfang her nichts vermissen lässt, was einer Integration im Wege stehen würde. Einziger Minuspunkt ist die etwas umständliche Bedienung, die jedoch nach kurzer Eingewöhnung nicht mehr ins Gewicht fällt. Durch die Möglichkeit, nicht nur Java und .NET zu verbinden, sondern auch andere CORBA-Anwendungen mit .NET kommunizieren zu lassen, wird IIOP.NET zu einem mächtigen Werkzeug, das man sich in jedem Fall einmal ansehen sollte. Für das Entwickeln von Enterprise-Applikationen, die neben Skalierbarkeit auch zusätzliche Dienste wie Transaktionsunterstützung oder Lastverteilung beim Zugriff auf J2EE-basierte Applikations-Server oder CORBA-basierte Anwendungen benötigen, eignet sich das Bridging-Flaggschiff Borland Janeva, das einen vollständigen CORBA-ORB in .NET bereitstellt, besonders gut. Auch zur Bereitstellung .NET-basierter CORBA-Server-Objekte ist es seit der jüngsten Version geeignet. Positiv hervorzuheben ist die gute Integration in die Entwicklungsumgebung C# Builder.

Weitere Produkte

In diesem Beitrag konnten aus Platzgründen nicht alle verfügbaren Bridging-Produkte besprochen werden. Mit Ja.NET [1] vergleichbar ist JNBridgePro der Firma JNBridge [2]. Den gleichen Ansatz wie IIOP.NET [4] und Janeva [3] verfolgt das Open-Source-Projekt Remoting.Corba [5]. Einen vollwertigen, in C# entwickelten CORBA Object Request Broker bietet die deutsche Firma MiddTec mit dem Produkt MiddCor [6]. Auch hierzu gibt es ein vergleichbares Open-Source-Projekt mit Namen The harmless ORB [7]. |||||||

[1] http://ja.net.intrinsyc.com/
[2] www.jnbridge.com/
[3] www.borland.com/janeva/
[4] http://iiop-net.sourceforge.net/
[5] http://remoting-corba.sourceforge.net/
[6] www.middtec.de/02_prod_middcor.htm
[7] http://sourceforge.net/projects/harmless/
[8] www.codemesh.com/en/JuggerNETCurrentRelease.html

Tabelle 1

Vergleich der besprochenen Produkte.

	Ja.NET	JuggerNET	IIOP.NET	Janeva
Hersteller	Intrinsyc	Codemesh	Elca Informatique	Borland
Preis	Ab 375 US-Dollar	Auf Anfrage	kostenlos	Entwicklungslizenz kostenlos; Serverlizenz 2.000 EUR
Website	http://ja.net.intrinsyc.com/	http://www.codemesh.com/en/JuggerNETCurrentRelease.html	http://iiop-net.sourceforge.net/	http://www.borland.com/janeva/
Aktuelle Version	1.5.1294	1.1	1.6.2	6
.NET -> Java	Ja	Ja	Ja	Ja
Java -> .NET	Ja	Nein	Ja	Ja
.NET -> CORBA	Nein	Nein	Ja	Ja
CORBA -> .NET	Nein	Nein	Ja	Ja
Verwendetes Remoting-Protokoll	TCP/IP (binär), http (SOAP)	— (nur In-Process)	IIOP	IIOP
IDE-Integration	Nein	Nein	Nein	VS.NET, C#-Builder