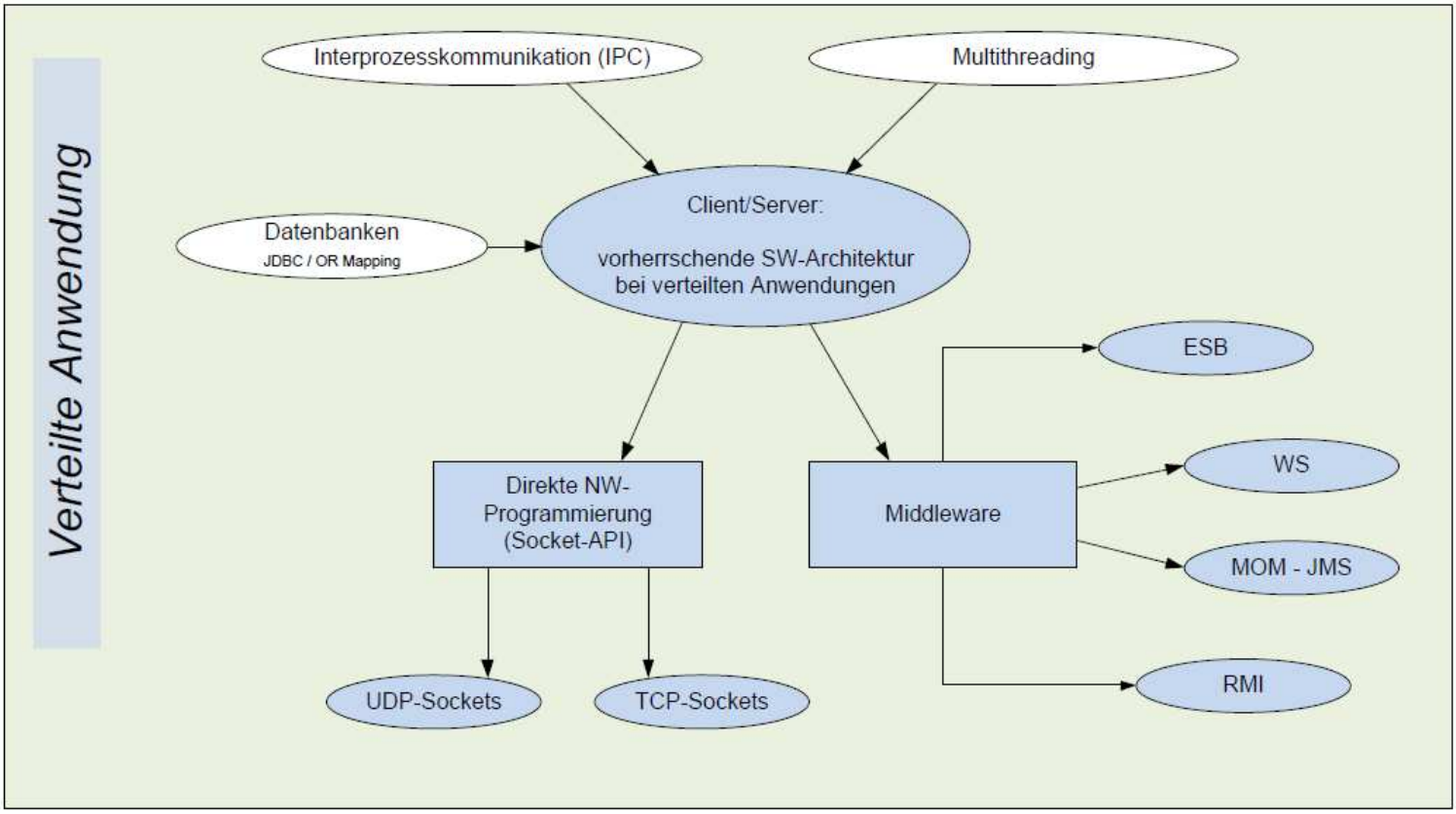
Lucerne University of
Applied Sciences and Arts
HOCHSCHULE
LUZERN

PIK – Zusammenfassung

Prinzipien und Entwicklung von SW-Komponenten



Interprozesskommunikation (IPC)

Unter **Interprozesskommunikation** versteht man Methoden zum Informationsaustausch, informatisch gesprochen Datenübertragung, von nebenläufigen Prozessen oder Threads. Im engeren Sinne versteht man unter IPC die Kommunikation zwischen Prozessen auf demselben Computer, deren Speicherbereiche aber strikt voneinander getrennt sind. Im weiteren Sinne bezeichnet IPC aber jeden Datenaustausch in **Verteilten Systemen**, angefangen bei Threads, die sich ein Laufzeitsystem teilen, bis hin zu Programmen, die auf unterschiedlichen Rechnern laufen und über ein Netzwerk kommunizieren. Für die Kommunikation ist dabei eine geeignete Prozesssynchronisation notwendig, insbesondere wenn verschiedene Prozesse potentiell gleichzeitig auf dieselben Ressourcen zugreifen können.

Multithreading

Der Begriff **Multithreading** bezeichnet das gleichzeitige Abarbeiten mehrerer Threads (das sind Ausführungsstränge innerhalb eines einzelnen Prozesses oder eines Tasks).

Middleware

Middleware (deutsch etwa „Zwischenanwendung“) bezeichnet in der Informatik anwendungsneutrale Programme, die so zwischen Anwendungen vermitteln, dass die Komplexität dieser Applikationen und ihre Infrastruktur verborgen wird.

ESB

Mit **Enterprise Service Bus** (**ESB**) bezeichnet man in der Informationstechnik (IT) eine Kategorie von Softwareprodukten, die die Integration verteilter Dienste (engl. *service*) in der Anwendungslandschaft eines Unternehmens (engl. *enterprise*) auf unten noch näher zu definierende Weise unterstützen.

MOM

Message Oriented Middleware (**MOM**) bezeichnet Middleware, die auf der asynchronen oder synchronen Kommunikation, also der Übertragung von Nachrichten *(Messages)* beruht. Das Format für die Nachrichten ist nicht festgelegt, in der Praxis hat sich jedoch XML als beliebtes Format etabliert.

Socket API

Der englische Begriff socket bedeutet übersetzt Steckdose, und analog zu den Anschlüssen des hausüblichen Wechselstromnetzes ist ein Socket im Zusammenhang mit Datennetzen ein Kommunikationsendpunkt. Im Gegensatz zu normalen Steckdosen handelt es sich jedoch nur um einen softwarebasierten .logischen. und keinen physischen Anschluß, der typischerweise auch eine bidirektionale Übertragung erlaubt.

Ein API (Application Program Interface) ist, wie eingangs angedeutet, eine Schnittstelle für die Programmierung von Anwendungen. Es besteht in der Regel aus mehreren Funktionen und Datenstrukturen, die jeweils eine bestimmte Funktionalität bereitstellen. Das Socket-API wird unter anderem benutzt, um Verbindungen zu anderen Rechnern aufzubauen oder Daten zu versenden.

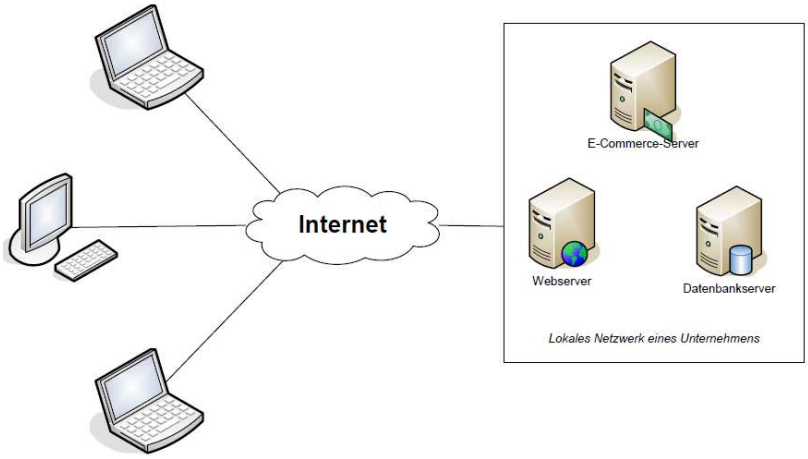
Verteilte Systeme und Anwendungen

Definition

Ein **verteiltes System (VS)** besteht aus einer Menge autonomer Computer, die durch Computernetzwerke miteinander verbunden sind und mit einer Software für die Koordination ausgestattet sind . (Beispiel: Intranet)

Ein verteiltes System ist ein System, in dem Hard- und Softwarekomponenten, die sich auf untereinander vernetzten Computer befinden, miteinander kommunizieren und Ihre Aktionen koordinieren, indem sie Nachrichten austauschen.

Ein **verteilte Anwendung (VA)** ist eine Anwendung, die aus verschiedenen Komponenten besteht und ein verteiltes System zur Lösung eines Anwendungsproblems benutzt. Die Komponenten einer verteilten Anwendung kommunizieren miteinander mit Hilfe von Nachrichten. Einem Standard-Anwender erscheint eine verteilte Anwendung wie eine gewöhnliche, nicht verteilte Anwendung (transparent).



Schichten einer Anwendung:

- **Datenhaltung**
  Datenbanken / Dateien
- **Datenverarbeitung**
  Geschäftslogik / Prozesssteuerung
- **Datenpräsentation**
  Diverse Datensichten
  Kommunikation mit Benutzer (UI)

Verteilte Systeme - Vorteile:

- **Besseres Abbild der Realität**
  logische vs. physische Struktur
  Leistungen werden dort erbracht, wo sie benötigt werden
- **Wirtschaftlichkeit und Lastverteilung**
  bessere Ausnutzung von Ressourcen
  Lastverbund
- **Bessere Skalierbarkeit**
  Erweiterung von bestehenden Systemen nach Bedarf
  System funktioniert mit wenigen und auch mit Vielen Komponenten (Webserver & Browser im WWW-Kontext)
- **Fehlertoleranz**
  Abfangen und Maskieren von Fehlern
  Lösungsansätze (HW- und SW-Redundanz, SW-Recovery)

Verteilte Systeme – Nachteile

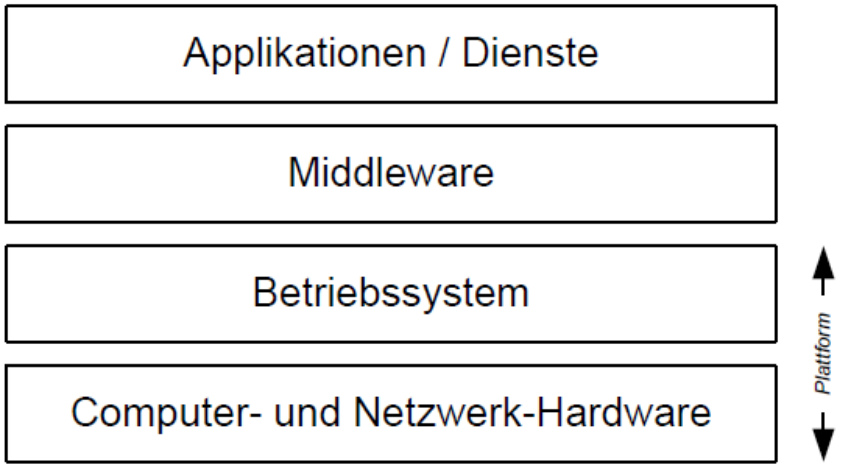
- **Höhere Komplexität durch Verteilung und Heterogenität**
  diverse Plattformen, HW-Architekturen, Programmiersprachen
- **Komplexe Netzinfrastruktur**
  Netzwerkarchitekturen & Protokolle
- **Höhere Sicherheitsrisiken (Verletzlichkeit)**
  Vertraulichkeit / Integrität / Authentizität
  e-banking & e-commerce

Modelle

Mit einem Modell werden allgemeine Eigenschaften und Design eines Systems beschrieben. Mit der Definition eines Modells wird Folgendes angegeben:

- Die wichtigsten Komponenten des Systems und deren Aufgaben
- Die Interaktion zwischen unterschiedlichen Komponenten des Systems
- Wie das Verhalten der Komponenten (einzeln und kollektiv) beeinflusst werden kann

Softwareschichten Modell



Applikationen

- Unabhängig von einer Plattform
- Sollte eine spezifisches Middleware-Modell verwenden

Middleware

- Verbirgt die Heterogenität des verteilten Systems
- Stellt ein Programmierungsmodell zur Verfügung (API)

Betriebssystem

- Ermöglicht den Zugriff auf die Systemressource

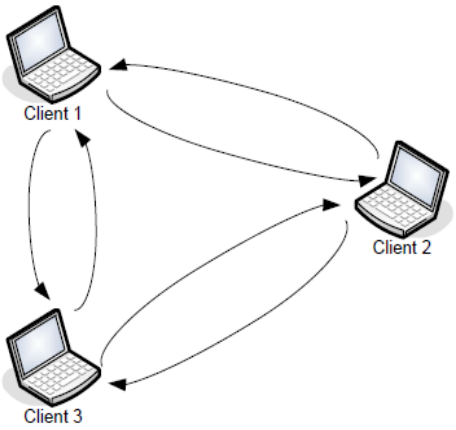
Architektur Modelle

Das Architekturmodell eines verteilten Systems

- vereinfacht und abstrahiert die Erfassung von Funktionen der einzelnen Komponenten
- definiert die Verteilung von Komponenten in einem Netz von Computern
- definiert die Beziehung von Komponenten untereinander (Rolle in der Kommunikation, Kommunikationsmuster)

Peer 2 Peer Modell

- Jeder Peer-Knoten hat
  - Sowohl den Applikationscode
  - als auch Koordinationscode
- Anwendungsbereich
  - Austauschplattform (Musik, Filme usw.)
- Vor- und Nachteile



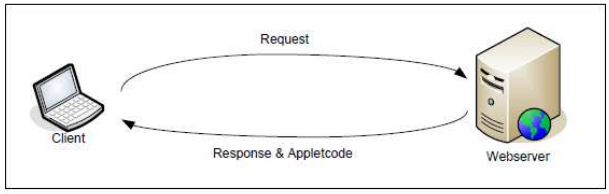
Client-Server Modell

- Es handelt sich um eine Softwarearchitektur (sowohl Client als auch Server können auf dem gleichen Knoten laufen).
- Server kann andere Server bei der Beantwortung einer Anfrage "bemühen" und in die Client-Rolle schlüpfen

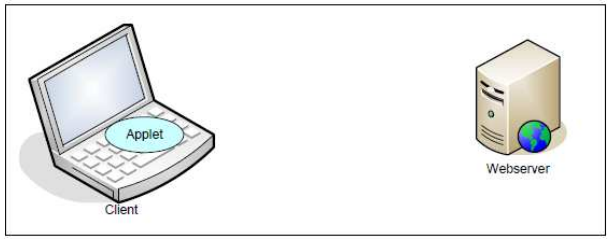
- Rollenwechsel
  - ❖ Metasuchmaschinen
  - ❖ Applikationsserver bei komplexen Transaktionen
- Ein Server kann i.d.R. mehrere Clients "gleichzeitig" bedienen (Nebenläufigkeit)

Variante Applet:

- Schritt 1:
  Client holt sich das Applet vom Webserver



- Schritt 2:
  Applet wird auf dem Client-System ausgeführt



Mobile Agenten:

Mobile Agenten sind kleine Programme bzw. Codestücke, welche sich von einem zu anderen Server bewegen um bestimmte Aktionen durchzuführen. Der Agent-Code wird auf dem Wirt-System ausgeführt. Nach der Ausführung wird der Agent "gefroren" und zusammen mit seinem Kontext zu einem anderen Wirt-System gesendet. Bei dem anderen Wirt-System angekommen, wir der Agent samt seinem Kontext „aufgetaut“ und der Programmcode ausgeführt. Dies alles wird im Auftrag vom Client bzw. für den Client gemacht

Spontane Netzwerke:

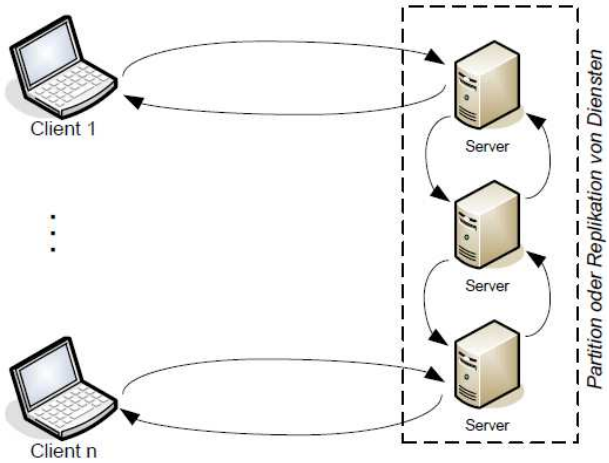
- Spontane Netzwerkverbindungen von mobilen Geräten in einer "fremden" Umgebung
- Zur Zeit Gegenstand intensiver Forschung
- gewinnt mit dem Vormarsch mobiler Geräte (PDA, Laptop, …) immer mehr an Bedeutung
- Wichtige Begriffe: Wireless LAN, Bluetooth

Mehrfacher Server:

Partition und Replikation von Daten und Diensten

Beispiele:

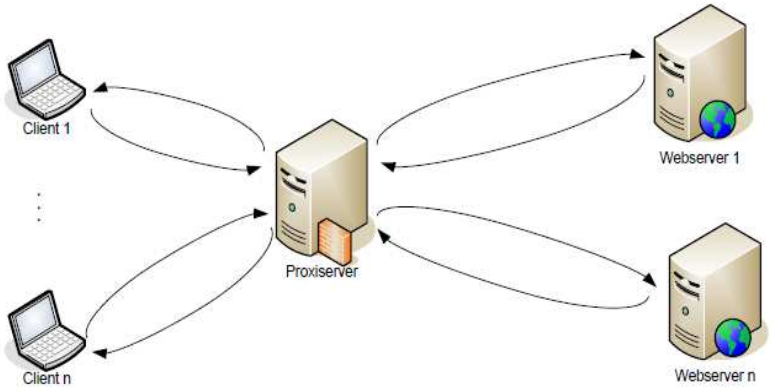
- Partition
- Replikation



Proxy Server:

Der Nutzen:

- Bessere Performance
- Bessere Verfügbarkeit (Cache)
- Erhöhte Sicherheit (Zugriffskontrolle)



Anforderungen an Modellauswahl

Performance

Antworten auf eine Anfrage müssen

- schnell zur Verfügung stehen und
- konsistent sein

Dies wird begünstigt durch

- den Einsatz von wenigen Komponenten
- die lokale Kommunikation immer wo möglich
- den Transfer bzw. Austausch von möglichst kleinen Datenmengen bzw. Dateneinheiten

Durchsatz von Daten

- die Geschwindigkeit bzw. die Rate, mit der die Rechenarbeit (Berechnung, Zustellung von Daten) erledigt wird
- wird von mehreren Komponenten, die an der Rechenarbeit beteiligt sind, bestimmt
- die schwächste Komponente bestimmt den Durchsatz

Lastbalancierung

- probiert die stark belasteten Komponenten zu entlasten und die anfallende Arbeit möglichst systemweit zu verteilen
- benötigt unter Umständen die Replikation von Daten, womit das ganze System komplizierter wird

Dienstgüte

Wenn ein Dienst verfügbar ist, wird probiert, den Dienst möglichst zu verbessern bzw. zu optimieren. Die Verbesserungen können folgende Punkte beinhalten:

- Zuverlässigkeit des Systems
- Verfügbarkeit des Systems
- Sicherheit
- Immer mehr echtzeitorientierte Parameter

Cashing und Replikation

Mit **Cashing** werden bestimmte Daten im Speicher gehalten und auf die Anfrage als Antwort geliefert (schneller Zugriff). Problem:

- Wie kann die Aktualität von Daten sicher gestellt werden?

Mit **Replikation** soll die Verfügbarkeit von Daten erhöht werden. Problem:

- Wie kann erreicht werden, dass alle Replikate den gleichen Stand haben?

Verlässlichkeit

Korrektheit

- Das System sollte das erwartete Verhalten zeigen (wie z. B. in der Spezifikation definiert)

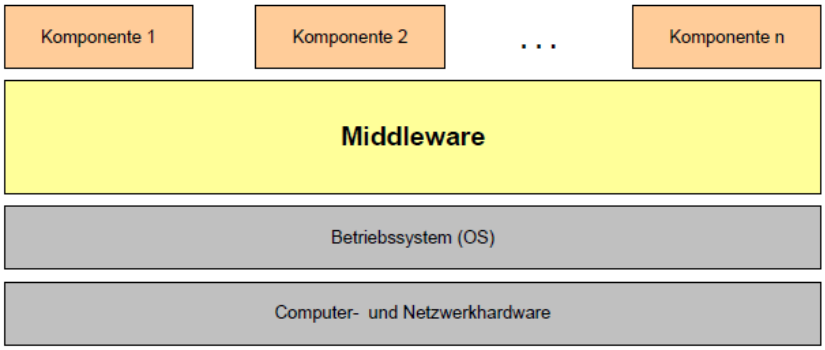
Sicherheit

- wo ist der sicherste Ablageplatz
- die Verfügbarkeit von Daten hat enorme Bedeutung

Fehlertoleranz

- inwieweit ein Dienst bzw. System noch zuverlässig arbeitet, nachdem ein Fehler aufgetreten ist

Middleware



Transparenz:

Das verteilte System wirkt wie eine Ganzheit. Der Benutzer nimmt es gar nicht wahr, dass es sich bei dem System, das von ihm benutzt wird, um ein verteiltes System handelt. Transparenztypen:

- Access Transparency (Zugriffstransp.)
- Location Transparency (Ortstransp.)
- Concurrency Transparency (Nebenläufigkeittransp.)
- Replication Transparency
- Failure Transparency (Fehler- bzw. Ausfalltransp.)
- Mobility Transparency (Mobilitätstransp.)
- Performance Transparency
- Scaling Transparency

DMBS in Java-Anwendungen

Datenbanken sind ein "fester" Bestandteil von Anwendungen. Aufgaben:

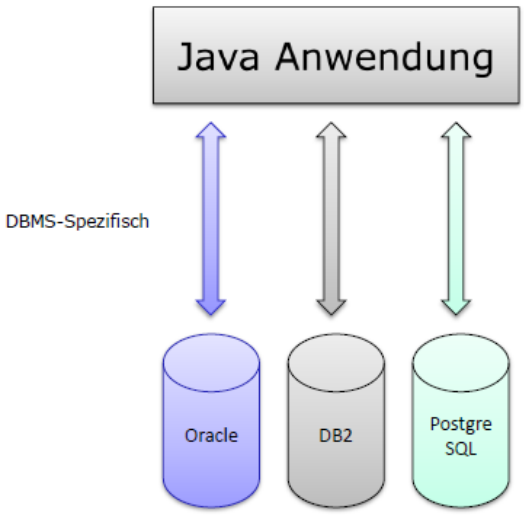
- dauerhafte Ablage von Daten
- sicherer Zugriff auf Daten

Anforderungen an DBMS:

- Leistungsfähigkeit
- Datensicherheit (konsistenter Zustand)
- Unterstützung von Transaktionen

Unterschiedliche DBMS haben unterschiedliche Zugriffsprotokolle:

- Bei einem **direkten** Zugriff aus der Applikation auf das DBMS ergeben sich Probleme bei der Umstellung auf ein anderes DBMS: *DBMS-Abhängigkeit*
- Bei einem DBMS-Wechsel, muss der Zugriff auf das neue DBMS im Programmcode teilweise oder ganz reimplementiert werden



<u>DBMS Unabhängigkeit:</u>

Eine Anwendung soll nicht von einem konkreten DBMS abhängig sein. Eine Zwischenschicht zwischen Anwendung und DBMS soll als "Übersetzer" dienen!

Lösungen:

- ODBC
- JDBC

JDBC (Java Database Connectivity)

JDBC ist ein Programmierpaket, das aus JDBC Treibermanager, JDBC Treiber Test Suite, JDBC-ODBC-Bridge und JDBC API besteht. Aufgaben der einzelnen Komponenten:

- JDBC Treibermanager verwaltet unterschiedliche DBMS-Treiber und ist für die Verbindung zwischen einem Java-Programm und Datenbank zuständig
- JDBC Treiber Test Suite dient zum Testen, ob ein Treiber alle Anforderungen erfüllt, die an einen JDBC-Treiber gestellt werden (Zertifizierung)
- JDBC-ODBC-Bridge ist ein Treiber-Typ, mit welchem im wesentlichen ODBC-Treiber benutzt werden
- JDBC API ist die Programmierschnittstelle mit Paketen *java.sql* & *javax.sql*

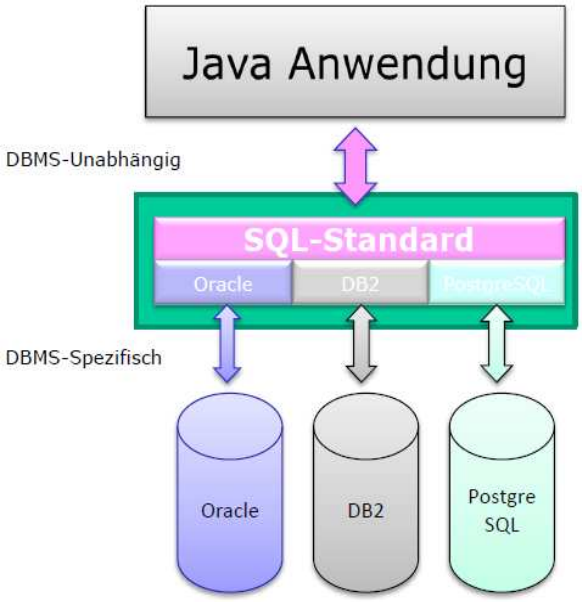
DBMS und Treiberkonzept:

Allgemeine Definition:

- Ein DB-Treiber ist eine Softwarekomponente, welche die Kommunikation zwischen einer Anwendung und einer Datenbank ermöglicht.

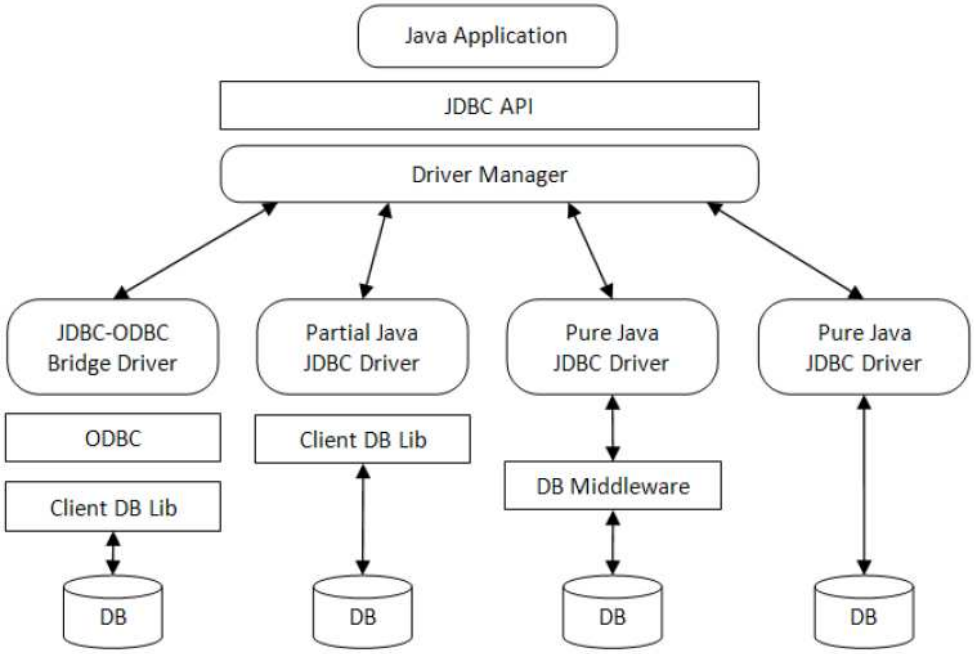
JDBC-Treiber:

- ist eine Komponente, welche die Kommunikation zwischen einer Java-Anwendung und einer Datenbank ermöglicht.
- stellt für den Client eine einheitliche Java-Schnittstelle zur Verfügung.



JDBC-Treibertypen:

- Typ 1: JDBC-ODBC-Treiber (Bridge)
- Typ 2: Native-API-Treiber
- Typ 3: JDBC-Net-Treiber
- Typ 4: Native-Protokoll-Treiber



Typ 1 und Typ 2:

- Übergangslösungen
- Enthalten Binärcode
- Nicht für Web-Anwendungen geeignet

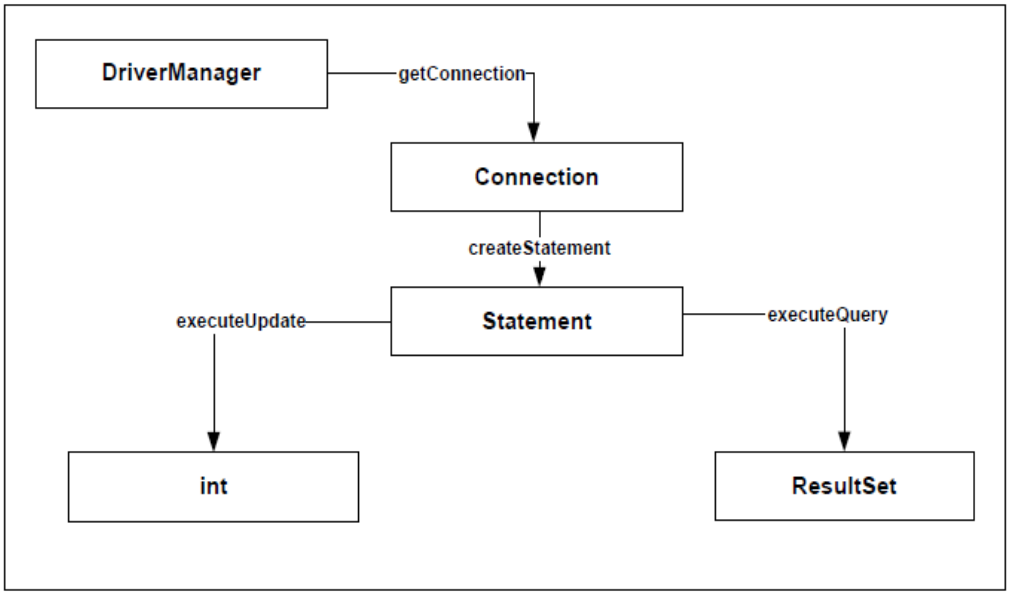
Typ 3 und Typ 4:

- Geeignet für die Web-Anwendungen
- In Java geschrieben

Typ 4:

- i.d.R. von DB-Hersteller geschrieben
- sehr effizient

Wichtigste JDBC Klassen und Schnittstellen:



Ablauf eines Zugriffs auf DB

Laden eines JDBC-Treibers

Automatisches Laden einer Treiberklasse:

- Alle in der *jdbc.drivers*-Systemeigenschaft angegebene Treiber werden vom Treibermanager beim Starten automatisch geladen

Explizites Laden einer Treiberklasse:

- Mit Hilfe der Methode *forName* der Klasse *Class*
- Durch die Erstellung einer Instanz der Klasse *Driver*

Beispiel für **PostgreSQL**-Treiber:

- Name der Treiberklasse: `org.postgresql.Drivers`
- Laden mit Hilfe der Methode `Class.forName:`

```
Class.forName("org.postgresql.Driver");
```

Beispiel für **MySQL**-Treiber:

- Name der Treiberklasse: `com.mysql.jdbc.Driver`
- Laden mit Hilfe der Methode `Class.forName:`

```
Class.forName("com.mysql.jdbc.Driver");
```

Verbindung aufbauen

Für Aufbau der Verbindung zu einer DB stehen folgende Methoden der Klasse *DriverManager* zur Verfügung:

- static Connection getConnection (String url) throws SQLException
- static Connection getConnection (String url, java.util.Properties properties) throws SQLException
- static Connection getConnection (String url, String user, String password) throws SQLException
- Alle Methoden werfen die *SQLException*
- Alle Methoden erwarten einen URL und geben ein Connection-Objekt zurück

Der URL der Datenbank

- ist die einzige treiberspezifische Information, welche von der JDBC-Schnittstelle verlangt wird
- ist wie folgt aufgebaut: `jdbc:<subprotocol>:<subname>`

Beispiele für URLs für PostgreSQL und MySQL:

- `jdbc:postgresql://localhost:5432/books_db`
- `jdbc:mysql://localhost/books_db`

Beispiel für den Aufbau einer Verbindung:

```
String url = "jdbc:postgresql://147.88.100.100:5432/raum_db";
String user = "student";
String pwd = "geheim";
// Aufbau der Verbindung
Connection con = DriverManager.getConnection(url, user, pwd);
```

Statement kreieren

Ein *Statement*-Objekt wird mittels *Connection*-Methode *createStatement* erzeugt:

```
Statement stm = con.createStatement();
```

Mit dem erzeugten *Statement*-Objekt kann

- der lesende (SELECT) und
- der schreibende (INSERT, DELETE, UPDATE)

Zugriff auf die DB ausgeübt werden.

Schreibender Zugriff

Schreibender Zugriff auf eine DB erfolgt mittels *Statement*-Methode *executeUpdate.* Sie liefert einen `int`-Wert zurück (die Anzahl der geänderten / gelöschten / eingefügten Tupel). Beispiel:

```
int anz = 0;
String delQuery = "DELETE FROM tbl_raum WHERE id_raum=2";
anz = stm.executeUpdate(delQuery)
```

Die Variable `anz` gibt die Anzahl der gelöschten Tupel an…

Lesender Zugriff

Lesender Zugriff auf eine DB erfolgt mittels *Statement*- Methode *executeQuery.* Sie liefert ein *ResultSet*-Objekt zurück. Das *ResultSet*-Objekt stellt die resultierende Ergebnismenge der Abfrage dar. Beispiel:

```
String query = "SELECT * FROM tbl_raum";
ResultSet rs = stm.executeQuery(query)
```

ResultSet (Ergebnismenge):

Ein *ResultSet* stellt die Ergebnismenge einer Abfrage dar. Die Methode *next* ermöglicht eine schrittweise Abarbeitung der Ergebnismenge (Tupel für Tupel):

```
public boolean next () throws SQLException
```

Der `boolean`-Rückgabewert gibt an, ob noch weitere, nicht angesprochene Tupel in der Ergebnismenge vorhanden sind. Nach dem Aufruf der Methode *executeQuery* steht der Tupel-Zeiger vor dem ersten Tupel (Zeile) der Ergebnismenge. Vor dem ersten Zugriff auf die Ergebnismenge muss zuerst die Methode *next* aufgerufen werden, um den Tupel-Zeiger auf das erste Tupel zeigen zu lassen!

Für die Ausgabe der Ergebnismenge-Daten, stellt die Schnittstelle *ResultSet* folgende generische *getXXX*-Methoden zur Verfügung:

- `public int getInt (int columnIndex) throws SQLException`
- `public int getInt (String columnName) throws SQLException`
- `public long getLong (int columnIndex) throws SQLException`
- `public long getLong (String columnName) throws SQLException`
  `...`
- `public String getString (int columnIndex) throws SQLException`
- `public String getString (String columnName) throws SQLException`
- `public Object getObject (int columnIndex) throws SQLException`
- `public Object getObject (String columnName) throws SQLException`

Zugriff auf eine Spalte kann mit Hilfe **der Spaltennummer** oder mit Hilfe **des Spaltennamens** erfolgen

Weitere Methoden der Schnittstelle *ResultSet*:

- `public void beforeFirst () throws SQLException`
- `public void afterLast () throws SQLException`
- `public boolean absolute (int m) throws SQLException`
- `public boolean previous () throws SQLException`
- `public void first () throws SQLException`
- `public void last () throws SQLException`
- `public boolean isFirst() throws SQLException`
- `public boolean isLast() throws SQLException`
- `public boolean isBeforeFirst() throws SQLException`
- `public boolean isAfterLast() throws SQLException`
- `public int getRow() throws SQLException`

Beispiel:

```
String query = null;
ResultSet rs = null;
query = "SELECT bezeichnung, anz_plaetze FROM tbl_raum";
rs = stm.executeQuery(query);
String str = null;
while(rs.next())
{
    str = "Raum: " + rs.getString("bezeichnung");
    str += ", Anz. Plaetze: " + rs.getInt("anz_plaetze");
    System.out.println(str);
}
```

Verbindung schliessen

Das Schliessen wird mit Hilfe der Methode *close* bewerkstelligt. Diese Methode ist in folgenden Schnittstellen vorhanden:

- Connection
- Statement
- ResultSet

Beim Schliessen einer Instanz der *Statement*-Schnittstelle wird die zugehörige *ResultSet*-Instanz implizit geschlossen. Beim Schliessen einer *Connection*-Instanz, werden alle *Statement*- und *ResultSet*-Instanzen implizit geschlossen!!

Transaktionen

Eine Transaktion kann aus mehreren atomaren Aktionen bestehen und nur dann sinnvoll sein, wenn alle atomaren Aktionen erfolgreich durchgeführt werden. JDBC stellt die Möglichkeit zur Verfügung:

- den Anfang und das Ende einer Transaktion explizit anzugeben und
- eine nicht gelungene Transaktion zurückzusetzen

Connection-Methoden:

- setAutocommit
- commit
- rollback

Beispiel:

```
try
{
    connection.setAutoCommit(false);
    statement.executeUpdate("INSERT ... ");
    statement.executeUpdate("UPDATE ... ");
    statement.executeUpdate("DELETE ... ");
    connection.commit();
}
catch (SQLException e)
{
    if (connection != null)
    {
        connection.rollback();
    }
}
```

DBMS und Property-Dateien

Das Ziel:

- Das DBMS soll ausgetauscht werden können
- Die Änderungen im Code (Anpassungen) dürfen nicht nötig sein

Lösung:

- DBMS spezifische Daten in eine Textdatei schreiben, die zu jedem Zeitpunkt angepasst werden kann
- Es handelt sich um eine *property*-Datei, die Einträge vom Typ **key-value** enthält
- Beim Zugriff auf das DBMS liest das Programm die DBMS spezifischen Daten aus dieser Datei aus
- Bei einem Austausch des DBMS ist kein Eingriff in den Programmcode nötig

Beispiel (db.properties):

```
#====================================
#= DBMS: MySQL-Datenbank =
#= DB-Name: raum_db =
#====================================
#=== DB-URL
jdbc.url=jdbc:mysql://147.88.111.111:3306/raum_db
#=== Treiberklasse
jdbc.drivers=com.mysql.jdbc.Driver
#=== Benutzername
jdbc.user=student
#=== Password
jdbc.password=geheim
```

Auslesen der Properties

Für das Auslesen von DBMS-Spezifischen Daten aus einer *property*-Datei steht die Klasse java.util.Properties zur Verfügung:

```
// Properties-Objekt erzeugen
Properties dbProperties = new Properties();

// Klassenloader holen
ClassLoader cLoader = this.getClass().getClassLoader();

// Properties laden
dbProperties.load(cLoader.getResourceAsStream("db.properties");

// Treiber-Klasse auslesen
String driverClass = dbProperties.getProperty("jdbc.drivers");

// Treiber laden
Class.forName(driverClass);

// URL, Benutzername und das Kennwort auslesen
String dbUrl = dbProperties.getProperty("jdbc.url");
String user = dbProperties.getProperty("jdbc.user");
String pwd = dbProperties.getProperty("jdbc.password");

// Verbindung zur Datenbank herstellen
con = DriverManager.getConnection(dbUrl, user, pwd);
```

Bei einem Austausch des DBMS muss einzig die entsprechende *property*-Datei ersetzt bzw. angepasst werden, während der Programmcode unangetastet bleibt. Die entsprechende Treiberklasse muss verfügbar und im CLASSPATH sein!!

Metadaten

Eine Datenbank enthält neben Nutzdaten auch zusätzliche Informationen über Datenbank, Tabellen, Spalten usw.: die Rede ist von *Metadaten.* JDBC stellt zwei Klassen zur Verfügung, die das Abfragen von Metadaten ermöglichen:

- DatabaseMetaData
- ResultSetMetaData

Mit Hilfe von ***DatabaseMetaData*** können zusätzliche Informationen über die Datenbank geholt werden, während mit ***ResultSetMetaData*** die zusätzlichen Informationen über die Ergebnismenge geholt werden können

DatabaseMetaData

Ein *DatabaseMetaData*-Objekt wird mit Hilfe der Methode getMetaData der Klasse *Connection* geholt und enthält diverse Informationen über die Datenbank: Welche Tabellen uns Spalten sind in der Datenbank enthalten, wie lautet der Primärschlüssel für eine bestimmte Tabelle usw. Beispiel (Tabellennamen anzeigen):

```
// Metadaten für die Datenbank holen
DatabaseMetaData dbMetaData = connection.getMetaData();

// Tabellennamen auslesen
ResultSet rSet = dbMetaData.getTables(null, null, null, new String[]{"TABLE"});

// Tabellennamen ausgeben
while(rSet.next())
{
    System.out.println("TABLE: " + rSet.getString("TABLE_NAME"));
}
```

ResultSetMetaData

Ein *ResultSetMetaData*-Objekt wird mit Hilfe der Methode getMetaData der Klasse *ResultSet* geholt und enthält diverse Informationen über die aktuelle Ergebnismenge: Anzahl Spalten, Namen und SQL-Typ der Spalten usw. Beispiel (Spaltennamen und SQL-Typ anzeigen):

```
// Metadaten des aktuellen ResultSet-Objekts holen
ResultSetMetaData rsMetaData = rSet.getMetaData();

// Anzahl Spalten auslesen
int anzahlSpalten = rsMetaData.getColumnCount();

// Spaltennamen und SQL-Typ ausgeben
if(rSet.next())
{
    for (int i = 1; i <= rsMetaData.getColumnCount(); i++)
    {
        System.out.println("COLUMN-NAME: " + rSet.getString(i) + "TYPE: "
        + rSet.getColumnTypeName());
    }
}
```

Parallele Ausführung von Prozessen

Parallele Ausführung bedeutet, dass mehrere Ausführungseinheiten (Prozesse) gleichzeitig ausgeführt werden. Anwendungsbereich:

- Serverseitiges Erbringen von Leistungen, wobei mehrere Clients gleichzeitig bedient werden
- Steuerung von Echtzeitautomaten, Robotern
- Ausführung von komplexen Abläufen, die in mehrere parallel ausführbare "Subprozesse" zerlegt werden können (Splitting)

Voraussetzungen für parallele Ausführung:

- das Mehrprozessorsystem und
- die Unterstützung durch das Betriebssystem

Mehrprozessorsysteme

- sind (waren) teuer und nicht sehr verbreitet
- dies ändert sich langsam mit dem Vormarsch von Multicore- Prozessoren

Betriebssystem muss in der Lage sein, die Ausführung auf mehrere Prozessor-Einheiten zu verteilten und zu verwalten. Das war nicht immer so:

Am Anfang hatte man die Batch-Betriebssysteme im Einsatz, die ein Programm von Anfang bis zum Ende ohne Unterbruch ausführten. Interaktives Arbeiten bzw. parallele Ausführung war nicht möglich. Erst mit der Einführung von Timesharing-Betriebssystemen konnte eine "parallele" Ausführung realisiert werden

Prozess und Taskwechsel

Ein Prozess muss in seiner Ausführung unterbrechbar sein:

- während Laufzeit muss es möglich sein, einem Prozess die Betriebsmittel für eine **kurze** Zeit zu entziehen, um diese einem anderen Prozess zur Verfügung zu stellen
- Wenn der Wechsel genug schnell durchgeführt wird, merkt ein Beobachter nicht, dass dem Prozess die Betriebsmittel entzogen wurden

Für einen Beobachter laufen alle Prozesse "parallel".

Timesharing-Betriebssysteme

Ein Prozess darf gar nicht merken, dass er unterbrochen wurde. Er muss, nachdem er die Betriebsmittel wieder bekommen hat, genau dort weiterfahren können, wo er unterbrochen wurde

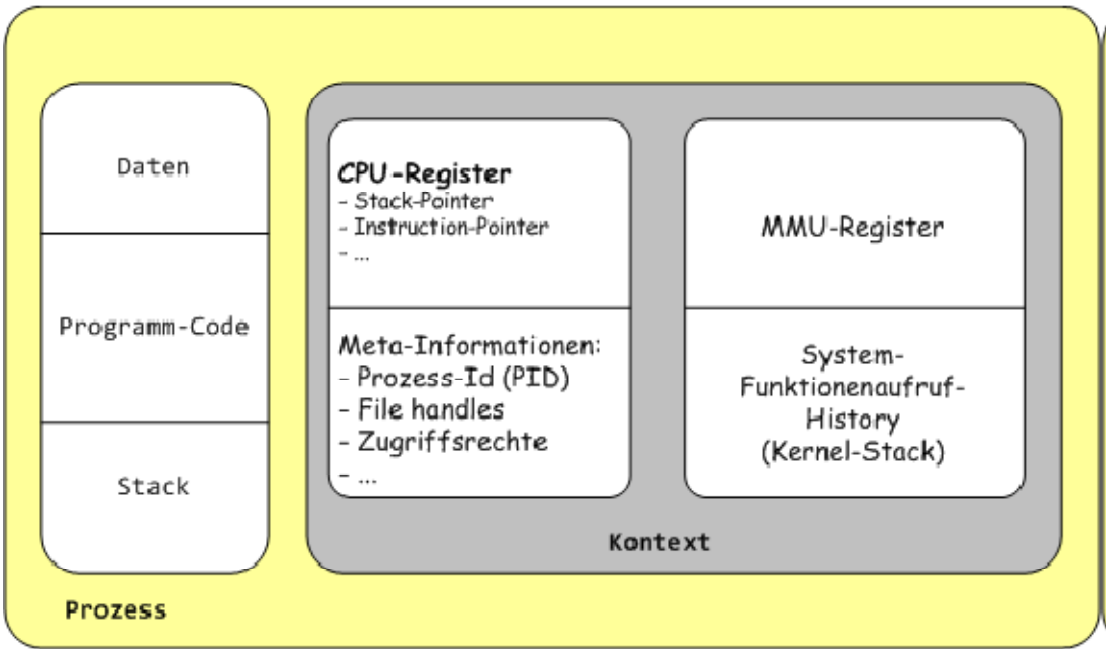
Die ersten Betriebssysteme, welche ein Prozesskonzept unterstützt haben, waren die Zeitscheiben-Betriebssysteme. Jedem Prozess wurden vom Scheduler für eine bestimmte Zeitscheibe (Time Slice) die Betriebsmittel zur Verfügung gestellt.

Wenn ein Prozess die Zeitscheibe (*time slice*) zum ersten Mal erhält, beginnt er mit der Ausführung. Wenn die Zeitscheibe verbraucht wird, bevor die Arbeit erledigt wird, wird dem Prozess die Betriebsmittel entzogen. Nachdem der Prozess die nächste Zeitscheibe erhalten hat, fährt er genau an der Stelle weiter, wo er unterbrochen wurde. Dies muss vom Betriebssystem gewährleistet werden.

Prozess und Prozessmittel

Jeder Prozess hat eine eigene Umgebung, in der die für die Ausführung benötigten Informationen und Daten verwaltet werden. Dies beinhaltet unter anderem:

- Prozessspezifische Daten (Id, Priorität, …)
- Code zum Ausführen
- Ablage für die während Laufzeit generierten Daten (Stack)
- Registerinhalte: Stack-Pointer (SP), Instruction-Pointer (IP) usw.
- Diverse Meta-Informationen
- Die für MMU relevanten Informationen

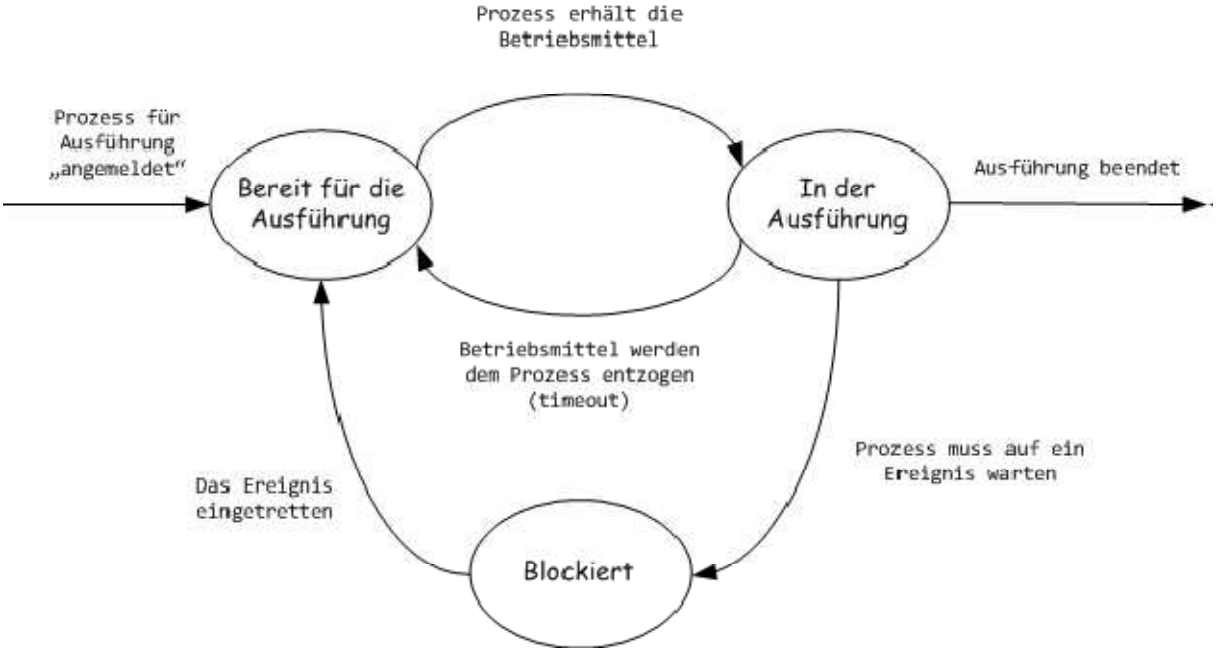


Prozess Zustände

Ein Prozess kann sich in 3 Zuständen befinden:

- Bereit für die Ausführung
- In der Ausführung
- Blockiert

Die Übergänge zwischen unterschiedlichen Zuständen werden von der Laufzeitumgebung gesteuert.

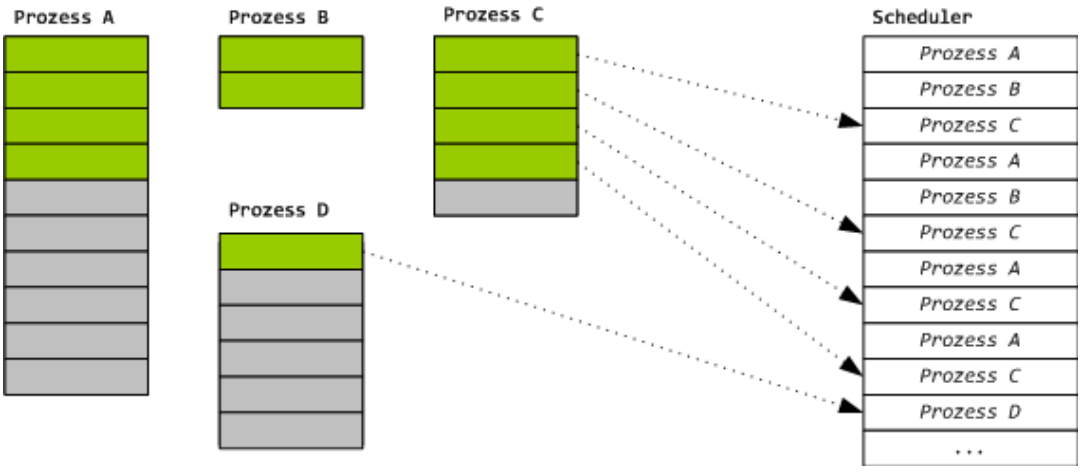


Prozessauslagerung (Taskwechsel)

Ein Timesharing-Betriebssystem ist in der Lage, mehrere Prozesse "parallel" auszuführen. Jedem Prozess werden die Betriebsmittel für eine bestimmte Zeit abwechslungsweise zur Verfügung gestellt. Die Reihenfolge von Prozessen bzw. die Strategie für die Bereitstellung von Betriebsmitteln kann unterschiedlich sein:

- First-Come First-Served (bzw. First In First Out: FIFO)
- Shortest Job First
- Prioritätsscheduling
- Round Robin

Wenn der Scheduler einem Prozess die Betriebsmittel entzieht und sie einem anderen Prozess zuweist, muss der Prozess-Kontext des auszulagernden Prozesses gerettet werden, um ihn beim neuen Erhalten der Betriebsmittel in identischer Form herstellen zu können. So merkt der Prozess vom Entzug der Betriebsmittel überhaupt nichts. Jede Prozesswechsel ist für das Betriebssystem (Scheduler) eine aufwendige Angelegenheit, die Zeit braucht.



Leichtgewichtige Prozesse – Threads

Ein klassischer Prozess (Betriebssystem-Prozess) stellt sowohl für das Memory-Management als auch für Scheduling eine Einheit dar und wird als ein schwergewichtiger Prozess bezeichnet. Moderne Betriebssysteme haben neben Prozessen auch Threads:

- in einem Prozess können mehrere Threads laufen
- der Kontextwechsel eines Threads ist weniger aufwendig, da nicht alles ausgelagert werden muss

Ein Thread wird als leichtgewichtiger Prozess bezeichnet.

Betriebsmittel eines Threads

Ein neuer Thread braucht:

- einen eigenen Stack (lokale Variablen, Rücksprungadressen)
- einen eigenen Befehlszeiger (Instruction-Pointer)
- einen eigenen Stackzeiger (Stack-Pointer)
- einen Satz von Register

Gemeinsam für alle Threads eines Prozesses sind:

- Programmcode
- Prozessspezifische Daten
- Datei-Informationen
- "globale" Daten

Threadzustände und Zustandsübergänge

Ein Thread hat folgende Zustände:

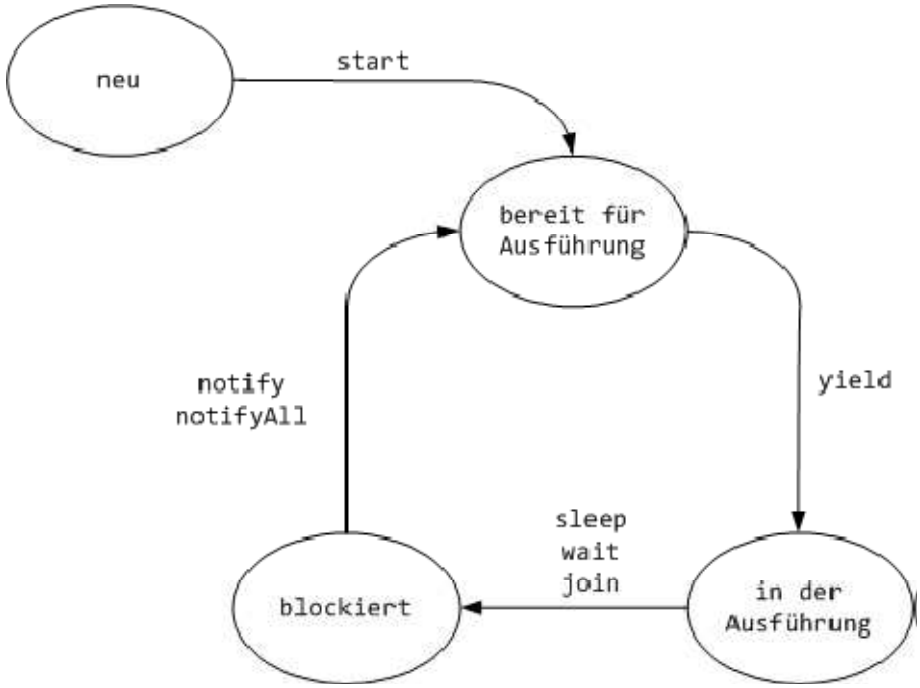
- neu
- bereit für die Ausführung
- in der Ausführung
- blockiert
- tot

Die Zustandsübergänge können durch entsprechende Methodenaufrufe (z. B. *sleep*) im Programm und durch das Betriebssystem (in Java JVM) hervorgerufen werden.

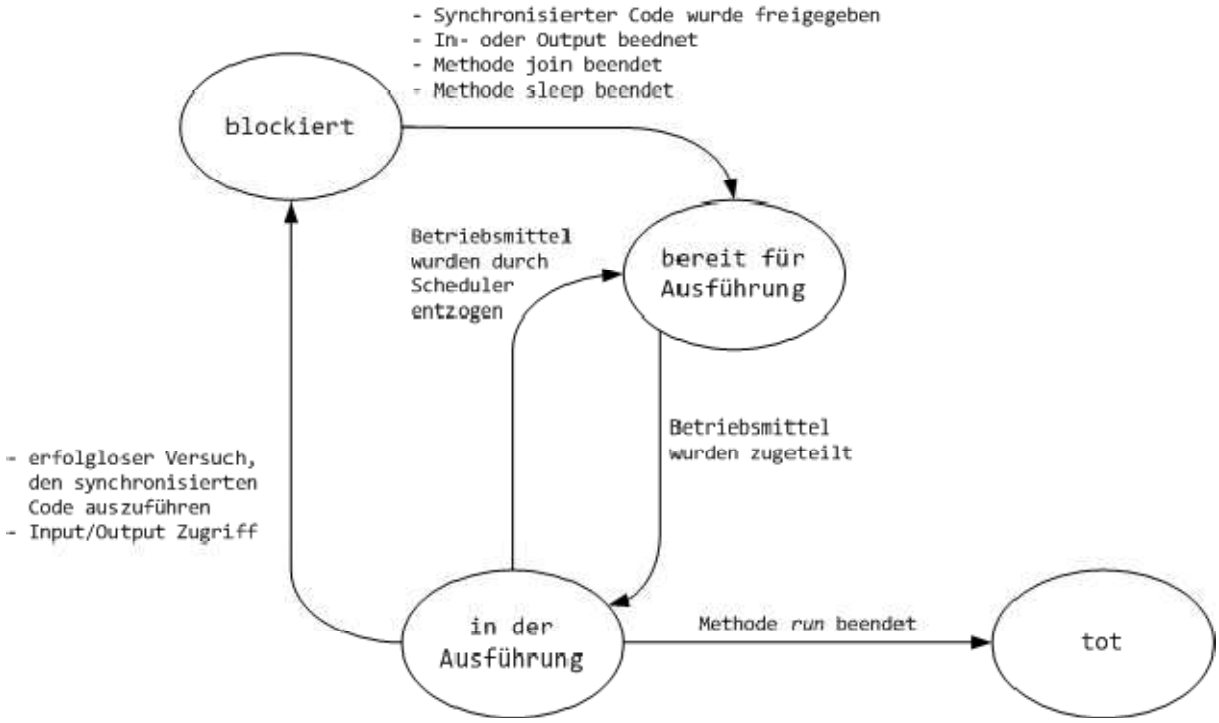
Der Zustand **neu** bedeutet dass der Thread durch den **new**-Operator erzeugt wurde und sich in seinem Anfangszustand befindet. Der Thread ist noch nicht lauffähig aber seine Datenfelder und Methoden können angesprochen werden.

Der Zustand **tot** bedeutet dass der Thread seine Arbeit erledigt hat. Seine Datenfelder und Methoden können angesprochen werden, ausser die Methode **run().** Ein Thread, welcher sich im Zustand tot befindet, kann nicht erneut gestartet werden (es wird eine Ausnahme geworfen).

Zustandsübergänge als Folge von Methodenaufrufen



Zustandsübergänge hervorgerufen durch VM



Threads in Java

Java unterstützt das Thread-Konzept. Falls das Betriebssystem das Thread-Konzept nicht unterstützt, kann dies von der JVM übernommen werden (ist allerdings weniger effizient). Falls das Betriebssystem das Thread-Konzept unterstützt, kann bei der Verwaltung von Threads direkt auf die Funktionalitäten des Betriebssystems zugegriffen werden (es hängt von der JVM-Implementierung ab).

Für den Umgang mit Threads stellt Java die Klasse **java.lang.Thread** und die Schnittstelle **java.lang.Runnable** zur Verfügung. Des Weiteren steht auch das Monitor-Konzept zur Verfügung, um den Zugriff auf gemeinsame Ressourcen sicher zu machen (synchronized)

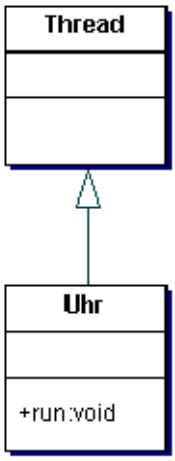
Ein Objekt kann auf zwei Arten threadfähig gemacht werden:

1. die generierende Klasse wird von der Klasse *Thread* abgeleitet, oder
2. die generierende Klasse implementiert die Schnittstelle *Runnable*

So kann sichergestellt werden, dass Instanzen einer Klasse als eigenständige Ausführungseinheiten (Threads) ausgeführt und vom Scheduler verwaltet werden.

Direktes Ableiten von der Klasse Thread

```
import java.text.SimpleDateFormat;

public class Uhr extends Thread
{
    public void run( )
    {
        SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss:SSS");

        while (true)
        {
            try
            {
                System.out.println("Zeit: " + sdf.format(new java.util.Date())
                Thread.sleep(1000);
            }
            catch (InterruptedException e)
            {
                // Ausnahmebehandlung ...
            }
        }
    }
}
```

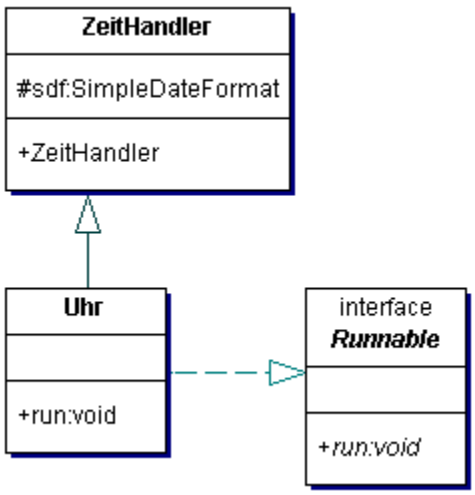


Thread-Ausführung starten

```
public class TestClass
{
   public static void main(String[] args)
   {
      // Klasse Uhr instanzieren
      Uhr timeThread = new Uhr();

      // Thread-Ausfuehrung starten
      timeThread.start();
   }
}
```

Es kann sein, dass eine Klasse nicht von der Klasse *Thread* abgeleitet werden kann. Diese Klassen müssen die Schnittstelle *Runnable* implementieren!!

Schnittstelle Runnable



Superklasse ZeitHandler:

```
import java.text.SimpleDateFormat;

public class ZeitHandler
{
   protected SimpleDateFormat sdf = null;

   public ZeitHandler()
   {
      sdf = new SimpleDateFormat("dd.MM.yyyy 'at' hh:mm:ss");
   }
}
```

Unterklasse Uhr:

```
public class Uhr extends ZeitHandler implements Runnable
{
   public void run()
   {
      while (true)
      {
         try
         {
            String dateAsString = sdf.format(new java.util.Date());
            System.out.println("Date & Time: " + dateAsString);
            Thread.sleep(1000);
         }
         catch (InterruptedException e)
         {
            // Ausnahmebehandlung …
         }
      }
   }
}
```

Thread Ausführung starten:

```
public class TestClass
{
   public static void main(String[] argv)
   {
      // Ein Runnable-Objekt erzeugen
      Uhr runnableObj = new Uhr();

      // Einen Thread erzeugen, wobei dem Konstruktor als
      // Parameter ein Runnable-Objekt uebergeben wird
      Thread timeObj = new Thread(runnableObj);

      // Ausfuehrung starten
      timeObj.start();
   }
}
```

Beenden der Ausführung:

Durch Aufruf der Methode *interrupt*, welche die Ausnahme *InterruptedException* auslöst. In der Ausnahme-Behandlung kann das Verlassen der Methode *run* beantragt werden (**return**).

Methode Interrupt aufrufen

```
public class TestClass
{
   public static void main(String[ ] args)
   {
      // Klasse Uhr instanzieren
      Uhr timeThread = new Uhr();

      // Thread-Ausfuehrung starten
      timeThread.start();

      try
      {
         // Haupt-Thread 30 Sekunden schlaffen lassen
         Thread.sleep(30000);

         // Den timeThread stoppen
         timeThread.interrupt();
      }
      catch(InterruptedException e)
      {
         // Ausnahmebehandlung …
      }
   }
}
```

Ausnahme InterruptedException "verwerten":

```
public class Uhr extends Thread
{
   public void run()
   {
      java.text.SimpleDateFormat sdf = new
         java.text.SimpleDateFormat("hh:mm:ss:SSS");

      while (true)
      {
         try
         {
            String dateAsString = sdf.format(new java.util.Date());
            System.out.println("Zeit: " + dateAsString);
            Thread.sleep(1000);
         }
         catch (InterruptedException e)
         {
            // Ausfuehrung der Methode 'run' beenden
            return;
         }
      }
   }
}
```

Privates Feld als Kommunikationsmittel

```
public class TestClass
{
   public static void main(String[ ] args)
   {
      // Klasse Uhr instanzieren
      Uhr timeThread = new Uhr();

      // Thread-Ausfuehrung starten
      timeThread.start();

      try
      {
         // Haupt-Thread 30 Sekunden schlaffen lassen
         Thread.sleep(30000);

         // Den timeThread stoppen
         timeThread.beenden();
      }
      catch(InterruptedException e)
      {
         // Ausnahmebehandlung ...
      }
   }
}


public class Uhr extends Thread
{
   private boolean beendet = false;

   public void run ()
   {
      java.text.SimpleDateFormat sdf = new
         java.text.SimpleDateFormat("hh:mm:ss:SSS");

      while (! beendet)
      {
         try
         {
            String dateAsString = sdf.format(new java.util.Date());
            System.out.println("Zeit: " + dateAsString);
            Thread.sleep(1000);
         }
         catch (InterruptedException e)
         {
            // Ausnahmebehandlung …
         }
      }
   }

   // Methode beenden
   public void beenden() {  beendet = true; }
}
```

Zugriff auf gemeinsame Ressourcen und Synchronisation

Falls Threads auf gemeinsame Ressourcen (Ausgabe- Geräte, Daten etc.) zugreifen, kann es zu Problemen kommen. Beispiel:

Annahme:

```
Hans muss nach der Schule mit dem Auto jemanden vom Flughafen abholen.
```

Zeit	Hans	Monika
07:00	Geht mit dem Fahrrad in die Schule.	Schläft wie ein Engel.
09:30		Wird wach, steht auf und fragt sich: "Was mache ich heute?"
10:30		Entscheidet sich, zu einer Kollegin zu fahren, die heute auch frei hat.
11:00		Verlässt das Haus, nimmt das Auto und fährt zu ihrer Kollegin weg.
14:30	Kommt nach Hause, isst etwas und zieht sich schnell um.	
15:00	Verlässt das Haus und geht mit dem Autoschlüssel in die Garage.	
15:05	Stellt mit Erschrecken fest, dass das Auto nicht in der Garage ist.	
21:30		Kommt nach Hause.
21:31	Was hier passiert, wird der Fantasie des Lesers überlassen!	

Die parallele Ausführung von Prozessen bzw. Threads kann zu ähnlichen Problemen führen, wenn Threads gemeinsame Ressourcen benutzen (in unserem Beispiel: das Auto). Da Zugriff auf gemeinsame Ressourcen nicht koordiniert erfolgt, kann ein Prozess bzw. Thread die inkonsistente Sicht der Daten erhalten (in unserem Beispiel findet Hans die leere Garage vor). Das Ergebnis hängt vom zeitlichen Ablauf (in unserem Beispiel kommt Hans nach 11:00 in die Garage).

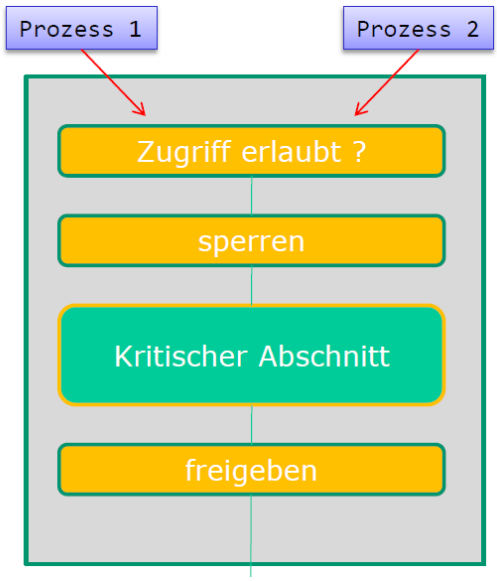
Falls ein Prozess / Thread auf gemeinsame Ressourcen zugreift und sie manipuliert, nennt man das ***race condition*** (auf Deutsch: *Wettkampfbedingung*).

Der Abschnitt, in dem die gemeinsamen Daten manipuliert werden, wird ***kritischer Abschnitt*** genannt.

Der Zugriff auf einen solchen kritischen Abschnitt muss gegenseitig ausschliessend (***mutual exclusion***) erfolgen: zu einem Zeitpunkt darf sich nur ein Prozess im kritischen Abschnitt befinden.

Vor dem Eintritt in einen solchen kritischen Abschnitt muss der Prozess um "Erlaubnis fragen".

- Eingang-Sektion:
  - Eintrittserlaubnis
  - Eintritt sperren
- Kritischer Abschnitt
  - Zugriff auf gemeinsame Ressourcen
- Ausgang-Sektion
  - Eintritt freigeben



Das Monitor-Konzept

Synchronisation mit Hilfe von Semaphoren

- wird schnell unübersichtlich und
- ist sehr fehleranfällig (kann schnell zu Deadlocks führen)

In Java wird das Monitorkonzept unterstützt, das im Jahr 1974 von Hoare als Synchronisationsmittel eingeführt wurde. Es handelt sich um ein Hochsprachenkonstrukt, dass allerdings einfach zu handhaben ist.

Wesentliche Eigenschaften eines Monitors:

- Kritische Abschnitte, die auf denselben Daten arbeiten, sind Methoden eines Monitors
- Ein Prozess betritt einen Monitor durch Aufruf einer Methode des Monitors

Zu einem gleichen Zeitpunkt kann sich nur ein Prozess in einem Monitor befinden. Jeder andere Prozess, der eine der Monitormethoden aufruft, wird suspendiert und muss in einer Queue warten, bis der Monitor vom Prozess, welcher sich im Monitor befindet, verlassen wird.

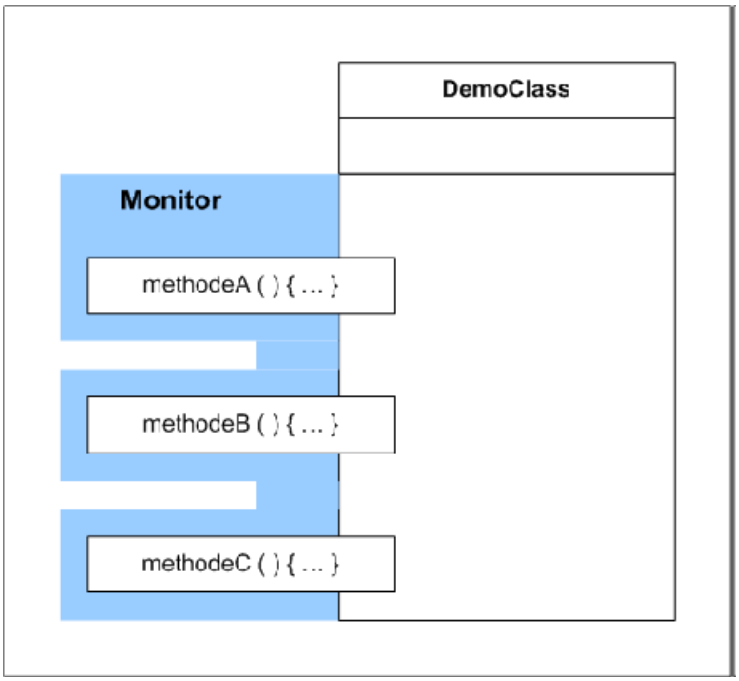
Das Monitorkonzept in Java wird mit dem Schlüsselwort **synchronized** umgesetzt. Mit dem Schlüsselwort **synchronized** kann der wechselseitiger Ausschluss sowohl für Methoden als auch für einzelne Codeblöcke realisiert werden. Falls nötig, kann der Zugriff auf alle Methoden synchronisiert werden (z. B. zugriff auf Collections).

Monitor für synchronisierte Klassenmethoden

Wenn eine oder mehrere Klassenmethoden mit dem Schlüsselwort *synchronized* versehen werden, wird ein Monitor um diese Methoden "herum gebaut". Die synchronisierten Klassenmethoden werden dadurch die Methoden des Monitors.

Zu einem bestimmten Zeitpunkt kann nur ein Thread eine der Methoden des Monitors (*synchronisierte Klassenmethoden*) ausführen. Für alle synchronisierten Klassenmethoden einer Klasse wird ein Monitor angelegt, welcher den Zugriff auf diese Methoden überwacht (ein Monitor pro Klasse).

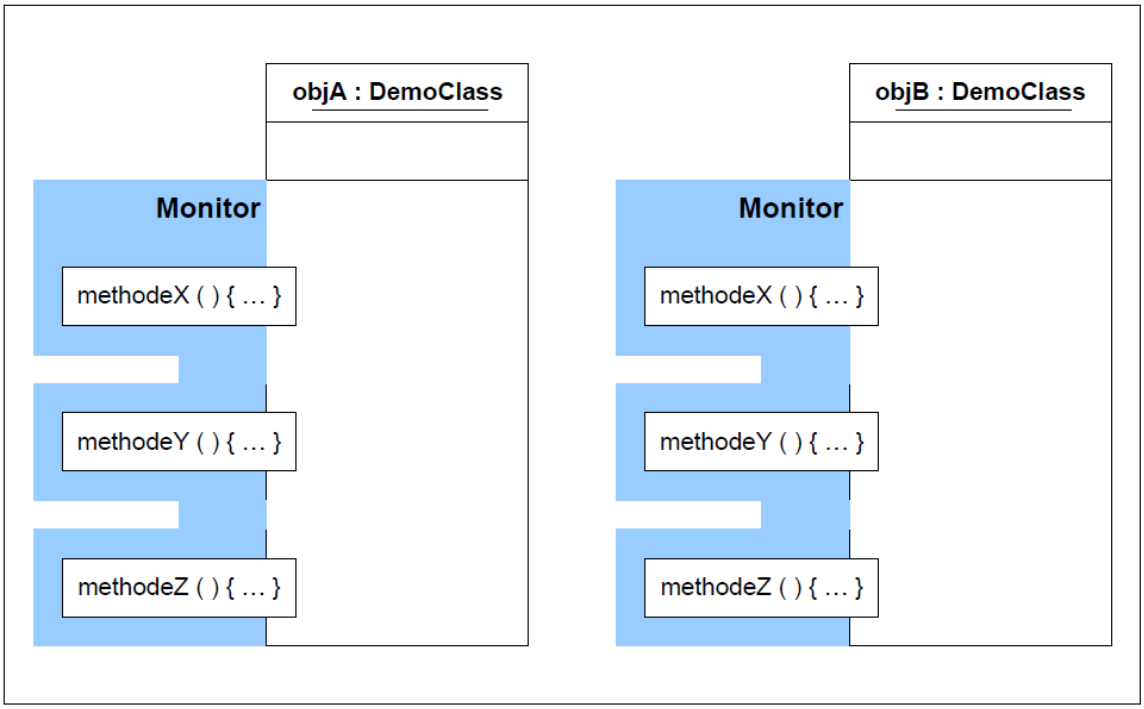
```
public class DemoClass{
    public static synchronized void methodeA() {
        // kritischer Abschnitt
    }

    public static synchronized void methodeB() {
        // kritischer Abschnitt
    }

    public static synchronized void methodeC() {
        // kritischer Abschnitt
    }

    // Weitere konkrete Methoden …
}
```



Monitor für Instanzmethoden

Falls eine oder mehrere Instanzmethoden mit dem Schlüsselwort *synchronized* versehen werden, so hat jede Instanz dieser Klasse einen eigenen Monitor, welcher den Zugriff auf die synchronisierten Instanzmethoden überwacht. Somit ist für die synchronisierten Instanzmethoden einer Klasse ein Monitor pro Instanz der Klasse vorhanden.

```
public class DemoClass{
    Public synchronized void methodeX() {
        // kritischer Abschnitt
    }

    Public synchronized void methodeY() {
        // kritischer Abschnitt
    }

    Public synchronized void methodeZ() {
        // kritischer Abschnitt
    }

    // Weitere konkrete Methoden …
}
```



Synchronisieren von einzelnen Codeblöcken und Schlüssel

In Java ist es möglich, die einzelnen Codeblöcken in unterschiedlichen Methoden zu synchronisieren:

```
public class Demo {
    public void doSomething() {
        // Code (nicht kritisch )

        synchronized (this) {
            // kritischer Abschnitt

        }

        // weiterer Code …
    }
}
```

Für das Synchronisieren eines Codeblocks wird ein Objekt als Schlüssel benötigt. Zu einem bestimmten Zeitpunkt kann ein Schlüssel für den Zugriff auf nur einen synchronisierten Block verwendet werden.

Mit einem Schlüssel kann der Zugriff auf mehreren Codeblöcke synchronisiert werden: die Schlüsselbrett-Rolle übernimmt der Monitor. Die Anzahl Monitore in einem Objekt stimmt mit der Anzahl der für die Synchronisation verwendeten Schlüssel überein.

Welches Objekt eignet sich als Schlüssel?

- Instanz der Klasse Class
- Ein übergebenes Schlüssel-Objekt
- Die Referenz auf das aktuelle Objekt (this)
- …

Wichtig:

Der Schlüssel, der in der Regel von mehreren Instanzen gebraucht wird, darf **nur einmal** Vorkommen!!!

Paket java.util.concurrent

Stellt zusätzliche Konstrukte für das threadsichere Arbeiten mit Objekten.

- Interfaces:
  - BlockingQueue
  - ConcurrentMap
  - …
- Klassen:
  - ArrayBlockingQueue
  - ConcurrentLinkedQueue
  - SynchronousQueue
  - ConcurrentHashMap
  - Semaphore

Interprozess-Kommunikation (IPC)

Eine Applikation besteht aus einem oder mehreren Prozessen. Prozesse sind Objekte des Betriebssystems und ermöglichen einer Anwendung den sicheren Zugriff auf die Ressourcen des Betriebssystems. Sie laufen in eigenem Speicherbereichen und sind dadurch voneinander isoliert. Für den Austausch von Daten zwischen Prozessen ist eine **Interprozesskommunikation** nötig!

IPC basiert auf dem Austausch von Nachrichten und Daten zwischen Prozessen:

- ein Prozess sendet die Nachricht bzw. die Daten (Sender)
- ein anderer Prozess empfängt die Nachricht bzw. die Daten (Empfänger)

IPC kann unterschiedlich umgesetzt werden:

- über Dateien
- über Pipes (nur lokal)
- über Sockets (sowohl lokal als auch entfernt)
- mit Hilfe einer Middleware

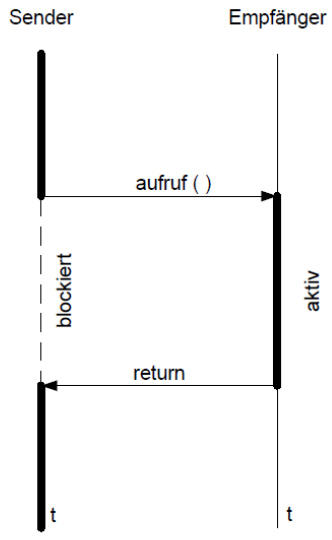
Kommunikationsmodelle

Das Kommunikationsmodel legt das Protokoll für den Ablauf der Kommunikation zwischen Prozessen (IPC) fest.

Synchrone Kommunikation:

- Wenn der Sender-Prozess eine Nachricht gesendet hat, muss er auf die Antwort warten.
- Wenn der Empfänger-Prozess ein "Empfangen" ausführt, wartet er so lange, bis eine Nachricht (eine Anfrage) empfangen wurde.

- Vorteile
  - Eher einfacher zum Implementieren
  - Synchronisation beim Zugriff auf gemeinsame Ressourcen gleich erledigt
- Nachteile
  - zum Teil ineffizient (das Warten)
  - braucht sichere und schnelle Netzwerkverbindung
  - Empfängerprozess muss verfügbar sein
  - enge Kopplung zwischen Sender und Empfänger



Synchrone Kommunikation

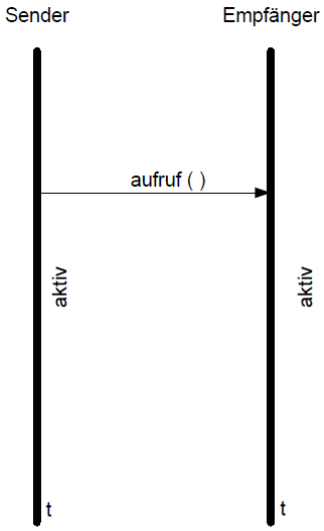
Asynchrone Kommunikation:

Der Sender-Prozess kann nach dem Senden der Nachricht sofort weiter arbeiten, ohne auf Antwort des Empfängers zu warten. Der Empfänger-Prozess kann sowohl blockierend als auch nicht-blockierend Nachrichten empfangen

Die Zustellung der Antwort:

- Der Empfänger wird aktiv und stellt sie dem Sender bei Gelegenheit asynchron zu.
- Der Sender holt sich bei Gelegenheit die Antwort selber vom Empfänger ab. Realisierung mittels Warteschlangen

- Vorteile
  - lose Kopplung von Prozessen
  - geringere Fehlerabhängigkeit
  - der Empfänger muss nicht empfangsbereit sein
  - effizienter (kein Warten)
- Nachteile
  - aufwendiger bzw. komplizierter in der Implementierung (Warteschlangen)
  - komplizierte Protokolle
  - nicht immer sinnvoll



Asynchrone Kommunikation

Datenrepräsentation

Ein verteiltes System läuft in der Regel in einer heterogenen Umgebung:

- unterschiedliche Hardwarearchitekturen
- unterschiedliche Betriebssysteme (Plattformen)
- unterschiedliche Programmiersprachen

Damit die Komponenten des verteilten Systems miteinander kommunizieren können, müssen sie alle ein gemeinsames Format für Daten benutzen. Jede Übermittlung (Senden / Empfangen) von Daten ist mit einer Übersetzung von Daten aus dem lokalen ins gemeinsame Format (und umgekehrt) verbunden.

Marshalling:

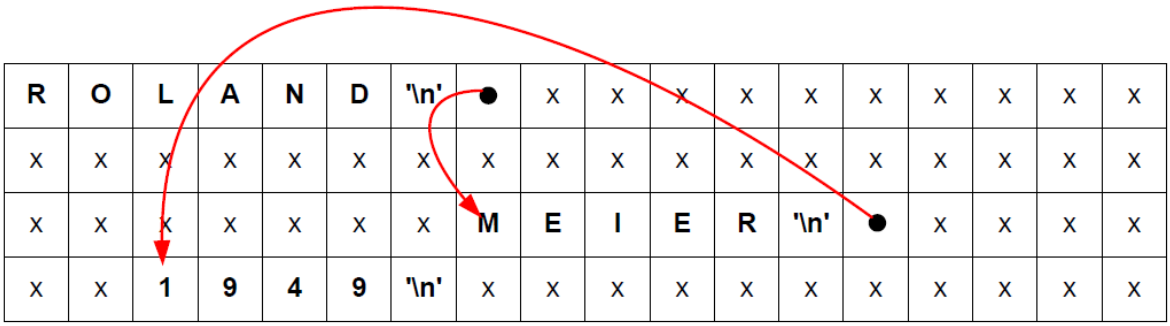
Unter ***marshalling*** versteht man die Transformation einer beliebigen Nachricht in eine übertragbare Nachricht, wobei die Datenstruktur in eine zusammenhängende Nachricht "planiert" wird und die zu sendenden Daten aus dem lokalen in das gemeinsame Format übersetzt werden. Daraus resultiert ein Strom von *bytes*, welcher übertragen werden kann: eine übertragbare Nachricht.

- Eine Nachricht (Datenstruktur) steht zusammenhängend im Speicher:

H	A	N	S	'\n'	P	O	R	T	M	A	N	N	'\n'	1	9	3	8	'\n'

Da es sich um eine zusammenhängende Folge von bytes handelt, kann diese Nachricht so übertragen werden, wie sie im Speicher steht.

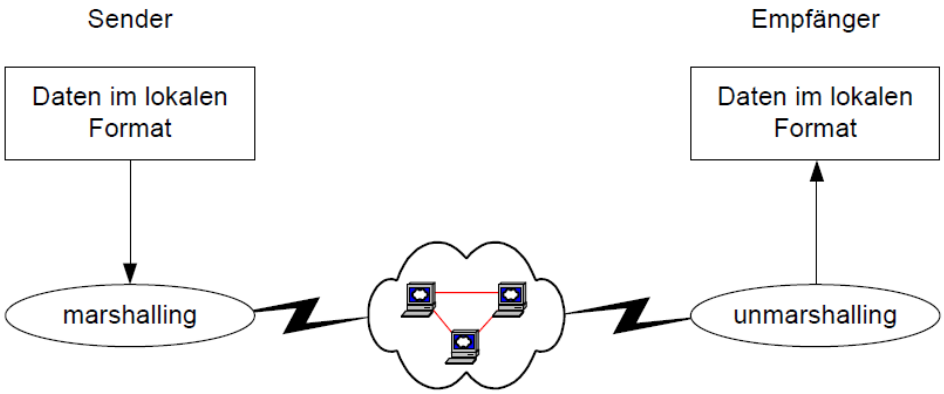
- Eine Nachricht (Datenstruktur) kann auch über den Speicher verteilt sein



Es ist eine nicht-zusammenhängende Folge von *bytes*, die in dieser Form **nicht** übertragen werden kann.

Unmarshalling

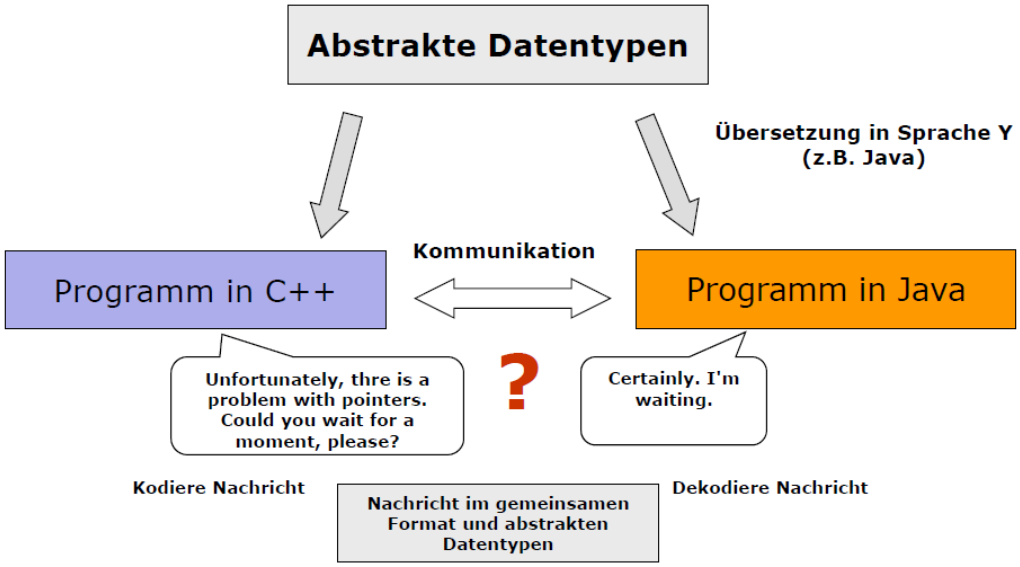
Auf der Seite des Empfängers müssen die empfangenen Daten aus dem *gemeinsamen* Format in das *lokale* Format des Empfängers übersetzt werden. Dieser Prozess wird ***unmarshalling*** genannt. Unmarshalling wird immer auf der Seite des Empfängers ausgeführt.



Es gibt mehrere Ansätze für ein gemeinsames Netzwerk-Datenformat. Die Grundidee:

- Es soll eine Menge von abstrakten Datentypen und eine Codierung für jeden dieser Datentypen definiert werden.
- Es sollen Werkzeuge für die automatische Überführung von abstrakten Datentypen in die Datentypen einer Programmiersprache und umgekehrt zur Verfügung gestellt werden.
- Es sollen Operationen zur Verfügung gestellt werden, welche die Daten aus dem lokalen ins gemeinsame Datenformat (und umgekehrt) übersetzen.

Wenn ein bestimmter Datentyp übertragen werden soll, wird die Codieroperation aufgerufen (die eine flache Struktur erzeugt) und das Ergebnis der Codierung in die Nachricht passend eingebettet und übertragen. Der Empfänger decodiert den empfangenen byte-Strom und erzeugt neue, lokale Repräsentation des empfangenen Typs.



Realisierung der IPC

Direkte Netzwerkprogrammierung mit Sockets

Setzt auf der Transportschicht (TCP/UDP) auf.

Vorteile:

- direkte Kontrolle aller Transportparameter
- grössere Flexibilität bei der Entwicklung neuer Protokolle
- bessere Performance

Nachteile:

- die Datenrepräsentation
- wenig komfortable Entwicklung von Anwendungen

Netzwerkprogrammierung mit Middleware

Middleware ist eine zusätzliche Schicht, die sich zwischen Transportschicht (TCP/UDP) und Anwendung befindet.

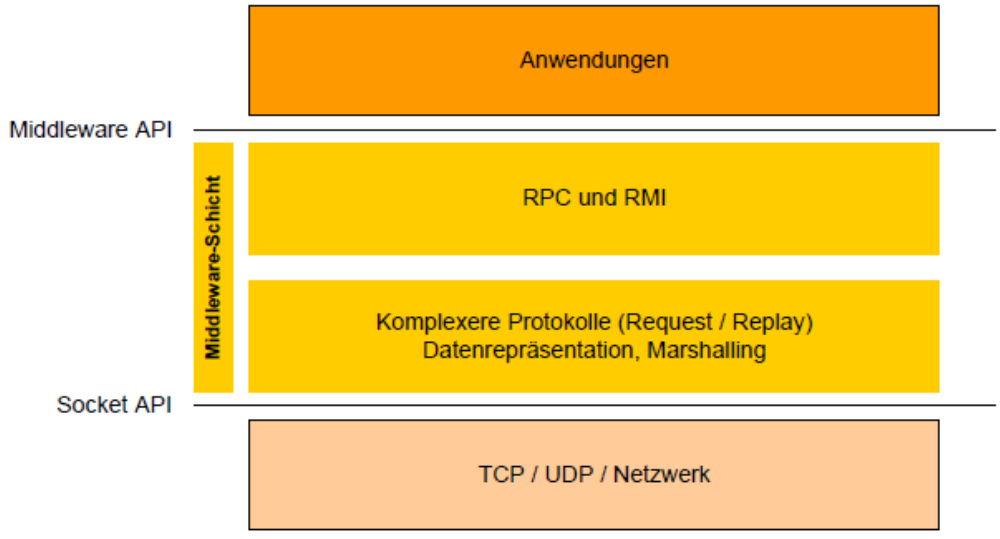
Vorteile:

- Bequeme Entwicklung von Anwendungen
- Lösung des Datenrepräsentationsproblems
- Evtl. weitere Dienste „von Haus aus“ verfügbar

Nachteile:

- Ein (unter Umständen grosser) Overhead muss in Kauf genommen werden
- Schlechtere Performance im Vergleich zu Socket-Programmierung

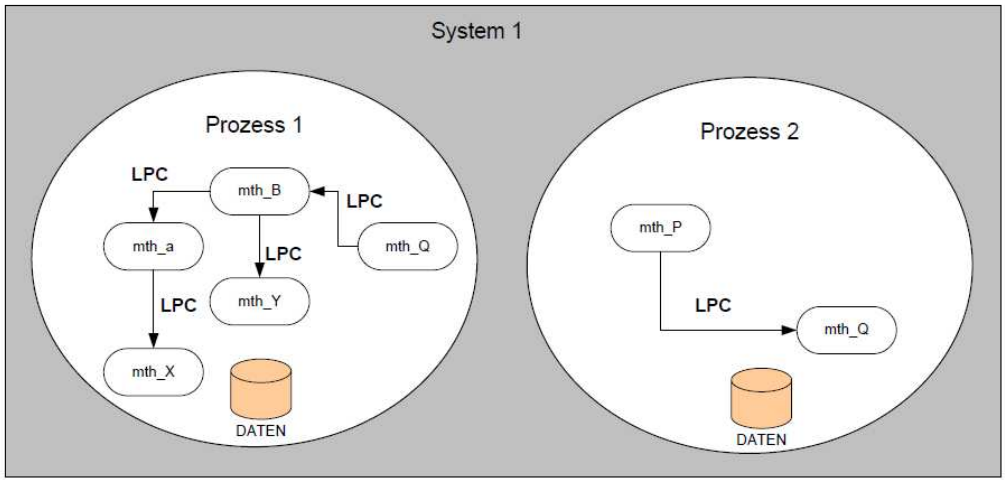
Schichten des Kommunikationssystem



Local- und Remote-Procedure-Call

Local Procedure Call (LPC)

Aufruf einer Prozedur, welche sich im gleichen Speicherraum (Prozess) befindet. Es ist Keine Interprozesskommunikation mit LPC möglich.



Vorteile:

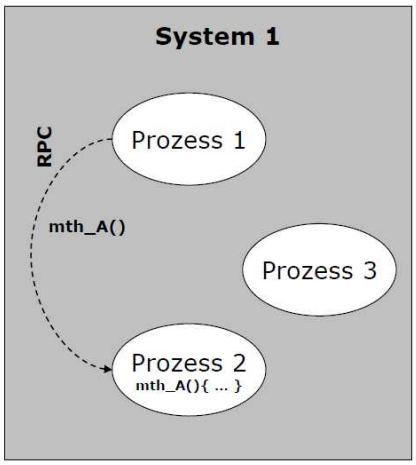
- Bessere Performance, da alle Prozeduren im gleichen Prozess bzw. Speicherraum
- Zuverlässige Kommunikation
- Einfachere Parameterübergabe, da lokaler Stack für alle Prozeduren gemeinsam und verfügbar
- Einfacher Zugriff auf gemeinsame Daten, da alle Prozessdaten gemeinsam

Nachteile:

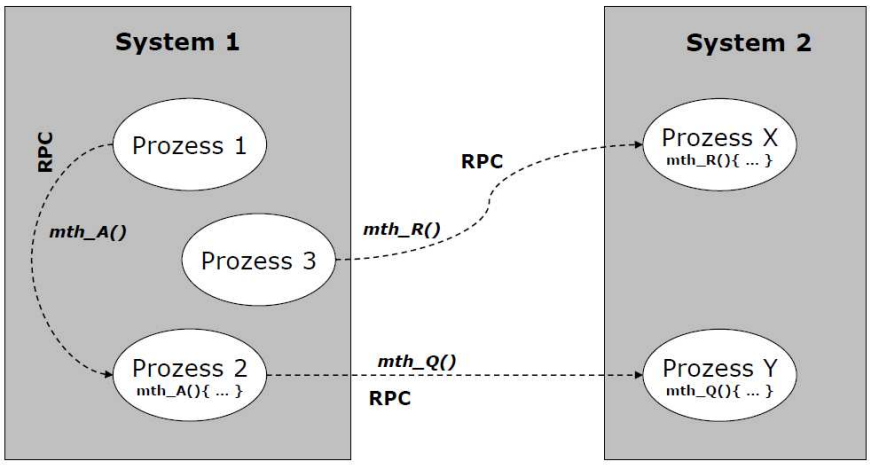
- Keine Interprozesskommunikation möglich

Remote Procedure Call (RPC)

RPC steht für den "entfernten Aufruf von Prozeduren„. Mit "entfernt" ist eine Prozedur gemeint, welche in einem anderen Prozess ausgeführt wird bzw. sich in einem anderen Speicherraum befindet. RPC stellt eine Art der Interprozesskommunikation dar. Die kommunikationswilligen Prozesse können auf dem gleichen System oder auf unterschiedlichen Systemen ausgeführt werden.



- Auf dem System 1 laufen drei Prozesse
- Jeder Prozess läuft in seinem eigenen (getrennten) Speicherraum
- Prozess 3 kommuniziert mit keinem weiteren Prozess
- Prozess 1 kommuniziert mit dem Prozess 2, indem er seine **mth_A** aufruft
- Es handelt sich um einen entfernten Aufruf (RPC) auf einem gleichen System



LPC VS. RPC

LPC ist einfacher zu implementieren, da sich die ganze Kommunikation (Nachrichtenaustausch) in einem gleichen Speicherraum (gleicher Prozess) abspielt.

RPC ist weniger performant:

- Kommunikation erfolgt in Form von Nachrichten, welche von einem zu anderem Prozess übertragen werden
- Parameter, welche übergeben werden müssen, müssen in die Nachricht passend verpackt werden

RPC ist komplexer in der Implementierung, da zusätzliche Protokolle für das Senden von Nachrichten benötigt!!

Anfrage- und Antwort-Nachricht

Die Struktur der Anfrage- bzw. Anwort-Nachricht muss "abgesprochen" werden und darf danach nicht ohne Rücksprache mit der "Gegenseite" geändert werden. Damit soll sichergestellt werden, dass die Kommunikation zwischen beteiligten Komponenten exakt und ohne Missverständnisse realisiert werden kann. Die Rede ist von Request- und Reply-Protokoll:

Request-Protokoll (Anfrage-Nachricht)

- definiert, wie die Anfrage-Nachricht aufgebaut werden muss
- enthält alle für den Aufruf der entferenten Operation relevanten Informationen (den Namen der Methode und die Parmeterliste) in einer passenden Form
- hängt im Aufbau (Struktur) von der Operation ab, die aufgerufen werden soll

Reply-Protokoll (Antwort-Nachricht):

- definiert, wie die Antwort-Nachricht aufgebaut werden soll
- enthält alle für die Antwort relevanten Informationen in einer passenden Form:
  - den Rückgabewert, der von der aufgerufenen Operation retourniert wird
  - die Informationen über die Ausnahme, die evtl. geworfen werden könnte
- ist im Aufbau vom Rückgabewert und Ausnahme, die geworfen werden könnte, abhängig

Pro Operation, die als entfernte Operation aufgerufen werden soll, muss je ein *Request*- und *Reply-Protokoll* definiert werden. Erst nach dem die Request- und Reply-Protokoll definiert sind, kann mit der Implementierung der Kommunikation gestartet werden.

Beispiel:

Entfernte Operation:

```
public boolean login (String username, String password) throws
Exception
```

Request-Protokoll:

Reply-Protokoll:

Direkte Netzwerk Programmierung mit Sockets

Socket: Begriff und Definition

Socket ist ein Endpunkt in der Kommunikationsverbindung zwischen zwei kommunikationswilligen Prozessen. Socketarten:

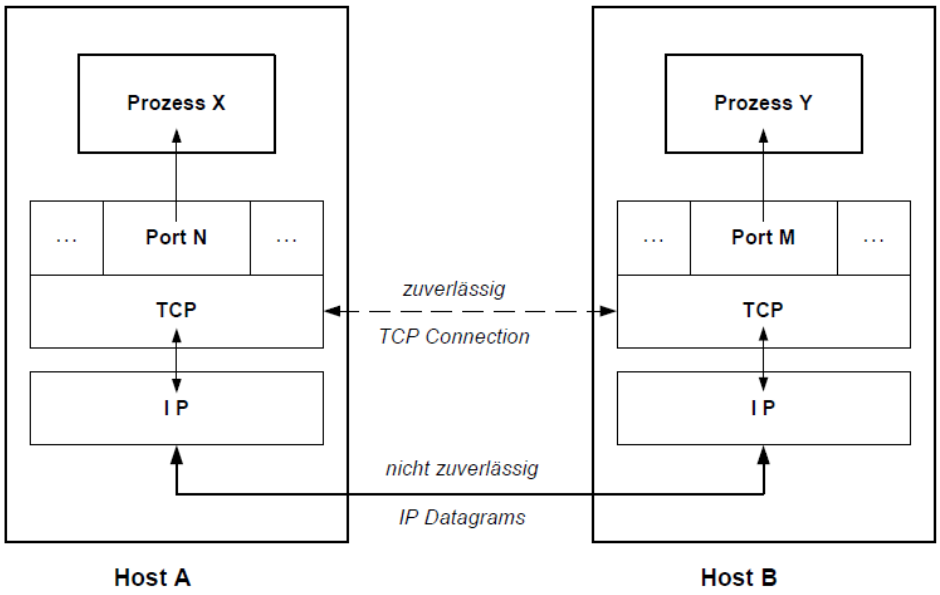
- Streamsockets (auch TCP-Sockets genannt)
- Datagramsockets (auch UDP-Sockets genannt)

Für die Herstellung der Verbindung:

- Host-Adresse (IP-Adresse)
- Port-Nummer

Durch die IP-Adresse und den Port wird ein Dienst auf einem Rechner eindeutig identifiziert. Auf einem Rechner können mehrere Sockets gleichzeitig offen sein.

Datenübertragung mit Sockets



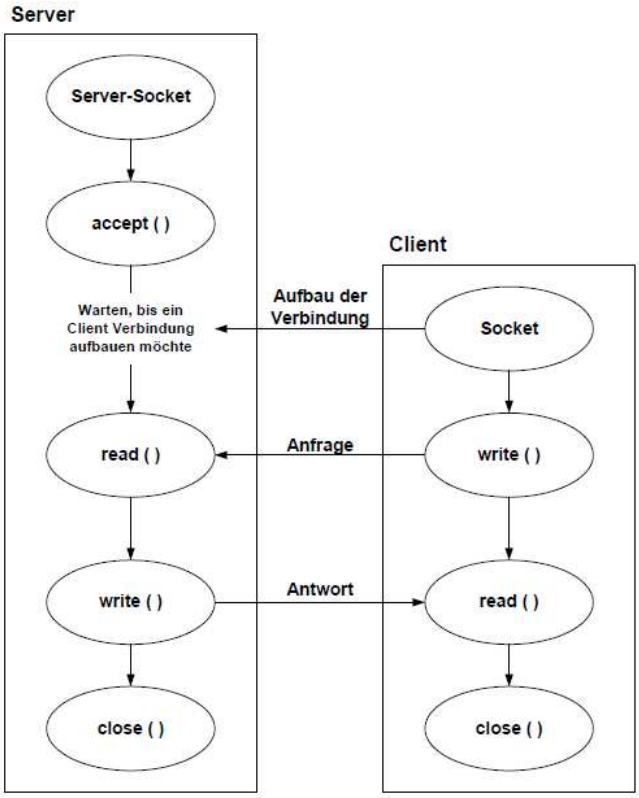
TCP-Sockets

TCP-Sockets sind verbindungsorientiert (TCP). Sie stellen Verbindungen zwischen zwei Programmen (Prozessen) her, welche sowohl auf einem gleichen als auch auf unterschiedlichen Rechnern ausgeführt werden können. Phasen der Datenübertragung:

- Aufbau der Verbindung
- Übertragung von Daten
- Abbau der Verbindung

Daten, welche übertragen werden, werden in einen Datenstrom geschrieben, und am Zielort aus dem Datenstrom (in gleicher Reihenfolge) gelesen. Es besteht keine Begrenzung bezüglich Grösse der übertragenen Daten

Socket-Implementierung in Java



Die Server-Anwendung muss einen Server-Socket erzeugen.

Der Server-Socket wird an einen vorgegebenen Port gebunden.

Der Server-Socket wartet, bis ein Client eine Verbindung aufbauen möchte.

Die Client-Anwendung erzeugt einen Client-Socket und versucht, eine Verbindung zum Server aufzubauen.

Wenn die Verbindung vom Server akzeptiert wurde, können Server und Client Daten austauschen.

Für den Austausch von Daten müssen Server und Client eine "gleiche Sprache" sprechen (Protokoll)

TCP-Sockets

In Java stehen zwei Klassen für die Verwendung von TCP-Sockets zur Verfügung:

- **java.net.ServerSocket**
- **java.net.Socket**

Server-Anwendung arbeitet mit der Klasse *ServerSocket*, welche alle für einen Server notwendigen Funktionalitäten enthält. Die Client-Anwendung arbeitet mit der Klasse *Socket.*

Klasse java.net.ServerSocket

Konstruktoren:

- ServerSocket () throws IOException
- ServerSocket (int port) throws IOException
- ServerSocket (int port, int backlog) throws IOException

Wichtigste Methoden:

- void bind (SocketAdress endpoint) throws IOException
- Socket accept () throws IOException
- void close () throws IOException
- InetAdress getInetAdress () oder int getLocalPort ()

Klasse java.net.Socket

Konstruktoren:

- Socket ( )
- Socket (InetAdress adress, int port) throws IOException
- Socket (String host, int port) throws IOException

Wesentliche Methoden:

- void bind (SocketAdress bindpoint) throws IOException
- void close ( ) throws IOException
- void connect (SocketAdress endpoint, int timeout) throws IOException
- InetAdress getInetAdress ( )
- InputStream getInputStream ( ) throws IOException
- OutputStream getOutputStream ( ) throws IOException oder int getPort ( ) usw…

Auszug aus dem Server Code

```
// ServerSocket erzeugen
ServerSocket server = new ServerSocket(port);

// ClientSocket holen, wenn eine Verbindung gewuenscht wird
Socket client = server.accept();

// Informationen ueber den Client ausgeben
String hostName = client.getInetAddress().getHostName();
int p = client.getPort();
System.out.println("Verbindung mit: " + hostName + ", Port: " + p + "\n");

// InputStream vom Client holen
InputStream is = client.getInputStream();

// vom Client zugestellten Daten ausgeben
int c = 0;
while ((c = is.read()) != -1)
  { System.out.print((char) c); }
```

Auszug aus dem Client Code

```
// Socket erzeugen und Verbindung zum Server aufbauen
Socket socket = new Socket("localhost", 1001);

// Benachrichtigung
System.out.println("Verbindung mit Server hergestellt!");

// OutputStream vom Socket holen
OutputStream os = socket.getOutputStream();

// Meldung dem Server senden
String msg = "Das ist eine Test-Meldung!";
os.write(msg.getBytes());

// Verbindung schliessen
socket.close();
```

UDP Sockets

UDP-Sockets sind nicht verbindungsorientiert. Daten werden in Pakete geschrieben. Datenpakete können maximal 8 KB gross sein (Header- und Nutzdaten zusammen). Grössere Nachrichten müssen in der Anwendung segmentiert bzw. reassembliert werden. Die Ankunft von Datenpaketen wird nicht bestätigt. Es kann passieren, dass manche Datenpakete nicht ans Ziel kommen.

In Java stehen folgende Klassen für die Verwendung von UDP-Sockets zur Verfügung:

- **java.net.DatagramSocket**
- **java.net.DatagramPacket**

Die Klasse *DatagramSocket* stellt den Socket zur Verfügung. Die Klasse *DatagramPacket* enthält die zu sendenden Daten.

Klasse java.net.DatagramSocket

Konstruktoren:

- DatagramSocket( )
- DatagramSocket (int port)
- DatagramSocket (int port, InetAddress iadr)

Wesentliche Methoden:

- void bind (SocketAddres sadr)
- void receive (DatagramPacket p)
- void send (DatagramPacket p)
- void disconnect ( )

Klasse java.net.DatagramPacket

Ein DatagramPacket-Objekt enthält die zu sendenden Informationen.

Konstruktoren:

- DatagramPacket (byte[] buf, int length)
- DatagramPacket (byte[] buf, int length, InetAddres address, int port)

Wesentliche Methoden:

- void setData (byte[] buf)
- void setData(byte[] buf, int offset, int length)
- byte[] getData ( )
- void setLength (int length)
- int getLength ()

Auszug aus dem Server Code

```
DatagramSocket socket = null;
DatagramPacket req = null, res = null;
byte[] buf = new byte[508];

// UDP-Socket erzeugen
socket = new DatagramSocket(9001);

// Request-DatagramPacket erzeugen
req = new DatagramPacket(buf, buf.length);
while(true)
  {

  // Request empfangen (entgegennehmen)
  socket.receive(req);

  // Response erzeugen
  res = new DatagramPacket(req.getData(), req.getLength(),
  req.getAddress(), req.getPort());

  // Response senden
  socket.send(res);
  }
```

Auszug aus dem Client Code

```
String msg = "Das ist eine Test-Meldung!";
byte [] resBuf = new byte[508];
int port = 9001;
InetAddress server = InetAddress.getByName("localhost");

// Socket erzeugen
DatagramSocket socket = new DatagramSocket();

// Request erzeugen
DatagramPacket req = new DatagramPacket(msg.getBytes(), msg.length(),
server, port);

// Request senden
socket.send(req);
DatagramPacket res = new DatagramPacket(resBuf, resBuf.length);

// Response empfangen
socket.receive(res);
System.out.println("Antwort: " + new String(res.getData()));
```

Multicast-Sockets

Sollte eine gleiche Nachricht an mehrere Partner gesendet werden, wäre ein paarweiser Austausch von Nachrichten mit anderen Partnern ineffizient. In einem solchen Fall wird die Zustellung der Nachricht mit dem so genannten Multicast realisiert. Bei einem Multicast wird die Nachricht an eine Gruppe bzw. alle Mitglieder dieser Gruppe in einem Schritt gesendet. Für die Multicast-Kommunikation stellt Java die Klasse *java.net.MulticastSocket* zur Verfügung.

Klasse java.net.MulticastSocket

Konstruktoren:

- MulticastSocket(int port)

Methoden:

- void joinGroup(InetAddress multicastAddr)
- void leaveGroup(InetAddress multicastAddr)
- void send(DatagramPacket p)
- void receive(DatagramPacket p)

Auszug aus dem Sender Code

```
String msg = "Diese Meldung geht an alle Gruppenmitglieder!";
MulticastSocket s = null;
DatagramPacket mOut = null;
String mcIp= "230.2.2.2";
InetAddress group = InetAddress.getByName(mcIp);

// Socker erzeugen
s = new MulticastSocket(4004);

// Der Gruppe beitretten
s.joinGroup(group);

// Nachricht erzeugen
mOut = new DatagramPacket(msg.getBytes(), msg.length(), group, 4004);

// Nachricht senden
s.send(mOut);

// Gruppe verlassen
s.leaveGroup(group); if (s != null) s.close();
```

Auszug aus dem Receiver-Code

```
MulticastSocket s = null;
DatagramPacket mIn = null;
String mcIp= "230.2.2.2";
InetAddress group = InetAddress.getByName(mcIp);

// Socket erzeugen
s = new MulticastSocket(4004);

// Der Gruppe beitretten
s.joinGroup(group);
mIn = new DatagramPacket(buf, buf.length, group, 4004);

// Nachricht empfangen
s.receive(mIn);

// Nachricht ausgeben
System.out.println("Empfangen: " + new String(mIn.getData()));

// Gruppe verlassen
s.leaveGroup(group); if (s != null) s.close();
```

Direkte Netzwerkprogrammierung Zusammenfassung

Vorteile:

- sehr gute Performance
- Protokolle können frei nach Bedarf implementiert werden, wodurch Anpassungen bzw. Optimierungen möglich sind

Nachteile:

- Sockets eignen sich für reine Datenübertragung
- Sockets sind nur die Schnittstelle zur Transportschicht
- Bieten keine Unterstützung für Kommunikationssteuerung und Datendarstellung
- Protokolle müssen auf der Anwendungsebene implementiert werden um Kommunikationspartner zu finden und Austausch von Nachrichten zu steuern
- Eigenschaften der Transportschicht sind zu berücksichtigen (TCP/ UDP)
- Programmierung relativ aufwendig und fehleranfällig

Mehrclientfähige Server mit TCP-Sockets

TCP-Sockets und parallele Ausführung

Auf der Server-Seite kann es nötig sein, mehrere Ausführungseinheiten (Threads) laufen zu lassen, um mehrere Clients "gleichzeitig" bedienen zu können. Dazu muss die Arbeit auf der Serverseite geteilt werden:

- der *Hauptthread* kümmert sich um Annahme der Anfrage, wonach
- die "Konversation" mit Clients von "*Worker-Threads*" übernommen wird

Die *Worker-Thrad*-Instanzen müssen threadfähig sein, um "parallele" Ausführung zu ermöglichen. Das Erbringen der Leistung für den Client muss in der generierenden *Worker-Thrad*-Klasse implementiert oder zumindest gesteuert werden. Pro Client-Anfrage wird **ein** Worker-Thread erzeugt, der sich um die Kommunikation mit dem Client eigenständig kümmert bzw. die Leistung für den Client erbringt. Das Erzeugen des Worker-Threads übernimmt der Haupt- Thread nach dem Eintreffen einer neuen Client-Anfrage.

Ablauf 1

Die Hauptklasse implementieren, in der

- der ServerSocket erzeugt und
- die gesamte Steuerung implementiert wird.

Diese Klasse könnte "Server" genannt werden. Die Worker-Thread-Klasse implementieren, deren Instanzen die Leistungen für die Clients erbringen.

Ablauf 2

Den Server (Instanz der Hauptklasse) starten und eine Instanz der Klasse ServerSocket erzeugen.

In einer Schleife die Methode *accept* auf der ServerSocket-Instanz aufrufen: der ServerSocket wartet, bis eine Anfrage kommt. Wenn eine Anfrage kommt

- wird eine Worker-Thread-Instanz erzeugt und seine Ausführung gestartet
- Je nach Kontext, können dem Konstruktor der Worker-Thread- Klasse auch Parameter übergeben werden (Referenz auf das Socket-Objekt, das von der Methode accept zurückgeliefert wurde, Referenz auf In- oder OutputStream des Sockets usw.)

Ablauf 3

Ab dem Moment übernimmt die Worker-Thread-Instanz die Konversation mit dem Client. Der ServerSocket "geht zurück" und wartet auf die neue Anfrage, indem auf der ServerSocket-Instanz erneut die Methode *accept* aufgerufen wird. Das ganze wiederholt sich so lange, bis die ServerSocket- Instanz aus der Schleife rauskommt (die Frage der Implementierung).

Middleware

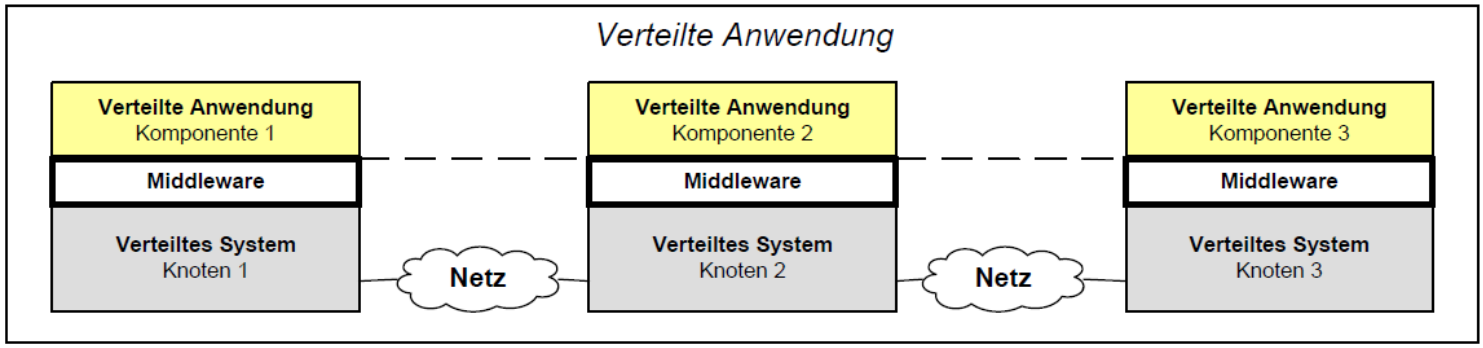
Einführung

Unter Middleware ist die Software zu verstehen, die zwischen der verteilten Anwendung und der darunterliegenden Schicht steht und "vermittelt". Die Aufgabe der Middleware ist:

- die Interaktion zwischen Anwendungskomponenten zu erleichtern, und
- die Komplexität der vernetzten Systemumgebung zu maskieren

Durch die Verwendung von Middleware wird für den Anwendungsentwickler die Umsetzung der Kommunikation einfacher, wobei alle Aspekte der Netzwerkprogrammierung so weit wie möglich (und sinnvoll) von der verteilten Anwendung verborgen werden. Der Anwendungsentwickler kann sich auf die Lösung des eigentlichen Anwendungsproblems konzentrieren.

Verteilte Anwendung und Middleware:



Middleware und deren Aufgaben

- Programmierer und Nutzer sollen durch Middleware von komplexen Netzwerkprotokollen, verteilten Speichern, konkurrierenden Ereignissen, Übertragungsfehlern usw. abgeschirmt werden.
- Middleware verwendet Protokolle, welche auf Nachrichten zwischen Prozessen (Komponenten) basieren.

Ortstransparenz:

- Der Client kann nicht erkennen, wo sich das aufgerufene Objekt befindet (im gleichen Prozess oder nicht)
- Sowohl der Client als auch der Server haben das Gefühl, mit einem lokalen Objekt zu kommunizieren

Kommunikationsprotokolle:

Die zu Grunde liegenden Transport-Protokolle, welche von Middleware verwendet werden, sollen von der verteilten Anwendung "verborgen" werden. Es handelt sich um ein "privates Anliegen" der Middleware.

Computer-Hardware:

Durch die externe Datendarstellung werden die Unterschiede in der Hardwarearchitekturen neutralisiert.

Betriebssysteme:

Middlewareschichten sollen von zu Grunde liegenden Betriebssystemen unabhängig sein.

Verwendung mehrere Programmiersprachen:

Middleware ermöglicht, dass die Anwendungen bzw. deren Komponenten, obwohl in unterschiedlichen Programmiersprachen entwickelt, miteinander kommunizieren können (CORBA).

Kategorien

Jede Middleware hat die Aufgabe, die Abstraktion der Netzwerkprogrammierung zu gewährleisten. Eine Middleware kann jedoch weitere, zusätzliche Dienste einer Anwendung zur Verfügung stellen. Es wird zwischen zwei Middleware-Kategorien unterschieden:

- Kommunikationsorientierte Middleware
- Anwendungsorientierte Middleware

Kommunikationsorientierte Middleware KOM

KOM konzentriert sich auf die Bereitstellung einer geeigneten Kommunikationsinfrastruktur für Komponenten einer verteilten Anwendung. Aufgaben:

- Kommunikation
- Marshalling und Unmarshalling
- Fehlerbehandlung bzw. Fehlerbehebung

Das Middleware-Protokoll setzt auf dem Transportprotokoll des verteilten Systems auf und steuert die Kommunikation zwischen den verteilten Middleware-Komponenten. Welche Aufgaben das Protokoll im Einzelnen zu erfüllen hat, hängt von der jeweiligen Middleware ab. Je mehr Aufgaben von der Middleware übernommen werden, desto komplexer die Middleware

Marshalling und Unmarshalling

- Mit *Marshalling* (Verpacken) werden die zu übertragenden Daten in ein übertragungsfähiges Format transformiert
- Mit *Unmarshalling* (Entpacken) wird die Wiederherstellung der gesendeten Daten aus dem empfangenen Datenstrom bezeichnet

Mit *Marshalling* und *Unmarshalling* wird sicher gestellt, dass die verteilten Anwendungskomponenten bei ihrer Kommunikation (über das Netz) mit gleichem Datenformat (gleicher "Sprache") arbeiten.

Heterogenität in der Hardware: Big-Endian und Little-Endian:

```
Die Zahl 1347 :00000000 00000000 00000101 01000011
```

Speicheradresse (Byte)	Big-Endian-Darstellung	Little-Endian-Darstellung
00	0000 0000	0100 0011
01	0000 0000	0000 0101
02	0000 0101	0000 0000
03	0100 0011	0000 0000

Heterogenität der Programmiersprachen:

- Unterschiedliche Datentypen
- Unterschiedliche Grösse für gleiche Datentypen

Fehlerbehandlung

Für verteilte Anwendungen ist die zuverlässige Übertragung von Nachrichten die Voraussetzung. Es gibt zwei Fehlertypen:

- **Fehler bei der Übertragung**
- **Ausfall von Komponenten**

Es muss geklärt werden, wer für die Behandlung von Fehlern zuständig ist:

- Verteiltes System
- Middleware
- Anwendung selbst

Architektur

Die Architektur der Middleware wird durch Kommunikationsmodelle und Programmierparadigmas bestimmt.

Kommunikationsmodell:

- asynchrone Kommunikation
- synchrone Kommunikation

Programmierparadigma:

- das prozedurale Paradigma (Prozeduren, direkter Zugriff)
- das objektorientierte Paradigma (Objekt, Identität, Attribute)

Das Programmiermodell ist die Sicht des Entwicklers auf die Architektur und definiert, wie die Architektur zur Entwicklung der Anwendung zu verwenden ist. Es wird zwischen drei Programmiermodelle unterschieden:

- Entfernte Prozeduraufrufe (RPC)
- Entfernte Methodenaufrufe (CORBA, RMI)
- Das nachrichtenorientierte Modell (MOM)

Anwendungsorientierte Middleware AOM

AOM stellt eine Erweiterung der KOM dar, die neben reiner Kommunikation eine Reihe zusätzlicher Dienste der verteilten Anwendung zur Verfügung stellt. Konzeptionell stellt die anwendungsorientierte Middleware eine kommunikationsorientierte Middleware, welche um Laufzeitfunktionalität und Dienstkomponenten erweitert wurde. Jeder dieser beiden Komponenten nimmt zusätzlich bestimmte Aufgaben wahr.

Aufgaben der Laufzeitumgebung

Das Betriebssystem als Laufzeitumgebung, die auf jedem Knoten vorhanden ist, ist nicht geeignet, alle Anforderungen der verteilten Anwendungen zu erfüllen.

Die Laufzeitumgebung einer Middleware baut auf den Funktionen des Betriebssystems auf und erweitert diese. Wichtigste Aufgaben der Laufzeitumgebung:

- Ressourcenverwaltung
- Nebenläufigkeit
- Verbindungsverwaltung usw.

Ressourcenverwaltung:

Betriebssysteme verwalten grundlegende Ressourcen (Speicher, Prozesse, Threads, Prozessorzeit, …). Die Ressourcen auf Ebene der Middleware können vom OS nicht optimal verwaltet werden. Diese Verwaltung muss von Middleware übernommen werden:

- Anlegen von getrennten Speicherbereichen mit individuellen Sicherheitskonzepten
- Anlegen von Verbindungsobjekten und Threads auf Vorrat, um sie bei Bedarf schnell einsetzen zu können usw.

Das Ziel der Ressourcenverwaltung ist die Verbesserung von Performance, Skalierbarkeit und Verfügbarkeit von Anwendungen!!

Nebenläufigkeit:

In der Regel werden verteilte Anwendungen von mehreren Anwender parallel benutzt. Um diese Parallelität zu gewährleisten werden die Aufrufe isoliert in separaten, nebenläufigen Threads oder Prozessen abgearbeitet. Threads und Prozesse sind Ressourcen des Betriebssystems: Es folgt, dass die Laufzeitumgebung auf die Funktionen des Betriebssystems zugreifen muss, um eine eigenständige Prozess- und Threadsverwaltung für Unterstützung der Nebenläufigkeit aufzubauen.

Verbindungsverwaltung:

In einer verteilten Anwendung werden auch Verbindungen, als Endpunkte von Kommunikations-kanälen innerhalb einer Anwendung hergestellt. Bei einer grossen Anzahl von Anwender ist eine fixe Zuteilung von Verbindungen zu Prozessen wenig sinnvoll. Ein Lösungsansatz ist die Verwaltung einer Anzahl von Verbindungen (auf Vorrat) in einem Pool

Vorgang:

- eine bestimmte Anzahl Verbindungen beim Starten anlegen
- falls benötigt, Verbindungsobjekte aus dem Pool nehmen
- falls nicht mehr benötigt, Verbindungsobjekte in den Pool zurück geben

Verfügbarkeit:

Mit hoher Verfügbarkeit wird verlangt, dass die Anwendung ihre Aufgabe jederzeit fehlerfrei und korrekt erfüllt. Es gibt jedoch Faktoren, welche die Verfügbarkeit einer Anwendung beeinflussen:

- Eine HW- oder SW-Komponenten fehlt bzw. kann nicht erreicht werden (Absturz, Wartung..)
- Eine HW- oder SW-Komponenten ist überlastet

Je nach Verfügbarkeitsanforderungen, sehen die Lösungen unterschiedlich aus (Redundanz).

Sicherheit:

Verteilte Anwendungen sind sehr angreifbar und somit unter Umständen relativ unsicher. Der Grund liegt darin, dass die Daten über unsicheren Strecken (Netzwerke) übertragen werden. Das Sicherheitsmodell ist aus diesem Grund lebenswichtig.

Im Minimum unterstützt ein Sicherheitsmodell die:

- Zugriffskontrolle (Authentifizierung), womit die Sicherstellung der Identität des Benutzers gemeint ist und
- Vergabe von Zugriffsrechten (Autorisierung), womit die Benutzungsrechte für bestimmte Dienste an die Benutzer vergeben werden

Dienste

Mit einem Dienst wird der Anwendung eine klar definierte Funktionalität zur Verfügung gestellt. Anwendungsorientierte Middleware bietet der Anwendung ihre Dienste implizit über die Laufzeitumgebung. Die Schnittstelle eines Dienstes wird in einer Spezifikation festgelegt. Hilfe dieser Schnittstellenspezifikation kann jede Anwendung den Dienst beliebig nutzen oder auch selbst implementieren.

Unterschiedliche Middleware-Implementierungen stellen unterschiedliche Dienste zur Verfügung. Je mehr Dienste zur Verfügung gestellt werden, desto mächtiger (und auch teuerer) die Middleware. Folgende Dienste sind die wichtigsten:

- Namensdienst
- Sitzungsverwaltung
- Transaktionsverwaltung
- Persistenz

Namensdienst:

Der Namensdienst (auch Verzeichnisdienst genannt) gehört zu den wichtigsten Diensten für eine verteilte Anwendung. Der Namensdienst ermöglicht, dass eine Ressource in einer bestimmten Umgebung (Intranet oder gar Internet) veröffentlicht werden kann, und zwar so, dass sie von interessierten Clients gefunden werden kann. Die Ressourcen, welche veröffentlicht werden sollen (beliebige Dienste, HW- oder SW-Komponenten), werden vom Namensdienst mit einem eindeutigen Namen versehen.

Die Referenz auf eine Ressource enthält alle für den Zugriff relevanten Informationen. Beim Zugriff auf eine veröffentlichte Ressource muss der Client den Namen, unter dem die Ressource veröffentlicht wurde, dem Namensdienst übergeben. Als Antwort auf diese Anfrage (auch **look up** genannt) erhält der Client die Referenz (IP-Adresse und die Portnummer) auf die gewünschte Ressource zurück, mit der die Kommunikation mit der Ressource realisiert werden kann

Sitzungsverwaltung:

Die Verwaltung von Sitzungen ist insbesondere für die interaktiven verteilten Anwendungen wichtig. Jedem einzelnen Anwender wird eine Sitzung zugeteilt, welche eine bestimmte Zeit gültig ist und alle für den Anwender relevanten Daten verwaltet. Der Dienst zur Sitzungsverwaltung hat die Aufgabe, Sitzungen vieler Anwender parallel zu verwalten.

Transaktionsverwaltung:

Die Transaktionsverwaltung spielt bei den verteilten Anwendungen , bei denen die Erhaltung der Datenkonsistenz gewährleistet werden muss, eine wichtige Rolle. Damit ist gemeint, dass alle persistenten Daten der Anwendung in ihrer Gesamtheit einen gültigen Zustand repräsentieren. Bei parallelen Zugriffen auf die Daten, könnte passieren, dass zwei Anwender einen gleichen Datensatz gleichzeitig ändern. Dies kann durch die Verwendung von Transaktionen verhindert werden. Eine Transaktion macht aus einer Reihe von atomaren Aktionen eine untrennbare Einheit.

Mit einer Transaktion wird **ACID** sicher gestellt:

- **A**tomicity (Atomarität): Das Alles-Oder-Nichts-Prinzip
- **C**onsistency (Konsistenz): Eine Transaktion überführt einen konsistenten Zustand immer in einen neuen konsistenten Zustand
- **I**solation (Isolation): Transaktionen laufen gegeneinander isoliert ab
- **D**urability (Dauerhaftigkeit): Der Zustand am Ende einer Transaktion wird dauerhaft gespeichert

Persistenz:

Mit Persistenz wird die Gesamtheit aller Mechanismen zur dauerhaften Speicherung von flüchtigen Daten im Hauptspeicher auf ein persistentes Speichermedium bezeichnet. In einer verteilten Anwendung werden Daten

- an den Persistenzdienst übergeben
- auf ein geeignetes Speicherformat abgebildet und
- an das Speichermedium oder die Datenbank weitergereicht

In der Praxis hat sich vor allem eine Art des Persistenzdienstes etabliert: die objektrelationale Mapper (OR Mapper). OR-Mapper sind Persistenzdienste, die sich auf die Abbildung von Objekten in relationalen Datenbanken konzentrieren.

Middleware Technologien

Es kann zwischen drei Typen von Middleware-Technologien unterschieden werden:

- Object Request Broker (ORB)
- Application Server (AS)
- Middleware-Plattformen

Object Request Broker (ORB) :

Der Object Request Broker setzt auf dem Programmiermodell der entfernten Methodenaufrufen auf. Er bietet die Kommunikationsinfrastruktur für die Verwaltung von verteilten Objekten und stellt zusätzliche Dienste zur Verfügung, die von der verteilten Anwendung genutzt werden können.

Beispiel für ORB:

Common **O**bject **R**equest **B**roker **A**rchitecture (**CORBA**)

Aplication Server (AS) :

Application Server (Anwendungsserver) werden jeweils für einen bestimmten Typ von Anwendungen entwickelt (z. B. JEE, .NET, …) und stellen die Kommunikationsinfrastruktur, die Laufzeitumgebung und diverse Dienste zur Verfügung. Reine Application Server sind als solche eher selten zu finden und werde in der Regel als Teil einer Middleware- Plattform zur Verfügung gestellt.

Middleware Plattformen:

Middleware-Plattform erweitert einen Applikation Server zu einer vollständigen Plattform für verteilte Anwendungen, indem sie die verteilten Anwendungen durch alle Schichten unterstützten. Zur Zeit existieren zwei Middleware-Plattformen, die zu Quasi-Standards geworden sind:

- JEE mit dem EJB-Komponentenmodell und
- .NET mit COM-Komponentenmodell

Remote Procedure Call (RPC)

Einführung

Mit Remote Procedure Call (RPC) ist eine Art der Interprozesskommunikation gemeint, mit der Aufruf von Operationen (Prozeduren) realisiert wird, die sich in einem anderen Prozess befinden. Da die aufgerufene Prozeduren nicht im eigenen Prozess (Speicherraum) ausgeführt werden, reden wir von einem **entfernten Prozeduraufruf.** Die kommunikationswilligen Prozesse (Aufrufer und Aufgerufene) können sich sowohl auf einem gleichen als auch auf unterschiedlichen Knoten befinden.

Remote Procedure Call (Entfernter Prozeduraufruf) wird oft in verteilten Systemen eingesetzt, mit dem Ziel, das verteilte System als eine Ganzheit aussehen zu lassen. Mit RPC kann z. B. der Zugriff auf das verteilte Dateisystem in einem Netzwerk so realisiert werden, dass der Benutzer das Gefühl hat, auf ein lokales Dateisystem zuzugreifen. Dies ist dank transparenten entfernten Prozeduraufrufen (RPC) möglich. Beispiel: NFS von Sun

Sockets ermöglichen eine einfache Datenübertragung, wobei die Schnittstelle auf folgende Operationen reduziert wird:

- Verbindung herstellen (*connect*)
- Daten lesen (*read*) und Daten schreiben (*write*)
- Verbindung abbauen (*diesconnect*)

Man ist gezwungen, mit Streams zu arbeiten (I/O), was nicht unserer Vorstellung entspricht: für uns ist logischer, die benötigten Methoden direkt aufzurufen.

Von RPC zu LPC

Sicht des Aufrufers (Client):

Wie kann eine entfernte Prozedur wie eine lokale Prozedur aufgerufen werden?

➔ Stellen wir dem Client eine Komponente zur Verfügung, die **im gleichen Prozess** ausgeführt wird und **als Server** aussieht. Diese Komponente muss **die gleiche Schnittstelle** wie der Server implementieren. Die Rede ist von einem **Stub** (auch ClientStub, Proxy bzw. Stellvertreter genannt)

Sicht des Aufgerufenen (Server):

Wie kann ein entfernter Aufruf wie ein lokaler Aufruf entgegengenommen werden?

➔ Stellen wir dem Server eine Komponente zur Verfügung, die **im gleichen Prozess** ausgeführt wird und **als Client** aussieht. Diese Komponente wird die Prozeduren des Servers als lokale Prozeduren aufrufen. Die Rede ist von einem **ServerStub** (auch Skeleton genannt).

Die Situation ist wie folgt:

Der Aufrufer (Client) macht einen lokalen Prozeduraufruf. Der Aufgerufene (Server) bekommt auch mit lokalen Prozeduraufruf zu tun.

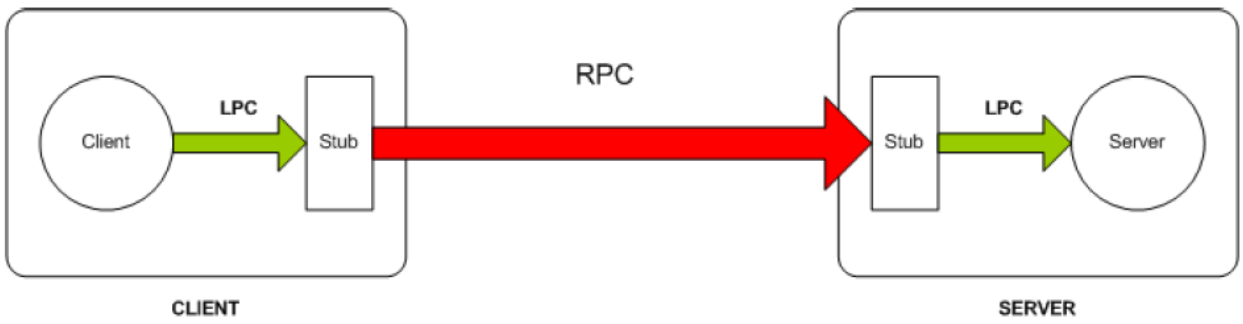
Die Folge:

Sowohl der Aufrufer (Client) als auch der Aufgerufene (Server) sehen nur die **lokalen Prozeduraufrufe**: die Welt ist in Ordnung, von RPC keine Spur!

Aber … Eine Interprozesskommunikation kann mit LPC nicht realisiert werden!

Die Lösung:

Die Interprozesskommunikation wird durch die beiden zusätzlichen Komponenten (**ClientStub** und **ServerStub**) realisiert, ohne das der Aufrufer (Client) und Aufgerufene (Server) davon etwas merken.



Kommunikation und RPC

Client

Der ClientStub implementiert die Server-Schnittstelle: so kann der Client alle Methoden des Servers auf dem ClientStub aufrufen. Der ClientStub

- nimmt den Aufruf der Server-Methode entgegen, verpackt sie passend (Marshalling) und sendet sie zum Server bzw. zum ServerStub (RPC)
- nimmt die Antwort des Servers entgegen, entpackt sie (Unmarshalling) und liefert den Rückgabewert an den Client zurück

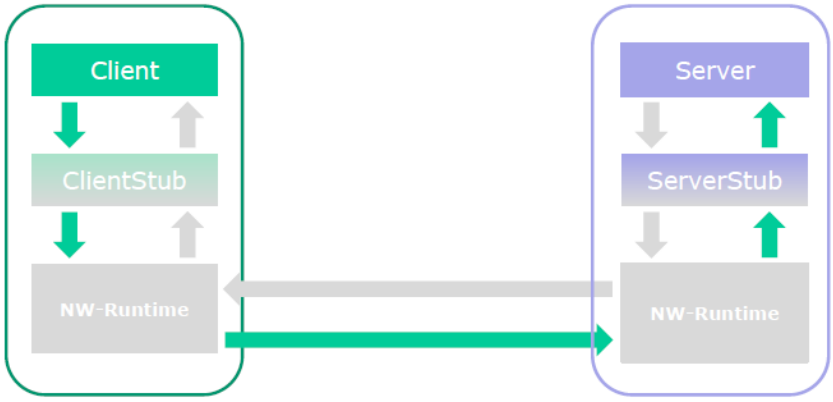
Server

Der Server muss alle Prozeduren implementieren, die in der entsprechenden Schnittstelle definiert wurden. Der ServerStub

- nimmt die Anfrage, die der ClientStub gesendet hat, entgegen und entpackt sie (Unmarshalling)
- ruft die gewünschte Methode auf dem Server auf (LPC) und nimmt die Antwort des Servers entgegen
- verpackt die Antwort passend (Marshalling) und sendet sie an den ClientStub zurück

Der ServerStub muss eine Referenz auf den Server haben, um seine Prozeduren aufrufen zu können.

Stubs



1. Client ruft die Server-Methode auf dem ClientStub auf
2. ClientStub verpackt den Aufruf in eine übertragbare Nachricht
3. Die Nachricht wird zum Server durch die Netzwerk-Runtime gesendet
4. Empfangene Nachricht wird zum ServerStub weiter geleitet
5. ServerStub entpackt die Nachricht und ruft die Server-Prozedur auf
6. Die aufgerufene Server-Prozedur liefert die Antwort zurück
7. ServerStub verpackt die Antwort in eine übertragbare Nachricht
8. Die Nachricht wird zum Client durch die Netzwerk-Runtime gesendet
9. Netzwerk-Runtime empfängt die Nachricht und leitet sie an den ClientStub weiter
10. ClientStub entpackt die Antwort und leitet sie an den Client weiter

Probleme mit RPC

Unterschiedliche Programmiersprachen

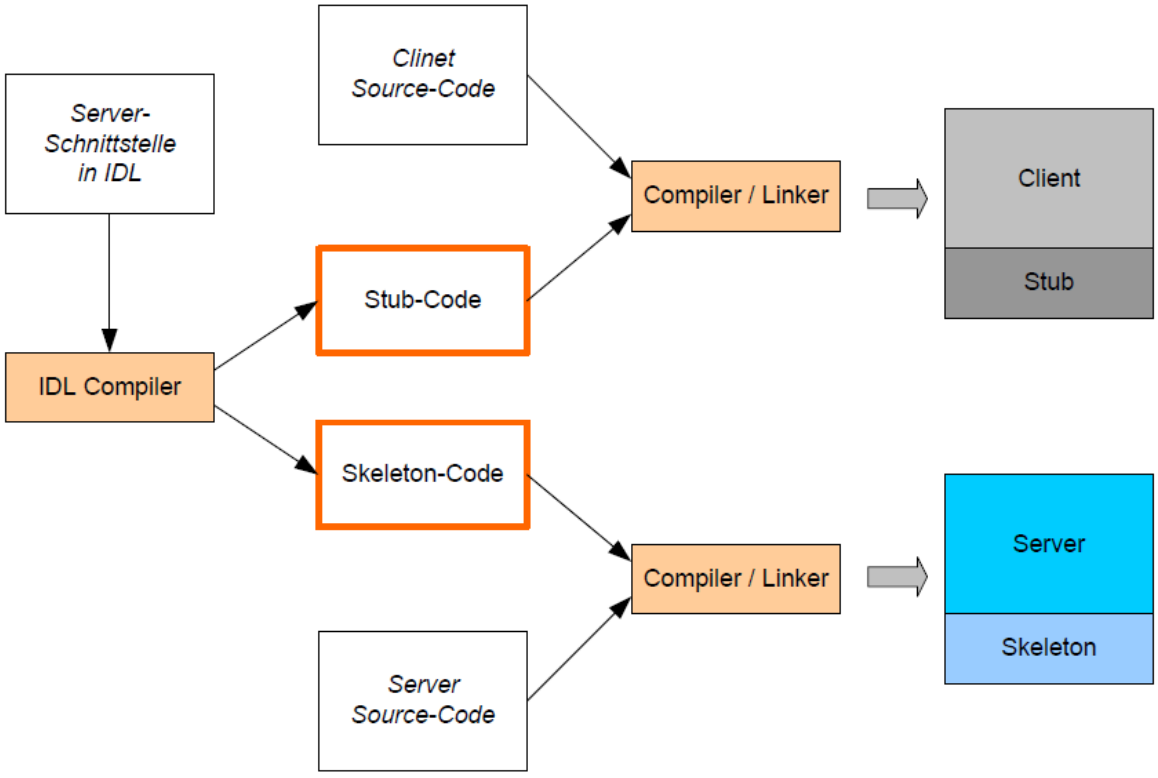
Programmiersprachen wie Java, C und C++ unterstützten das RPC-Konzept nicht von Haus aus. Für die Generierung von Stubs wird ein zusätzlicher Compiler (Precompiler) eingesetzt. Als Basis für die Generierung von Stubs muss die Schnittstelle zur Verfügung gestellt werden, die vom Server realisiert wird. Für die Definition von Schnittstellen wird die *Interface Definition Language* (IDL) eingesetzt.

Interface Definition Language (IDL)

IDL ist eine deklarative Sprache und stellt die Syntax zur Beschreibung von Schnittstellen einer Komponente zur Verfügung. Sie setzt "abstrakte" Konstrukte und Datentypen ein, die nicht aus einer konkreten Programmiersprachen kommen. Mit IDL ist es möglich, Prozeduren "abstrakt" zu beschreiben, ohne dabei eine konkrete Programmiersprache zu verwenden

Anhand dieser Beschreibung ist es möglich, mit einem speziellen Compiler den Client- und den Server-Stub zu generieren. Pro eingesetzte Programmiersprache muss ein solcher Compiler zur Verfügung stehen, der unter anderem für die Übersetzung von lokalen Datentypen in abstrakte Datentypen (die IDL einsetzt) und umgekehrt zuständig ist. Beispiel: CORBA IDL

IDL und Erstellungsprozess:



RPC und Parameterübergabe

- Übergabe von Werten (pass by value)
  - Der Wert wird in die Nachricht entsprechend kopiert
  - Auf der anderen Seite wird der Wert "rekonstruiert"
- Übergabe von Referenzen (pass by reference)
  - Grundsätzlich nicht möglich
  - Ausnahme: falls gemeinsamer Speicher vorhanden (shared memory)

RPC und Datenrepräsentation

Verteilte Anwendungen werden in der Regel in einer heterogenen Umgebung ausgeführt. Mögliche Problemquellen:

- Unterschiedliche HW-Architektur (Anordnung von Bytes)
- Unterschiedliche Programmiersprachen bei der Entwicklung von einzelnen Komponenten verwendet
- Unterschiedliche Zeichensätze

Für heterogene Systeme ist ein gemeinsamer Format für Daten nötig. Beispiele:

- eXternal Data Repräsentation (XDR) von Sun
- ASN.1 (ISO Abstract Syntax Notation)

XDR unterstützt *implicit typing*:
Nur Werte werden übermittelt (keine Infos zum Datentyp des Parameters).

ASN.1 unterstützt *explicit typing*:
Mit jedem Parameterwert wird auch die Datentyp-Information vermittelt.

RPC und Binding

Der Kommunikationspartner muss "gefunden werden".

- Wie ist das Host-System zu finden?
- Wie findet man das gewünschte Prozess auf dem Host-System?

Lösungsansatz:

Jeder Server verwaltet (in einer passender Form) selbst alle Services, die lokal (auf diesem Server) ausgeführt werden. Bevor das Service in Anspruch genommen wird, wird beim Server nachgefragt, ob ein solches Service verfügbar sei und wie es angesprochen wird (die Portnummer des Services)

Die Portnummer des Services darf aus Portierungsgründen nicht vorgegeben sein!

- Wie kann der Client die Portnummer des gewünschten Services erfahren?

Die Lösung heisst **Binding**-Dienst : der Verwalter von Services mit deren Portnummern. Beispiel für ein Binder: *portmapper* bei Sun RPC.

Ein Binder hat drei Aufgaben:

- Anmeldung von gestarteten und somit verfügbaren Services (*register*)
- Anfrage nach Portnummer der Services beantworten (*lookup*)
- Abmeldung von Services, die nicht mehr zur Verfügung stehen (*unregister*)

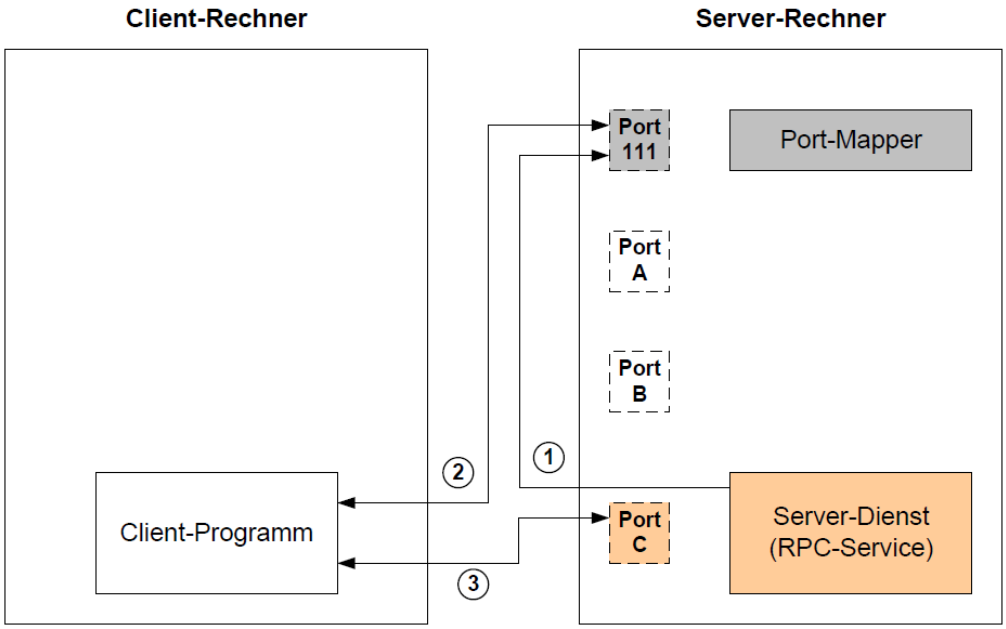
Ein High Level RPC-Aufruf besteht aus zwei Teilen:

- Aufruf des Binders (Server-Portmappers)
- Aufruf des Server-Dienstes über die so ermittelte Portnummer

Oder, anders gesagt

Jeder High Level RPC-Aufruf (für den Benutzer sichtbar) besteht aus zwei reale Aufrufe. Diese werden vom Benutzer als ein Aufruf wahrgenommen.

Beispiel Sun RPC



Die Portnummer des Binders muss zum Voraus bekannt sein: in unserem Fall läuft der Binder auf dem Port 111

- Schritt 1:
  Der Server startet das Service und registriert es beim Binding-Dienst
- Schritt 2:
  Der Client fragt beim Binder die Portnummer des gewünschten Services ab
- Schritt 3:
  Der Client hat die Portnummer vom Binder erhalten und spricht das gewünschte Service an (ruft die gewünschte Prozedur auf)

RPC und Fehlerbehandlung

Bei einem lokalen Aufruf kann nicht viel passieren, da alles im gleichen Speicher abläuft. Bei einem entfernten Aufruf kann ein Fehler an mehreren Orten passieren:

- Bei der Zustellung der Anfrage
- Während der Ausführung auf dem Server
- Während der Zustellung der Antwort

Während der Zustellung der Anfrage:

- die Anfrage geht verloren
- die Zustellung der Anfrage dauert länger als "erlaubt"

Während der Ausführung auf dem Server :

- vor der Ausführung der Prozedur
- während der Ausführung der Prozedur
- nach der Ausführung der Prozedur, jedoch vor dem Senden der Antwort

Während der Zustellung der Antwort:

- Die Antwort geht verloren
- Die Antwort kommt verspätet (nach *time out*) ein, was das erneute Senden der Anfrage zur Folge hat

Wie sich ein Fehler auf das ganze auswirkt, hängt davon ab, ob es sich um Aufruf von Prozeduren handelt, die beliebig oft (*idempotent*) oder nur einmal (*nicht idempotent*) ausgeführt werden dürfen. **Idempotente Prozesse** dürfen beliebig oft ausgeführt werden (Beispiel: das Lesen aus einer Datei). **Nicht idempotente Prozesse** dürfen nur einmal ausgeführt werden (Beispiel: die Überweisung eines bestimmten Geldbetrags von einem auf das andere Konto).

Die Fehlerbehandlung hängt stark davon ab, ob es sich dabei um einen *idempotenten* oder *nicht idempotenten* Prozess (Prozedur) handelt

Weitere Probleme mit RPC

Performance:

RPC ist langsamer als LPC. Die Antwortzeiten hängen auch von der Belastung des Netzwerks stark, die sich ständig ändert.

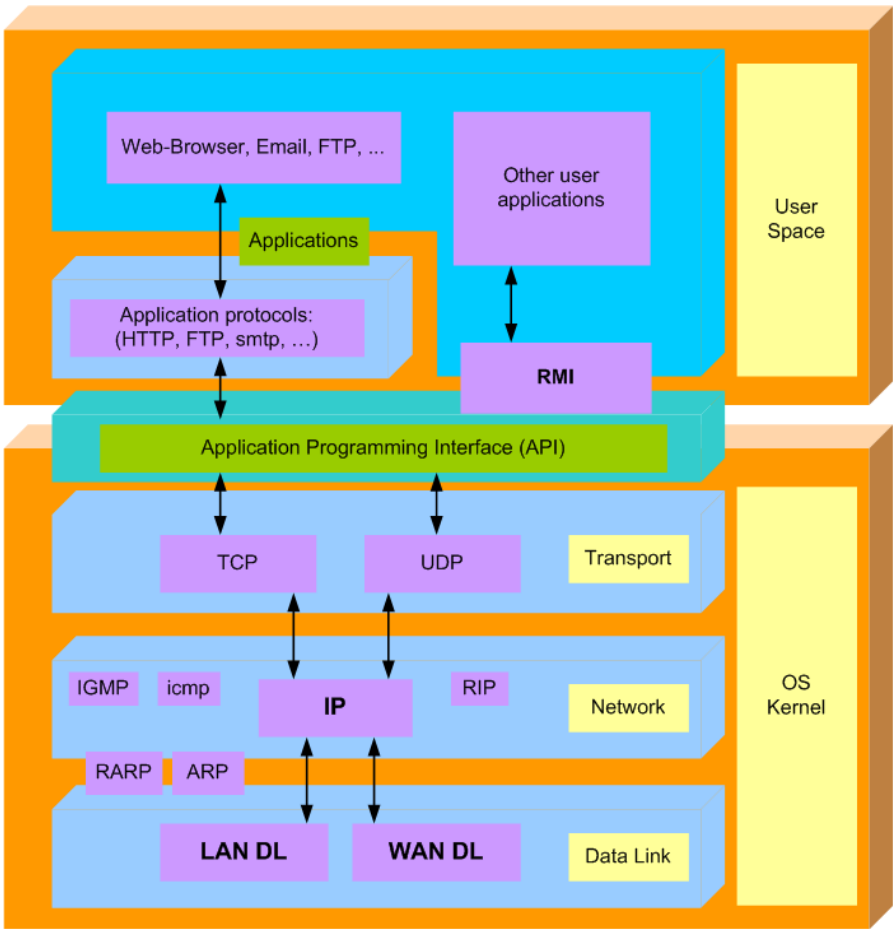
Sicherheit

Daten (Nachrichten) werden im Netz übertragen und sind somit angreifbar. Zudem Probleme mit Authentifizierung von Kommunikationspartner.

Remote Methode Invocation (RMI)

Einführung

Remote Method Invocation (RMI) ist ein Mechanismus in Java, mit welchem entfernte Objekte bzw. deren Angebote genutzt werden können und ein einfaches Framework für die Entwicklung von verteilten Anwendungen in Java. Die Entwicklung von verteilten Anwendungen mit RMI ist nur in Java möglich (Java to Java only). Mit *Internet Inter-Orb Protocol* (RMI-IIOP) kann eine RMI-Java-Anwendung mit Anwendungen und Komponenten kommunizieren, die *Common Object Request Broker Architecture* (CORBA) verwenden.



Vorteile:

Mit RMI

- werden Details der Netzwerkkommunikation "ausgeblendet"
- wird die Verteilung von Objekten durch einen Namensdienst (RMI-Registry) ermöglicht
- wird das dynamische Laden vom Code ermöglicht

Die Java-Standardedition (JSE) enthält alle Klassen und Interfaces, die für die Entwicklung von verteilten RMI-Anwendungen nötig sind. Entfernte Objekte (Server) sind "multithreaded". RMI bietet (von Haus aus) eine synchrone Kommunikation zwischen Client und Server (entferntes Objekt)

Nachteile:

RMI basiert ausschliesslich auf OO-Konzepten der Sprache Java (*Java-to- Java only*), die Integration mit anderen Verteilungstechniken, insbesondere mit anderen Programmiersprachen, sehr schwierig und an sich nur über IIOP möglich.

RMI verwendet den Namensdienst. Der mitgelieferte Namensdienst ist sehr einfach und eignet sich nicht für alle Anwendungen.

Die Synchronisation bei konkurrierenden Zugriffen wird nicht von RMI realisiert und liegt im Verantwortungsbereich des Entwicklers.

Kommunikation

Client

- nimmt Dienste von entferntem Objekten in Anspruch
- fragt dazu beim Namensdienst nach, ob ein passendes Objekt für den gewünschten Dienst registriert wurde
- ruft Methoden des Entfernten Objekts auf

Server

- erzeugt das entfernte Objekt und meldet es beim Namensdienst an (Registrierung)
- meldet das entfernte Objekt beim Namensdienst ab, wenn es nicht mehr benötigt wird (De-Registrierung)

Entfernte Objekte:

Werden vom Server erzeugt (falls ein separater Server vorhanden) und beim Namensdienst registriert. Sie laufen auf dem Server-Knoten und stellen bestimmte Dienste zur Verfügung. Ein entferntes Objekt empfindet die Client-Aufrufe, welche aus einer anderen JVM kommen, als lokale Aufrufe (Location Transparency).

Stub und Skeleton:

Werden durch *rmic* aus der Klasse erzeugt, welche das entfernte Objekt implementiert. Sie kümmern sich um die Übertragung von Daten über das Netzwerk.

Generierung von Stubs in unterschiedlichen JDK-Versionen:

- mit *rmic v1.1* werden sowohl Stub als auch Skeleton erzeugt
- mit *rmic v1.2* wird nur Stub erzeugt, während der Skeleton dynamisch (während Laufzeit) erzeugt wird
- seit jdk1.5.0 werden sowohl Stub als auch Skeleton dynamisch erzeugt

Beispiel für Generierung mit *rmic v1.1*:

```
C:\Temp\bin>rmic –v1.1 –keep server.CalculatorImpl
```

Marshalling und Unmarshalling:

Unter Marshalling versteht man das Verpacken von Nachrichten, bevor sie zum entfernten Objekt (Server) gesendet werden. Unter Unmarshalling versteht man das Entpacken von Nachrichten, welche vom Client in verpackter Form eingetroffen sind, bevor sie dem echten EO übermittelt werden.

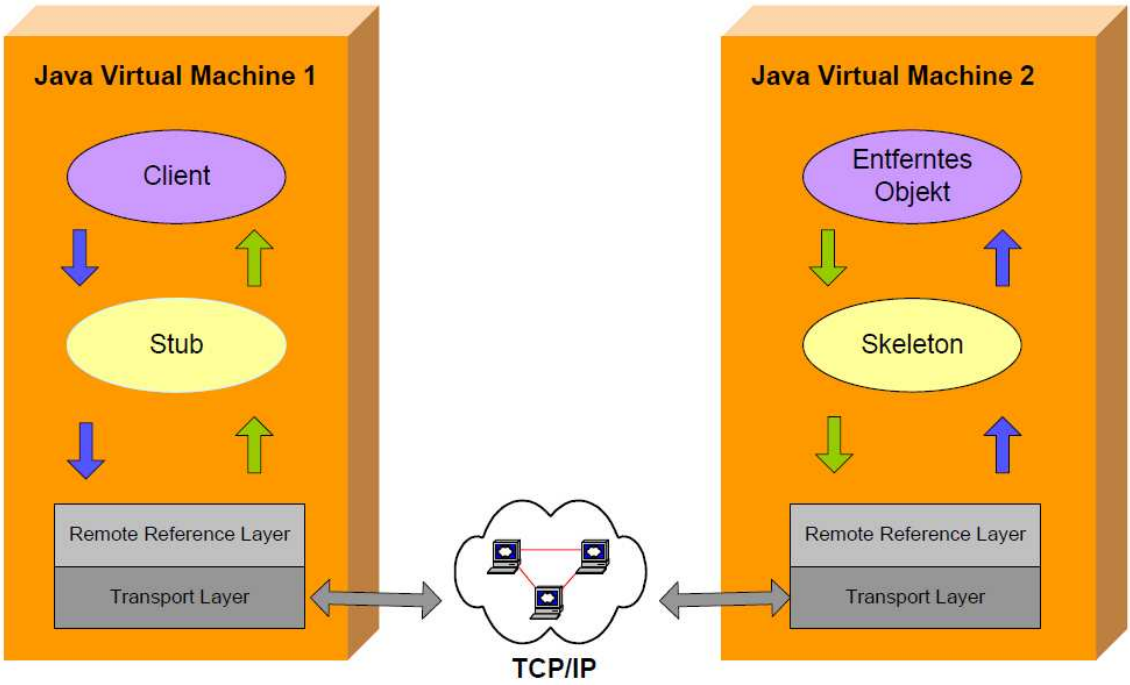
Das Ver- und Entpacken von Nachrichten (Marshalling und Unmarshalling) wird automatisch von Stub und Skeleton erledigt. Bei RMI muss sich der Entwickler darum nicht kümmern.

Namensdienst (RMI-Registry):

Aufgaben des Namensdienstes

- das Registrieren von EO zu ermöglichen (*binding*)
- das Finden des EO zu ermöglichen (*lookup*)

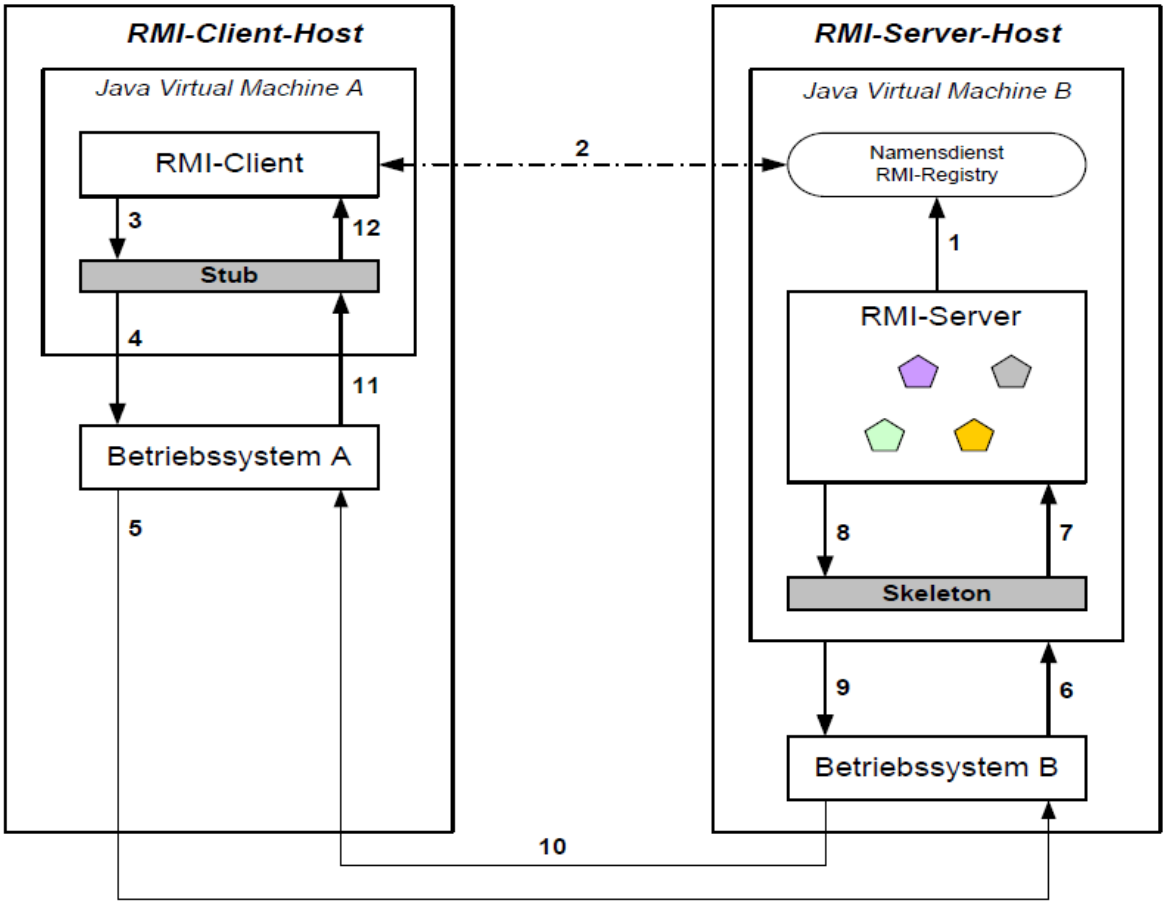
Namensdienst wird auf dem Host-System (Server) ausgeführt. Der Server kann das EO bei dem Namensdienst registrieren. Der Client kann beim Namensdienst nachfragen, ob ein passendes EO für den gewünschten Dienst zur Verfügung steht bzw. registriert wurde. Kann mit dem Telefonauskunftsdienst verglichen werden.



Entfernter Aufruf

- Der RMI-Server erzeugt das entfernte Objekt (EO) und registriert es beim Namensdienst.
- Der RMI-Client möchte ein EO benutzen und fragt beim Namensdienst, ob und welches EO für den gewünschten Dienst zur Verfügung steht.
- Der Namensdienst sendet dem Client die Referenz auf ein entferntes Objekt (falls vorhanden), das den gewünschten Dienst erbringen kann
- Der RMI-Client besitzt die Referenz auf das entfernte Objekt und kann die gewünschte Methoden dieses Objekts aufrufen
- Das Stub-Objekt nimmt die Anfrage des RMI-Clients entgegen, verpackt die übergebenen Parameter (marshalling) und sendet den Methodenaufruf an den RMI-Server, wobei er dazu die zugrunde liegenden OS-Funktionen aufruft.
- Das Betriebssystem überträgt die Daten über das physikalische Netzwerk zum entferntem Host, der RMI-Client wartet auf die Antwort (blockierend)
- Das entfernte Betriebssystem nimmt die Anfrage entgegen und leitet die Anfrage und das entsprechende Skeleton-Objekt weiter.
- Das Skeleton-Objekt entpackt die enthaltene Anfrage und generiert daraus den entsprechenden Methodenaufruf. Zudem ruft es die gewünschte Methode des echten Objekts auf.
- Das echte Objekt (entferntes Objekt) erbringt den geforderten Dienst und liefert das Ergebnis an den Aufrufer (Methode aus dem Skeleton-Objekt) zurück
- Das Skeleton-Objekt nimmt die Antwort entgegen (Rückgabewert, Exception …), verpackt die Antwort der aufgerufenen Methode für die Übertragung (marshalling) und übergibt die verpackte Antwortnachricht für die Übertragung an das Betriebssystem.
- Das entfernte Betriebssystem sendet die Antwort an das lokale Betriebssystem
- Das lokale Betriebssystem nimmt die Antwortnachricht entgegen und leitet sie an das entsprechende Stub-Objekt weiter.
- Das Stub-Objekt entpackt die empfangene Antwortnachricht und extrahiert den Rückgabewert (oder Exception) daraus (unmarshalling). Zudem liefert es die Antwort an den RMI-Client zurück.
- Der RMI-Client nimmt die Antwort (den Rückgabewert) entgegen und kann weiter arbeiten
- Falls eine Ausnahme auf der Serverseite geworfen wurde, wird der Client-Stub diese extrahieren und werfen, während der Client auf die geworfene Ausnahme entsprechend reagieren muss.

Alles auf einen Blick:



Implementierung

Definition der Schnittstelle

Methoden, die von einem entfernten Objekt zur Verfügung gestellt werden sollen, werden in einer passenden Schnittstelle definiert. Diese Schnittstelle muss von der Schnittstelle ***java.rmi.Remote*** abgeleitet werden. Alle Methoden der Schnittstelle können ***java.rmi.RemoteException*** werfen und müssen diese deklarieren. Die Rede ist von einer **entfernten** Schnittstelle.

Beispiel für eine entfernte Schnittstelle:

```
import java.rmi.*;

public interface Adder extends Remote
{
   int add(int x, int y) throws RemoteException;
}
```

Definition der entfernten Klasse

Die Klasse, deren Instanzen als entfernte Objekte eingesetzt werden sollen muss

- von der Klasse *java.rmi.UnicastRemoteObject* abgeleitet werden,
- den Standardkonstruktor zur Verfügung stellen, der seinerseits die Ausnahme *java.rmi.RemoteException* deklarieren muss und
- die entfernten Schnittstelle implementieren

Aus der entfernten Klasse werden mit Hilfe des **rmic**-Tools (RMI Compiler) anschliessend *Stub* und *Skelleton* generiert. Namensgebung:

- Der Name der entfernten Klasse wird aus dem Namen der entfernten *Schnittstelle* + *Impl* zusammengesetzt
- Es handelt sich allerdings um eine Empfehlung bzw. Konvention

Beispiel:

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

public class AdderImpl extends UnicastRemoteObject implements Adder
{
   public AdderImpl() throws RemoteException
   { }

   public int add(int x, int y) throws RemoteException
   {
      return x + y ;
   }
}
```

Kompilierung der entfernten Klasse

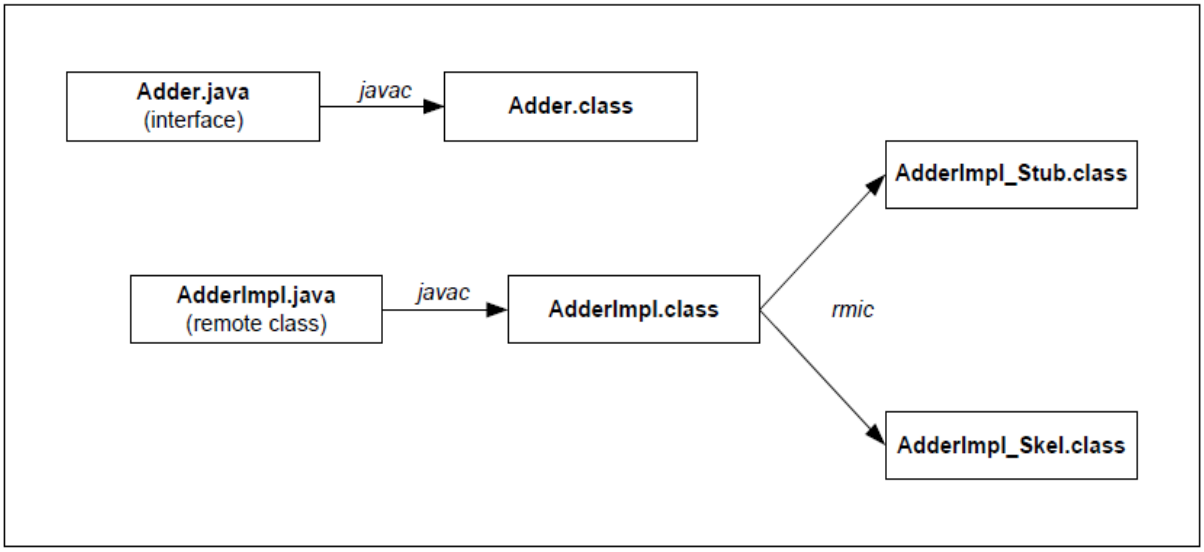
Kompilieren (javac):

- Aufruf: `javac *.java`
- produziert `Adder.class` und `AdderImpl.class`

Generierung der *Stub*- und Skeleton-Klasse (rmic):

- Seit **jdk1.5.0** fällt die explizite Generierung der *Stub*- und Skeleton-Klasse weg
- Die **AdderImpl_Stub.class** und die **AdderImpl_Skel.class** werden während der Laufzeit (dynamisch) automatisch erzeugt

Alles auf einen Blick:



Namensdienst (Anmeldung des Entfernten Objekts)

Damit ein entferntes Objekt registriert werden kann, muss der Namensdienst auf dem Server gestartet werden

```
start rmiregistry [port]
```

Der Namensdienst-Prozess läuft und steht für die Registrierung von entfernten Objekten zur Verfügung. Der Server kann gestartet werden, wonach er das entfernte Objekt (ein oder mehrere) generiert und beim Namensdienst anmeldet (registriert). **Hinweis:** Port 1099.

RMI-Server Implementierung

```
public static void main(String[] args)
{
  try
  {
    // Entferntes Objekt erzeugen
    Adder adder = new AdderImpl();
    // Entferntes Objekt beim Namensdienst registrieren
    Naming.rebind("AdderObjekt", adder);
    // Ausgabe – Server bereit
    System.out.println("Adder bound");
  }
  catch (RemoteException re)
  {
    re.printStackTrace();
  }
  catch (MalformedURLException me)
  {
    me.printStackTrace();
  }
}
```

RMI-Client Implementierung

Der RMI-Client muss

- den *SecurityManger* installieren, um sich von "bösartigen" *Stubs* zu schützen
- den Namensdienst mit Hilfe der Klasse *java.rmi.Naming* finden
- beim Namensdienst nachfragen, ob das gewünschte entfernte Objekt vorhanden ist und eine Referenz auf dieses Objekt erhalten (Methode *lookup*)
- den Typ der erhaltenen Referenz auf den Typ der entfernten Schnittstelle casten (Methode *lookup* liefet *java.rmi.Remote*-Typ zurück)

Über die erhaltene Referenz kann das entfernte Objekt benutzt werden. Die Angabe der Adresse des Namendienstes wird in Form einer URL realisiert:

```
rmi://192.168.1.25:1099/AdderObjekt
```

Die Bestandteile der URL sind:

- rmi: das Protokoll
- IP-Adresse des Servers, auf dem der Namensdienst ausgeführt wird
- Port, auf dem der Namensdienst läuft (Standardport: 1099)
- Der Name, unter dem das entfernte Objekt bei dem Namensdienst registriert wurde

```
public class Main {
  public static void main(String[] args) {
    // SecurityManager installieren
    System.setSecurityManager(new RMISecurityManager());
    // URL definieren
    String url = "rmi://196.168.1.25:1099/AdderObjekt";
    // Referenz auf das entfernte Objekt holen
    Adder adderObj = (Adder) Naming.lookup(url);
    // Methode 'add' des entfernten Objekts aufrufen
    int sum = adderObj .add(13, 37);
    // Ergebnis ausgeben
    System.out.println("13 + 37 = " + sum);
  }
}
```

Java-Security und Policy-Datei

In einem verteilten System wird oft der Code von einem anderen Server (fremder Code) geladen und auf dem lokalen System ausgeführt. Das kann gefährlich sein, da der Code durch einen Angreifer durch seinen, bösartigen Code, ausgetauscht weden kann. Das Downloaden und Ausführen des fremden Codes wird vom ***RMISecurityManager*** laufend überwacht.

Der *RMISecurityManager* wurde beim Client "installiert" und braucht noch klare Anweisungen, welche Aktionen vom fremden Code ausgeführt werden dürfen und welche nicht. Diese Anweisungen werden in einer so genannten ***policy*-Datei** festgehalten. Eine *policy*-Datei kann viele Einträge enthalten. ->Grundregel: nur das erlauben, was nötig!!

Eine sehr einfache und auch grosszügige Policy-Datei: *adder.policy*

```
grant {
  permission java.security.AllPermission;
};
```

Eine weitere Variante wäre wie folgt:

```
grant {
  permission java.net.SocketPermission „*:1024-", "connect, accept";
};
```

Diese Datei soll (am besten) im <ROOT>Verzeichnis des RMI-Clients gespeichert werden.

Verteilung von Klassen

Nachdem alle Klassen erstellt und kompiliert wurden, müssen diese verteilt werden. Auf der Server-Seite müssen folgende Klassen im CLASSPATH zu finden sein:

- entfernte Schnittstelle (*Adder.class*)
- entfernte Klasse (*AdderImpl.class*)
- Server-Klasse (die Methode *'main*' enthält)

Auf der Client-Seite müssen folgende Klassen im CLASSPATH zu finden sein:

- entfernte Schnittstelle (*Adder.class*)
- die Client-Klasse (die Methode *'main*' enthält)
- die *adder.policy* im ROOT-Verzeichnis

Diese Verteilung von Klassen ist nicht die einzige, jedoch für dieses Beispiel die einfachste

RMI-Server starten

Namensdienst starten:

```
start rmiregistry
```

Server starten (Serverklasse: *Server.class*):

```
java Server
```

Hinweis: Der Namensdienst muss aus dem ROOT-Verzeichnis des Servers gestartet werden.!!

RMI-Client starten

Ausgangslage:

Die entfernte Schnittstelle (*Adder.class*) befindet sich auf dem Client-System und ist im CLASSPATH zu finden. Die Policy-Datei (*adder.policy*) befindet sich im ROOT-Verzeichnis des Clients. Client starten (Client-Klasse: *Client.class*):

```
java –Djava.security.policy=adder.policy Client
```

Es fällt auf, dass die *policy*-Datei beim Starten des RMI-Clients als Parameter für JVM übergeben werden muss. Die *policy*-Datei kann auch im Programmcode angegeben werden, indem die System-Property **`java.security.policy`** im Programm vor dem Installieren des RMISecurityManagers passen gesetzt wird. Sie muss den Namen der *policy*-Datei (Path) enthalten, wie das folgende Beispiel zeigt:

```
System.setProperty("java.security.policy", "ressources/myApp.policy");
```

Start von RMI-Server und –Clients

Das Starten von RMI-Server und –Clients kann auch einfacher realisiert werden, wenn

- der Namensdienst (RMI Registry) aus dem Programmcode gestartet wird und
- die *policy*-Datei für den Client im Programmcode angegeben wird

Für das kreieren einer Registry-Instanz (starten des Namensdienstes) steht die Klasse ***java.rmi.LocateRegistry*** zur Verfügung. Das Erzeugen der RMI-Instanz erfolgt mit Hilfe der Klassenmethode:

```
LocateRegistry.createRegistry (int port)
```

Annahmen:

- Server-Host: **147.88.100.100**
- Registry-Port: **1099**
- Client-Host: **147.88.100.200**
- Policy-Datei: **adder.policy**

Hinweise:

Der Registry-Port kann nach Bedarf angepasst werden. Pro JVM darf nur eine Registry-Instanz ausgeführt werden

Der Servercode könnte wie folgt aussehen:

```
// Entferntes Objekt erzeugen
Adder adder = new AdderImpl();
// Registry-Instanz erzeugen bzw. starten
Registry reg = LocateRegistry.createRegistry(port);
// Entferntes Objekt beim Namensdienst registrieren
if (reg != null)
{
  reg.rebind("AdderObjekt", adder);
  System.out.print("Adder bound!");
}
```

Der Clientcode könnte wie folgt aussehen:

```
// policy-Datei angeben
System.setProperty("java.security.policy", "adder.policy");
// SecurityManager installieren
System.setSecurityManager(new RMISecurityManager());
// URL definieren
String url = "rmi://147.88.100.100:port/AdderObjekt";
// Referenz auf das entfernte Objekt holen
Adder adderObj = (Adder) Naming.lookup(url);
// Methode 'add' des entfernten Objekts aufrufen
int sum = adderObj .add(13, 37);
// ...
```

Server starten (Serverklasse: *Server.class*):

```
java Server
```

Client starten (Client-Klasse: *Client.class*):

```
java Client
```

Übertragung von Objekten

Objekte werden aus einem in den anderen Speicherraum übertragen. Parameter und Rückgabewerte können primitive Datentypen, Lokale (gewöhnliche) Objekte und Referenzen auf Entfernte Objekte sein. Frage:

Wie kann ein Objekt aus einem Speicherraum in einen andren Speicherraum übertragen werden?

Für Übertragung von Parametern und Rückgabewerten zu bzw. von entfernten Methoden gelten folgende Regeln:

- Primitive Datentypen (**int**, **long**, …) werden *by value* übertragen
- Lokale Objekte
  - werden *by value* übertragen (echte Kopien)
  - Müssen serialisiert (*Stub*), übertragen und deserialisiert (*Skeleton*) werden
- Entfernte Objekte werden *by reference* übertragen
  - Das bedeutet, dass nur der Stub des entfernten Objekts übertragen wird und nicht die echte Kopie des entfernten Objekts.
  - Die *Stub*-Klasse wird vom RMI-Namensdienst dem RMI-Client nach seiner Anfrage zugestellt

Konsequenz:

Falls gewöhnliche Objekte als Parameter zu entfernten Methoden oder Rückgabewerte von entfernten Methoden übertragen werden müssen, müssen sie vom Typ ***java.io.Serializable*** sein.

Message Oriented Middleware (MOM)

MOM ermöglicht die Übertragung von Nachrichten, deren Format nicht festgelegt (vorgeschrieben) ist. Mit der MOM ist es möglich, sowohl die synchrone als auch asynchrone Kommunikation in einer verteilten Anwendung zu realisieren. Durch die Verwendung der MOM wird die Kopplung zwischen einzelnen Teilsystemen reduziert. Die Verwendung einer MOM bringt einige Vorteile mit sich:

- Lose Kopplung zwischen Kommunikationspartner
- Das ganze System wird stabiler, da Abhängigkeiten zwischen Teilsystemen reduziert
- Bessere Skalierbarkeit
- Die Belastung einzelner Komponenten kann besser abgefedert werden (Load Balancing)

Der grosse Nachteil ist der, dass der Ausfall der MOM zum Ausfall des ganzen Systems führt!

Java Message Service (JMS)

Java Message Service ist eine Schnittstelle, mit der das Senden und Empfangen von Nachrichten mit Hilfe einer MOM standardisiert werden soll. Sie ist Teil der JEE und befindet sich zur Zeit in der Version 1.1 (seit März 2002). Mit JMS werden zwei Ansätze zum Versenden von Nachrichten unterstützt:

- Nachrichtenschlangen (Message Queue)
- Anmelde-Versendesystem (Publish-Subscribe)

Wenn die Java Message Service Schnittstelle implementiert wird, steht eine konkrete Message Oriented Middleware bzw. ein **JMS Provider** zur Verfügung. Ähnliche Ansätze findet man auch bei JDBC und JNDI. Mit einer solchen MOM kann die Kommunikation zwischen unterschiedlichen Applikation bzw. deren Komponenten realisiert werden.

Ein Java Messages System besteht aus folgenden Teilen:

- Directory-Service (JNDI)
- JMS-Server (JMS Provider)
- JMS Clients
- Messages
- ConnectionFactory
- Destinationen (Queue und Topic)

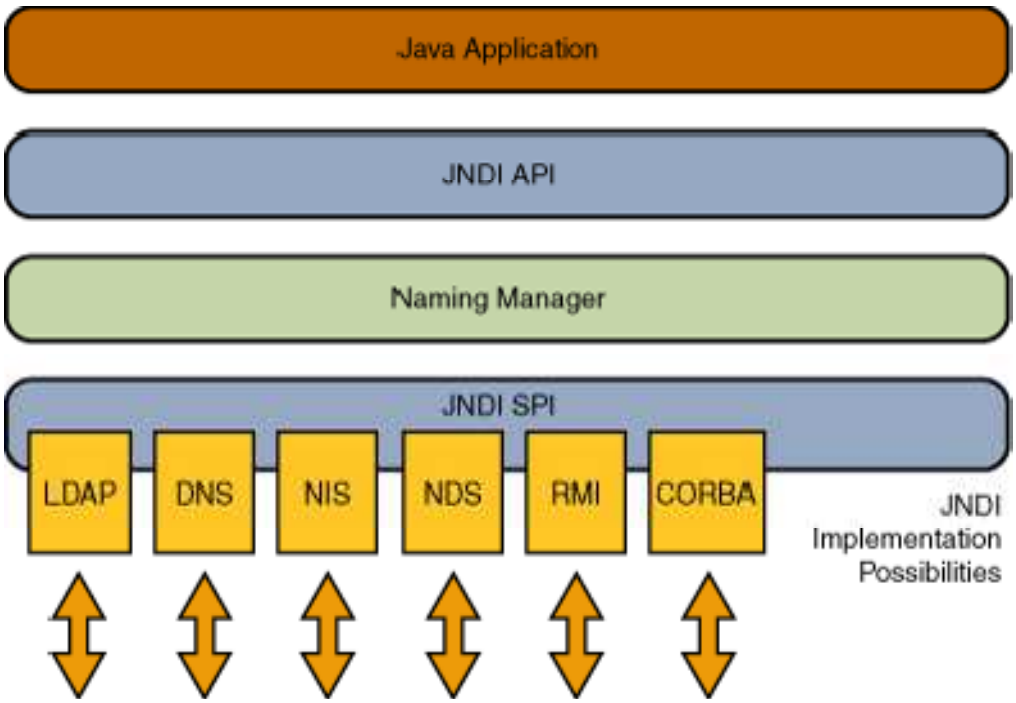
ConnectionFactories und Destinationen werden auch **administrierte** Objekte genannt, da sie von einem Administrator erzeugt und bei dem entsprechenden Verzeichnisdienst (JNDI-Provider) angemeldet werden.

Java Naming and Directory Interface (JNDI)

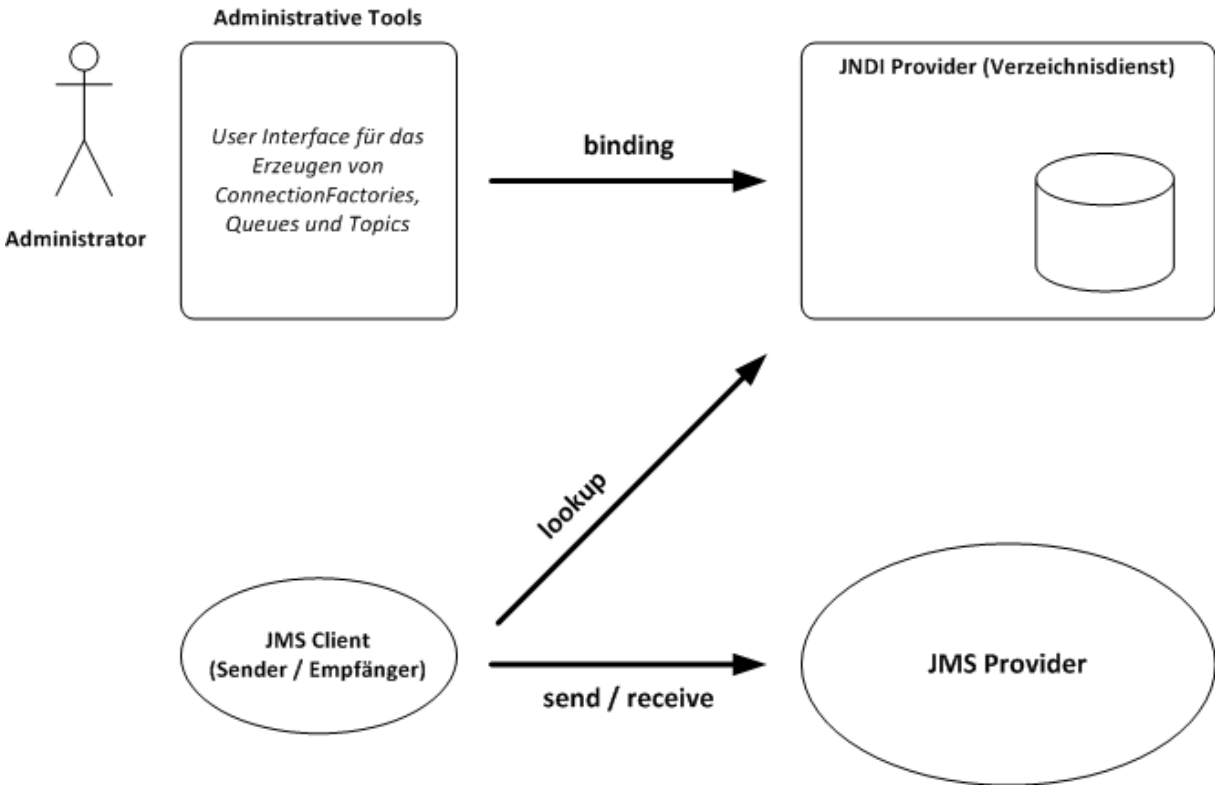
JNDI ist eine Schnittstelle für verzeichnisorientierte Dienste und soll den Zugriff auf Verzeichnisdienste vereinfachen bzw. standardisieren. In einem JNDI-Verzeichnisdienst (JNDI Provider) können unterschiedlichste Objekte verwaltet werden. Bei einem JNDI-Provider publizierte Objekte können von Clients abgefragt werden (*lookup*).

Java Naming and Directory Service

The JNDI architecture consists of an API and a service provider interface (SPI). Java applications use the JNDI API to access a variety of naming and directory services. The SPI enables a variety of naming and directory services to be plugged in transparently, thereby allowing the Java application using the JNDI API to access their services. See the following figure.

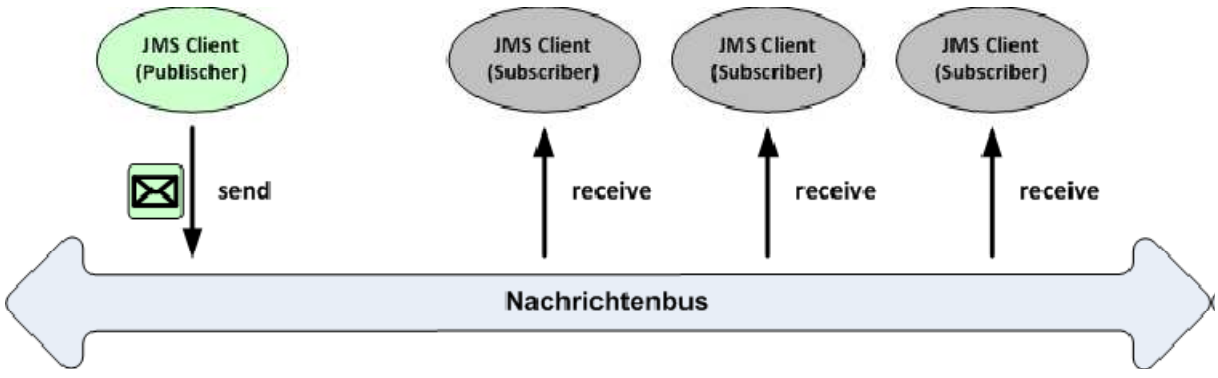


Eine Java Message System mit dessen Komponenten



Nachrichten Senden und Empfangen

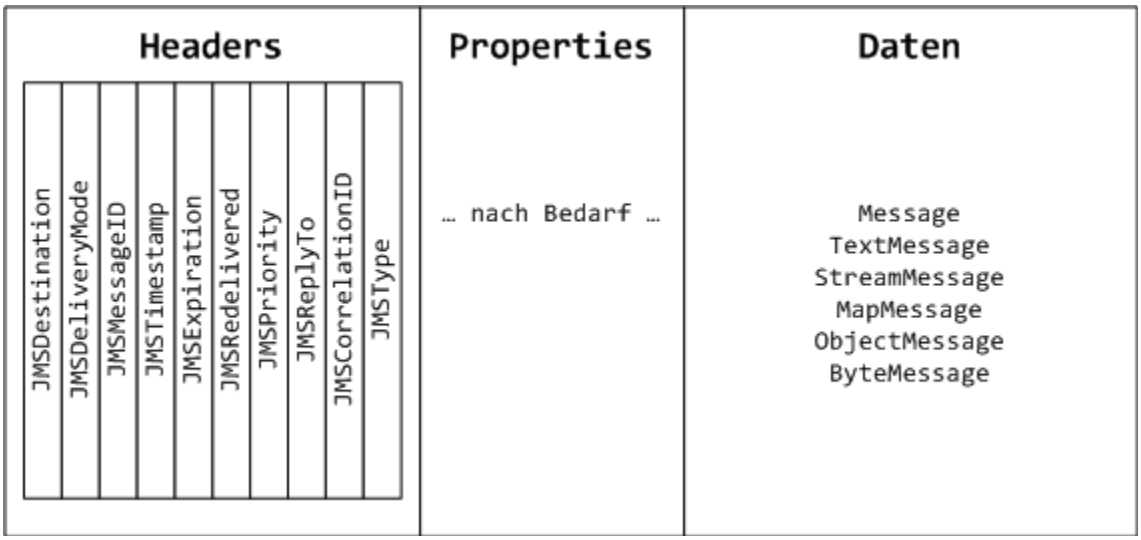
Nachrichten werden von JMS-Clients (Sender bzw. Empfänger) an bestimmte Destinationen gesendet, die vom Typ Queue oder Topic sein können. Der JMS-Provider stellt einen Nachrichtenbus zur Verfügung, mit dessen Hilfe die eingehenden Nachrichten an jene zugestellt werden, die sich als Empfänger (Konsumenten) angemeldet haben. Dazu wird die eingegangene Nachricht dupliziert und an den angemeldeten Konsumenten weitergeleitet, wenn er online ist. Falls er nicht online ist, wird die Nachricht "aufbewahrt", bis er wieder online ist (Store and Forward).



Message – Bestandteile

Eine JMS-Nachricht besteht aus folgenden Teilen: Headers, Properties und Nutzdaten

Headers	Die Adressinformationen und diverse Metadaten
Properties	Zusätzliche Eigenschaften, die fallbezogen definiert werden
Nutzdaten	Der Inhalt, der an sich von Interesse ist



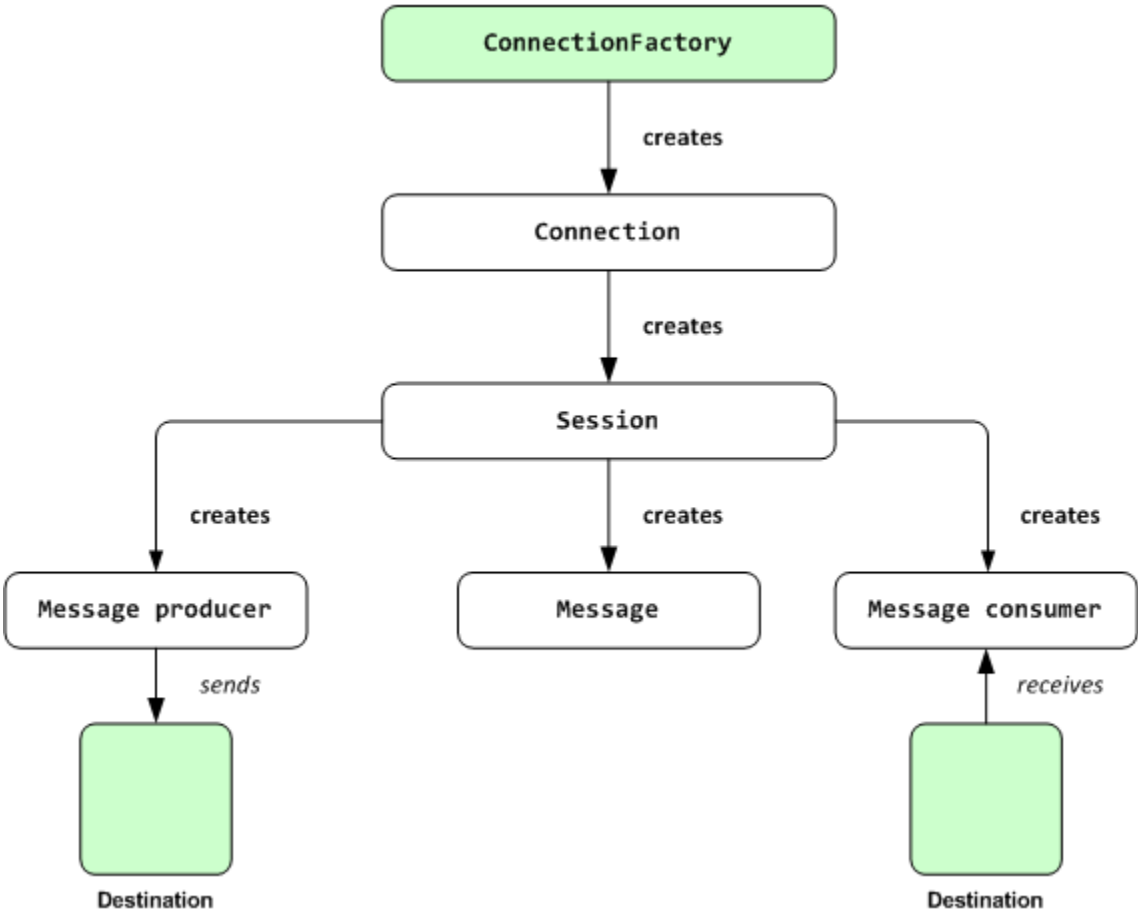
Administrierte Objekte

- **ConnectionFactory**:
  Wird für die Herstellung der Verbindung zwischen einem JMS-Client und JMS-Provider benutzt
- **Destination**:
  Die Ablage für Nachrichten, die Sender und Empfänger beim Nachrichtenaustausch benutzen (*Queues* und *Topics*)

ConnectionFactories und *Destinationen* werden vom Administrator erzeugt und beim JNDI-Dienst angemeldet (*binding*): daher **administrierte** Objekte.

JMS-Klassen im Überblick

Gemeinsam	Point-to-Point (P2P)	Publish-Subscribe (P/S)
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver	TopicSubscriber



Nachrichtenmodelle

Es wird zwischen zwei Nachrichtenmodelle unterschieden: Point to Point (P2P) und Publish-Subscribe (PS). Point to Point (P2P):

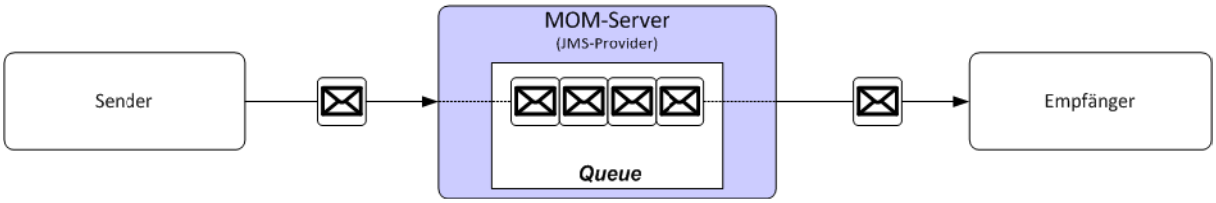
- Sender und Receiver
- Kommunikation über einer Queue als Destination: der Sender stellt die Nachricht in die Queue, der Empfänger nimmt die Nachricht an, sobald er bereit ist

Publish-Subscribe (PS):

- Kommunikation über einem Topic (Thema) als Destination
- Ein oder mehrere Publisher stellen Nachrichten in ein Topic: Ein oder mehrere Subscriber nehmen die Nachrichten aus dem Topic

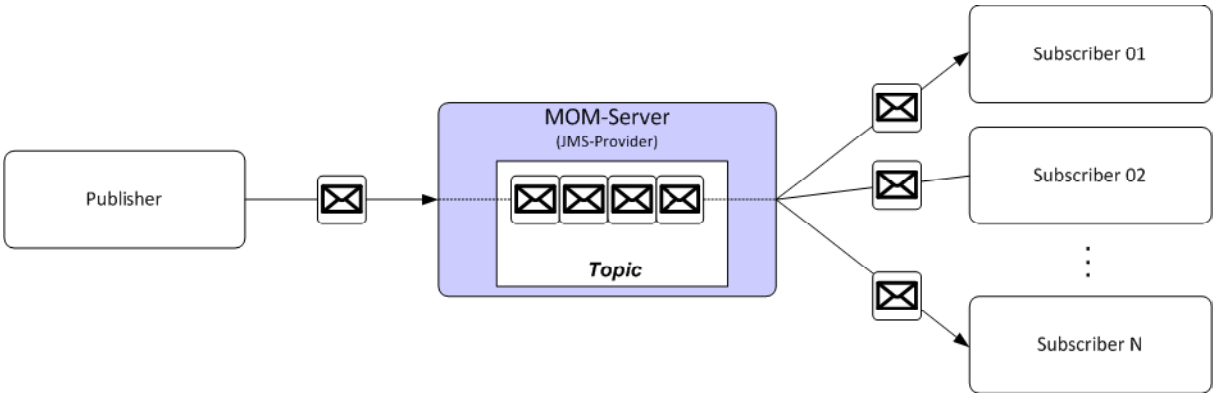
Point-to-Point

Nachrichten werden zwischen einem Sender und einem Empfänger ausgetauscht. Beide Kommunikationspartner nutzen die gleiche Nachrichtenschlange (Queue) und sind somit "fest verdrahtet".



Publish-Subscribe

Nachrichten können von mehreren Empfängern abonniert bzw. konsumiert werden. Der *Publischer* weiss nicht, von welchem *Subscriber* die von ihm publizierte Nachricht konsumiert wird.



Point-to-Point vs. Publish-Subscribe

Point-to-Point:

Nachrichten werden aufbewahrt, bis der Empfänger bereit ist, diese abzuholen: kein Verlust von Nachrichten, wenn der Empfänger nicht empfangsbereit ist.

Publish-Subscribe:

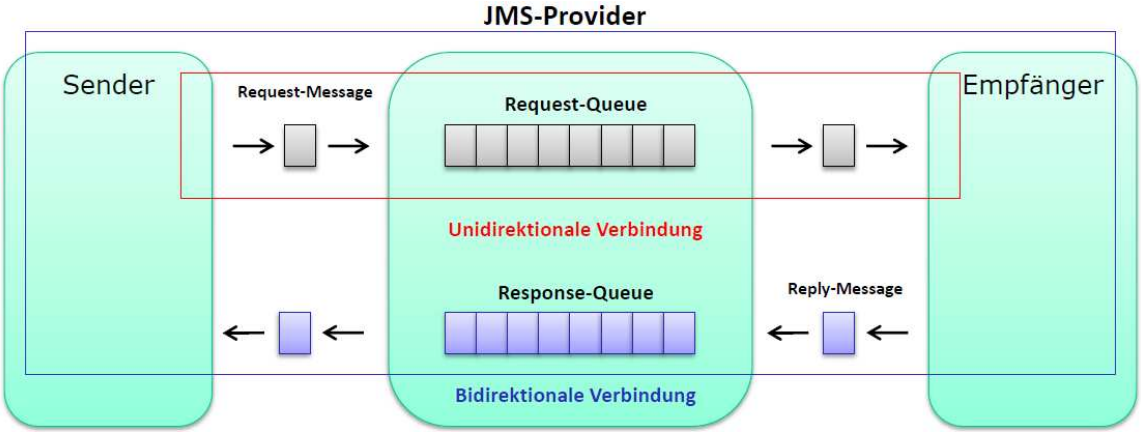
- Alle Empfänger (Subscriber) bekommen Nachrichten, die in ein Topic publiziert werden, und zwar nur wenn sie mit dem Topic aktiv verbunden sind (online)
- Wenn sie nicht "online" sind, werden sie Nachrichten, die während dieser Zeit publiziert werden, nicht erhalten: so sind solche Nachrichten für Subscriber für immer verloren
- Option: dauerhaftes Abonnieren (*persistent subscription*)

One-Way und Request-Reply

Bei der asynchronen Kommunikation wird in der Regel die One- Way MEP (Message Exchange Pattern) verwendet: Der Sender senden die Nachricht und erwartet keine Antwort

Mit P2P Nachrichtenmodell kann jedoch auch das **Request-Reply** MEP realisiert werden:

- Der Sender braucht die Antwort, um weiter arbeiten zu können
- Dazu wird eine temporäre Queue angelegt:
  - Der Empfänger stellt die Antwort in diese Queue, der Sender liest die Antwort aus der Queue (Rollenwechsel)
  - Die Queue wird anschliessend gelöscht



JMS Provider ActiveMQ

ActiveMQ ist eine frei verfügbare Implementierung der JMS-Spezifikation 1.1. Features:

- Unterstützung für viele Programmiersprachen
- Point-to-Point und Publish-Subscribe Nachrichtenmodell
- Synchrone und asynchrone Zustellung von Nachrichten
- Persistierung von Nachrichten mittels JDBC
- Web UI für Administrator
- XML-Basierte Konfigurationsmöglichkeit
- Unterstützung von mehreren Protokollen (TCP, HTTP, STOMP …)
- Unterstützung für Web-Sockets (Jetty)
- Unterstützung für WebServices-Frameworks CXF und Axis

ActiveMQ stellt auch einen einfachen JNDI-Server zur Verfügung.

Setzten von Properties für InitialContext(jndi.properties)

Für das Erzeugen der *InitialContext*-Instanz müssen folgende Properties angegeben werden:

- `java.naming.factory.initial`
- `java.naming.provider.url`

Dazu stehen zwei Möglichkeiten zur Verfügung:

- Entweder werden die beiden Properties in eine `Hashtable` abgelegt, die dem Konstruktor der Klasse `InitialContext` übergeben wird, oder
- Die beiden Properties werden in der `jndi.properties` Datei angegeben, die ihrerseits im CLASSPATH sein muss

Die Verwendung der `jndi.properties` Datei ist einfacher, da kein Eingriff in den Code nötig ist, falls etwas evtl. angepasst werden muss. Beispiel für `jndi.properties`:

```
#============================================================#
#                    JNDI Properties                         #
#============================================================#
# InitialContextFactory
java.naming.factory.initial=org.apache.activemq.jndi.ActiveMQInitial
ContextFactory
# Provider URL
java.naming.provider.url=tcp://10.9.35.119:61616
# Queue (Syntax: queue.[jndiName]=[phisicalName]
queue.myWeatherdataQueue=myWeatherdataQueue
```

Nachrichten austauschen

Code des Senders:

```
/* IntialContext erzeugen, ConnectionFactory und Queue 'myWeatherdataQueue' holen */
InitialContext jndiContext = new InitialContext();
ConnectionFactory conFactory = (ConnectionFactory) jndiContext.lookup("ConnectionFactory");
Queue queue = (Queue) jndiContext.lookup("myWeatherdataQueue");

/* Connection erstellen und anschliessend Session von der Connection holen */
Connection con = conFactory.createConnection();
Session session = (Session) con.createSession(false, Session.AUTO_ACKNOWLEDGE);

/* Sender erstellen */
MessageProducer sender = session.createProducer(queue);

/* Die ObjectMessage aus dem WeatherdataMessage-Objekt erzeugen */
ObjectMessage objMsg = session.createObjectMessage(weatherdataMessage);

/* Message senden */
sender.send(objMessage);
```

Code des Empfängers:

```
/* IntialContext erzeugen, ConnectionFactory und Queue 'myWeatherdataQueue' holen */
InitialContext jndiContext = new InitialContext();
ConnectionFactory conFactory = (ConnectionFactory) jndiContext.lookup("ConnectionFactory");
Queue queue = (Queue) jndiContext.lookup("myWeatherdataQueue");

/* Connection erstellen und anschliessend Session von der Connection holen */
Connection con = conFactory.createConnection();
Session session = (Session) con.createSession(false, Session.AUTO_ACKNOWLEDGE);

/* Empfänger erstellen */
MessageConsumer receiver= session.createConsumer(queue);

/* Connection starten */
connection.start();

/* Message empfangen */
ObjectMessage objMsg= (ObjectMessage)receiver.receive();
// Message verarbeiten ...
```

Pull- und Push-Prinzip

Empfänger-Komponente (Consumer) kann auf das Eintreffen aktiv warten (**pull**-Prinzip) oder durch die Callback-Methode über das Eintreffen einer nachricht informiert werden (**push**-Prinzip).

Callback-Methode:

- Schnittstelle javax.jsm.MessageListener
- Methode onMessage:
  `public void onMessage(Message message)`

Aufbau des Systems

Damit die Kommunikation realisiert werden kann, werden folgende Komponenten benötigt:

- JMS Provider (in diesem Fall **ActiveMQ**)
- JNDI Verzeichnisdienst (in diesem Fall im JMS-Provider integriert)
- Kommunikationswillige Komponenten (JMS-Clients)

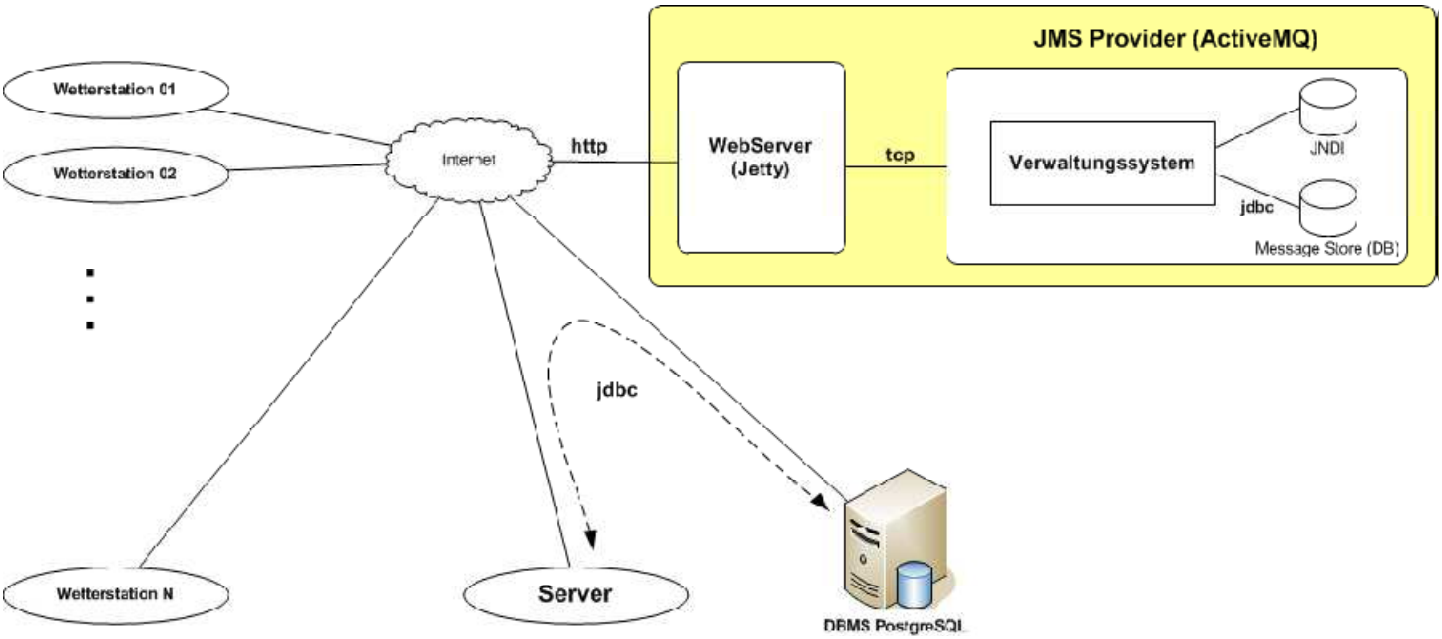
Standardmässig wird die Kommunikation über *openwire*-Protokoll (Port 61616) realisiert. Falls die Kommunikation über http realisiert werden soll, muss dies in *activemq.xml* noch explizit konfiguriert werden:

```
<transportConnectors>
<transportConnector name="openwire" uri="tcp://0.0.0.0:61616"/>
<!-- HTTP-Connector: Muss explizit konfiguriert werden -->
<transportConnector name="http" uri="http://0.0.0.0:8080"/>
</transportConnectors>
```

Die Abwicklung der Kommunikation über http ermöglicht der integrierte WebServer (Jetty).
ActiveMQ – Web UI http://hostIp:8161/admin

Demo-Applikation

In mehreren Skigebieten sollen Wetterdaten laufend erfasst und einer zentralen Komponente zugestellt werden, die alle gemessenen Werte in einen vorgegebenen Format überführt und in die Datenbank dauerhaft ablegt. So aufbereitete Daten werden von anderen Applikationen zu diversen Zwecke benötigt. Für die Zustellung von Wetterdaten soll eine MOM eingesetzt werden. Des Weiteren soll für die Übertragung von Daten aus Sicherheitsgründen das HTTP Protokoll verwendet werden. Die Applikation muss so erstellt werden, dass die Anzahl Messstationen beliebig erhöht werden kann, ohne dass der Eingriff in den Applikationscode nötig wird.



Webservices

Einführung

Web Services sind

- im Internet verfügbare Dienste, die von Clients über URLs angesprochen werden können
- Komponenten, die bestimmte Funktionalitäten über wohldefinierten Schnittstellen zur Verfügung stellen

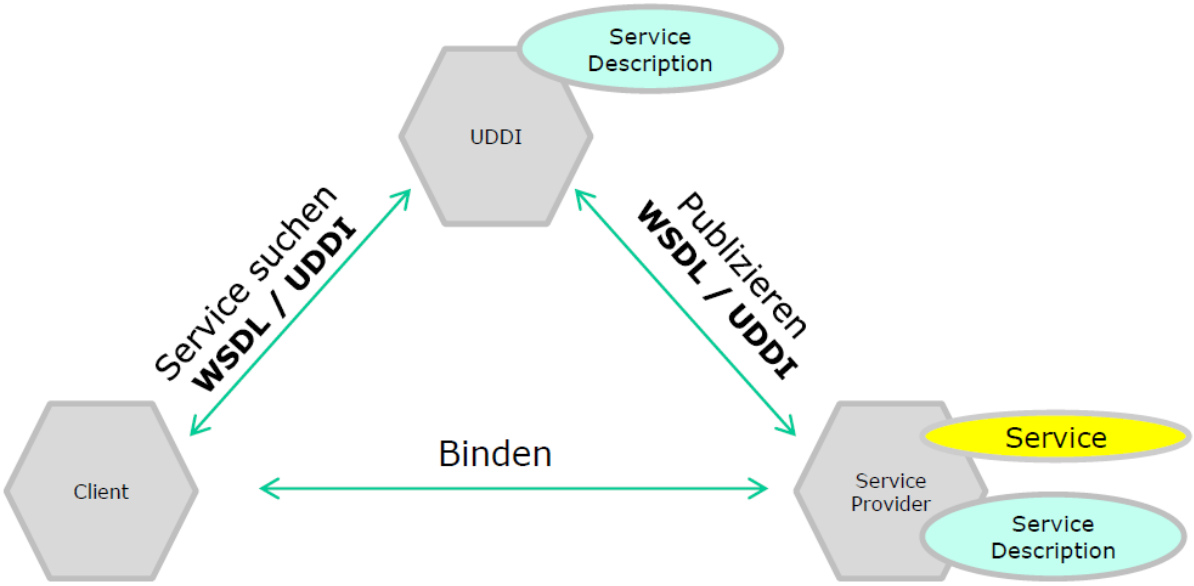
Web Services dienen "als Unterstützung zur Zusammenarbeit zwischen verschiedenen Anwendungsprogrammen, die auf unterschiedlichen Plattformen und/oder Frameworks betrieben werden"

Web Services basieren auf offenen Protokollen, sind "selbstbeschreibend" (WSDL) und können registriert und gefunden werden (UDDI). Die Basisplattform für Web Services: XML + http.

Grundelemente (Plattform) :

- SOAP (**S**imple **O**bject **A**ccess **P**rotocol)
- UDDI (**U**niversal **D**escription, **D**iscovery und **I**ntegration)
- WSDL (**W**eb **S**ervice **D**escription **L**anguage)

Web Services Architektur:



SOAP

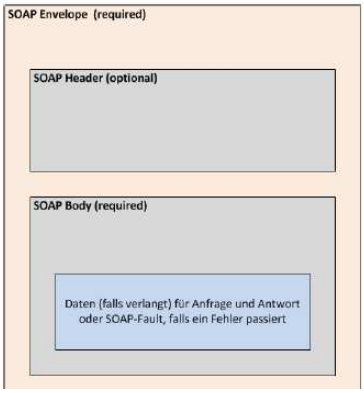
SOAP ist ein leichtgewichtiges Kommunikationsprotokoll für den Austausch von Daten in einer dezentralisierten, verteilten Umgebung. SOAP basiert auf XML und wurde für die Kommunikation im Internet entwickelt. Es ist Plattform- und programmierspracheunabhängig und wird in Kombination mit HTTP als Übertragungsprotokoll von Firewalls "durchgelassen".

Eine SOAP-Nachricht ist ein XML-Dokument und besteht aus folgenden Teilen:

- Envelope (Umschlag)
- Header (Header-Informationen)
- Body (Anfrage- und Antwort-Informationen)
- Fault (Informationen zu den Fehlern)

Beispiel:

```
<?xml version="1.0" encoding="UTF-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

    <soap:Header>
        <!-- Header-Informationen -->

    </soap:Header>

    <soap:Body>
        <!-- Body-Informationen -->

        <soap:Fault>
            <!-- Fehler-Informationen -->

        </soap:Fault>
    </soap:Body>
</soap:Envelope>
```



SOAP-Envelope

- markiert das XML-Dokument als SOAP-Nachricht
- enthält die Namensraumangaben für SOAP-Envelope und "encoding style"

SOAP-Header

- ist optional: falls vorhanden, muss als erstes Element erscheinen
- enthält anwendungsspezifische Daten (Authentifizierung, Zahlungsdaten …)
- kann auch weitere Angaben enthalten (*mustUnderstand*, *actor*, *encodingStyle*)

SOAP-Body

- muss vorhanden sein und enthält die Nutzdaten
- kann optional das Subelement Fault (Fehlerinformationen) enthalten

Beispiel für SOAP Request:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPrice>
    <m:StockName>IBM</m:StockName>
  </m:GetStockPrice>
</soap:Body>
</soap:Envelope>
```

Beispiel für SOAP Response:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>
</soap:Envelope>
```

WSDL

WSDL ist eine XML-basierte Sprache für die Beschreibung von Web Services (analog zu IDL). Sie ist nicht von darunterliegendem Protokoll abhängig. Für Menschen ist WSDL nicht besonders verständlich. Mit WSDL wird auch angegeben

- wie der Web Service zu finden ist (URL) und
- welche Operationen (Methoden) der Service zur Verfügung stellt

Ein WSDL Dokument besteht aus mehreren Elementen:

- ***Types***
- ***Messages***
- ***portType***
- ***binding***

WSDL Types:

- Definiert die benutzten Datentypen
- soll mit XML-Schema realisiert werden (max. Portabilität)

WSDL Messages:

- enthält Daten der einzelnen Operationen
- kann mit Parametern "verglichen" werden

WSDL Ports:

- das wichtigste WSDL-Element: beschreibt den Web Service, Operationen die Ausgeführt werden können und Nachrichten, die gesendet werden
- kann mit einem Modul (einer Bibliothek) verglichen werden

WSDL Bindings:

- definiert das Nachrichtenformat und
- das Protokoll für jeden Port

UDDI

UDDI ist ein plattformunabhängiger Verzeichnisdienst (sehr mächtig), wird aber auf Internet-Ebene weniger gebraucht. UDDI ermöglicht

- das Beschreiben von Web Services
- die Registrierung (Anmeldung, Publizierung) von Web Services
- die Suche nach Web Services

In der Praxis wird ein UDDI-Dienst eher auf Unternehmensebene betrieben.

Webservice - Beispiel in Java

Zeit-Abfrage Service

Eingesetztes Framework:

- Metro: die RI für JAX-WS
- Homepage: http://metro.java.net/

WebServer:

- Klasse javax.xml.ws.Endpoint (siehe API)

Service-Aufgabe:

Aktuelle Zeit für diverse Lokationen weltweit liefern

Vorgehensweise:

Code First: Vom Java-Code zu WSDL

Schritt 1: Interface definieren

```
package time.model;
import javax.jws.*;
import java.util.List;

@WebService
public interface Time {
  @WebMethod
  long getCurrentTime(@WebParam(name = "cityName") String cityName);

  @WebMethod
  List<String> getAvailableCityNames();
}
```

Schritt 2: Implementierende Klasse erstellen

```
package time.business;
import javax.jws.WebService;
import time.model.Time;

@WebService(endpointInterface = "time.model.Time")
public class TimeImpl implements Time {
  public long getCurrentTime(String cityName) {
    // Implementierung ...
  }
  public List<String> getAvailableCityNames() {
    // Implementierung ...
  }
}
```

Schritt 3: WebService publizieren

```
package time;

public class Publisher {
  public static void main(String[] args) {
    // Service-Objekt erstellen
    Time service = new TimeImpl();
    // URI definieren
    String uri= "http://localhost:9090/timeService";
    // WebService publizieren
    Endpoint ePoint = Endpoint.publish(uri,service);
    // Dialog zum Beenden des WebServices anzeigen
    JOptionPane.showMessageDialog(null, "Server beenden");
    // WebService-Ausführung beenden
    ePoint.stop();
  }
}
```

Schritt 4: WSDL anzeigen – I

URI: http://localhost:9090/timeService?wsdl

```
<definitions targetNamespace="http://business.time/" name="TimeImplService">
  <import namespace="http://model.time/" location="http://localhost:9090/timeService?wsdl=1"/>
  <binding name="TimeImplPortBinding" type="ns1:Time">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    <operation name="getCurrentTime">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
    <operation name="getAvailableCityNames">
      <soap:operation soapAction=""/>
      <input>
        <soap:body use="literal"/>
      </input>
      <output>
        <soap:body use="literal"/>
      </output>
    </operation>
  </binding>
  <service name="TimeImplService">
    <port name="TimeImplPort" binding="tns:TimeImplPortBinding">
      <soap:address location="http://localhost:9090/timeService"/>
    </port>
  </service>
</definitions>
```

```
- <definitions targetNamespace="http://business.time/" name="TimeImplService">
    <import namespace="http://model.time/" location="http://localhost:9090/timeService?wsdl=1"/>
  - <binding name="TimeImplPortBinding" type="ns1:Time">
      <soap:binding transport="http://schemas.xmlsoap.org/soap/http" style="document"/>
    - <operation name="getCurrentTime">
        <soap:operation soapAction=""/>
      - <input>
          <soap:body use="literal"/>
        </input>
      - <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    - <operation name="getAvailableCityNames">
        <soap:operation soapAction=""/>
      - <input>
          <soap:body use="literal"/>
        </input>
      - <output>
          <soap:body use="literal"/>
        </output>
      </operation>
    </binding>
  - <service name="TimeImplService">
    - <port name="TimeImplPort" binding="tns:TimeImplPortBinding">
        <soap:address location="http://localhost:9090/timeService"/>
      </port>
    </service>
  </definitions>
```

Schritt 4: WSDL anzeigen – II

URI: http://localhost:9090/time?wsdl=1

```
- <definitions targetNamespace="http://model.time/">
  - <types>
    - <xsd:schema>
        <xsd:import namespace="http://model.time/" schemaLocation="http://localhost:9090/timeService?xsd=1"/>
      </xsd:schema>
    </types>
  - <message name="getCurrentTime">
      <part name="parameters" element="tns:getCurrentTime"/>
    </message>
  - <message name="getCurrentTimeResponse">
      <part name="parameters" element="tns:getCurrentTimeResponse"/>
    </message>
  - <message name="getAvailableCityNames">
      <part name="parameters" element="tns:getAvailableCityNames"/>
    </message>
  - <message name="getAvailableCityNamesResponse">
      <part name="parameters" element="tns:getAvailableCityNamesResponse"/>
    </message>
  - <portType name="Time">
    - <operation name="getCurrentTime">
        <input message="tns:getCurrentTime"/>
        <output message="tns:getCurrentTimeResponse"/>
      </operation>
    - <operation name="getAvailableCityNames">
        <input message="tns:getAvailableCityNames"/>
        <output message="tns:getAvailableCityNamesResponse"/>
      </operation>
    </portType>
  </definitions>
```

Schritt 4: WSDL anzeigen – III

URI: http://localhost:9090/time?xsd=1

```
−<xs:schema version="1.0" targetNamespace="http://model.time/">
    <xs:element name="getAvailableCityNames" type="tns:getAvailableCityNames"/>
    <xs:element name="getAvailableCityNamesResponse" type="tns:getAvailableCityNamesResponse"/>
    <xs:element name="getCurrentTime" type="tns:getCurrentTime"/>
    <xs:element name="getCurrentTimeResponse" type="tns:getCurrentTimeResponse"/>
  −<xs:complexType name="getAvailableCityNames">
      <xs:sequence/>
    </xs:complexType>
  −<xs:complexType name="getAvailableCityNamesResponse">
    −<xs:sequence>
        <xs:element name="return" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  −<xs:complexType name="getCurrentTime">
    −<xs:sequence>
        <xs:element name="cityName" type="xs:string" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  −<xs:complexType name="getCurrentTimeResponse">
    −<xs:sequence>
        <xs:element name="return" type="xs:long"/>
      </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Schritt 5: Client-Artefakte generieren – I

Vorgehen:

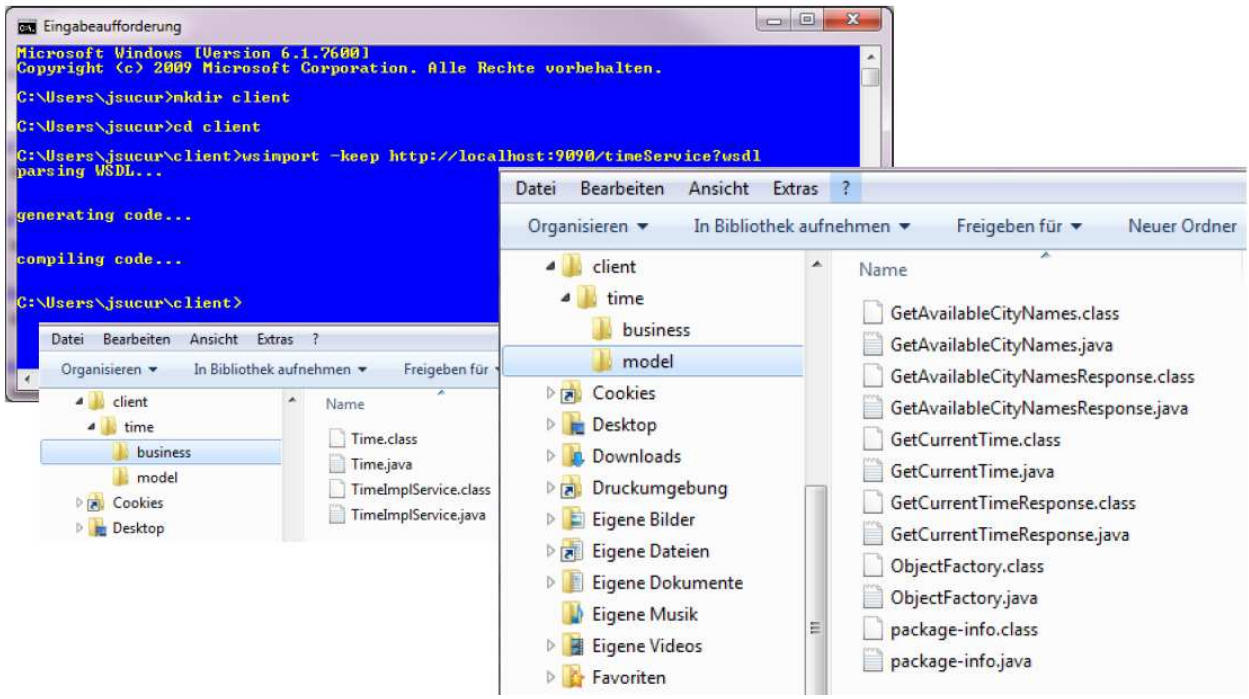
- An Hand der WSDL werden die Klassen generiert, die auf der Client-Seite für die Kommunikation mit dem WebService benötigt werden (Stub-Klassen).
- Die Generierung wird durch das Tool *wsimport* vorgenommen.
- Client implementieren

Praktisch:

ein neues Projekt für die Client-Komponente erstellen und im Quellcode-Verzeichnis des Client-Projekts wsimport aufrufen:

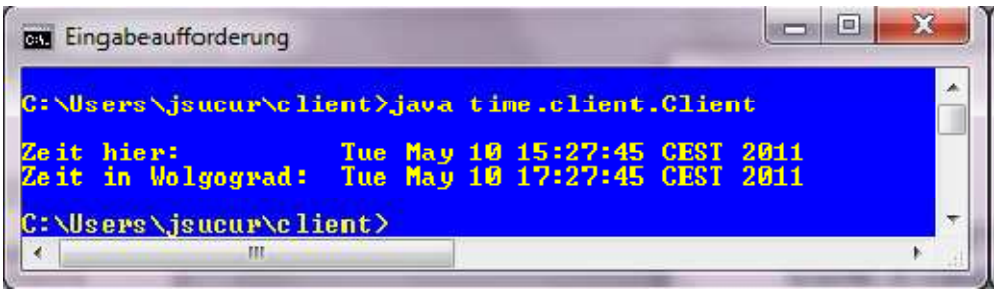
```
wsimport -keep http://localhost:9090/timeService?wsdl
```

Die nötigen Client-Artefakte werden generiert und kompiliert (mit **–keep** wird auch der Quellcode beibehalten).



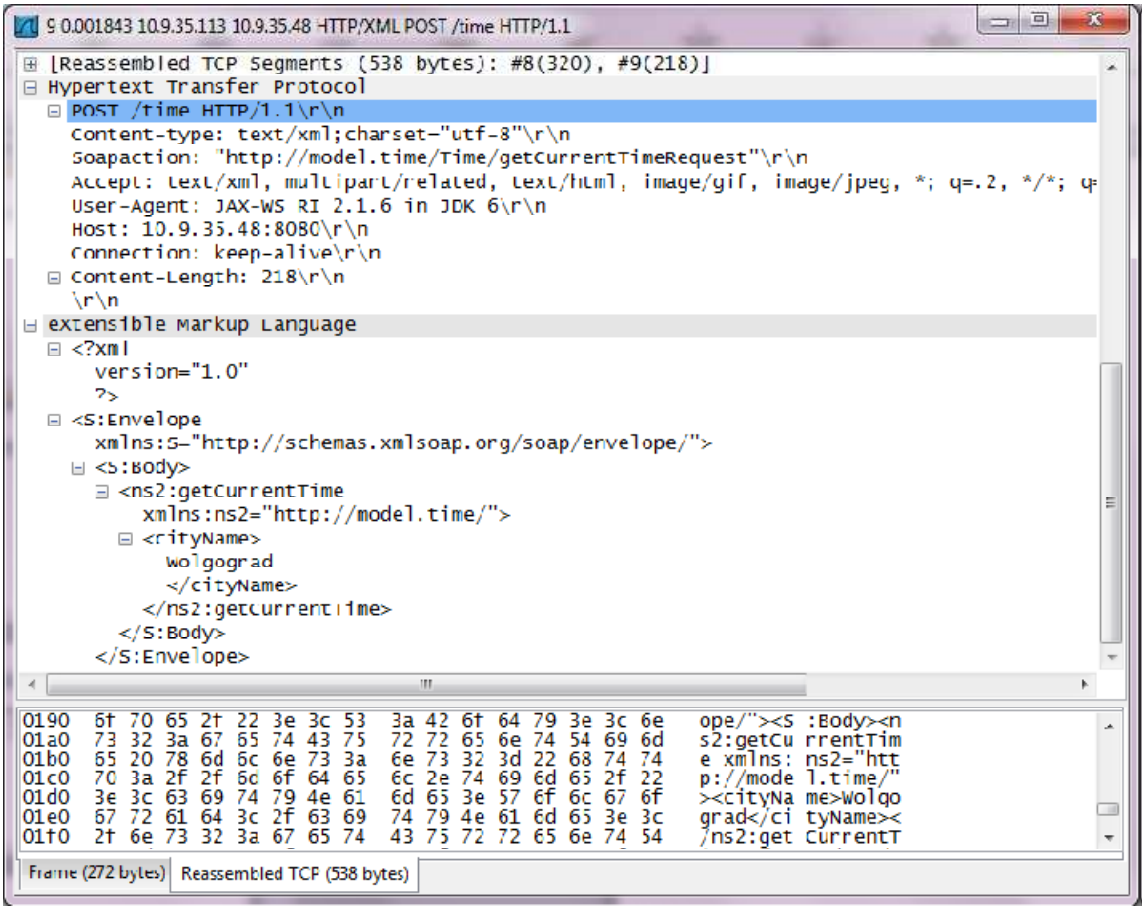
Schritt 6: Client implementieren

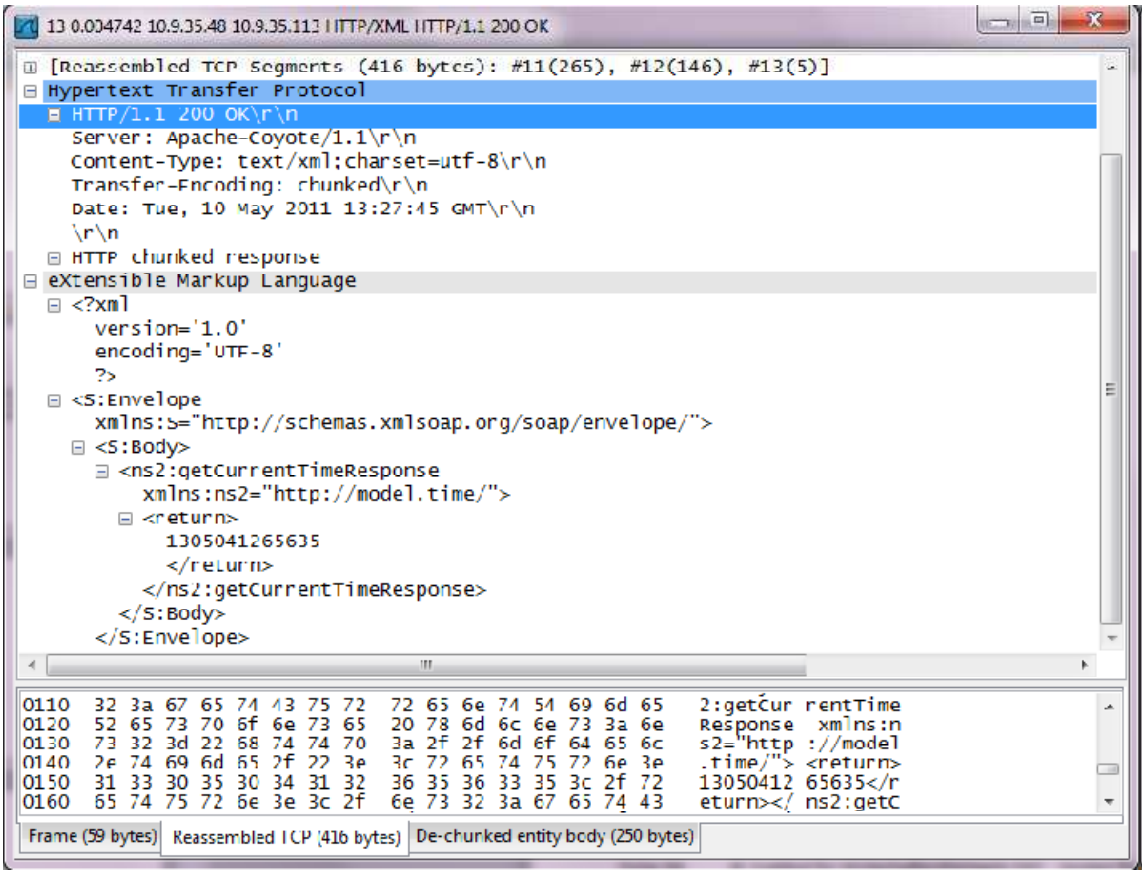
```
package time.client;
import java.util.*;
import time.business.Time;
import time.business.TimeImplService;

public class Client {
  public static void main(String[] args) {
    /* Service kreieren */
    TimeImplService service = new TimeImplService();
    /* Proxy kreieren (Client-Stub) */
    Time proxy = service.getTimeImplPort();
    /* Aktuelle Zeit in Wolgograd abfragen */
    long timeInMillis = proxy.getCurrentTime("Wolgograd");
    Date d = new Date(timeInMillis); /* Ausgabe */
    System.out.println("Zeit hier: " + new Date());
    System.out.println("Zeit in Wolgograd: " + d);
  }
}
```



SOAP-Messages

SOAP-Message – Request + Response





JAX-WS API

javax.jws enthält diverse Anotationen:

- Oneway
- WebMethod
- WebParam
- WebResult
- WebService

javax.xml.ws enthält diverse Artefakte:

- Interfaces: AsyncHandler, Response …
- Endpoint, Holder, WebServicePermission …
- Diverse Annotationen

Übertragung von primitiven und komplexen Datentypen (Objekten) möglich. Annotation @XmlType:

```
javax.xml.bind.annotation.XmlType
```

Message-Exchange-Pattern (MEP)

- RequestReplay (default)
- OneWay muss explizit mit Annotation **@Oneway** gekennzeichnet werden

```
javax.jws.Oneway
```

Strukturierte Datentypen:

```
@XmlType (propOrder={ "phone", "email" })
public class Contact{
  private String phone;
  private String email;
  // weitere Implementierung ...
}
```

OneWay-MEP:

```
@WebService
public interface MessageSender{
  @Oneway
  void sendMessage(@WebParam(name="message") String message)
}
```

Asynchrone Aufrufe auch möglich

- die Programmausführung kehrt sofort zurück
- das Ergebnis wird asynchron zur Verfügung gestellt

Realisierung:

- Pooling:
  - Client fragt periodisch bei seinem Stub, ob das Ergebnis verfügbar ist
  - Die Verfügbarkeit wird durch die Methode *isDone* der Schnittstelle Response abgefragt
- Callback:
  - Die Client-Anwendung muss einen *AsyncHandler* zur Verfügung stellen, der
  - die Schnittstelle javax.xml.ws.Asynchandler<T> implementiert und
  - das Resultat in der Methode handleResponse verwertet

Java & .Net

Es existieren diverse Java-WebServices-Platformen

- Metro: http://jax-ws.java.net/
- Apache CXF: http://cxf.apache.org/
- Apache Axis2: http://axis.apache.org/axis2/java/core/
- Vergleich: http://www.predic8.de/axis2-cxf-jax-ws-vergleich.htm

Für das Hosten stehen diverse Server zur Verfügung (Tomcat, Jboss, …).In der "Microsoft-Welt" sind die WebServices Teil der Sprache C# (dadurch auch einfach im Gebrauch). Für das Hosten wird der Microsoft-Server IIS benötigt

WebServices mit Tomcat publizieren

Tomcat

Ist eine **R**eferenz-**I**mplementierung für Servlet-Container. Aktuelle Version: 7.x

Installation und Konfiguration:

- Umgebungsvariable CATALINA_HOME setzen
- evtl. bnötigten Bibliotheken dem Tomcat passend hinzufügen

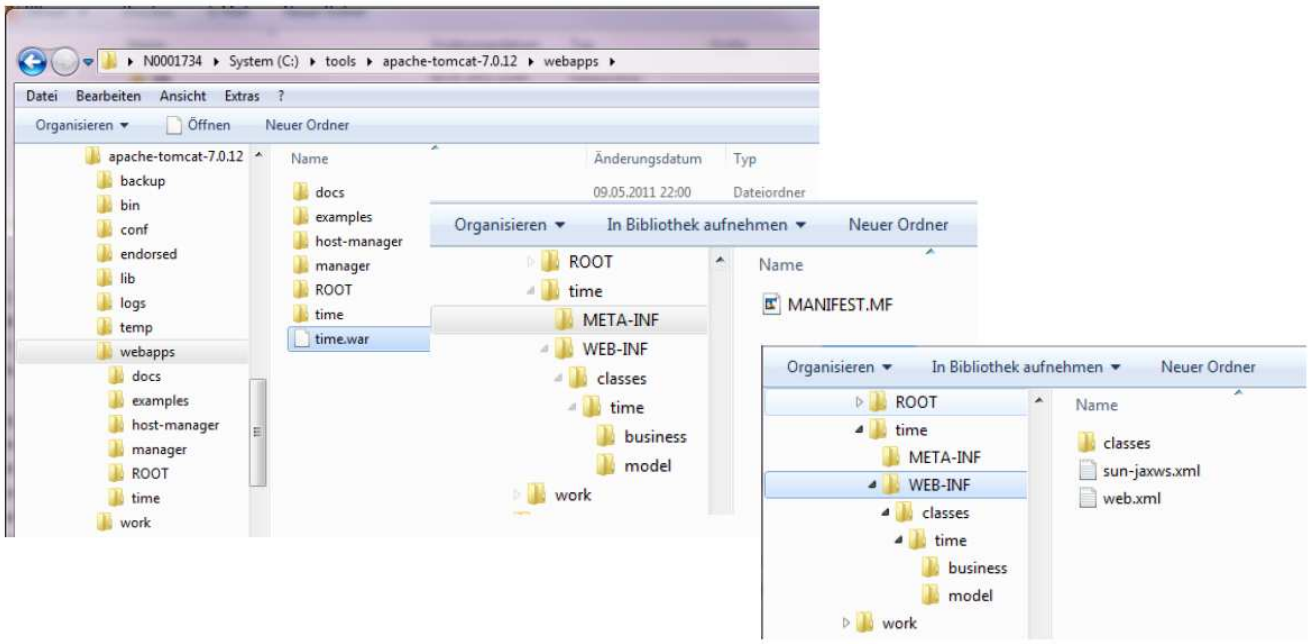
Starten:

<CATALINA_HOME>\bin\startup.bat

Kontrollieren:

http://localhost:8080 (Port-Nr. evtl. anpassen)

Web-Anwendungen werden i.d.R. als WebArchive Dateien (*.war) verpackt und ins Verzeichnis **CATALINA_HOME>\webapps** abgelegt. Die Struktur von **war**-Archiven ist vorgegeben und muss eingehalten werden.



Web-Deployment-Descriptor: web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
        <!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
        "http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">

<web-app>
      <listener>
            <listener-class>com.sun.xml.ws.transport.http.servlet.WSServletContextListener</listener-class>
      </listener>
      <servlet>
            <servlet-name>TimeServlet</servlet-name>
            <servlet-class>com.sun.xml.ws.transport.http.servlet.WSServlet</servlet-class>
            <load-on-startup>1</load-on-startup>
      </servlet>
      <servlet-mapping>
            <servlet-name>TimeServlet</servlet-name>
            <url-pattern>/*</url-pattern>
      </servlet-mapping>
      <session-config>
            <session-timeout>180</session-timeout>
      </session-config>
</web-app>
```

JAX-WS RI Deployment-Descriptor: sun-jaxws.xml

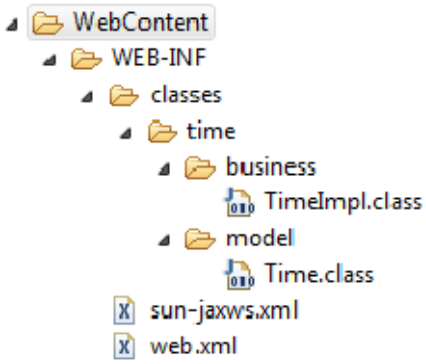
```
<?xml version="1.0" encoding="UTF-8"?>
<endpoints xmlns="http://java.sun.com/xml/ns/jax-ws/ri/runtime" version="2.0">
     <endpoint name="time" implementation="time.business.TimeImpl" url-pattern="/*" />
</endpoints>
```

Das Element `endpoints` enthält ein oder mehrere `endpoint`-Elemente, die je einen WSDL-Port repräsentieren. Das Element endpoint enthält die Angaben zu der implementierenden Klasse (Service-Klasse) und URL-Mapping.

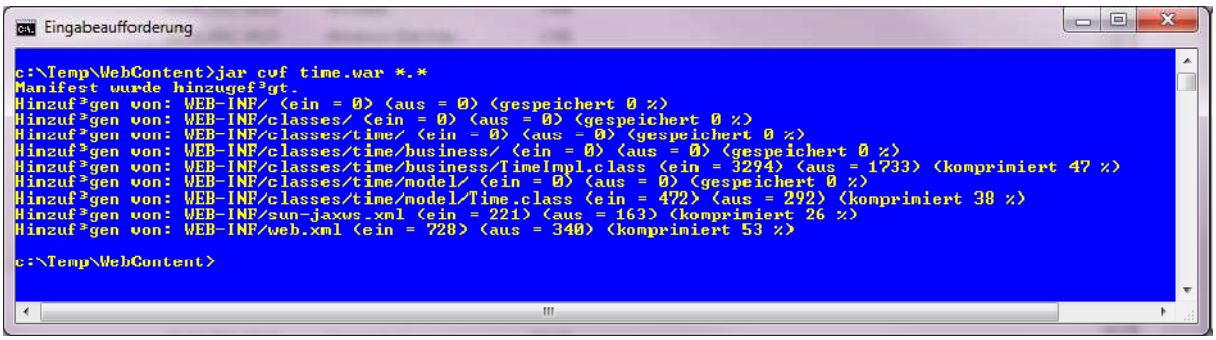
war-Datei "von Hand" erstellen

Voraussetzung:

Alle benötigten Artefakte im WebContent-Verzeichnis enthalten, wobei die Struktur der Web-Anwendung eingehalten wird.



time.war generieren:



Die benötigten (fremden) Bibliotheken müssen dem Tomcat-Server zur verfügung gestellt werden. Entweder durch das Kopieren von Bibliotheken in entsprechende Verzeichnisse oder das Anpassen der *shared.loader*-Property in der *catalina.properties* Datei:

```
shared.loader=<METRO_HOME>/lib/*.jar
```

Die (gerade generierte) Datei time.war in **<CATALINA_HOME>/webapps** kopieren und den Server starten. Kontrolle:

http://localhost:8080/time?wsdl sollte das WSDL-Dokument im Browser anzeigen

Build-Tool Ant

Während Entwicklung werden in einem Projekt oft zahlrieche wiederholende Aufgaben erledigt: Kompilieren von Klassen, Erstellen der API-Dokumentation, Kopieren von jar-Dateien, Erstellen von Archive-Datei (jar / war / ear) etc. Solche Arbeiten werden mit der Zeit sehr mühsam und können von entsprechenden Tools übernommen werden. In der Java-Welt hat sich das Tool **Apache Ant** durchgesetzt: http://ant.apache.org.

Grundidee:

- Zu erledigende Aufgaben werden als `tasks` definiert, die `tasks` werden in der `build.xml` zusammengefasst
- Durch die Möglichkeit, die Abhängigkeiten zwischen einzelnen Tasks zu definieren, kann eine sinnvolle Reihenfolge von Tasks sichergestellt werden
- Durch die Definition von `targets` können gezielt einzelne Tasks ausgeführt werden

Im Bedarfsfall kann man eigene Tasks definieren: es existieren jedoch zahlreiche Tasks, die verwendet werden können:

http://ant.apache.org/manual/index.html

Der Verwendung eines solchen Tools in einem Projekt bringt viele Vorteilte und ist an sich ein Muss! Zu Ant gibt es zahlreiche Tutorials im Internet und auch Bücher: eine Einarbeitung braucht etwas Zeit, zahlt sich aber danach entsprechend aus.