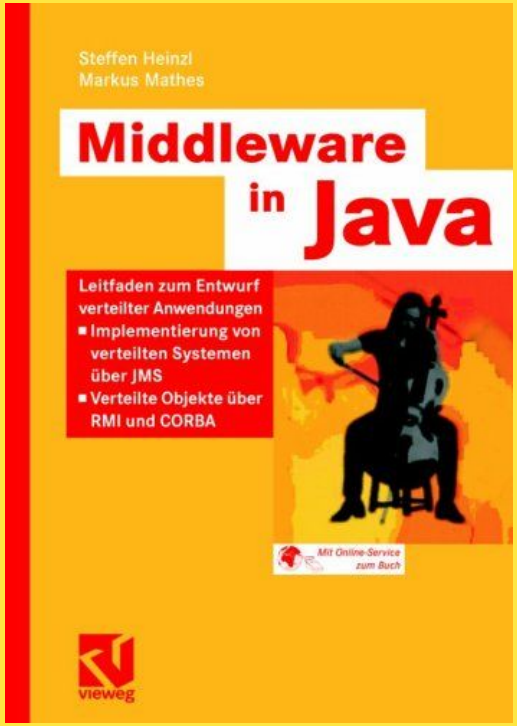
Kapitel 6:
Verteilte Objekte durch RMI



Prinzip von RMI

- RMI (Remote Method Invocation) erlaubt den Aufruf von Methoden von Objekten, die sich auf einem ***anderen*** – einem ***entfernten*** – Rechnersystem befinden.
- Man unterscheidet zwischen ***entfernten*** und ***lokalen Methoden*** bzw. ***entfernten*** und ***lokalen Objekten***.
- Der Aufruf einer entfernten Methode kann nicht vom Aufruf einer lokalen Methode unterschieden werden. Folglich realisiert RMI ***Ortstransparenz***.

Vor- und Nachteile von RMI

Vorteile:

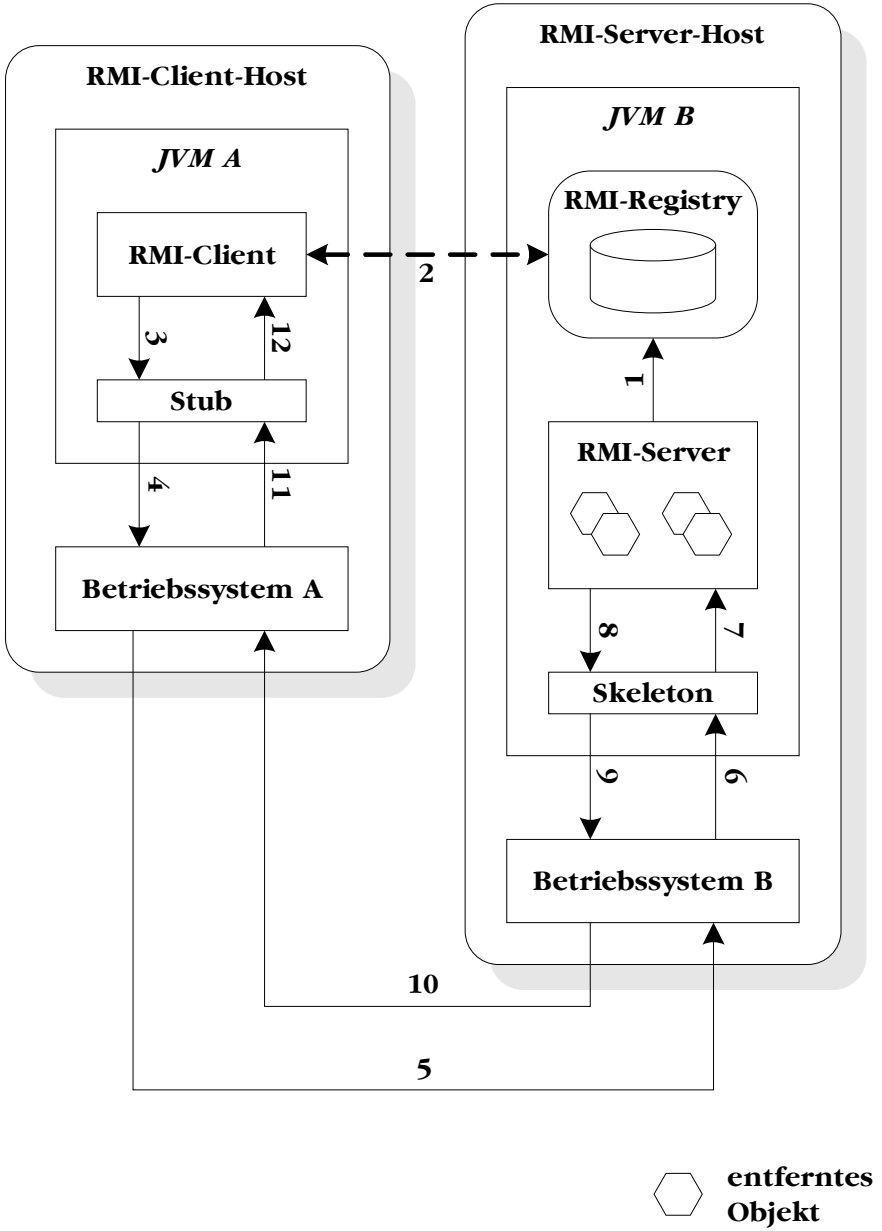
- hohes Maß an Ortstransparenz
- Einsatz eines Namensdienstes (RMI-Registry)
- dynamisches Laden von Code möglich
- Quasi-Standard

Nachteile:

- Integration mit anderen Verteilungstechniken schwierig
- Synchronisation der entfernten Aufrufe durch Programmierer notwendig
- Aufruf entfernter Methoden kann fehlschlagen

Struktur einer RMI-Anwendung (1)

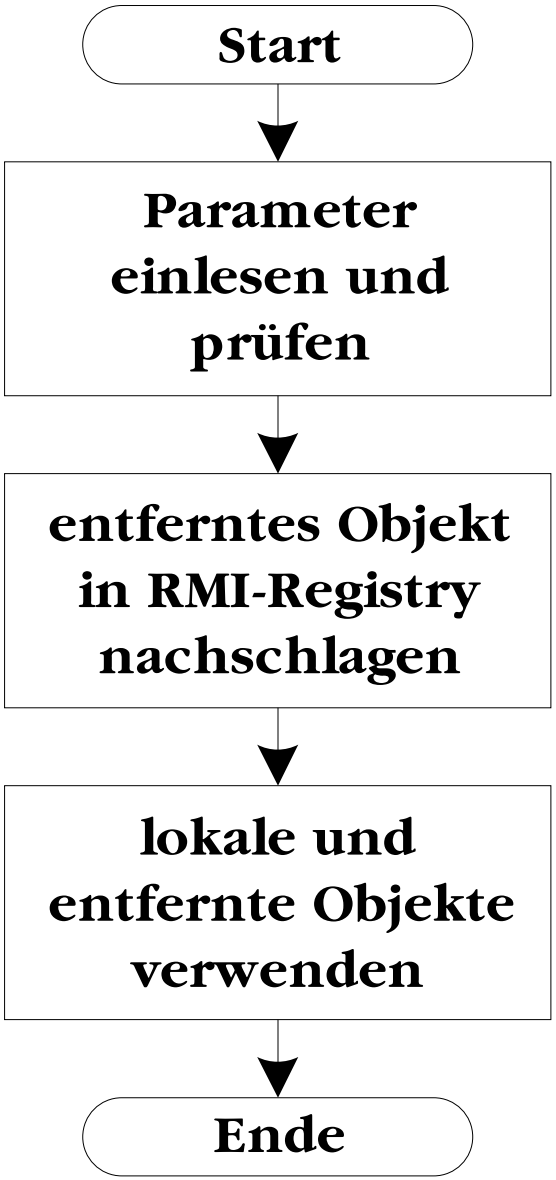
- RMI-Client
- Stub (-Objekte)
- Skeleton (-Objekte)
- RMI-Registry
- RMI-Server und entfernte Objekte



Struktur einer RMI-Anwendung (2)

- ***Stub***
  - Ein Stub-Objekt ist ein ***(client-seitiger) Stellvertreter*** für das entfernte Objekt beim RMI-Server.
- ***Skeleton***
  - Ein Skeleton-Objekt ist ein ***(server-seitiger) Stellvertreter*** für das aufrufende Objekt.
- ***RMI-Registry***
  - Die RMI-Registry implementiert einen ***Namens-dienst***, der eine Abbildung von Dienstnamen auf entfernte Objekte realisiert.

Arbeitsweise eines RMI-Clients



Entwicklung einer RMI-Anwendung

Definition der Schnittstelle entfernter Objekte

⇓

Implementierung der entfernten Schnittstelle

⇓

Generierung der Stubs/Skeletons

⇓

Starten des RMI-Servers

⇓

Starten des RMI-Clients

Definition der Schnittstelle entfernter Objekte

- Alle entfernt aufrufbaren Methoden müssen zunächst in einem Interface definiert werden, das von dem Interface `Remote` erbt.
- Das Interface `Remote` selbst ist leer und dient nur zur Markierung von so genannten ***Remote-Interfaces*** (***Marker-Interface***).
- Beispiele:
  - Remote-Interface für DAYTIME
  - Remote-Interface für TIME
  - Remote-Interface für ECHO

Remote-Interface für DAYTIME

```
package RMIExamples.MultiserviceServer;

import java.rmi.*;

public interface Daytime extends Remote
{
  /* Semantik: Liefert Systemzeit in lesbarer Form */
  public String getDaytime() throws RemoteException;
}
```

Remote-Interface für TIME

```
package RMIExamples.MultiserviceServer;

import java.rmi.*;

public interface Time extends Remote
{
  /* Semantik: Liefert Systemzeit in Millisekunden */
  public long getTime() throws RemoteException;
}
```

Remote-Interface für ECHO

```
package RMIExamples.MultiserviceServer;

import java.rmi.*;

public interface Echo extends Remote
{
  /* Semantik: Liest einen String und gibt ihn
   *           unverändert zurück
   */
  public String sendData(String data)
    throws RemoteException;
}
```

Implementierung des Remote-Interface DAYTIME

```
package RMIExamples.MultiserviceServer;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.text.*;

public class DaytimeImpl extends UnicastRemoteObject implements Daytime
{

  /* Konstruktor */
  public DaytimeImpl() throws RemoteException
  {

  }

  public String getDaytime() throws RemoteException
  {
    // Zeitformat: Tag, Datum, Zeit
    SimpleDateFormat formatter = new SimpleDateFormat("EEE, dd.MM.yyyy, HH:mm:ss:SS");

    return formatter.format(new Date());
  }
}
```

rmic (1)

- Mit Hilfe von `rmic` lassen sich Stubs und Skeletons ***automatisch*** aus Klassendateien generieren.
- Syntax: `rmic <Optionen> <Klassen>`
- Optionen:

```
-keep / -keepgenerated
-v1.1 / -vcompat / -v1.2
-iiop / -idl
-g / -nowarn / -verbose
-nowrite
-classpath <Pfad>
-bootclasspath <Pfad>
-d <Verzeichnis>
```

rmic (2)

Aufruf von ...	... erzeugt
`rmic -keep RMIExamples.MultiserviceServer.EchoImpl`	`EchoImpl_Stub.java` `EchoImpl_Stub.class`
`rmic -keep RMIExamples.MultiserviceServer.DaytimeImpl`	`DaytimeImpl_Stub.java` `DaytimeImpl_Stub.class`
`rmic -keep RMIExamples.MultiserviceServer.TimeImpl`	`TimeImpl_Stub.java` `TimeImpl_Stub.class`

DaytimeImpl_Stub (1)

```
// Stub class generated by rmic, do not edit. Contents subject to change without notice.

package RMIExamples.MultiserviceServer;

public final class DaytimeImpl_Stub extends java.rmi.server.RemoteStub
  implements RMIExamples.MultiserviceServer.Daytime, java.rmi.Remote {

  private static final long serialVersionUID = 2;
  private static java.lang.reflect.Method $method_getDaytime_0;

  static {
    try {
      $method_getDaytime_0 =
        RMIExamples.MultiserviceServer.Daytime.class.getMethod("getDaytime", new java.lang.Class[] {});
    }
    catch (java.lang.NoSuchMethodException e) {
      throw new java.lang.NoSuchMethodError("stub class initialization failed");
    }
  }
```

DaytimeImpl_Stub (2)

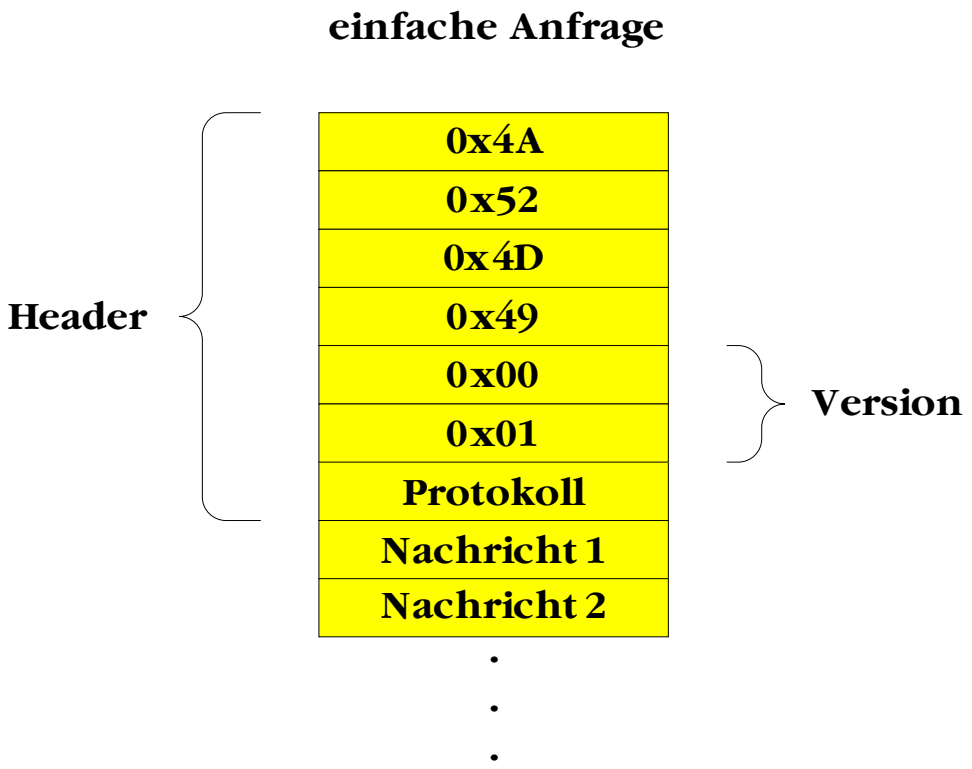
```
  // constructors
  public DaytimeImpl_Stub (java.rmi.server.RemoteRef ref) {
    super(ref);
  }

  // methods from remote interfaces
  // implementation of getDaytime()
  public java.lang.String getDaytime() throws java.rmi.RemoteException {
    try {
      Object $result = ref.invoke(this, $method_getDaytime_0, null, 6856047024248350588L);
      return ((java.lang.String) $result);
    }
    catch (java.lang.RuntimeException e) {
      throw e;
    }
    catch (java.rmi.RemoteException e) {
      throw e;
    }
    catch (java.lang.Exception e) {
      throw new java.rmi.UnexpectedException("undeclared checked exception", e);
    }
  }
}
```

RMI Wire-Protocol (1)

- Das RMI Wire-Protocol definiert die zwischen RMI-Client und RMI-Server ausgetauschten Nachrichten.
- Die Kommunikation zwischen RMI-Client und RMI-Server erfolgt über einen so genannten ***Stream***.
- Man unterscheidet:
  - ***output stream*** (vom RMI-Client zum RMI-Server)
  - ***input stream*** (vom RMI-Server zum RMI-Client)
- Der RMI-Client sendet ***Anfragenachrichten***, der RMI-Server antwortet mit ***Antwortnachrichten***.

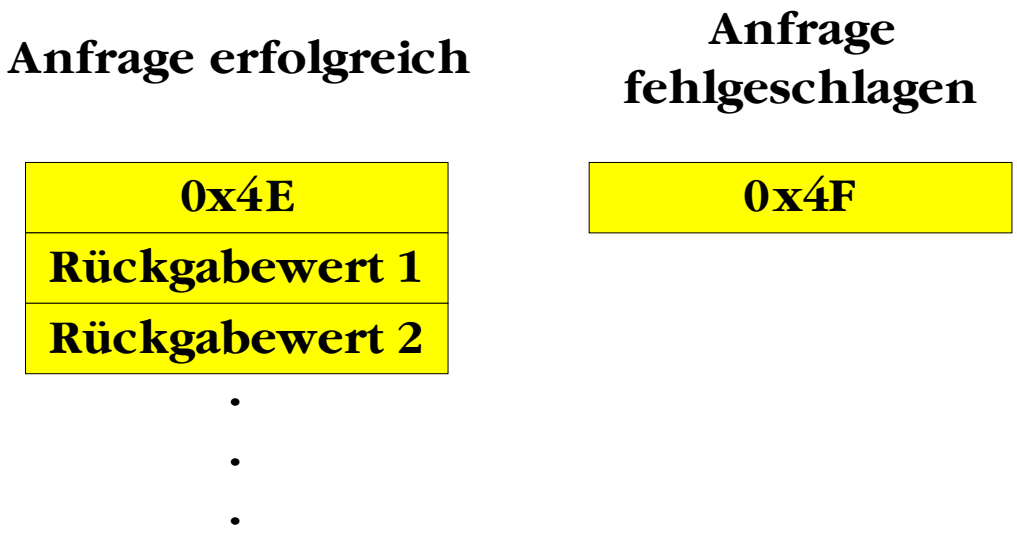
RMI Wire-Protocol (2)

- Einfache Anfrage besteht aus
  - ***Header***
  - ***Menge von Nachrichten***
- Protokollnummer definiert Aufbau von Nachrichten
  - `StreamProtcol`
  - `SingleOpProtocol`
  - `MultiplexProtocol`



RMI Wire-Protocol (3)

- Die Anfrage eines RMI-Clients kann entweder ***erfolgreich sein*** oder ***fehlschlagen***.
- Der RMI-Server signalisiert dies mittels einer entsprechenden ***Antwortnachricht***.



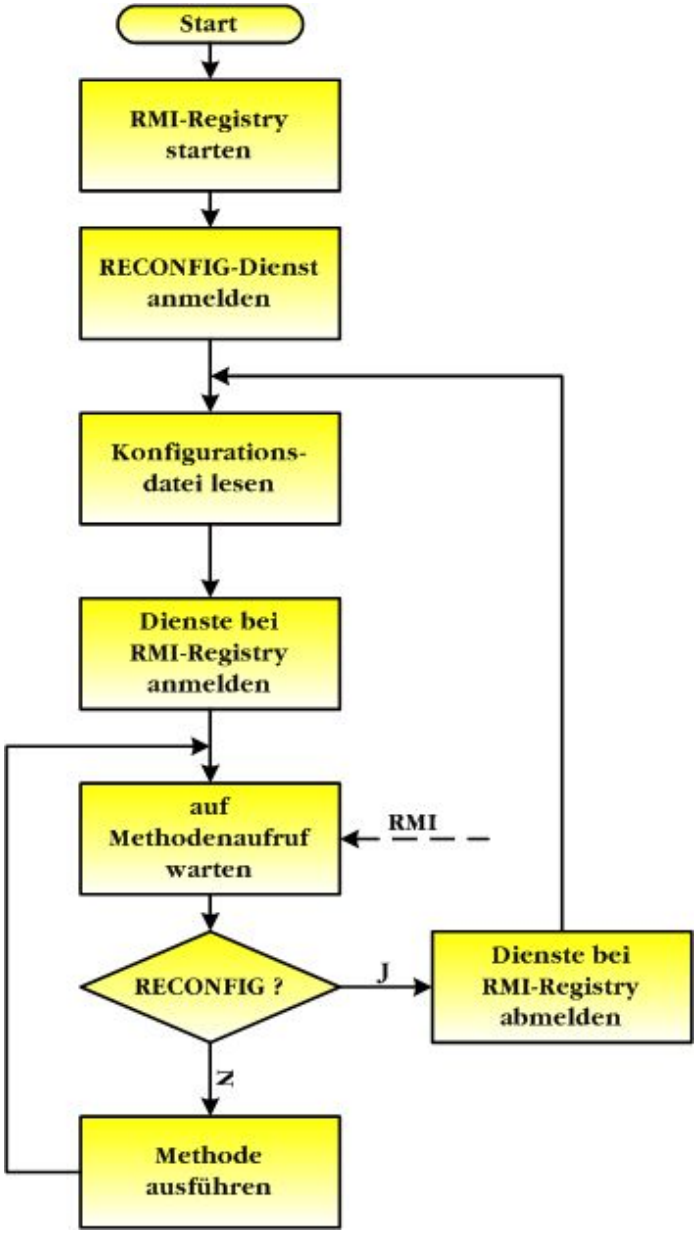
Implementierung einer Dienstauskunft

```
package RMIExamples;

import java.rmi.*;
import java.net.*;

public class ServiceList
{
  public static void main(String args[])
  {
    try
    {
      switch(args.length)
      {
        case 1:
          String services[] = null;
          // Service-Liste anfordern
          services = Naming.list(args[0]);
          System.out.println(services.length +
            " Dienst(e) registriert:");
          for(int i = 0; i < services.length;
            i++)
          {
            System.out.println("   " +
              services[i]);
          }
          break;
        default:
          System.err.println("Syntax:");
          System.err.println("  java " +
            " ServiceList <Registry-URL>");
          System.err.println("Beispiel: ");
          System.err.println("  java " +
            ServiceList //localhost:1099\n");
          System.exit(-1);
          break;
      }
    }
    catch(RemoteException e)
    {
      System.err.println(e.toString());
      System.exit(-1);
    }
    catch(MalformedURLException e)
    {
      System.err.println(e.toString());
      System.exit(-1);
    }
  }
}
```

Fallbeispiel: Multiservice-Servers

Mittels RMI läßt sich einfach ein ***Multiservice-Server*** entwickeln, d.h. ein Server, der mehrere Dienste gleichzeitig anbietet.

Zusätzlich kann der Server ***dynamisch rekonfigurierbar*** implementiert werden.



Konfigurationsdatei service.cfg

- Der Multiservice-Server verwendet eine Konfigurationsdatei namens ***service.cfg*** folgender Form:

```
# service.cfg
echo;//localhost:1099/echo;EchoImpl
daytime;//localhost:1099/daytime;DaytimeImpl
time;//localhost:1099/time;TimeImpl
# Ende service.cfg
```

- Ein korrekter Eintrag in der Konfigurationsdatei hat den Aufbau `Dienstname;Dienst-URL;Dienstanbieter`.
- Als Kommentarzeichen dient `#`.

Literatur

- William Grosso: *Java RMI*; O'Reilly 2001
- Ressourcen von Sun Microsystems Inc. zu Java RMI; http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html
- Guido Krüger: *Handbuch der Java-Programmierung (4. Auflage)*; Addison Wesley 2004; http://www.javabuch.de
- Sun Microsystems Inc.: *JDK 5.0 Documentation*; http://java.sun.com/j2se/1.5.0/docs/index.html, http://java.sun.com/j2se/1.5.0/docs/api/index.html
- Christian Ullenboom: *Java ist auch eine Insel*; Galileo Computing 2004; http://www.galileocomputing.de/openbook/javainsel4/
- Sun Microsystems Inc.: *Java Remote Method Invocation Specification*; http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html

Aufgaben

In „***Middleware in Java***“ finden Sie

- Wiederholungs-,
- Vertiefungs-,
- Programmieraufgaben

zu den vorgestellten Themen.

Zur Festigung und Vertiefung des Erlernten wird eine Bearbeitung der Aufgaben empfohlen.