**Grundlagen**

* Definition Verteiltes System:
  + mehrere Einzelkomponenten auf unterschiedlichen Rechnern
  + Nachrichtenaustausch, gemeinsame Zielsetzung
  + unabhängige Komponenten, die für den Benutzer wie ein einzelnes System erscheinen
  + Merkmale:
    - Zugreifbarkeit, gemeinsame Ressourcennutzung
      * Ressourcen (z.B. Datenbanken, Dienste etc.) für alle Benutzer gleichzeitig zugreifbar
    - Nebenläufigkeit
      * parallele Abarbeitung von Teilaufgaben
    - Transparenz
      * Verbergung von verschiedenen Abläufen und Ereignissen
        + Zugriffstransparenz Art und Weiß wie auf eine Ressource zugegriffen wird
        + Ortstransparenz Ablageort der Ressource
        + Migrationstransparenz Ressource kann an anderen Ort verschoben werden
        + Relokationstransparenz Ressource kann verschoben werden, während sie benutzt wird
        + Replikationstransparenz Ressource wird repliziert
        + Nebenläufigkeitstransparenz Ressource wird von mehreren Benutzern gleichzeitig benutzt
        + Fehlertransparenz Ausfall und Wiederherstellung der Ressource
      * keine 100% Transparenz: nicht sinvoll & möglich (z.B. Anzeigen von Latenz)
    - Offenheit
      * Standardisierte Schnittstellen
      * Interoperabilität: Dienst kann von mehreren Komponenten / Herstellern erbracht werden
      * Portabilität, Konfigurierbarkeit, Erweiterbarkeit
    - Skalierbarkeit
      * Größenmäßig, Geografisch, Administrativ
        + vertikale Skalierung: scale up (HW-Ressourcen erweitern)
        + horizontale Skalierung: scale in/out (heterogene Rechner / Server, z.B. für Event anmieten)
      * Zentralisierung schränkt Skalierbarkeit ein
    - Fehlertoleranz
      * lokale Fehler sollen keine Auswirkungen auf Gesamtsystem haben
* Klassen:
  + Verteilte Computersysteme (Cluster / homogene & Grid / heterogene Rechenknoten)
  + Verteilte Informationssysteme (transaktionale Unternehmensanwendungen)
  + Verteilte Pervasive Systeme (Sensoren)
* Ein Bild, das Text, Schild, Himmel, draußen enthält.

  Automatisch generierte BeschreibungReaktives Manifest:
  + Responsive zeitgerechte Antwortbereitschaft, konsistente Antwortzeiten
  + Resilient bleibt bei Ausfällen von HW und SW antwortbereit
  + Elastic bleibt unter sich ändernden Lastbedingungen antwortbereit
  + Message Driven asynchrone Nachrichtenübermittlung zur Sicherstellung von Isolation

**Architektur**

* Arten:
  + **Zentralisierte Architektur:**
    - Client-Server-Modell (2-Schicht-Architektur)
    - n-Schicht-Architektur möglich
  + Dezentrale Architektur:
    - Peer-to-Peer
  + Hybride Architektur:
    - Edge-Server-Systeme (z.B. ISP-Dienste)
* Client-Server-Modell:
  + ein Server bedient gleichzeitig 1-n Clients, die die Dienste und Daten nutzen
  + Request/Reply Kommunikationsmuster
    - Client ruft einen Dienst mit Übermittlung der Paramter auf
    - Sever antwortet und übermittelt das Ergebnis
  + bei n-schichtigen Architekturen:
    - Server übernimmt sowohl Serverrolle (für Client) als auch Clientrolle (für nachfolgende Server)
    - Arten:
      * Kette (rekursiv) 🡪 Kommunikation nur mit einer Hierachieebene drüber / drunter
      * Kette (transitiv) 🡪 letzter Server schickt reply direkt an Client
      * Baum 🡪 mehrere Server / Clients greifen auf einen Server zu, Antworten müssen koordiniert werden
  + **3-Schichten-Architektur**
    - vertikale Verteilung
      * Präsentationsschicht: Benutzerschnittstelle, nur z.B. Eingabevalidierung
      * Anwendungsschicht: fachliche Logik (Berechnung mit Daten)
      * Persistenzschicht: Datenbank- und Persistenztechnologie
    - horizontale Verteilung
      * Client & Server werden unterteilt in logisch gleiche Teile
      * Innerhalb einer Schicht kommt es zu Parallelisierung
* Möglichkeiten der Umsetzung:
  + Betriebssystem:
    - Netzwerkbetriebssystem:
      * locker gekoppelt, heterogene Multi-Computer
      * lokale Dienste für entfernte Clients
      * z.B. Linus, MacOS, Windows ...
    - Verteiltes Betriebssystem:
      * gekoppeltes Betriebssystem für Multiprozessoren und homogene Multicomputer
      * für Nutzer erscheint es wie ein logisches Betriebssystem
  + Anwendungen:
    - Verteilte Anwendung:
      * verschieden Komponenten erfüllen eine gemeinsame Aufgabe
      * für Benutzer transparente Verteiltheit
    - Middleware:
      * Ausgliederung der Dienste zur Umsetzung einer verteilten Anwendung mit Standardschnittstellen
      * Verteiltheit für Entwickler transparent

**Prozesse und Threads**

* Definition:
  + Programm: Menge von Anweisungen einer höheren Programmiersprache
  + Prozesse: entstehen bei Ausführung eines Programms, können Kindprozesse starten
  + Threads: Prozesse, die innerhalb einges gemeinsamen Adressraums laufen
    - context switch:
      * „context“ ist Zustand vom Prozess
      * context switch ist Übergang von einem Thread zum nächsten
      * beim context switch wird Zustand gespeichert und der Zustand des neuen Threads geladen
  + Unterschied Thread / Prozess:
    - Thread wird Prozessen vorgezogen
    - Kontextwechsel von Threads einfacher, weil TCB (Thread Control Block) kleiner ist als PCB (Process ...)
* Threads in verteilten Systemen:
  + Client:
    - Benutzereingaben sind parallel zur Verarbeitung im Hintergrund möglich
    - verbergen langer Latenzzeiten, parallele Kommunikaten über mehrere Kanäle
  + Server:
    - blockierende Aufrufe von Methoden ohne Prozess zu blockieren
    - z.B. parallele Abarbeitung von Requests (gleichzeitige Aufnahme und Abarbeitung von Anfragen)
* Erzeugung und Aufruf von Threads in Java:

public class KLASSE **implements Runnable** {

@Override

**public void run()** { *// dieser Code wird bei Aufruf ausgeführt*

*//CODE*

}

}

public class STARTER {

Thread myThread = **new Thread(new KLASSE())**;

*// Runnable Klasse muss zugewiesen werden*

**myThread.start()**; *// zum Starten vom Thread, führt run() aus*

}

* + alternativ:

public class KLASSE **extends Thread** {

@Override

*// Klasse Thread implementiert Runnable-Interface*

*// run()-Methode als Implementierung*

**public void run()** {

*//CODE*

}

}

public class STARTER {

KLASSE myThread = **new KLASSE()**;

*// direkter Aufruf der Klasse*

**myThread.start()**;

}

* + Daemon:
    - Daemon-Threads werden mit Programmende hart beendet, andere Threads laufen bis zum Ende des letzten User-Threads
    - Aufruf durch:

myThread.**setDaemon(true)**

* geteilte Ressourcen:
  + Race Condition
    - zwei oder mehr Treads teilen sich eine Ressource:
    - Starvation, Lost Update, etc. wenn mehrere Prozesse schreiben wollen
  + Critical Section:
    - Der Aufruf, der eine „race condition“ verursacht muss synchronisiert werden, der simultane Zugriff muss unterbunden werden
  + Thread-safety
    - ein Prozess, der frei von race conditions ist
  + Monitor zur Synchronisation
    - regelt den Zugriff auf einen Codeblock, gleichzeitiger Zugriff nur von einem Thread möglich

**Object monitor = new Object();** *//kann für mehrere synchronized benutzt werden*

public void add (long x) {

**synchronized(monitor)**{ *// Zugriff gleichzeitig nur durch einen Thread*

**while**(count < 5){ *// threads warten, wenn count < 5 ist*

**try {**

**monitor.wait();**

**} catch (InterruptedException ex) {}**

}

this.count = this.count + x; *// Änderung erst nach while-Schleife*

**monitor.notifyAll();** *// Alle wartenden Threads werden benachrichtigt*

}

}

* + - notwendig, wenn es mehrere schreibende Zugriffe gibt
    - für mehrere Methoden, die auf eine Variable zugreifen, benutzt man **denselben** **monitor**
    - mit notifyAll() werden alle Threads aufgeweckt 🡪 keine Warteschlange
      * notifyAll()muss benutzt werden, wenn:
        + Wartebedingungen der Threads unterschiedlich sind, aber die Ressource gleich
        + Potenziell mehrere Threads weiterlaufen können
      * sonst können Deadlocks auftreten:
        + Threads warten gegenseitig auf Zuteilung von Ressourcen
        + vgl. Parkhaus:

Auto wartet darauf, reinfahren zu können, aber keins fährt raus, weil es auf monitor wartet

* + - * Deadlock-Vermeidung:
        + komplette Anforderung: entweder alle Ressourcen oder keine
        + geordnete Anforderung: bestimmte Reihenfolge der angeforderten Betriebsmittel
        + analysierte Anforderung: vorherige Analyse, welche Belegung ohne Verklemmung möglich ist
* Java-Concurrency-Framework:
  + API-Klassen zu Standardproblemen der Thread-Synchronisierung
  + Locks und Conditions
    - implementiert die synchronized, wait und notify Funktionen

**Lock monitor = new ReentrantLock();**

**Condition leser = monitor.newCondition(), schreiber = monitor.newCondition();**

public Object get(){

**monitor.lock();** *// am Anfang locken des monitors*

try {

while(...) { try {

**leser.await();** } catch ...

} *// vgl. monitor.wait()*

*// Ausführung*

**schreiber.signal();** *// vgl. monitor.notifyAll()*

} finally {

**monitor.unlock();** *// unlocken des monitors bis zum nächsten Aufruf*

}}

* + zeitliche Synchronisation
    - CountDownLatch
      * Synchronisiert gemeinsamen Beginn oder Ende von Threads
      * Threads warten mit await(), zählen mit countDown() herunter und starten dann
    - CyclicBarrier
      * synchronisiert auch gemeinsamen Beginn oder Ende
      * await() blockiert Aufruf, bis genügen Threads await() aufgerufen haben, dann werden alle gestartet
      * letzter Thread gibt Blockierung der anderen frei
      * kann durch reset() wiederverwendet werden
  + atomic-Klassen
    - thread-safe Implementierung für Standardfälle
  + Threadpool: keine fachliche Sperrbedingung, sammelt Threads, führt Thread aus wenn Kapazitäten frei sind

**Kommunikation – Sockets**

* TCP: verbindungsorientiert (Request/Reply), UDP: verbindungslos (Send/Receive)
* Socket bezeichnet Programmierschnittstelle (API) des OS zur Internetprozesskommunikation über TCP / UDP
* Verbindungsaufbau:
  + Server lauscht über Server-Socket
  + sobald Client Verbindung aufbaut, wird neuer Socket erstellt mit TCP / UDP, alter Socket lauscht weiter

**Implementierung: Server**

**try(ServerSocket serverS = new ServerSocket(<PORT>))** {

**while(true)** { *// lauscht dauerhaft*

**try** {

**Socket** **s = serverS.accept();**

*// Input & Output-Stream definieren*

**InputStream in = s.getInputStream();**

**BufferedReader reader = new** **BufferedReader(new InputStreamReader(in));**

**OutputStream out = s.getOutputStream();**

**PrintWriter writer = new Printwriter(out);**

writer(println(„Hallo“);

writer.flush(); *// senden des Pakets*

String antwort = reader.readLine();

System.out.println(„Antwort der Gegenseite: “ + antwort);

**s.close();**

} **catch (IOException e) { e.printStackTrace(); }**

**}**

**} catch (IOException e) { e.printStackTrace(); }**

**Implementierung: Client**

**try {**

**Socket s = new Socket(„<DOMAIN>“, <PORT>);**

**InputStream in = s.getInputStream();**

**BufferedReader reader = new** **BufferedReader(new InputStreamReader(in));**

**OutputStream out = s.getOutputStream();**

**PrintWriter writer = new Printwriter(out);**

String eingang = read.readLine();

writer.println(„Ich habe empfangen: “ + eingang);

writer.flush();

**} catch (IOException e) { e.printStackTrace(); }**

* Framing 🡪 zur Entscheidung, wann eine Nachricht vollständig eingegangen ist
  + Delimiter-based: Nachricht wird durch Begrenzer abgeschlossen, z.B. „\0“ in C-Strings
  + Explicit length: Nachricht beginnt mit Feld fixer Länge, in der Länge der Nachricht steht
* Möglichkeiten der Server-Implementierungen:
  + sequenzieller Server
    - lässt nur einen Client nach dem anderen zugreifen, Anfragen werden nacheinander abgearbeitet
  + Paralleler Server
    - durch Aufbau neuer Sockets können Clients parallel zugreifen
    - zwei Aufgaben:
      * Annahme und Verteilung von Anfragen
      * Abfragen von Anfragen

**Kommunikation – RPC**

* Eigenschaften:
  + Alle Funktionsaufrufe sollen wie lokale Funktionsaufrufe funktionieren
  + Funktion wird von Client aufgerufen, auf Server abgearbeitet und zu Client zurückgesendet
* Architektur:
  + Client-Stub:
    - Funktionssignatur wie eigentliche Funktion
    - ist für die Datentransformation (Marshalling, Unmarshalling) und die Kommunikation mit dem Server zuständig
  + Server-Stub / Skeleton:
    - nimmt Anfragen des Betriesbssystems entgegen
    - übernimmt die Kommunikations- und Datentransferaufgaben
    - Ruft die lokale Funktionalität zur Ausführung der Anfrage auf
  + Datentransformation:
    - Marshalling / Encode:
      * Umwandlung von lokaler Datencodierung in Zielcodierung
    - Unmarshalling / Decode:
      * Rückumwandlung in die lokale Datencodierung
* Schnittstellenbeschreibung:
  + so neutral wie möglich
  + mindestens Beschreibung der Daten- und Funktionsstrukturen
  + z.B. Interface Definition Language IDL
* Bindung mit Verzeichnisdienst:
  + Orts- und Migrationstransparenz soll gewahrt werden, deshalb keine feste Codierung der Server-Adresse im Client
  + Implementierung der Schnittstelle über Verzeichnisdienst anfragen
* Interaktionssemantik:
  + Request/Reply
    - Client sendet Request und wartet, bis Reply vom Server zurückkommt
    - synchrone Kommunikation
  + Request/Future
    - Client sendet Request, Server bestätigt Empfang
    - Client arbeitet weiter und prüft periodisch, ob Ergebnis vorliegt
    - zurückgestellte synchrone Kommunikation
  + Request/Callback
    - Client sendet Request & Adresse für Callback, Server sendet Ergebnis an Callback-Adresse
    - asynchrone Kommunikation
  + Oneway
    - Client sendet Request, Rückgabe erfolgt nicht
    - Fire and forget, Ein-Weg-Kommunikation

**Kommunikation – Java RMI (Remote Method Invocation)**

* RPC-artige Implementierung in Java
* Schritte zur Erstellung:
  + Interface definieren

**public interface *INTERFACE* extends Remote {**

**public String method(String s) throws RemoteException;**

**}**

* + Server-Klasse implementieren

**public class CLASS implements *INTERFACE* {**

**@Override**

**public String method(String s) throws RemoteException { ... }**

**}**

* + Stub erzeugen & registrieren

***INTERFACE* *object* = new CLASS();** *// Objekt wird erzeugt*

***INTERFACE* stub = (RemoteInterface)UnicastRemoteObject.exportObject(*object*,0);**

**Registry r = LocateRegistry.createRegistry(<PORT>);** *// 1x create, sonst get*

**r.bind(„Codename“, stub);** *// für Verbindung*

* + Client implementieren

**Registry r = LocateRegistry.getRegistry(„localhost“, <PORT>);**

***INTERFACE* skeleton = (*INTERFACE*) r.lookup(„Codename“);** *// Verbindung*

**String antwort = skeleton.method(„Max Muster“);** *//Methodenaufruf*

* Parameteraustausch:
  + standardmäßig werden Parameter „by value“ übergeben
    - Objekte müssen serialisierbar sein:
      * Klasse wird durch implements serializable (Marker Interface) als serialisierbar markiert
      * Werte des Objekts werden in Stream geschrieben, bei De-Serialisierung aus Stream in leeres Objekt
      * RMI nutzt Serialisierung standardmäßig, also müssen alle Parameter serialisierbar sein
      * Serializable-Interface:
        + Container speicher Attributwerte in Outputstream
        + Standard-Serialisierung von JVM: passiert automatisch, aber langsam & speicherintensiv
        + Daten, die nicht übergeben werden sollen, können durch transient ausgeschlossen werden

z.B. private **transient** Decimal gehalt

* + - Ausführung ist lokal auf Werten, die übertragen wurden
    - Klasse by value:

public class Kunde **implements serializable** { ... }

* + Übergabe „by reference“
    - Stub des Clients wird beim Aufruf übergeben und durch Funktion des Servers angesprochen
    - Ausführung von Client zu Server remote
    - Interface auf Server:

public interface ServerIF extends Remote {

public void anlegen(**KundeIF** k) throws... *//Stub wird mit übergeben*

}

* + - Klasse by reference:

public interface **KundeIF** **extends Remote** { String getName() throws... }

public class Kunde **implements KundeIF** { ... }

**Beispiel Client-Implementierung (call by reference – Übergabeparameter):**

Server-Seite:

public interface KundeIF extends Remote { String getName() throws... }

public interface ServerIF extends Remote {

public void anlegen(**KundeIF** k) throws RemoteException; }

public class Kunde implements KundeIF {

private String name;

*@Override*

public String getName() throws RemoteException { return this.name; }

}

Client-Seite:

public class Client {

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

*// Skeleton anbinden*

Registry registry = LocateRegistry.getRegistry(„hostdomain.de“, 1099);

ServerIF serverStub = (ServerIF) registry.lookup(„service-name“);

*// kundenStub erstellen & Funktion aufrufen*

Kunde k = new Kunde(„Max Muster“);

KundeIF **kundenStub** = (KundeIF) UnicastRemoteObject.exportObject(**k**, 0);

*// exportObject bekommt Kundenobjekt zugewiesen*

serverStub.anlegen(**kundenStub**); *// Stub wird statt Objekt übergeben*

}}

**Beispiel Client-Implementierung (call by reference – Rückgabeparameter):**

Server-Seite:

public interface KundeIF extends Remote { String getName() throws... }

public interface ServerIF extends Remote {

public **KundeIF** kundeSuchen(int kundenNr) throws RemoteException; }

public class Kunde implements KundeIF {

private String name;

*@Override*

public String getName() throws RemoteException { return this.name; }

public void setName(String name) { this.name = name; }

}

public class Server implements ServerIF {

public **KundeIF** kundeSuchen(int kundenNr) throws RemoteException {

Kunde k = new Kunde();

String name = *// aus Datenbank laden o. Ä.*

k.setName(name);

*// Stub für Rückgabe an Client, kann dann genutzt werden*

**KundeIF kundenStub = (KundeIF)** UnicastRemoteObject.exportObject(k, 0);

**return kundenStub;**

}}

* + Entscheidungsfaktoren:
    - wie oft werden Details benötigt?
    - wie groß ist das Objekt?
    - wie häufig ändert sich etwas?
    - sollen Änderungen direkt am Origianlobjekt durchgeführt werden?
* Callback:
  + Client übergibt beim Funktionsaufruf einen Stub von einem neuen Client
  + Server sendet Besätigungsobjekt zu späterem Zeitpunkt an Client-Stub zurück
  + Aufruf erst nach Bestätigung zu Ende, aber zwischendurch keine blockierende Wirkung

**Beispiel Callback-Implementierung:**

Server-Seite:

public interface ServerIF extends Remote {

public void erteileAuftrag(Auftrag a, **CallbackIF referenz**) throws...;

*// referenz ist neuer client, der die bestaetigung erhält*

}

public interface CallbackIF extends Remote {

public void setBestaetigung(Bestaetigung b) throws RemoteException;

}

Client-Seite:

public class Client implements **CallbackIF** {

public static void main(String[] args) {

*// Skeleton anbinden*

Registry registry = Locate.getRegistry(„reg.othr.de“, 1099);

ServerIF serverStub = (ServerIF) registry.lookup(„service-name“);

*// Callbackstub erstellen mit neuem Client*

Auftrag **auftrag** = new Auftrag(„10 Leberkäs-Semmeln bitte“);

**CallbackIF c =** **new Client();**

CallbackIF **callbackStub** = (CallbackIF)UnicastRemoteObject.exportObject(**c**, 0);

serverStub.erteileAuftrag(**auftrag**, **callbackStub**);

}

*@Override*

public void setBestaetigung(Bestaetigung b) throws RemoteException {

*// z.B. in Datenbank Bestaetigung sichern*

}

}

* + Callback:
    - hohe Praxisrelevanz, da keine blockierenden Aufrufe
    - zusätzliche, server-artige Implementierung nötig
    - Zuordnung von Aufruf zu Bestätigung bzw. Callback wichtig
* RMI-Codebase:
  + alle Komponenten müssen für System bekannt sein, dafür werden jar-Dateien erstellt

**Kommunikation – SOA (serviceorientierte Architektur)**

* Definition
  + Services sind klar gegeinander abgegrenzte und aus betriebswirtschaftlicher Sicht sinnvolle Funktionen
  + Bestandteile:
    - Service: Abbildung konkreter Fachlichkeit in Geschäftsprozessen implementiert
    - Application Frontend: eigene GUI innerhalb einer SOA, häufig Webanwendung
    - Service-Bus: zentrale Kommunikationskomponente einer SOA
    - Service-Repository: Verzeichnisdienst zum Registrieren der angebotenen Services
    - Service-Inventory: Verwaltung und Planungssoftware für eine SOA
  + Find-Bind-Execute:
    - Anbieter veröffentlicht Services an Vermittler
    - Nutzer sucht über Vermittler Services
    - Nutzer frägt Beschreibung von Anbieter ab und nutzt Service
  + Service-Orchestrierung:
    - Service ist idealerweise von der Ablauflogik getrennt
    - Steuerungsschicht stellt notwendige Daten für Services zur Verfügung
* Webservices:
  + **SOAP**- / WSDL-Webservices
    - WSDL: (Web Services Definition Language)
      * stellt Model und XML-Format zur Beschreibung von Webservices zur Verfügung
    - SOAP: (stateless, oneway message exchange paradigm)
      * Envelope SOAP-Version und XML-spezifische Parameter
      * Header meist Steuerungsinformationen für den Nachrichtenaustausch
      * Body Nutzdaten in XML codiert für den Serviceaufruf
  + **REST**ful-Webservices
    - Representational State Transfer, Architekturstil, kann auf Webservices übertragen werden
    - Ziele: Skalierbarkeit, Generalisierung, unabhängiges Deployment von Komponenten, geringe Latency, erhöhte Sicherheit, Kapselung von Legacy-Anwendungen
    - Konvention bei REST:
      * Transport: HTTP
      * Methoden: CRUD
      * Gruppierung von Funktionalität nach Geschäftsobjekten
    - Konzepte:
      * Resource: Geschäftsobjekte, jede Information die benannt werden kann
      * Resource Identifier: eindeutige ID
      * Representation: Darstellung einer Resource in Datenformat, z.B. XML oder JSON
      * State of a Represenation: derzeitiger oder gewollter Status
      * Components: Service-Provider und -Consumer
    - CRUD zu HTTP-Methoden
      * HTTP: POST GET PUT DELETE
      * CRUD: CREATE READ UPDATE DELETE
    - JAX-RS:
      * JavaAPI for Restful Webservices, die die Funktionalität liefert, auf HTTP-Reqeusts zu reagieren und passende HTTP-Responses zu schicken
        + @Path Pfad
        + @GET, @POST, @PUT, @DELETE CRUD
        + @Produces und @Consumes Medientyp muss angehängt werden
        + @PathParam / @QueryParam Binding von Parametern
      * muss implementiert werden, z.B. von Jersey
      * Code:
        + z.B. Aufruf für Aufforderung:

GET /students/123 HTTP/1.1 Accept: application/xml

**@GET** *// GET-Aufruf wird vorangestellt*

**@Path(„students/{id}“)** *// Wenn Pfad übereinstimmt: Aufruf Funktion*

**@Produces(MediaType.APPLICATION\_XML)** *// Rückgabetyp wird festgelegt*

public Student read(**@PathParam(„id“)** int mnr){

*// Parameter id wird in mnr variable geschrieben*

Student s;

*// s aus Datenbank lesen*

return s; *// gibt Studen Objekt zurück, wird zu XML-Objekt umgewandelt*

}

* + - Ausgabe beinhaltet Header & XML- / Json-Objekt:

HTTP/1.1 200 OK

Content-type: **application/xml**

Content-length+: <ZEICHENANZAHL>

\r\n

...

* Notationen:

JSON:

{

„objektname“ : {

„mainID“: 12345,

„integer“: 123,

„string“: „STRING“,

„array“ : [123, 456, 789],

„objekt“ : {

„string2“: „STRING“,

„string3“: „STRING“

}

}

}

XML:

<?xml version=“1.0“ ... >

<objektname mainID=“12345“>

<integer>123</integer>

<string>STRING</string>

<array>

123, 456, 789

</array>

<objekt>

<string2>STRING</string2>

<string3>STRING</string3>

</objekt>

</objektname>

**Transaktionen**

* Ziel: persistente, konsistente, zuverlässige Datenhaltung
* ACID:
  + Atomicity Transaktion ist unteilbar, entweder vollständig oder gar nicht ausgeführt
  + Consistency: Transaktion hinterlässt konsistenten Datenbestand
  + Isolation: bei paralleler Ausführung muss Ergebnis genauso sein wie bei sequenzieller
  + Durability: Änderungen am Datensatz durch eine Transaktion sind dauerhaft
* Strategien:
  + privater Arbeitsbereich Änderungen an Kopie, nach Abschluss festschreiben
  + Write-Ahead-Log Operationen auf Original-Dateien, Protokollierung im Log
* Arten von Transaktionen:
  + Flache Transaktion ACID-Eigenschaft, Normalfall sind SQL-Befehle
  + Verschatelte Transatkion besteht aus unabhängigen Teiltransaktionen mit ACID-Eigenschaften
  + Verteilte Transaktion flache Transaktion in verteiltem System
* Beispiele:
  + Datenbankmanagementsysteme (DBMS)
  + Workflowsysteme
  + Unternehmensanwendungen
  + B2B-Anwendungen
* Technologien
  + JDBC
    - Verbindungsaufbau und Zugriff auf Tabellen, SQL-Statements an Datenbank senden
  + Transaction-Processing-Monitor (TP-Monitor)
    - für verteilte Transaktionen: Koordination
    - Phasen:
      * 1. Abstimmungsphase
        + Teilnehmer führen Operation vorläufig durch, antworten mit READY oder ABORT
      * 2. Entscheidungsphase
        + wenn alle READY gesendet haben, wird COMMIT durchgeführt
        + wenn nicht alle READY sind, ABORT oder ROLLBACK
      * 2 Phasen Protokoll
    - Consistency und Isolation durch Kontrollverfahren sichergestellt
  + Nebenläufigkeitssteuerung
    - Operationen, die Konflikt erzeugen können, müssen korrekt geplant werden
    - Lösungsmöglichkeiten:
      * pessimistischer Ansatz Operationen synchronisieren, bevor sie begonnen werden
      * optimistischer Ansatz Operationen ausführen, am Ende synchronisieren und prüfen

**Replikation und Caching**

* Replikation:
  + Mehrfachspeicherung von Daten
  + Ziele: Performanz, Verfügbarkeit, Verlässlichkeit
  + Replikation vs. Konsistenz:
    - man verzichtet auf strikte Konsistenz, um Ziele zu erreichen
    - wie weit man von Konsistenz abweicht, muss individuell entschieden werden
* Caching:
  + Pufferspeichern mit geringer Zugriffszeit
  + Sonderform von Replikation: nicht Besitzer, sondern Benutzer entscheidet sich für Replikation
  + Ausprägungen:
    - Orte: Datenbank, Heap, Browser, Festplatte, CPU
    - Arten: Lokaler Cache, Data Grid, Datenbank-Persistenz-Schicht
    - Verteilungsmethoden:
      * Replication: Jeder bekommt Kopie von allem
      * Invalidation: Bei Änderungen müssen andere Knoten löschen
      * Distribution: Jeder nur einen Anteil, gleichmäßige Verteilung
* Konsitenz:
  + Clientzentrierte Konsistenzmodelle:
    - Monotones Lesen: Jede Operation liest den gleichen Wert von Element x
    - Monotones Schreiben: Schreiboperation auf x wird erst abgeschlossen, bevor nächste beginnt
    - Read your own writes: Prozess kann beim Lesen die eigenen Schreiboperationen sehen
    - Write follows read: Schreiboperation garantiert auf kürzlich gelesenem Wert
* BASE-Prinzip:
  + BA basically available System verfügbar, schnelle Antwortzeiten
  + S soft state Aktualität eines Objekts nimmt kontinuierlich ab, Bewertung durch Attribut
  + E eventually consistent keine strikte Konsistenz, Konsistenz wird erst nach Schreibprozess hergestellt
  + Gegenstück zu ACID
* CAP-Theorem
  + Consistency, Availability, Partition tolerance
  + Unmöglich, in einem verteilten System alle drei Eigenschaften zu garantieren (maximal 2 gleichzeitig)

**MapReduce**

* Verarbeitung großer Datenmengen in verteilten Umgebungen
* Phasen:
  + Map: analysiert key-value Paare, weist jedem Key alle zugehörigen Values zu
  + Sort & Shuffle: sortiert Paare nach Keys, gruppiert Paare für jeden Key und fasst Values zusammen
  + Reduce: empfängt neue k-v-Paare, wird für jeden unique key ausgeführt, Ergebnis neue k-v-Paare
  + (Collator: fässt alle Ergebnisse zusammen, MapReduce-Zyklus wird nochmal durchlaufen)
* Code:

**map**(String key, String value):

*// key: document name*

*// value: content of document*

for each word **w** in value:

**emit**(**w**, „1“); *// gibt Wort & „1“ als Paar raus*

*// dazwischen Sortierung der Paare*

**reduce**(String key, List<String> value):

*// key: unique word*

*// value: list of counts*

int result = 0;

for each v in value:

restult += Integer.valueOf(v); *// zählt Anzahl Vorkomnisse des Worts*

**Microservices**

* Strukturierung der Anwendung so wie die Organisation strukturiert ist und kommuniziert (Conway’s Law)
* Architekturstil Microservices:
  + Service läuft unabhängig von anderen Services als eigener Prozess
  + Jeder Service hat eigene Laufzeitumgebung (Web-, Anwendungsserver, Datenbank)
  + Kommunikation zwischen Services über leichtgewichtige Mechanismen, z.B. REST-Schnittstellen
  + Services werden unabhängig zur Ausführung gebracht
* Vergleich Monolith vs. Microservices:
  + Monolith:
    - n Services, 1 Prozess
    - Skalierung durch Replikation der ges. Anwendung
    - zentrierte, relationale Datenhaltung
  + Microservice:
    - n Services, n Prozesse
    - Skalierung durch Replikation nach Bedarf
    - Jeder Microservice kann eigene Persistenzeinheit besitzen
      * Polyglotte Persistenz (Datenhaltung bei Microservices)
        + Microservice hat eigene Datenhaltung, richtet sich nach Zweck des Microservices
        + kann z.B. relational oder NoSQL sein
* DevOps
  + Individuelle Zusammenstellung der Services möglich, effizientere Arbeitsabläufe und Prozesse
  + kann auf die Bedürfnisse der IT-Abteilung angepasst werden
  + **DevOps** = **Dev**elopment + **Op**erations
* Umsetzung:
  + technische Möglichkeiten z.B. Java Enterprise Edition mit JAX-RS
  + Virtualisierung für Cloud-Ausrichtung 🡪 Docker
* **Docker:**
  + Virtualisiert eine Kontrollgruppe wie VirtualBox, aber ohne Betriebssystem
    - benutzt Linux-Kernel zur Virtualisierung der Prozesse
  + Image lässt sich in Docker-Umgebung laden, darin können Prozesse bzw. Microservices laufen

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

**Threads**

* Erzeugung und Aufruf von Threads in Java:

public class KLASSE **implements Runnable** {

@Override

**public void run()** { *// dieser Code wird bei Aufruf ausgeführt*

*//CODE*

}

}

public class STARTER {

Thread myThread = **new Thread(new KLASSE())**;

*// Runnable Klasse muss zugewiesen werden*

**myThread.start()**; *// zum Starten vom Thread, führt run() aus*

}

* + alternativ:

public class KLASSE **extends Thread** {

@Override

*// Klasse Thread implementiert Runnable-Interface*

*// run()-Methode als Implementierung*

**public void run()** {

*//CODE*

}

}

public class STARTER {

KLASSE myThread = **new KLASSE()**;

*// direkter Aufruf der Klasse*

**myThread.start()**;

}

* + Monitor zur Synchronisation

**Object monitor = new Object();** *//kann für mehrere synchronized benutzt werden*

public void add (long x) {

**synchronized(monitor)**{ *// Zugriff gleichzeitig nur durch einen Thread*

**while**(count < 5){ *// threads warten, wenn count < 5 ist*

**try {**

**monitor.wait();**

**} catch (InterruptedException ex) {}**

}

this.count = this.count + x; *// Änderung erst nach while-Schleife*

**monitor.notifyAll();** *// Alle wartenden Threads werden benachrichtigt*

}

}

* + Locks und Conditions

**Lock monitor = new ReentrantLock();**

**Condition leser = monitor.newCondition(), schreiber = monitor.newCondition();**

public Object get(){

**monitor.lock();** *// am Anfang locken des monitors*

try {

while(...) { try {

**leser.await();** } catch ...

} *// vgl. monitor.wait()*

*// Ausführung*

**schreiber.signal();** *// vgl. monitor.notifyAll()*

} finally {

**monitor.unlock();** *// unlocken des monitors bis zum nächsten Aufruf*}}

**Sockets**

**Implementierung: Server**

**try(ServerSocket serverS = new ServerSocket(<PORT>))** {

**while(true)** { *// lauscht dauerhaft*

**try** {

**Socket** **s = serverS.accept();**

*// Input & Output-Stream definieren*

**InputStream in = s.getInputStream();**

**BufferedReader reader = new** **BufferedReader(new InputStreamReader(in));**

**OutputStream out = s.getOutputStream();**

**PrintWriter writer = new Printwriter(out);**

writer(println(„Hallo“);

writer.flush(); *// senden des Pakets*

String antwort = reader.readLine();

System.out.println(„Antwort der Gegenseite: “ + antwort);

**s.close();**

} **catch (IOException e) { e.printStackTrace(); }**

**}**

**} catch (IOException e) { e.printStackTrace(); }**

**Implementierung: Client**

**try {**

**Socket s = new Socket(„<DOMAIN>“, <PORT>);**

**InputStream in = s.getInputStream();**

**BufferedReader reader = new** **BufferedReader(new InputStreamReader(in));**

**OutputStream out = s.getOutputStream();**

**PrintWriter writer = new Printwriter(out);**

String eingang = read.readLine();

writer.println(„Ich habe empfangen: “ + eingang);

writer.flush();

**} catch (IOException e) { e.printStackTrace(); }**

**RMI**

* + Interface definieren

**public interface *INTERFACE* extends Remote {**

**public String method(String s) throws RemoteException;**

**}**

* + Server-Klasse implementieren

**public class CLASS implements *INTERFACE* {**

**@Override**

**public String method(String s) throws RemoteException { ... }**

**}**

* + Stub erzeugen & registrieren

***INTERFACE* *object* = new CLASS();** *// Objekt wird erzeugt*

***INTERFACE* stub = (RemoteInterface)UnicastRemoteObject.exportObject(*object*,0);**

**Registry r = LocateRegistry.createRegistry(<PORT>);** *// 1x create, sonst get*

**r.bind(„Codename“, stub);** *// für Verbindung*

* + Client implementieren

**Registry r = LocateRegistry.getRegistry(„localhost“, <PORT>);**

***INTERFACE* skeleton = (*INTERFACE*) r.lookup(„Codename“);** *// Verbindung*

**String antwort = skeleton.method(„Max Muster“);** *//Methodenaufruf*

* + - Klasse by value:

public class Kunde **implements serializable** { ... }

* + - Klasse by reference:

public interface ServerIF extends Remote {

public void anlegen(**KundeIF** k) throws... *//Stub wird mit übergeben* }

public interface **KundeIF** **extends Remote** { String getName() throws... }

public class Kunde **implements KundeIF** { ... }

**Beispiel Client-Implementierung (call by reference – Übergabeparameter):**

Server-Seite:

public interface KundeIF extends Remote { String getName() throws... }

public interface ServerIF extends Remote {

public void anlegen(**KundeIF** k) throws RemoteException; }

public class Kunde implements KundeIF {

private String name;

*@Override*

public String getName() throws RemoteException { return this.name; }

}

Client-Seite:

public class Client {

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

*// Skeleton anbinden*

Registry registry = LocateRegistry.getRegistry(„hostdomain.de“, 1099);

ServerIF serverStub = (ServerIF) registry.lookup(„service-name“);

*// kundenStub erstellen & Funktion aufrufen*

Kunde k = new Kunde(„Max Muster“);

KundeIF **kundenStub** = (KundeIF) UnicastRemoteObject.exportObject(**k**, 0);

*// exportObject bekommt Kundenobjekt zugewiesen*

serverStub.anlegen(**kundenStub**); *// Stub wird statt Objekt übergeben*

}}

**Beispiel Client-Implementierung (call by reference – Rückgabeparameter):**

Server-Seite:

public interface KundeIF extends Remote { String getName() throws... }

public interface ServerIF extends Remote {

public **KundeIF** kundeSuchen(int kundenNr) throws RemoteException; }

public class Kunde implements KundeIF {

private String name;

*@Override*

public String getName() throws RemoteException { return this.name; }

public void setName(String name) { this.name = name; }

}

public class Server implements ServerIF {

public **KundeIF** kundeSuchen(int kundenNr) throws RemoteException {

Kunde k = new Kunde();

String name = *// aus Datenbank laden o. Ä.*

k.setName(name);

*// Stub für Rückgabe an Client, kann dann genutzt werden*

**KundeIF kundenStub = (KundeIF)** UnicastRemoteObject.exportObject(k, 0);

**return kundenStub;**

}}

**Beispiel Callback-Implementierung:**

Server-Seite:

public interface ServerIF extends Remote {

public void erteileAuftrag(Auftrag a, **CallbackIF referenz**) throws...;

*// referenz ist neuer client, der die bestaetigung erhält*

}

public interface CallbackIF extends Remote {

public void setBestaetigung(Bestaetigung b) throws RemoteException;

}

Client-Seite:

public class Client implements **CallbackIF** {

public static void main(String[] args) {

*// Skeleton anbinden*

Registry registry = Locate.getRegistry(„reg.othr.de“, 1099);

ServerIF serverStub = (ServerIF) registry.lookup(„service-name“);

*// Callbackstub erstellen mit neuem Client*

Auftrag **auftrag** = new Auftrag(„10 Leberkäs-Semmeln bitte“);

**CallbackIF c =** **new Client();**

CallbackIF **callbackStub** = (CallbackIF)UnicastRemoteObject.exportObject(**c**, 0);

serverStub.erteileAuftrag(**auftrag**, **callbackStub**);

}

*@Override*

public void setBestaetigung(Bestaetigung b) throws RemoteException {

*// z.B. in Datenbank Bestaetigung sichern*

}

}

**JAX-RS (für REST):**

* + - * + z.B. Aufruf für Aufforderung:

GET /students/123 HTTP/1.1 Accept: application/xml

**@GET** *// GET-Aufruf wird vorangestellt*

**@Path(„students/{id}“)** *// Wenn Pfad übereinstimmt: Aufruf Funktion*

**@Produces(MediaType.APPLICATION\_XML)** *// Rückgabetyp wird festgelegt*

public Student read(**@PathParam(„id“)** int mnr){

*// Parameter id wird in mnr variable geschrieben*

Student s;

*// s aus Datenbank lesen*

return s; *// gibt Studen Objekt zurück, wird zu XML-Objekt umgewandelt*

}

* + - Ausgabe beinhaltet Header & XML- / Json-Objekt:

HTTP/1.1 200 OK

Content-type: **application/xml**

Content-length+: <ZEICHENANZAHL>

\r\n

...

* Notationen:

JSON:

{

„objektname“ : {

„mainID“: 12345,

„integer“: 123,

„string“: „STRING“,

„array“ : [123, 456, 789],

„objekt“ : {

„string2“: „STRING“,

„string3“: „STRING“

}

}

}

XML:

<?xml version=“1.0“ ... >

<objektname mainID=“12345“>

<integer>123</integer>

<string>STRING</string>

<array>

123, 456, 789

</array>

<objekt>

<string2>STRING</string2>

<string3>STRING</string3>

</objekt>

</objektname>

**MapReduce**

**map**(String key, String value):

*// key: document name*

*// value: content of document*

for each word **w** in value:

**emit**(**w**, „1“); *// gibt Wort & „1“ als Paar raus*

*// dazwischen Sortierung der Paare*

**reduce**(String key, List<String> value):

*// key: unique word*

*// value: list of counts*

int result = 0;

for each v in value:

restult += Integer.valueOf(v); *// zählt Anzahl Vorkomnisse des Worts*