Einführung in Web Services

Seminararbeit

von
Sascha Schnaufer
aus Mannheim

vorgelegt am
Lehrstuhl für Praktische Informatik IV
Prof. Dr. W. Effelsberg
Fakultät für Mathematik und Informatik
Universität Mannheim

August 2004

Betreuer:
Dipl.-Inform. Andreas Kamper (Karlsruhe)
Dipl.-Wirtsch.-Inf. Jürgen Vogel (Mannheim)

Inhaltsverzeichnis

1 Einleitung 1

2 Verteilte Anwendungen 1

2.1 Eigene Entwicklungen 2

2.1 Remote Method Invocation (RMI) 3

2.2 Common Object Request Broker Architecture (Corba) 6

2.3 Webservices im Vergleich zu RMI und Corba 8

3 Webservice Beispiel in .NET 9

3.1 Aufbau des Beispiel-Webservice 10

3.2 Implementierung der Server-Anwendung 11

3.3 Implementierung der Client-Anwendung 12

3.4 Kommunikation zwischen Server und Client 12

3.5 Die WSDL Datei 14

Literaturverzeichnis 18

1 Einleitung

Nachdem im ersten Teil der Arbeit der grundsätzliche Aufbau von Webservices gezeigt wurde, soll hier der Schwerpunkt auf den praktischen Einsatz dieser Technologie liegen.
Webservices werden auch im Intranet-Umfeld eingesetzt und konkurrieren dort mit anderen Technologien wie Corba und RMI. Um die technologischen Unterschiede aufzuzeigen wird eine Beispielanwendung mit RMI und als Webservice implementiert und verglichen.

2 Verteilte Anwendungen

Schon bei einfachen Softwareprojekten, wie beispielsweise einer Adressenverwaltung, möchten mehrere Anwender die Möglichkeit haben eine gemeinsame Datenbasis zu benutzen. Die Adressen müssen dann zwischen den verschiedenen Rechnern der Benutzer ausgetauscht werden. Meist wird in einem solchen Fall eine Client-Server-Architektur gewählt. Die Funktionalität wird dann in zwei Anwendungen aufgeteilt. Die Server-Anwendung stellt den Client-Anwendungen Dienste, in diesem Beispiel die Adressen, zur Verfügung. Bei einem Faxprogramm könnte als Dienst die Möglichkeit Faxe zu verschicken bereitgestellt werden. So müsste nur der Rechner auf dem die Server-Anwendung läuft über ein Faxmodem oder eine ISDN-Karte verfügen.
Neben dem teilen von Ressourcen, wie gewisser Hardware Komponenten, ermöglicht die Architektur eine gute Trennung der Programmfunktionen. Hierbei unterscheidet man abstrakt die Logik des Programms, das User-Interface und das Datenmanagement[1]. Bei der Client-Server-Architektur werden meist im Server die Logik und das Datenmanagement gekapselt und die Client-Anwendung nur als User-Interface benutzt. Diese Aufteilung ermöglicht es bei vielen Updates, die nur die grundsätzliche Logik des Programms betreffen, diese nur auf dem Server zu installieren ohne dass ein Update der Clients notwendig ist. Mit dieser Aufgabenteilung können auch leichter Client-Anwendungen für ein anderes Betriebssystem realisiert werden, da nur ein kleiner Teil des Programms neu programmiert werden muss.

[1] Vgl. Kuhlins, Stefan [2]: Kapitel 1.

Bei der Implementierung von verteilten Anwendungen entstehen aber auch Problempunkte. So muss davon ausgegangen werden, dass unterschiedliche Hardware, unterschiedliche Betriebssysteme, unterschiedliche Programmiersprachen und unterschiedliche Kommunikationsprotokolle zum Einsatz kommen. Ziel ist es in diesem heterogenen Umfeld die Interoperabilität zu gewährleisten.

2.1 Eigene Entwicklungen

Es ist schon mit einfachen Mitteln möglich eine Client-Server-Architektur zu realisieren. Zuerst muss festgelegt werden auf welchem Weg die Anwendungen miteinander kommunizieren sollen. Hierfür bietet sich, schon wegen der Verbreitung, das TCP[2] Protokoll an. Nachdem ein beliebiger Port festgelegt wurde, kann die Server-Anwendung mit der Client-Anwendung kommunizieren. Doch damit sind noch keine Dienste bereitgestellt. Zuerst müssen hierfür Nachrichten definiert werden. Eine einfache Möglichkeit ist, alle angebotenen Funktionen durchzunummerieren und diese Nummer zu senden wenn die entsprechende Funktion aufgerufen werden soll.
Soll beispielsweise die Methode **`String getKundenName( String Kundennummer )`** veröffentlicht werden, so geben wir dieser Funktion die eindeutige Nummer 1. Möchte nun der Client diese Funktion nutzen, baut er, wenn noch nicht geschehen, eine TCP Verbindung mit dem vereinbarten Port auf und schickt ein Paket mit dem Inhalt: '1, Kundennummer' und erhält ein Paket mit dem Aufbau '1, Kundenname' zurück.

```
Beispiel:
Anfragepaketes: '1, K50015'
Antwortpaketes: '1, Müller'
```

Bei dieser Art der Implementierung fällt sehr viel Definitionsarbeit an und der gesamte Kommunikationscode, inklusive der Typenprüfung der Parameter und Rückgabewerte, muss selbst programmiert werden. Dadurch sind solche Lösungen unflexibel und fehleranfällig.
Es werden nun Technologien vorgestellt, die dem Programmierer die Möglichkeit geben Methoden zu veröffentlichen und dabei die gesamte Kommunikation übernehmen.

[2] Vgl. Transmission Control Protocol [4]

2.1 Remote Method Invocation (RMI)

RMI[3] basiert auf der Java Virtual Machine (JVM) und ermöglicht die einfache Entwicklung von verteilten Java-Anwendungen. Die Vorteile liegen in der natürlichen Integration in die Sprache und der automatischen Typenprüfung.

Abb. 1: Verteilte Anwendung mit RMI

Die gesamte Kommunikation wird von RMI übernommen. Es werden nicht, wie bei der Eigenentwicklung, Methoden veröffentlicht sondern Objekte. Die Server-Anwendung erzeugt Objekte und wartet auf Methodenaufrufe von den Client-Anwendungen. Möchte eine Client-Anwendung auf ein entferntes Objekt zugreifen, bekommt es eine Referenz und kann damit wie mit lokalen Objekten arbeiten.

Es sind folgende Schritte notwenig um mit RMI zu arbeiten:

1. Remote Java-Interface einer Klasse definieren
2. Interface implementieren
3. Server implementieren
4. Client implementieren

Alle vier Schritte werden nun anhand einer Beispielanwendung durchgeführt.

1 Remote Interface definieren:

Damit die Client-Anwendung die verfügbaren Methoden kennt, muss ein Interface (siehe Abb. 2) definiert werden. Dieses erbt von dem `Remote Interface`, welches im `java.rmi` Paket enthalten ist. Alle Methoden müssen eine `RemoteException` werfen, die auch im `java.rmi` Paket enthalten ist. In diesem Beispiel wird ein Zähler definiert, der es ermöglicht den Zählerwert um eins zu erhöhen, auf null zurückzusetzen und den aktuellen Wert auszulesen. Wie in einem Interface üblich findet hier noch keine Implementierung statt.

[3] Vgl. RMI Architecture and Functional Specification [11]

```
import java.rmi.*;
public interface RMICounter extends Remote
{
  void reset() throws RemoteException;
  void increment() throws RemoteException;
  int value() throws RemoteException;
}
```

Abb. 2: RMI Remote Interface

2 Interface implementieren:

Nun findet die eigentliche Implementierung des Zählers statt. Hierfür wird eine Klasse programmiert (siehe Abb. 3), die das eben definierte Interface implementiert und von **`UnicastRemoteObject`** erbt. Durch diesen Ansatz wird ein Teil des für die Veröffentlichung benötigten Codes geerbt.

```
import java.rmi.*;
import java.rmi.server.*;

public class RMICounterImpl extends UnicastRemoteObject
  implements RMICounter
{
  private int counter;
  public RMICounterImpl() throws RemoteException { super(); }
  public void reset() throws RemoteException { counter = 0; }
  public void increment() throws RemoteException { ++counter; }
  public int value() throws RemoteException { return counter; }
}
```

Abb. 3: RMI Remote Klasse

3 Server implementieren

Nachdem die Klasse programmiert wurde, wird nun die Server-Anwendung geschrieben (siehe Abb. 4). Die **`RMI-Registry`** dient als Middleware zwischen den Anwendungen. Sie übernimmt also die gesamte Kommunikation und kann als externes Programm oder durch den Aufruf der statischen Methode **`createRegistry(PortNr.)`** innerhalb der Server-Anwendung gestartet werden. Nun wird ein Objekt der realisierten Klasse erzeugt. Dabei handelt es sich um ein normales lokales Java Objekt. Um den Client-Anwendungen den Zugriff zu ermöglichen, wird das Objekt an einen Namensstring gebunden. In diesem Beispiel wird das Objekt **`myCount`** an den Namen “Counter“ gebunden.

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

public class RMICounterServer
{
  public static void main(String args[])
  {
    try
    {
      LocateRegistry.createRegistry(1099);
      RMICounter myCount = new RMICounterImpl();
      Naming.rebind("Counter", myCount);
      System.out.println("Server ready");
    }
    catch(RemoteException x) { … }
    catch(java.net.MalformedURLException x) { … }
  }
}
```

Abb. 4: RMI Server-Anwendung

Um die Server-Anwendung und die Client-Anwendung ausführen zu können, muss zuerst noch mit Hilfe von dem Kommandozeilenprogramm **`rmic`** Hilfsklassen erzeugt werden. Im Beispiel erzeugt der Aufruf von **`rmic RMICounterImpl`** die Klassen **`RMICounterImpl_Skel.class`** und **`RMICounterImpl_Stub.class`**, die für die Kommunikation benötigt werden. Vereinfacht ausgedrückt leiten diese Klassen den jeweiligen Methodenaufruf innerhalb der Client-Anwendung, mit Hilfe der RMI-Registry, an die Server-Anwendung weiter. Analog wird der Rückgabewert übertragen.

4 Client implementieren

In der Client-Anwendung wird nun mit **`Naming.lookup(RMI URL)`** eine Referenz des entfernten Objekts ermittelt. Dabei gibt die URL das verwendete RMI Protokoll, den Rechner auf dem das Objekt veröffentlicht wurde und den Namen, an den das Objekt gebunden wurde, an. Im Beispiel (siehe Abb. 5) wird der Servername als Kommandozeilenparameter übergeben. Nun ist in der Variable **`myCount`** eine Referenz auf das entfernte Objekt. Der Programmierer kann dieses Objekt genauso nutzen wie lokale. Zuerst wird im Beispiel der Zählerwert mit **`reset()`** auf null gesetzt, dann um eins erhöht und zuletzt ausgegeben.

Erwartungsgemäß wird diese Client-Anwendung den Wert eins ausgeben. Würde der Aufruf der **`reset()`** Methode auskommentiert werden, würde bei einem zweiten Aufruf der Wert zwei ausgegeben. Es handelt sich um ein und dasselbe Objekt für alle

Client-Anwendungen, egal auf welchem Rechner sie ausgeführt werden. Des Weiteren ist die Lebensdauer des Objekts in keiner Weise an die Client-Anwendung gebunden, sondern ausschließlich durch den Server vorgegeben.

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class RMICounterClient
{
  public static void main(String args[])
  {
    try
    {
      RMICounter myCount = (RMICounter) Naming.lookup("rmi://" +
                                        args[0] + "/Counter");
      myCount.reset();
      myCount.increment();
      System.out.println("Result = " + myCount.value());
    }
    catch(NotBoundException x) { … }
    catch(java.net.MalformedURLException x) { … }
    catch(RemoteException x) { … }
  }
}
```

Abb. 5: RMI Client-Anwendung

RMI ermöglicht es mit relativ wenig Aufwand verteilte Anwendungen zu erstellen. Die Integration in die Programmiersprache ist nahezu perfekt. Denn nachdem die Referenz auf das entfernte Objekt ermittelt wurde, lässt sich mit diesem wie mit einem lokal vorhanden Objekt arbeiten. Zu den Nachteilen zählt andererseits, dass keine andere Programmiersprache als Java benutzt werden kann.

2.2 Common Object Request Broker Architecture (Corba)

Corba wurde von der Objekt Management Group (OMG) spezifiziert, einer Vereinigung die unter anderem das Ziel hat die Entwicklung einer offenen Architektur für verteilte Anwendungen unter Ausnutzung der Objekttechnologie voranzutreiben. Die OMG umfasst mehr als 800 IT Unternehmen wie beispielsweise Sun, IBM und Microsoft.

Die Ansätze von RMI und Corba sind recht ähnlich, weshalb hier nur kurz auf Corba eingegangen wird. Der grundsätzliche Unterschied liegt darin, dass Corba nicht an eine spezielle Programmiersprache gebunden ist. Da Corba erst einmal eine reine Spezifikation ist, wird für eine Corba-Anwendung eine Implementierung dieser benötigt. Benutzt werden

kann jede Programmiersprache, für die es einen passenden Object Request Broker (ORB)[4] und einen Interface Definition Language (IDL)[5] Compiler gibt.

Die ORB Spezifikation beschreibt den Mechanismus, der das Aufrufen von Diensten ermöglicht. Dieser Mechanismus ist hardwareunabhängig, betriebssystemunabhängig und programmiersprachenunabhängig. Es ist also möglich diese Spezifikation auf jedem beliebigem Gerät zu implementieren und mit anderen Implementierungen über das Inter-Inter-ORB-Protokoll (IIOP)[6] zu kommunizieren.

Es wird wie bei RMI ein Interface deklariert, welches aber unabhängig von der verwendeten Programmiersprache sein soll. Deshalb wird die eigenständige IDL zum definieren des Interfaces benutzt. Der IDL Compiler erzeugt dann anhand der IDL-Datei Hilfsklassen (analog zu **`rmic`** bei RMI). Der Syntax von IDL unterscheidet sich von den meisten objektorientierten Programmiersprachen nur geringfügig. Das vom RMI Bespiel bekannte Interface wird im Folgenden in IDL dargestellt (siehe Abb. 6).

```
module CounterApp {
  interface Counter {
    void reset();
    void increment();
    long value();
  };
};
```

Abb. 6: IDL Interface

Es muss aber beachtet werden, dass IDL ein eigenständiges Typensystem besitzt, welches mit dem der Programmiersprache nicht übereinstimmen muss.

IDL Typ	Java Typ	Beschreibung
long	int	32 Bit
unsigned long	int	32 Bit ohne Vorzeichen
string	java.lang.String	Zeichenkette

Tabelle1. Auszug aus der IDL nach Java Mapping Tabelle[7]

[4] Vgl. CORBA Component Model v3.0 [7]
[5] Vgl. Catalog of OMG IDL [9]
[6] Vgl. CORBA - IIOP Specification[6]
[7] Vgl. IDL to Java Language Mapping [8] S. 18.

Prinzipielle Vorgehensweise:

1. Definition des Interfaces in IDL
2. Übersetzen des Interfaces (per IDL Compiler)
3. Interface implementieren
4. Server implementieren
5. Client implementieren

Dieser Ablauf ähnelt der Vorgehensweise bei RMI, mit dem Zwischenschritt über IDL, der die Programmiersprachenunabhängigkeit ermöglicht.
Im Java SDK ist ein ORB und ein IDL Compiler bereits enthalten. Wenn das Interface ebenfalls in Java realisiert wird, ist es möglich RMI Objekte über das IIOP freizugeben und so in Corba Client-Anwendungen zu nutzten[8].

2.3 Webservices im Vergleich zu RMI und Corba

Im ersten Teil dieser Arbeit wurde der Aufbau von Webservices beschrieben und wird deshalb nicht erneut im Detail dargestellt.
Sowohl bei RMI als auch bei Corba wurde das gesamte Kommunikationssystem neu entworfen und spezifiziert. Dies unterscheidet sie von Webservices, bei denen versucht wurde bisherige Standards und existierende Technologien zu nutzen. So wird meist als „Transportprotokoll“ das bewährte HTTP[9] benutzt und die gesamte Kommunikation wird im XML[10] Format ausgetauscht und ist damit auch für den Menschen lesbar. Durch den Einsatz von HTTP sind Webservices Firewall freundlich, da diese meist schon entsprechend konfiguriert sind. Für den Fall, dass auch der Inhalt der ausgetauschten SOAP[11] Nachrichten von der Firewall untersucht werden soll, sind SOAP-Level Firewalls verfügbar[12]. Im Gegensatz zu RMI und Corba besitzen Webservices ohne weitere Maßnahmen keinen Zustand. Auch werden keine Objekte für Client-Anwendung angeboten, sondern eine Menge beliebiger Funktionen.
Während mit RMI und Corba meist eine verteilte Anwendung realisiert werden soll, die Client-Anwendung und die Server-Anwendung werden also von der gleichen

[8] Vgl. RMI over IIOP [10]
[9] Vgl. Hypertext Transfer Protocol - HTTP/1.1 [3]
[10] Vgl. Extensible Markup Language (XML) [13]
[11] Vgl. Simple Object Access Protocol (SOAP) 1.1 [15]
[12] Z.B.: Reactivity XML Firewall (http://www.reactivity.com/products/)

Entwicklergruppe entworfen, sind Webservices besonders dafür ausgelegt einen Dienst anzubieten, der auch oder sogar speziell von Unbekannten genutzt wird. Eine Fuggesellschaft könnte einen Webservice anbieten, der es Entwicklern ermöglicht die Flugpläne abzufragen, um diese Informationen in ihren Anwendungen zu verarbeiten. Denkbar wäre beispielsweise eine Anwendung für Reisebüros. Aufgrund dieser Ausrichtung von Webservices ist in der Spezifikation das webbasierte Informationssystem Universal Description, Discovery and Integration (UDDI) enthalten. Mit dem webbasierten Interface von UDDI kann ein Entwickler einen passenden Webservice finden. Da UDDI selbst ein Webservice ist, kann eine Eintragung, Aktualisierung oder Suche auch automatisiert werden.[13]

Schon etablierte Standards zu nutzen, bietet die Möglichkeit an deren Umfang, Verbreitung und Weiterentwicklung zu partizipieren. Um die Übertragung bei einem Webservice zu verschlüsseln kann beispielsweise einfach HTTPS benutzt werden. Der verwendete Webserver muss nur entsprechend konfiguriert werden. Analoges gilt für XML, da fast für jede Programmiersprache ein XML Parser vorhanden ist der genutzt werden kann.

RMI und Corba übertragen ihre Daten in binärer Form und sparen damit den Overhead von XML Dokumenten, der durch die Textdarstellung und die tagbasierte Struktur entsteht, ein. Auch die Verarbeitung von XML Dokumenten ist wesentlich rechenintensiver, da sie immer geparst werden müssen.

Jede der vorgestellten Technologien hat Vorteile. Welche überwiegen hängt von den Anforderungen des jeweiligen Projektes ab.

3 Webservice Beispiel in .NET

Die Funktionsweise eines Webservices soll nun anhand eines Beispiels verdeutlicht werden. Es wird die Implementierung des Server und des Clients gezeigt, die zugehörige Web Services Description Language (WSDL)[14] Datei und alle Nachrichten die zwischen Server und Client, bei einem Methodenaufruf gesendet werden.

[13] Vgl. UDDI Specifications [5]
[14] Vgl. Web Services Description Language [12]

3.1 Aufbau des Beispiel-Webservice

Um die ausgetauschten Nachrichten zwischen Server und Client auszuwerten, wird der TCPMonitor, eine Anwendung des Axis Projekts zum loggen von TCP Verbindungen, zwischen beide geschaltet.

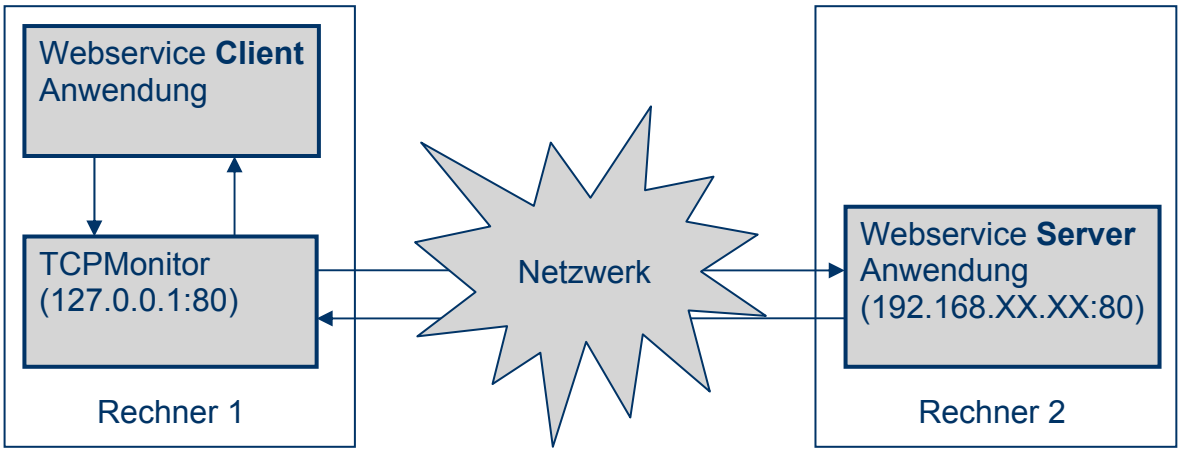

Abb. 7: Der Aufbau des Beispiel-Webservice

Wie der Abb. 7 zu entnehmen ist, läuft die Client-Anwendung auf dem Rechner 1 und der Webservice auf dem Rechner 2. Die Client-Anwendung wird nun die Verbindung nicht direkt zum Server aufbauen, sondern zum TCPMonitor, der auf dem gleichen Rechner installiert ist. Der TCPMonitor baut dann eine Verbindung zu Rechner 2 auf und überträgt alle von der Client-Anwendung erhaltenen Daten und schickt analog alle Daten des Servers an die Client-Anwendung. Die ausgetauschten Nachrichten, die alle nur aus Text bestehen, werden nun im TCPMonitor angezeigt (siehe Abb. 8).

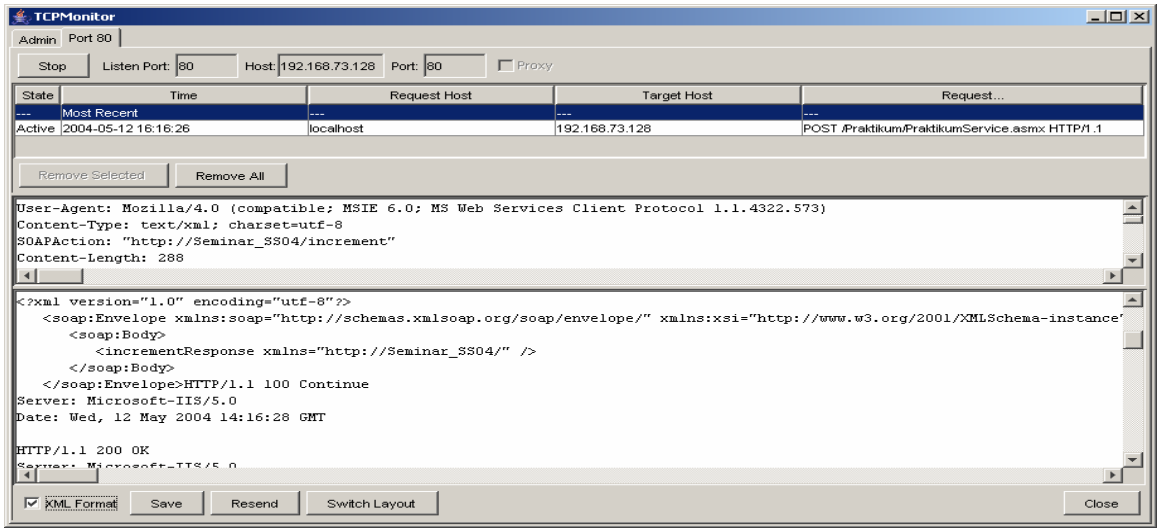

Abb. 8: Der TCPMonitor

3.2 Implementierung der Server-Anwendung

Die Implementierung dieses Webservice ist in der Programmiersprache C# des .NET Frameworks realisiert. Nur zwei Methoden werden angeboten: eine zum erhöhen eines Zählers um eins und eine um den aktuellen Zählerwert zu ermitteln.
In C# genügt es die eigene Klasse von **`System.Web.Services.WebService`** abzuleiten um einen Webservice zu erzeugen. Die Methoden die veröffentlicht werden sollen, müssen mit dem Attribut **`[WebMethod]`** vor der eigentlichen Methoden Deklaration markiert werden. Vor der Klassendeklaration kann optional noch eine Beschreibung und der Namespace des Webservices angegeben werden (Vgl. Abb. 9).

```
using System;
...

namespace Praktikum
{
  [WebService(Namespace="http://Seminar_SS04/",Description="Webservice
                  für Präsentationszwecke im Seminar Webservices SS04")]
  public class SeminarService : System.Web.Services.WebService
  {
     private int count = 0;
    ...
    [WebMethod]
    public void increment()
    { count++; }

    [WebMethod]
    public int countValue()
    { return count; }
  }
}
```

Abb. 9: Implementierung des Servers

Wenn als Entwicklungsumgebung das Microsoft Visual Studio .NET benutzt wird, werden alle benötigten Dateien automatisch auf dem Webserver abgelegt und eingerichtet. Greift man auf den Webservice mit einem Webbrowser zu, wird automatisch eine Webseite mit einer Liste der verfügbaren Methoden und weiteren Informationen über den Webservice erzeugt. Die Methoden können auch über den Webbrowser per HTTP Post direkt getestet werden. Diese Funktion wird nur ermöglicht wenn der Webzugriff lokal auf dem gleichen Rechner stattfindet.

3.3 Implementierung der Client-Anwendung

Um in dem Projekt der Client Anwendung den Webservice nutzen zu können, muss eine so genannte Web-Reference hinzugefügt werden. Visual Studio bietet die Möglichkeit in einem browserähnlichen Fenster die URL des Webservices einzutragen. Von dort wird dann automatisch die WSDL Datei herunter geladen und eine Proxy-Klasse erzeugt. Diese Klasse kapselt alle Methoden des Webservice und verschickt bei einem Methodenaufruf automatisch SOAP Nachrichten.

Die Kapselung in einer Klasse darf aber nicht darüber hinwegtäuschen, dass ein Webservice an sich keinen Zustand besitzt. Um dies zu verdeutlichen ruft der Client zweimal die Methode **`increment()`** auf und gibt jeweils den aktuellen Zählerwert in einer MessageBox aus (siehe Abb.10). Wie bei einer zustandslosen Implementierung des Webservices zu erwarten ist, wird beide Mal der Wert 0 ausgegeben.

```
using System;
...

namespace PraktikumWinApp
{
    public class Form1 : System.Windows.Forms.Form
    {
      private System.Windows.Forms.Button button1;
      private WebReferenceNeu.SeminarService sWebS =
                    new WebReferenceNeu.SeminarService();
      ...

      private void button1_Click(object sender, System.EventArgs e)
      {
        sWebS.increment();
        MessageBox.Show("count=" + sWebS.countValue().ToString(),"Info");
        sWebS.increment();
        MessageBox.Show("count=" + sWebS.countValue().ToString(),"Info");
      }
   }
}
```

Abb. 10: Implementierung des Clients

3.4 Kommunikation zwischen Server und Client

Um die Kommunikation zwischen dem Webservice und die Client-Anwendung zu verdeutlichen, wurden die verschickten Nachrichten mitgeloggt. Da im Beispiel-Webservice keine Methoden mit Parametern benutzt wurden, wird die Kommunikation anhand einer weiteren Methode mit der Signatur **`public String getKundenNr( String`**

kundenName) durchgespielt. Der obere Teil der Nachricht (Abb. 10) besteht aus dem HTTP Header in seinem typischen Aufbau. Der Content-Type ist auf text/xml gesetzt.

```
POST /Praktikum/PraktikumService.asmx HTTP/1.1
VsDebuggerCausalityData: AwAAAMaH7L4RZ0FLgZQo1hqStKAAAAAAAAAA …
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; ...)
Content-Type: text/xml; charset=utf-8
SOAPAction: "http://Seminar_SS04/getKundenNr"
Content-Length: 334
Expect: 100-continue
Host: 192.168.73.128
```

```
<?xml version="1.0" encoding="utf-8"?>
   <soap:Envelope
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
       <soap:Body>
          <getKundenNr xmlns="http://Seminar_SS04/">
             <kundenName>MÃ¼ller</kundenName>
          </getKundenNr>
       </soap:Body>
   </soap:Envelope
```

Abb. 11: SOAP Nachricht für den Aufruf von getKundenNr()

Im unteren Teil wird eine SOAP Nachricht übertragen, die alle Informationen für den Methodenaufruf enthält. Die Methodeninformation wird als eigenes Element, welches an den Namespace des Webservices gebunden ist, im SOAP **body** Element übertragen. Der Parameter vom Typ String wird als Unterelement von der Methode eingebetet.

Die Antwort des Webservice (siehe Abb. 12), die den Rückgabewert der Methode enthält, ist analog zu der Anfrage aufgebaut. Im SOAP-Body ist nun ein Element mit dem Namen **getKundenNrResponse** enthalten, welches ebenfalls an den Namespace des Webservice gebunden ist. Dieses Element repräsentiert die Nachricht mit dem Rückgabewert. Der Wert selbst wird, wie schon der Parameter, als Unterelement eingebettet.

```
HTTP/1.1 100 Continue
Server: Microsoft-IIS/5.0
Date: Wed, 12 May 2004 16:45:13 GMT

HTTP/1.1 200 OK / Server: Microsoft-IIS/5.0 /Date: Wed, 12 May...
X-AspNet-Version: 1.1.4322 / VsDebuggerCausalityData:...
Cache-Control: private, max-age=0
Content-Type: text/xml; charset=utf-8
Content-Length: 370
```

```
<?xml version="1.0" encoding="utf-8"?>
   <soap:Envelope
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema">
       <soap:Body>
          <getKundenNrResponse xmlns="http://Seminar_SS04/">
             <getKundenNrResult>K50001</getKundenNrResult>
          </getKundenNrResponse>
       </soap:Body>
   </soap:Envelope
```

Abb. 12: SOAP Nachricht für den Rückgabewert von getKundenNr()

3.5 Die WSDL Datei

Die bisher betrachteten Elemente im SOAP-Body sind vom Webservice abhängig und können folglich nicht im SOAP Standart normiert werden. Deshalb werden diese Informationen in der WSDL Datei angegeben, welche den gesamten Webservice beschreibt. Diese beinhaltet den Aufbau der Elemente im SOAP-Body, falls SOAP benutzt wird, das verwendete Transportprotokoll, hier HTTP, alle angebotenen Methoden, eigene Datentypen, die URL des Webservices, den Namespace des Webservices und einen Beschreibungstext.

Auch wenn die gesamte WSDL Datei meist von der Entwicklungsumgebung erzeugt wird, werden nun Auszüge untersucht um die Funktionsweise zu verdeutlichen.

Zuerst werden mit XML-Schema[15] die Elemente des SOAP-Body definiert (siehe Abb.13). Das Namespacepräfix **`s`** ist an http://www.w3.org/2001/XMLSchema gebunden. Es wird das Element **`getKundenNr`**, **`getKundenNrResponse`** und deren Unterelemente definiert. Für beide wird der complexType von XML-Schema benutzt, da ein Unterelement definiert werden soll. Die Angabe von sequence wird benutzt um eine Liste von Unterelementen anzugeben. In diesem Fall nur das Element **`kundenName`**, vom Typ String. Es wird das

[15] Vgl. XML Schema [14]

programmiersprachenunabhängige Typensystem von XML-Schema benutzt. Das Attribut **minOccurs** gibt an wie oft das Element **kundenName** in dem Element **getKundenNr** mindestens vorkommen muss. Analog gibt **maxOccurs** an wie oft es maximal vorkommen darf.

```
<types>
  <s:schema ... targetNamespace="http://Seminar_SS04/">
    <s:element name="getKundenNr">
      <s:complexType>
        <s:sequence>
          <s:element minOccurs="0" maxOccurs="1" name="kundenName"
                                                 type="s:string" />
        </s:sequence>
      </s:complexType>
    </s:element>
    <s:element name="getKundenNrResponse"> ...
    </s:element>
  </s:schema>
</types>
```

Abb. 13: Typendeklaration in der WSDL Datei

Die beiden bisher definierten Elemente stehen aber noch in keinem Kontext. Es muss erst angegeben werden, dass sie in einer SOAP Nachricht verwendet werden sollen. Hierfür definieren wir zwei Nachrichten mit den Namen **getKundenNrSoapIn** und **getKundenNrSoapOut** (siehe Abb. 14). Das Namespacepräfix **s0** ist an den Namespace des Webservices gebunden, also an die Typendeklaration von weiter oben. Beide Nachrichten bestehen jeweils nur aus einem Element. Die eine aus dem Typ **getKundenNr** und die andere aus dem Typ **getKundenNrResponse**.

```
  <message name="getKundenNrSoapIn">
    <part name="parameters" element="s0:getKundenNr" />
  </message>
  <message name="getKundenNrSoapOut">
    <part name="parameters" element="s0:getKundenNrResponse" />
  </message>
```

Abb. 14: Nachrichtendeklaration in der WSDL Datei

Beide Nachrichten stehen aber noch in keinerlei Verhältnis. Es muss also noch angegeben werden, dass eine von beiden für den Methodenaufruf steht und die anderen für den Rückgabewert. Dies wird mit dem **portType** Element realisiert (siehe Abb. 15). In diesem Fall liegt eine so genannte Request−Response Kommunikation vor. Es gibt aber darüber hinaus noch weitere drei Kommunikationsarten, auf die hier aber nicht eingegangen

werden soll. Wir geben nun unsere beiden Nachrichten innerhalb des **portTypes** an. Die für den Methodenaufruf als input Message (=Request) und die für den Rückgabewert als output Message (=Response).

```
  <portType name="SeminarServiceSoapType">
    <operation name="getKundenNr">
      <input message="s0:getKundenNrSoapIn" />
      <output message="s0:getKundenNrSoapOut" />
    </operation>
  </portType>
```

Abb. 15: Vorgangsdeklaration in der WSDL Datei

Im **binding** Element wird nun einem **portType** ein Transportprotokoll zugewiesen. In diesem Fall HTTP (siehe Abb. 16).

```
<binding name="SeminarServiceSoapBinding"
         type="s0:SeminarServiceSoapType">
    <soap:binding
     transport="http://schemas.xmlsoap.org/soap/http" style="document" />
    <operation name="getKundenNr">
      <soap:operation soapAction="http://Seminar_SS04/getKundenNr"
                                  style="document" />
      ...
    </operation>
  </binding>
```

Abb. 16: Vorgangsdeklaration in der WSDL Datei

Im letzten Schritt wird nun der eigentliche Webservice definiert (siehe Abb. 17). Eine Textbeschreibung kann in das Element **documentation** eingefügt werden. Das eben definierte **binding** wird nun an die Adresse des Webservices gebunden. Weil die Client-Anwendung über den TCPMonitor, der auf dem gleichen Rechner läuft, kommuniziert, wird hier der localhost benutzt.

```
  <service name="SeminarService">
    <documentation>Seminar Webservices SS04</documentation>

    <port name="SeminarServiceSoap"
          binding="s0:SeminarServiceSoapBinding">
      <soap:address location="http://127.0.0.1/...Service.asmx" />
    </port>
  </service>
</definitions>
```

Abb. 17: Vorgangsdeklaration in der WSDL Datei

Die untersuchte WSDL Datei wurde von Visual Studio automatisch generiert. Leider ist an einigen Stellen die Benennung nicht ganz eindeutig gewesen. So wurde sowohl das **`binding`** als auch das **`porttype`** Element gleich benannt. Dies wurde um die Übersicht zu verbessern geändert.

Literaturverzeichnis

[1] **Apache Project**: *Apache Axis*, 2004 http://ws.apache.org/axis/ (14.07.2004).

[2] **Kuhlins, Stefan**: *Skript zur Vorlesung Verteilte Objekte*, Universität Mannheim, WS 2002/2003

[3] **Network Working Group**: *Hypertext Transfer Protocol - HTTP/1.1*; Network Working Group, RFC 2068, 1997, http://www.ietf.org/rfc/rfc2068.txt (14.07.2004).

[4] **Network Working Group**: *Transmission Control Protocol*; Network Working Group, RFC 793, 1981, http://www.faqs.org/rfcs/rfc793.html (14.07.2004).

[5] **OASIS**: *UDDI Specifications*; 2002-2004, http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm (14.07.2004).

[6] **OMG**: *CORBA - IIOP Specification v3.02*; Object Management Group, 2004, http://www.omg.org/technology/documents/formal/corba_iiop.htm (14.07.2004).

[7] **OMG**: *CORBA Component Model v3.0*; Object Management Group, 2004, http://www.omg.org/technology/documents/formal/components.htm (14.07.2004).

[8] **OMG**: *IDL to Java Language Mapping v1.2*; Object Management Group, 2002, http://www.omg.org/cgi-bin/doc?formal/02-08-05 (14.07.2004).

[9] **OMG**: *Catalog of OMG IDL / Language Mappings Specifications*; Object Management Group, http://www.omg.org/technology/documents/idl2x_spec_catalog.htm (14.07.2004)

[10] **Sun**: *RMI over IIOP*; Sun Microsystems, Documentation Home Page, 2002, http://java.sun.com/j2se/1.4.2/docs/guide/rmi-iiop/ (14.07.2004).

[11] Sun: *RMI Architecture and Functional Specification*; Sun Microsystems, 2003, http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html (14.07.2004).

[12] W3C: *Web Services Description Language (WSDL) 1.1*; World Wide Web Consortium, Note, 2001, http://www.w3.org/TR/wsdl (14.07.2004).

[13] W3C: *Extensible Markup Language (XML) 1.1*; World Wide Web Consortium, Recommendation, 2004, http://www.w3.org/TR/xml11/ (14.07.2004).

[14] W3C: *XML Schema*; World Wide Web Consortium, Recommendation, 2001, http://www.w3.org/TR/xmlschema-0/ (14.07.2004).

[15] W3C: *Simple Object Access Protocol (SOAP) 1.1*; World Wide Web Consortium, Note, 2000, http://www.w3.org/TR/SOAP (14.07.2004).

[16] W3C: *Web Services Architecture*; World Wide Web Consortium, Working Group Note, 2004, http://www.w3.org/TR/ws-arch/ (14.07.2004).