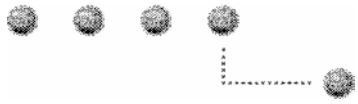


Berner Fachhochschule

Hochschule für
Technik und Architektur Bern

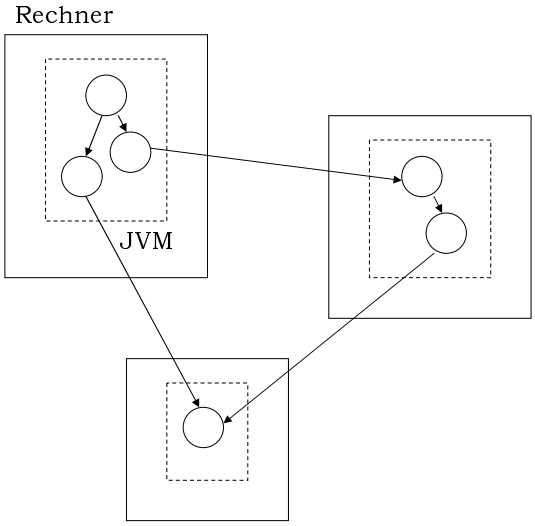
Verteilte Applikationen mit Java

Dr. Stephan Fischli

Winter 2003/2004

Einführung

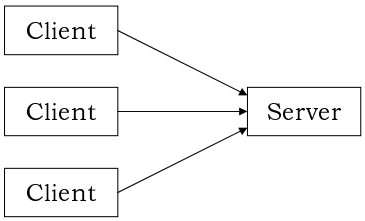
Verteilte Applikationen



Eine verteilte Applikation besteht aus mehreren Programmteilen, die auf verschiedenen Rechnern laufen und miteinander kommunizieren, um eine gemeinsame Aufgabe zu lösen.

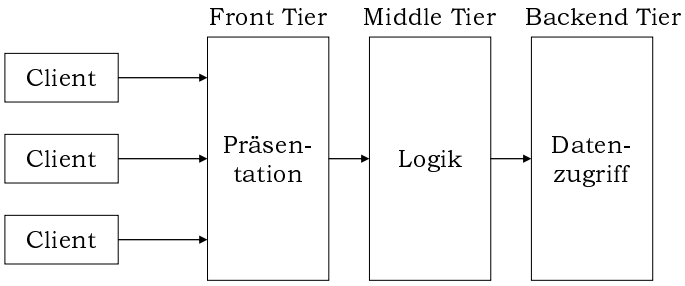
Die Entwicklung verteilter Applikationen ist wesentlich komplexer als diejenige lokaler Applikationen. So braucht es für die Kommunikation zusätzliche Technologien, und wegen der parallelen Ausführung müssen die Zugriffe auf gemeinsame Ressourcen synchronisiert werden.

Client/Server-Architektur



Bei der Client/Server-Architektur stellt ein Server seine Dienste zur Verfügung, welche von Clients in Anspruch genommen werden können. Die meisten klassischen Internet-Applikationen basieren auf dieser Architektur.

Multitier-Architektur

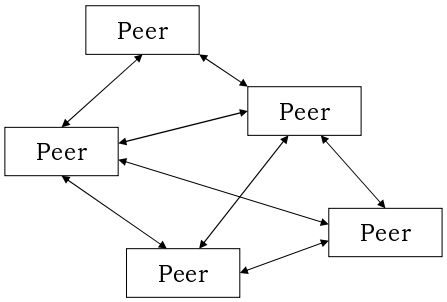


Die Multitier-Architektur ist eine Weiterentwicklung der Client/Server-Architektur, bei welcher die Server-Funktionalität auf mehrere Schichten (Tiers) verteilt wird:

- Der Front-Tier ist verantwortlich für die Darstellung von Daten und die Interaktion mit den Clients
- Der Middle-Tier enthält die eigentliche Applikationslogik
- Der Backend-Tier implementiert den Zugriff auf die Daten, die zum Beispiel in einer Datenbank enthalten sind

Grössere Web-Sites weisen oft eine Multitier-Architektur auf.

Peer-to-Peer Architektur



In einer Peer-to-Peer Architektur sind alle Rechner gleichberechtigt, es gibt keine ausgezeichneten Server oder Clients. Peer-to-Peer Architekturen eignen sich zum Beispiel für das Sharing oder den direkten Austausch von Ressourcen.

Unterstützung durch Java

- Threads
- Sockets
- Interfaces
- Reflection
- Objekt-Serialisierung
- Remote Method Invocation

Java unterstützt die Entwicklung verteilter Applikationen:

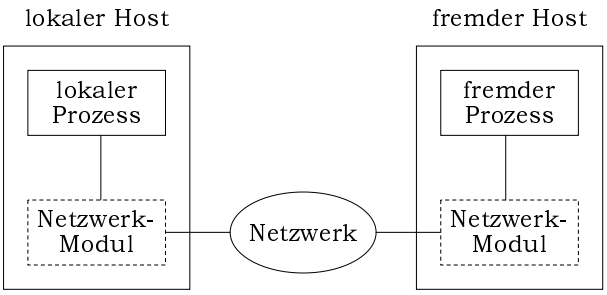
- Mit Threads können Programmteile parallel ausgeführt werden
- Sockets bilden eine einfache Schnittstelle zum Netzwerk
- Interfaces ermöglichen die Entkopplung von Programmteilen
- Reflection erlaubt das dynamische Laden von Programmteilen zur Laufzeit
- Mittels Objekt-Serialisierung können Objekte über das Netzwerk geschickt werden
- Remote Method Invocation realisiert die transparente Kommunikation entfernter Objekte

Literatur

- Marko Boger
  Java in verteilten Systemen
  dpunkt, 1999
- Jim Farley
  Java Distributed Computing
  O'Reilly, 1998
- Elliotte Rusty Harold
  Java Network Programming
  O'Reilly, 2000
- William Grosso
  Java RMI
  O'Reilly, 2001
- Richard Monson-Haefel, David Chappell
  Java Message Service
  O'Reilly, 2000

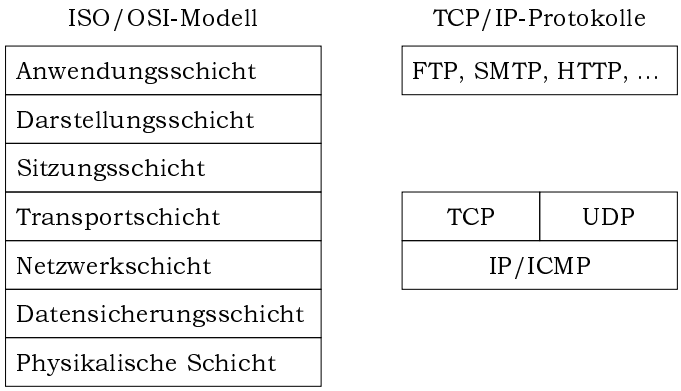
Netzwerk-Grundlagen

Verbindungsmodell



Eine Netzwerkverbindung kann durch das 5-Tupel (Protokoll, lokaler Host, lokaler Prozess, fremder Host, fremder Prozess) charakterisiert werden. Das Format der Host-Adressen und die Identifikation der kommunizierenden Prozesse wird durch das verwendete Protokoll definiert. In der TCP/IP-Protokollfamilie werden dafür Internet-Adressen und Portnummern verwendet.

TCP/IP-Protokolle



Die TCP/IP-Protokolle stellen offene Standards dar, die Hardware- und Betriebssystem-unabhängig sind, und werden in sogenannten „Requests for Comments“ (RFC) publiziert. Die wichtigsten Protokolle sind:

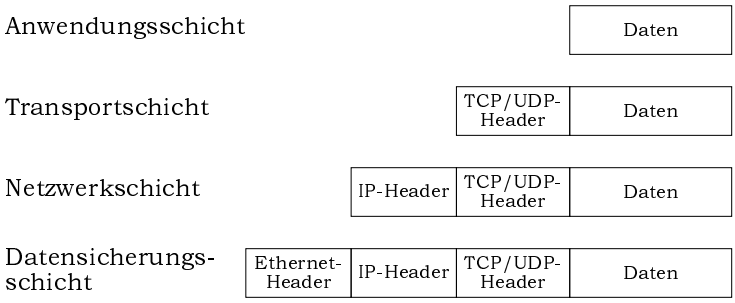
IP	Internet Protocol (RFC 791)
ICMP	Internet Control Message Protocol (RFC 792)
TCP	Transmission Control Protocol (RFC 793)
UDP	User Datagram Protocol (RFC 768)
FTP	File Transfer Protocol (RFC 959)
TELNET	Network Terminal Protocol (RFC 854)
SMTP	Simple Mail Transfer Protocol (RFC 821)
HTTP	Hypertext Transfer Protocol (RFC1945)

Eigenschaften der TCP/IP-Protokolle

	TCP	UDP	IP
verbindungsorientiert	ja	nein	nein
meldungsorientiert	nein	ja	ja
zuverlässig	ja	nein	nein
Flusskontrolle	ja	nein	nein

- Bei einem verbindungsorientierten Protokoll bauen die beiden Kommunikationspartner eine Verbindung auf, die während dem Datenaustausch bestehen bleibt und danach wieder abgebaut wird.
- Bei einem meldungsorientierten Protokoll werden die Meldungsgrenzen beibehalten, die der Sender festgelegt hat.
- Ein zuverlässiges Protokoll garantiert, dass die Daten fehlerfrei und in derselben Reihenfolge beim Empfänger ankommen, wie sie der Sender geschickt hat.
- Eine Flusskontrolle stellt sicher, dass der Sender die Daten nicht schneller schickt, als sie der Empfänger verarbeiten kann.

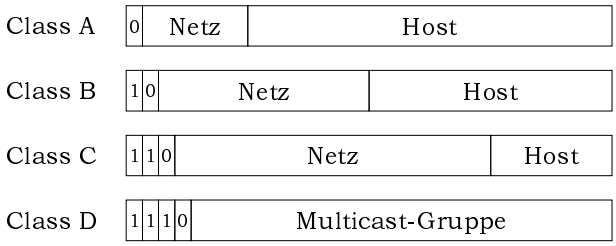
Rahmenbildung



In jeder Schicht des ISO/OSI-Modells fügt der Sender den Daten einen Header mit Kontrollinformation hinzu, der beim Empfänger wieder entfernt wird.

Der IP-Header enthält u.a. die Internet-Adressen des Sender- und Empfänger-Hosts sowie einen Protokoll-Identifier der Transportschicht. Die TCP- und UDP-Header enthalten die Portnummern der beiden kommunizierenden Prozesse.

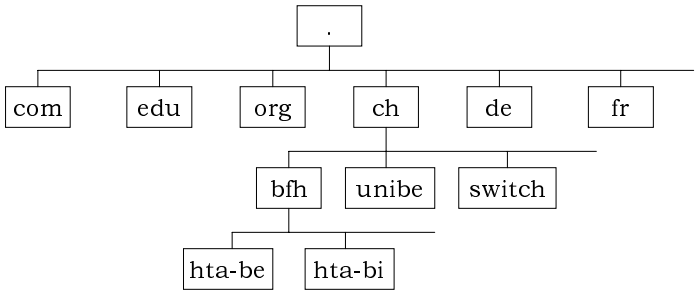
Internet-Adressen



Jeder Host im Internet hat eine 32 Bit-Internet-Adresse, die aus einem Netzwerk-Identifier und einem dazu relativen Host-Identifier besteht. Die Länge dieser beiden Identifier hängt von der Klasse der Adresse ab, die durch die ersten Bits bestimmt wird.

Die vier Bytes einer Internet-Adresse werden üblicherweise als Dezimalzahlen geschrieben und durch einen Punkt voneinander getrennt, z.B. 130.59.10.30

Domain Name System (DNS)



Fast jeder Host im Internet hat neben seiner Internet-Adresse auch einen Namen wie nic.switch.ch. Diese Namen sind Teil des Domain Name System, das hierarchisch aufgebaut ist und aus einem Root Domain, verschiedenen Toplevel Domains, Domains zweiter Stufe usw. besteht. Für die Umsetzung der Hostnamen in die entsprechenden Internet-Adressen gibt es spezielle Name Server, die jeweils für einen Domain zuständig sind.

Portnummern

Bereich	Kategorie
1 - 255	reservierte Portnummern
1 - 1023	privilegierte Portnummern
1024 - 5000	kurzlebige Portnummern
5001 - 65535	benutzerdefinierte Portnummern

Portnummern sind 16 Bit-Dezimalzahlen, die von TCP und UDP zur Identifikation der kommunizierenden Prozesse verwendet werden.

- Reservierte Portnummern sind „wohlbekannte“ Portnummern, die für Standard-Anwendungen wie FTP (21), TELNET (23), SMTP (25), HTTP (80) usw. reserviert sind.
- Privilegierte Portnummern können nur von privilegierten Prozessen verwendet werden.
- Kurzlebige Portnummern werden vom System automatisch an Client-Prozesse vergeben, die nicht auf eine bestimmte Portnummer angewiesen sind.
- Benutzerdefinierte Portnummern können von nicht-privilegierten Server-Prozessen beansprucht werden, die eine feste Portnummer benötigen.

Klasse InetAddress

InetAddress	
InetAddress	getByName(String host)
InetAddress[]	getAllByName(String host)
InetAddress	getLocalHost()
String	getHostName()
String	getHostAddress()
byte[]	getAddress()

Die Klasse InetAddress repräsentiert Internet-Adressen und ist verantwortlich für die Umwandlung von Hostnamen in Internet-Adressen und umgekehrt.

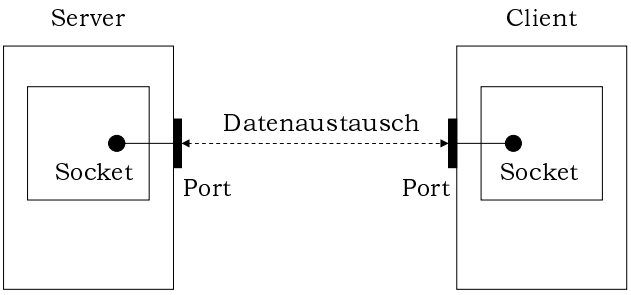
DNS-Lookup

```
import java.net.InetAddress;

public class DNSLookup {
    public static void main(String[] args) throws Exception {
        InetAddress[] addrs = InetAddress.getAllByName(args[0]);
        for (int i = 0; i < addrs.length; i++) {
            if (Character.isDigit(args[0].charAt(0)))
                System.out.println(addrs[i].getHostName());
            else
                System.out.println(addrs[i].getHostAddress());
        }
    }
}
```

Das Programm DNSLookup wandelt eine als Argument übergebene Internet-Adresse in Hostnamen um oder umgekehrt.

Socket-Programmierung

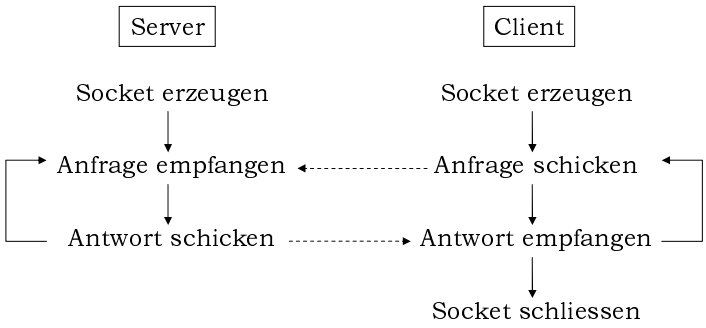
Verbindungslose Kommunikation



Sockets bilden die Programmierschnittstelle zum Netzwerk und stellen die logischen Endpunkte einer Netzwerkverbindung dar.

Bei einer verbindungslosen Kommunikation erzeugt der Server einen Socket, über den er die Anfragen der Clients empfängt, diese verarbeitet und die Antworten zurückschickt. Der Server kommuniziert also mit verschiedenen Clients über denselben Socket.

Ablauf



Ein verbindungsloser Server wird meistens iterativ implementiert, das heisst, er verarbeitet eine Anfrage nach der andern.

Klasse DatagramSocket

DatagramSocket	
DatagramSocket()	
DatagramSocket(int port)	
InetAddress	getLocalAddress()
int	getLocalPort()
void	send(DatagramPacket dp)
void	receive(DatagramPacket dp)
void	close()

Ein DatagramSocket wird verwendet, um UDP-Datengramme zu verschicken und zu empfangen. Für einen Server-Prozess kann ein DatagramSocket mit einem festen Port verknüpft werden.

Klasse DatagramPacket

DatagramPacket	
DatagramPacket(byte[] buf, int len)	
DatagramPacket(byte[] buf, int len, InetAddress addr, int port)	
InetAddress	getAddress()
int	getPort()
byte[]	getData()
int	getLength()

Ein DatagramPacket repräsentiert ein UDP-Datengramm. Neben den eigentlichen Daten enthält ein DatagramPacket die Empfänger- oder Absenderadresse.

Verbindungsloser Echo-Server

```
import java.net.*;

public class UDPEchoServer {
   public static void main(String[] args) throws Exception {

      // create socket
      int port = Integer.parseInt(args[0]);
      DatagramSocket socket = new DatagramSocket(port);
      System.out.println("Echo Server ready");

      while (true) {
         // receive message
         byte[] buffer = new byte[1024];
         DatagramPacket message =
            new DatagramPacket(buffer, buffer.length);
         socket.receive(message);

         // print message
         String client = message.getAddress().getHostName();
         String line =
            new String(message.getData(), 0, message.getLength());
         System.out.println("Message from " + client + ": " + line);
```

Das Programm UDPEchoServer implementiert einen Server, der über einen Datagram-Socket Meldungen empfängt und diese an die entsprechenden Clients zurückschickt.

Verbindungsloser Echo-Server (ff.)

```
                // send message back
                socket.send(message);
            }
        }
    }
```

Verbindungsloser Echo-Client

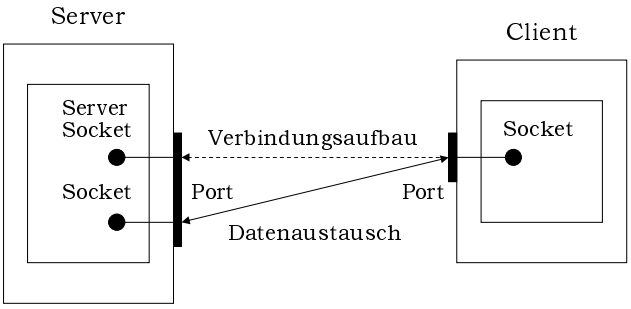
```
import java.io.*;
import java.net.*;

public class UDPEchoClient {
   public static void main(String[] args) throws Exception {

      // create socket
      InetAddress server = InetAddress.getByName(args[0]);
      int port = Integer.parseInt(args[1]);
      DatagramSocket socket = new DatagramSocket();

      BufferedReader stdin =
         new BufferedReader(new InputStreamReader(System.in));
      while (true) {
         // send message
         String line = stdin.readLine();
         if (line.equals(".")) break;
         DatagramPacket message = new DatagramPacket(
            line.getBytes(), line.length(), server, port);
         socket.send(message);
```

Das Programm UDPEchoClient liest Meldungen ein, schickt diese über einen Datagram-Socket an einen Server und gibt die vom Server erhaltenen Antworten wieder aus.

Verbindungsloser Echo-Client (ff.)

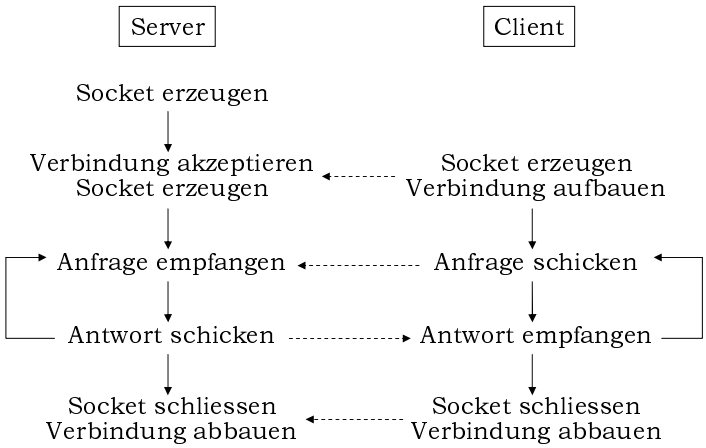
```
            // receive response
            byte[] buffer = new byte[1024];
            message = new DatagramPacket(buffer, buffer.length);
            socket.receive(message);
            line = new String(message.getData(), 0, message.getLength());
            System.out.println(line);
        }
        socket.close();
    }
}
```

Verbindungsorientierte Kommunikation



Bei einer verbindungsorientierten Kommunikation erzeugt der Server einen Socket, auf dem er Verbindungsanforderungen von Clients akzeptiert. Für jede erfolgte Verbindung erhält er einen neuen Socket, der mit dem entsprechenden Client verbunden ist.

Ablauf



Ein verbindungsorientierter Server wird meistens parallel implementiert, das heisst, er erzeugt für jede Verbindung einen Thread, der die Kommunikation mit dem entsprechenden Client übernimmt.

Klasse ServerSocket

ServerSocket	
ServerSocket(int port)	
Socket	accept()
int	getLocalPort()
InetAddress	getInetAddress()
void	close()

Ein ServerSocket wird von einem Server verwendet, um auf seinem wohlbekannten Port Verbindungsanforderungen von Clients zu akzeptieren.

Klasse Socket

Socket	
Socket(String host, int port)	
Socket(InetAddress addr, int port)	
InetAddress	getLocalAddress()
int	getLocalPort()
InetAddress	getInetAddress()
int	getPort()
InputStream	getInputStream()
OutputStream	getOutputStream()
void	close()

Mit einem Socket kann einerseits ein Client eine Verbindung zu einem Server aufbauen. Andererseits erhält nach dem Verbindungsaufbau der Server einen Socket, welcher mit dem Client verbunden ist. Client und Server können dann Ein- und Ausgabestreams erzeugen, um über ihren Socket Daten auszutauschen.

Verbindungsorientierter Echo-Server

```
import java.io.*;
import java.net.*;

public class TCPEchoServer {
   public static void main(String[] args) throws Exception {

      // create server socket
      int port = Integer.parseInt(args[0]);
      ServerSocket serverSocket = new ServerSocket(port);
      System.out.println("Echo Server ready");

      while (true) {
         // accept connection
         Socket socket = serverSocket.accept();
         String client = socket.getInetAddress().getHostName();
         System.out.println("Connection by " + client);

         // start thread
         ServerThread thread = new ServerThread(socket);
         thread.start();
      }
   }
}
```

Das Programm TCPEchoServer implementiert einen Server, der einen ServerSocket erzeugt, auf diesem Client-Verbindungen akzeptiert und nach erfolgter Verbindung einen ServerThread startet.

Verbindungsorientierter Echo-Server (ff.)

```
class ServerThread extends Thread {
   private Socket socket;

   public ServerThread(Socket socket) {
      this.socket = socket;
   }
   public void run() {
      try {
         // create socket streams
         BufferedReader sockin = new BufferedReader(
               new InputStreamReader(socket.getInputStream()));
         PrintWriter sockout =
            new PrintWriter(socket.getOutputStream(), true);

         while (true) {
            // receive message
            String message = sockin.readLine();
            if (message == null) break;

            // send message back
            sockout.println(message);
         }
```

Die Klasse ServerThread implementiert einen Thread, der über einen Socket Meldungen empfängt und diese an den Client zurückschickt.

Verbindungsorientierter Echo-Server (ff.)

```
                socket.close();
                System.out.println("Connection closed");
            }
            catch (IOException e) {
                System.err.println(e);
            }
        }
    }
```

Verbindungsorientierter Echo-Client

```
import java.io.*;
import java.net.*;

public class TCPEchoClient {
   public static void main(String[] args) throws Exception {

      // create socket
      InetAddress server = InetAddress.getByName(args[0]);
      int port = Integer.parseInt(args[1]);
      Socket socket = new Socket(server, port);

      // create socket streams
      BufferedReader sockin = new BufferedReader(
         new InputStreamReader(socket.getInputStream()));
      PrintWriter sockout =
         new PrintWriter(socket.getOutputStream(), true);

      BufferedReader stdin =
         new BufferedReader(new InputStreamReader(System.in));
```

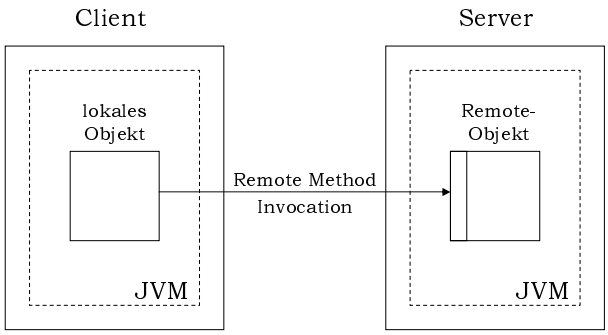
Das Programm TCPEchoClient baut über einen Socket eine Verbindung zu einem Server auf. Danach liest es Meldungen ein, schickt diese an den Server und gibt die vom Server erhaltenen Antworten wieder aus.

Verbindungsorientierter Echo-Client (ff.)

```
        while (true) {
            // send message
            String message = stdin.readLine();
            if (message.equals(".")) break;
            sockout.println(message);

            // receive response
            message = sockin.readLine();
            System.out.println(message);
        }
        socket.close();
    }
}
```

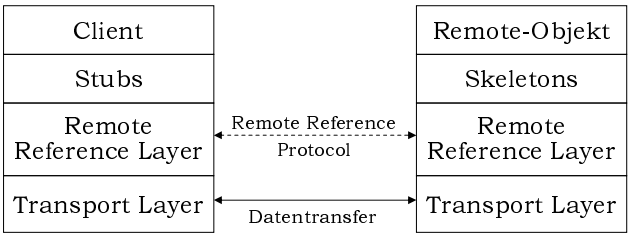
Remote Method Invocation

Einführung



Remote Method Invocation (RMI) ist der Aufruf einer Methode eines Remote-Objekts. Ein Remote-Objekt ist dabei ein Objekt, dessen Methoden auch von andern virtuellen Java-Maschinen aus aufgerufen werden können als nur von derjenigen, in der das Objekt lebt. Welche Methoden dies sind, wird in einem oder mehreren Remote-Interfaces festgehalten, die das Remote-Objekt implementiert.

Architektur

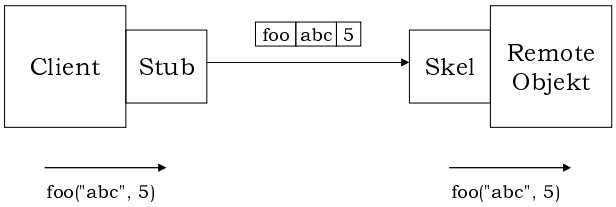


Ein Stub ist ein Objekt, das auf der Client-Seite stellvertretend für das Remote-Objekt steht. Es implementiert dieselben Interfaces wie das Remote-Objekt und ist somit typkompatibel zu diesem. Ruft der Client eine Methode des Remote-Objekts auf, so gibt der Stub den Aufruf an das Skelett auf der Server-Seite weiter, das dann die Methode des Remote-Objekts aufruft.

Im Remote Reference Layer werden die lokalen Referenzen auf die Stubs in Referenzen auf die entsprechenden Remote-Objekte übersetzt.

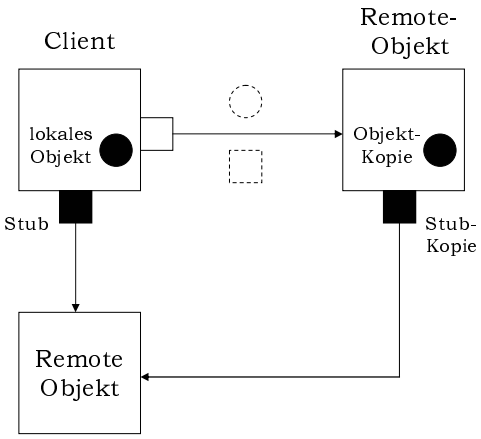
Der Transport Layer ist zuständig für die Erstellung und Aufrechterhaltung der Datenverbindung.

Marshaling und Unmarshaling



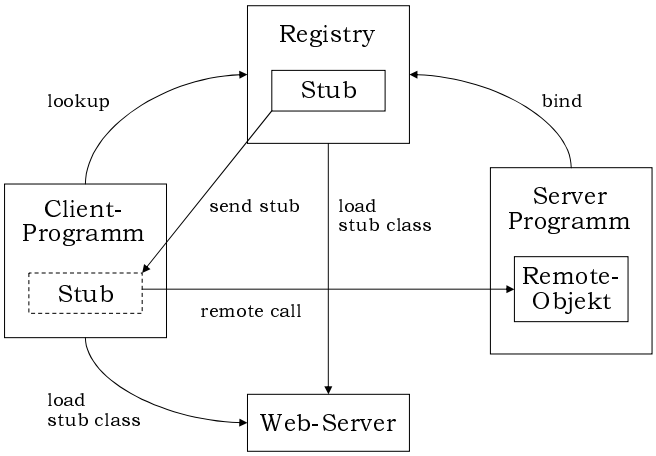
Die Stubs und Skeletons sind auch zuständig für das sogenannte Marshaling und Unmarshaling von Parametern, also für die Umwandlung von Argumenten und Returnwerten einer Remote-Methode in Byte-Streams und umgekehrt.

Objektparameter



Wird einer Remote-Methode ein lokales Objekt als Parameter übergeben, so wird dieses serialisiert und der Empfänger des Aufrufs erhält eine Kopie des Objekts (Wertübergabe). Bei einem Remote-Objekt als Parameter wird hingegen nur der Stub übertragen, mit welchem der Empfänger dann auf das entsprechende Objekt zugreifen kann (Referenzübergabe).

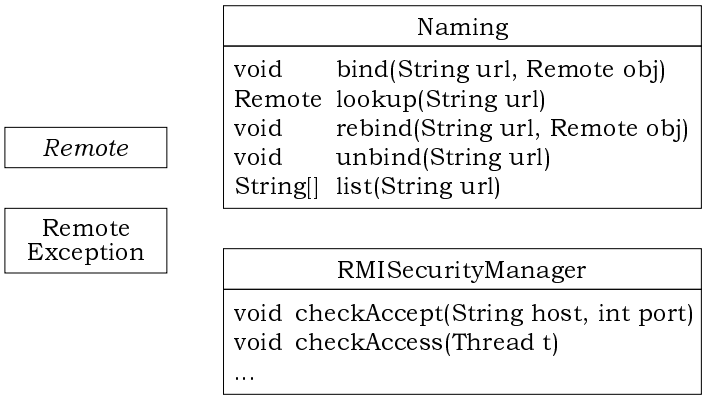
Kommunikationsablauf



Um auf ein Remote-Objekt zugreifen zu können, benötigt ein Client den entsprechenden Stub. Diesen erhält er entweder als Returnwert eines gewöhnlichen Methodenaufrufs oder als Resultat einer Abfrage in der Registry, in welcher der Server seine Remote-Objekte eingetragen hat.

Zudem ist es möglich, dass ein Client den Code der Stubklasse zur Laufzeit von einem Webserver herunterlädt. Weil dies ein potentielles Sicherheitsrisiko darstellt, muss der Client vorgängig einen Security-Manager installieren.

Package java.rmi



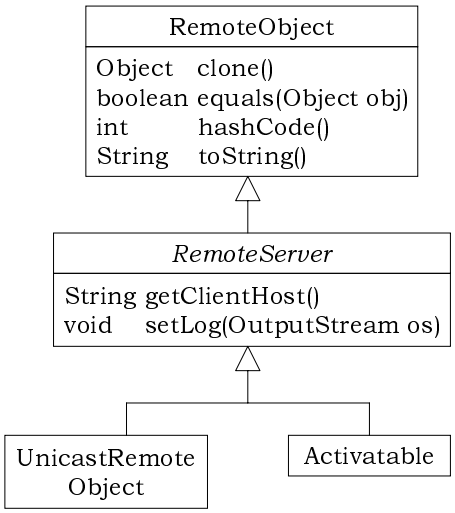
Das Interface Remote definiert keine Methoden, sondern kennzeichnet lediglich Objekte als Remote-Objekte.

Die Klasse RemoteException ist die Basisklasse einer Reihe von Exception-Klassen, die beim Aufruf von Remote-Objekten geworfen werden können.

Die Klasse Naming ist eine einfache Registry, die URLs auf Remote-Objekte abbildet. Ein URL eines Remote-Objekts hat dabei die Form rmi://host:port/name. Fehlt der Hostname oder die Portnummer, so wird als Default der lokale Host bzw. die Portnummer 1099 verwendet.

Die Klasse RMISecurityManager implementiert einen Security-Manager, der nichts erlaubt ausser der Definition von Klassen und den Zugriff auf diese.

Package java.rmi.server



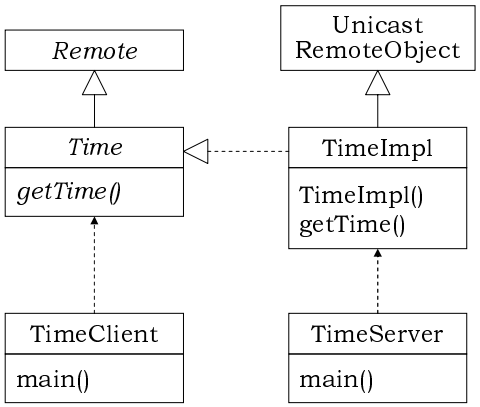
Die Klasse RemoteObject implementiert das Verhalten der Klasse Object für Remote-Objekte.

Die Klasse RemoteServer ist eine abstrakte Basisklasse für Server-Implementationen. Sie definiert u.a. Methoden, um Remote-Objekte zu erzeugen und zu exportieren.

Die Klasse UnicastRemoteObject implementiert ein Remote-Objekt, das ausschliesslich auf einem Host lebt, für die Kommunikation TCP-Sockets verwendet und dessen Remote-Referenzen nur solange gültig sind, wie der Serverprozess läuft.

Die Klasse Activatable implementiert ein Remote-Objekt, dessen Remote-Referenz gültig bleibt, auch wenn der zugehörige Serverprozess vorübergehend nicht läuft.

Beispiel: Time-Server



Bei der Implementation einer RMI-Applikation müssen folgende Schritte ausgeführt werden:

1. Definieren des Remote-Interfaces als Schnittstelle zwischen Client und Server
2. Implementieren des Remote-Objekts und des Server-Programms
3. Implementieren des Client-Programms
4. Kompilieren des Server-Programms und Erzeugen der Stubs und Skeletons
5. Starten der Registry und des Servers
6. Kompilieren des Client-Programms und Starten des Clients

Definieren des Remote-Interfaces

```
import java.rmi.*;

public interface Time extends Remote {
    String getTime() throws RemoteException;
}
```

Das Remote-Interface definiert diejenigen Methoden eines Remote-Objekts, die von Clients in andern virtuellen Java-Maschinen aufgerufen werden können.

Ein Remote-Interface wird vom Interface Remote abgeleitet. Alle Methoden müssen deklarieren, dass sie eine RemoteException werfen, die vom RMI-System ausgelöst wird, wenn ein Netzwerk- oder Protokoll-Fehler auftritt.

Implementieren des Remote-Objekts

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class TimeImpl extends UnicastRemoteObject implements Time {
    public TimeImpl() throws RemoteException {
        super();
    }
    public String getTime() throws RemoteException {
        return new Date().toString();
    }
}
```

Das Remote-Objekt implementiert die Methoden des entsprechenden Remote-Interfaces. Es kann zusätzliche Methoden implementieren, die aber nur lokal aufgerufen werden können. Da der Konstruktor eines Remote-Objekts eine RemoteException werfen kann, muss er implementiert werden, auch wenn er leer ist.

Implementieren des Server-Programms

```
import java.rmi.Naming;
import java.rmi.server.RemoteServer;

public class TimeServer {
    public static void main(String[] args) throws Exception {
        TimeImpl time = new TimeImpl();
        RemoteServer.setLog(System.out);
        String url = "rmi://localhost:" + args[0] + "/SimpleTime";
        Naming.rebind(url, time);
    }
}
```

Das Server-Programm erzeugt ein oder mehrere Remote-Objekte und trägt sie in der lokalen Registry ein. Zudem können die Remote-Aufrufe geloggt werden.

Implementieren des Client-Programms

```
import java.rmi.Naming;

public class TimeClient {
    public static void main(String[] args) throws Exception {
        String url = "rmi://" + args[0] + ":" + args[1] + "/SimpleTime";
        Time time = (Time)Naming.lookup(url);
        System.out.println(time.getTime());
    }
}
```

Das Client-Programm fragt die benötigten Remote-Objekte in der Registry des Servers ab. Anschliessend können sie wie lokale Objekte verwendet werden.

Kompilieren und Starten der Programme

Kompilieren des Servers und Erzeugen der Stubs und Skeletons:

```
> javac TimeImpl.java TimeServer.java
> rmic TimeImpl
```

Starten der Registry und des Servers:

```
> rmiregistry <port>
> java TimeServer <port>
```

Kompilieren und Starten des Clients:

```
> javac TimeClient.java
> java TimeClient <serverhost> <port>
```

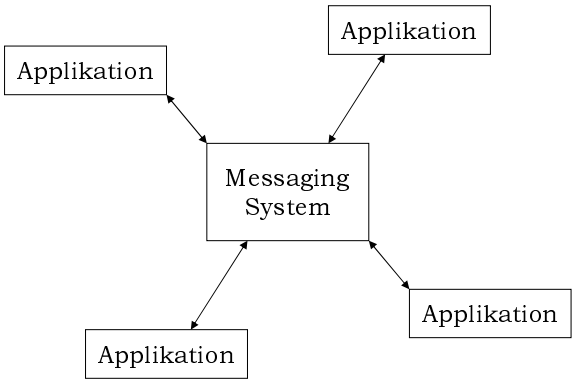
Der RMI-Compiler erzeugt aus dem Bytecode des Remote-Objekts den Stub und das Skeleton. Diese haben denselben Namen wie das Remote-Objekt ergänzt um die Endungen _Stub bzw. _Skel.

Die Registry kann auf einem bestimmten Port gestartet werden, Default ist 1099.

Sowohl die Registry als auch der Client müssen die Interface- und Stub-Klasse des Remote-Objekts in ihrem Klassenpfad haben, oder es wird beim Starten des Servers zusätzlich ein URL angegeben, von welchem diese heruntergeladen werden können.

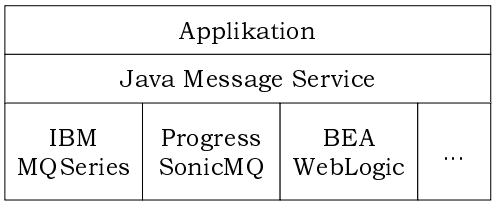
Java Message Service

Messaging-Systeme



Applikationen oder Software-Komponenten können miteinander kommunizieren, indem sie über ein Messaging-System Meldungen austauschen. Solche Meldungen werden asynchron ausgeliefert, sodass ein Absender nicht warten muss, bis sie beim Empfänger ankommen und von diesem verarbeitet werden. Da ein Absender seine Meldungen an eine virtuelle Destination schickt, muss er die Empfänger nicht einmal kennen.

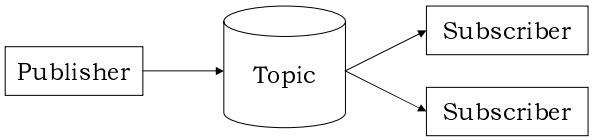
Java Message Service



Der Java Message Service (JMS) definiert eine standardisierte Programmierschnittstelle für Messaging-Systeme verschiedener Hersteller. Dabei werden folgende Begriffe verwendet:

- Ein JMS-Provider ist ein Messaging-System, welches für das Routing und die Auslieferung von Meldungen zuständig ist
- Ein JMS-Client ist eine Java-Applikation, welche mittels JMS auf ein Messaging-System zugreift
- Eine JMS-Applikation ist ein Software-System, welches aus mehreren JMS-Clients und einem JMS-Provider besteht
- Ein JMS-Produzent ist ein JMS-Client, welcher Meldungen schickt
- Ein JMS-Konsument ist ein JMS-Client, welcher Meldungen empfängt

Publish/Subscribe-Modell

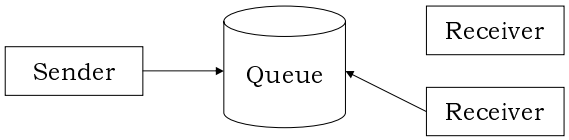


Beim Publish/Subscribe-Modell schickt ein Produzent (Publisher) seine Meldungen an ein Topic, von wo sie an alle registrierten Konsumenten (Subscriber) ausgeliefert werden.

Das Publish/Subscribe-Modell hat folgende Eigenschaften:

- Jede Meldung kann mehrere Konsumenten haben, die Meldungen werden also vervielfacht
- Die Meldungen werden über einen Callback-Mechanismus automatisch an die Konsumenten ausgeliefert (Push-Modell)
- Es besteht eine zeitliche Abhängigkeit zwischen Produzent und Konsumenten, da die Meldungen nur an die momentan registrierten Konsumenten ausgeliefert und nicht gespeichert werden (ausser bei einer dauerhaften Registrierung)

Point-to-Point-Modell

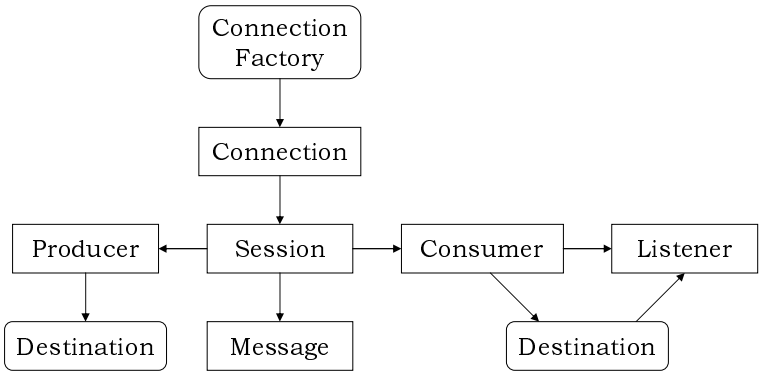


Beim Point-to-Point-Modell schickt ein Produzent (Sender) seine Meldungen an eine Queue, von wo sie die Konsumenten (Receiver) abholen können.

Das Point-to-Point-Modell hat folgende Eigenschaften:

- Jede Meldung hat genau einen Konsumenten
- Ein Konsument holt normalerweise eine Meldung aktiv von der Queue ab (Pull-Modell), kann sie sich aber auch automatisch zustellen lassen
- Es besteht keine zeitliche Abhängigkeit zwischen Produzent und Konsumenten, da die Meldungen in der Queue gespeichert bleiben, bis sie von einem Konsumenten abgeholt werden

Programmiermodell

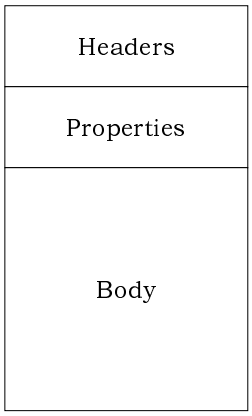


Das JMS-Programmiermodell besteht aus folgenden Komponenten:

- Eine Connection-Factory wird von einem Client verwendet, um eine Verbindung zum entsprechenden Provider zu erzeugen
- Eine Connection kapselt eine virtuelle Verbindung zum Provider
- Eine Session bildet einen single-threaded Kontext, um Meldungen zu schicken und zu empfangen
- Ein Producer kann Meldungen schicken, ein Consumer Meldungen empfangen
- Ein Listener ist ein asynchroner Event-Handler für Meldungen
- Eine Destination ist das Ziel, an welches ein Producer seine Meldungen schickt bzw. die Quelle, von welcher ein Consumer Meldungen empfängt

Connection-Factories und Destinations sind administrierte Objekte. Auf sie greift ein Client über das Java Naming and Directory Interface (JNDI) zu.

Aufbau einer Meldung



Eine JMS-Meldung besteht aus drei Teilen:

- Die Headers enthalten Identifikations- und Routing-Information, insbesondere die Destination der Meldung
- Im Properties-Teil können Applikations- oder Provider-spezifische Informationen abgelegt werden, zum Beispiel zur Filterung von Meldungen
- Der Body enthält die eigentlichen Nutzdaten der Meldung, wobei verschiedene Typen möglich sind

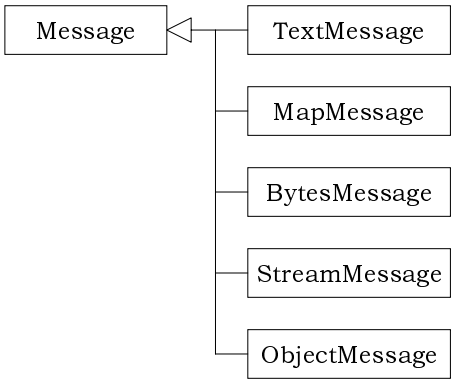
Headers

- JMSDestination
- JMSDeliveryMode
- JMSMessageID
- JMSTimestamp
- JMSExpiration
- JMSRedelivered
- JMSPriority
- JMSReplyTo
- JMSCorrelationID
- JMSType

Die meisten Header werden vom JMS-Provider automatisch zugeordnet, andere müssen explizit gesetzt werden:

- JMSDestination identifiziert die Destination (Topic oder Queue) der Meldung
- JMSDeliveryMode gibt die Art der Auslieferung an (persistent oder nicht-persistent)
- JMSMessageID identifiziert die Meldung eindeutig
- JMSTimestamp enthält den Zeitpunkt, als die Meldung geschickt wurde
- JMSExpiration definiert die Zeitdauer, nach welcher die Meldung nicht mehr ausgeliefert wird
- JMSRedelivered zeigt an, dass die Meldung bereits einmal ausgeliefert wurde
- JMSPriority enthält die Priorität der Meldung
- JMSReplyTo bezeichnet die Destination, an welche Antworten geschickt werden sollen
- JMSCorrelationID stellt eine Verknüpfung mit einer früheren Meldung her
- JMSType charakterisiert die Struktur oder den Inhalt der Meldung

Meldungstypen



Es gibt fünf Typen von JMS-Meldungen:

- Eine TextMessage besteht aus einem String, zum Beispiel einem XML-Dokument
- Eine MapMessage besteht aus Name-Wert-Paaren, wobei die Namen Strings und die Werte primitive Typen sind
- Eine BytesMessage besteht aus einem nicht-interpretierten Byte-Strom
- Eine StreamMessage besteht aus einem Strom primitiver Datentypen, die sequentiell geschrieben und gelesen werden
- Eine ObjectMessage besteht aus einem serialisierbaren Objekt

Einfacher Publisher

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class SimplePublisher {
   public static void main(String[] args) throws Exception {

      // lookup connection factory and topic
      Context context = new InitialContext();
      TopicConnectionFactory factory = (TopicConnectionFactory)
         context.lookup("TopicConnectionFactory");
      Topic topic = (Topic)context.lookup(args[0]);

      // create connection, session and publisher
      TopicConnection connection = factory.createTopicConnection();
      TopicSession session = connection.createTopicSession(
         false, Session.AUTO_ACKNOWLEDGE);
      TopicPublisher publisher = session.createPublisher(topic);
```

Das Programm SimplePublisher implementiert einen JMS-Client, welcher Meldungen einliest und diese an ein Topic schickt.

Einfacher Publisher (ff.)

```
        BufferedReader in =
            new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            // publish message
            String text = in.readLine();
            if (text.equals(".")) break;
            TextMessage message = session.createTextMessage();
            message.setText(text);
            publisher.publish(message);
        }
        connection.close();
    }
}
```

Einfacher Subscriber

```
import javax.jms.*;
import javax.naming.*;

public class SimpleSubscriber {
   public static void main(String[] args) throws Exception {

      // lookup connection factory and topic
      Context context = new InitialContext();
      TopicConnectionFactory factory = (TopicConnectionFactory)
         context.lookup("TopicConnectionFactory");
      Topic topic = (Topic)context.lookup(args[0]);

      // create connection, session and subscriber
      TopicConnection connection = factory.createTopicConnection();
      TopicSession session = connection.createTopicSession(
         false, Session.AUTO_ACKNOWLEDGE);
      TopicSubscriber subscriber = session.createSubscriber(topic);

      // receive messages
      subscriber.setMessageListener(new TextListener());
      connection.start();
   }
}
```

Das Programm SimpleSubscriber implementiert einen JMS-Client, welcher asynchron Meldungen von einem Topic empfängt und diese ausgibt.

Einfacher Subscriber (ff.)

```
class TextListener implements MessageListener {
   public void onMessage(Message message) {
      try {
         // print message
         String text = ((TextMessage)message).getText();
         System.out.println(text);
      }
      catch (Exception e) {}
   }
}
```

Die Klasse TextListener implementiert einen Message-Listener, welcher eine Callback-Methode zum Empfangen von Meldungen zur Verfügung stellt.

Starten der Applikation

Kompilieren der Clients:

```
> set CLASSPATH=<j2eehome>\lib\j2ee.jar;.
> javac SimplePublisher.java SimpleSubscriber.java
```

Erzeugen des Topic und Starten des JMS-Providers:

```
> set PATH=<j2eehome>\bin
> j2eeadmin –addJmsDestination <topicname> topic
> j2ee –verbose
```

Starten der Clients:

```
> java –Dorg.omg.CORBA.ORBInitialHost=<serverhost>
      –Dorg.omg.CORBA.ORBInitialPort=<serverport>
      SimplePublisher/SimpleSubscriber <topicame>
```

Als JMS-Provider kann der Applikationsserver der J2EE-Referenzimplementation von Sun verwendet werden. Der Port, auf welchem der Server läuft, ist im Konfigurationsfile orb.properties defaultmässig auf 1050 festgelegt.

Beim Starten der Clients können der Rechner und der Port des Servers als Properties mitgegeben werden. Default sind der lokale Host und die Portnummer 1050.

Einfacher Sender

```
import java.io.*;
import javax.jms.*;
import javax.naming.*;

public class SimpleSender {
   public static void main(String[] args) throws Exception {

      // lookup connection factory and queue
      Context context = new InitialContext();
      QueueConnectionFactory factory = (QueueConnectionFactory)
         context.lookup("QueueConnectionFactory");
      Queue queue = (Queue)context.lookup(args[0]);

      // create connection, session and sender
      QueueConnection connection = factory.createQueueConnection();
      QueueSession session = connection.createQueueSession(
         false, Session.AUTO_ACKNOWLEDGE);
      QueueSender sender = session.createSender(queue);
```

Das Programm SimpleSender implementiert einen JMS-Client, welcher Meldungen einliest und diese an eine Queue schickt.

Einfacher Sender (ff.)

```
        BufferedReader in =
            new BufferedReader(new InputStreamReader(System.in));
        while (true) {
            // send message
            String text = in.readLine();
            if (text.equals(".")) break;
            TextMessage message = session.createTextMessage();
            message.setText(text);
            sender.send(message);
        }
        connection.close();
    }
}
```

Einfacher Receiver

```
import javax.jms.*;
import javax.naming.*;

public class SimpleReceiver {
   public static void main(String[] args) throws Exception {

      // lookup connection factory and queue
      Context context = new InitialContext();
      QueueConnectionFactory factory = (QueueConnectionFactory)
         context.lookup("QueueConnectionFactory");
      Queue queue = (Queue)context.lookup(args[0]);

      // create connection, session and receiver
      QueueConnection connection = factory.createQueueConnection();
      QueueSession session = connection.createQueueSession(
         false, Session.AUTO_ACKNOWLEDGE);
      QueueReceiver receiver = session.createReceiver(queue);
```

Das Programm SimpleReceiver implementiert einen JMS-Client, welcher synchron Meldungen von einer Queue empfängt und diese ausgibt.

Einfacher Receiver (ff.)

```
            connection.start();
            while (true) {
               // receive message
               Message message = receiver.receive();
               try {
                  String text = ((TextMessage)message).getText();
                  System.out.println(text);
               }
               catch (Exception e) {}
            }
         }
      }
```

Starten der Applikation

Kompilieren der Clients:

```
> set CLASSPATH=<j2eehome>\lib\j2ee.jar;.
> javac SimpleSender.java SimpleReceiver.java
```

Erzeugen der Queue und Starten des JMS-Providers:

```
> set PATH=<j2eehome>\bin
> j2eeadmin –addJmsDestination <queuename> queue
> j2ee –verbose
```

Starten der Clients:

```
> java –Dorg.omg.CORBA.ORBInitialHost=<serverhost>
      –Dorg.omg.CORBA.ORBInitialPort=<serverport>
      SimpleSender/SimpleReceiver <queuename>
```