RPC-Fehlersemantik-Klassen

- Operationale Sichtweise:
  - Wie wird nach einem Timeout auf (vermeintlich?) nicht eintreffende Nachrichten, wiederholte Requests, gecrashte Prozesse reagiert?

---

1. Maybe-Semantik:
   - Keine Wiederholung von Requests
   - Einfach und effizient
   - Keinerlei Erfolgsgarantien → nur ausnahmsweise anwendbar
   - Mögliche Anwendungsklasse: z.B. Auskunftsdienste (Anwendung kann es evtl. später noch einmal probieren, wenn keine Antwort kommt)

RPC-Fehlersemantik-Klassen (2)

2. At-least-once-Semantik:
   - Automatische Wiederholung (evtl. mehrfach) von Requests
   - Keine Duplikatserkennung (zustandsloses Protokoll auf Serverseite)
   - Akzeptabel bei idempotenten Operationen (z.B. Lesen einer Datei)

1) und 2) werden etwas euphemistisch oft als „best effort" bezeichnet

---

3. At-most-once-Semantik:
   - Erkennen von Duplikaten (Sequenznummern, log-Datei etc.)
   - Keine wiederholte Ausführung der Prozedur, sondern evtl. erneutes Senden des (gemerkten) Reply
   - Geeignet auch für nicht-idempotente Operationen

RPC-Fehlersemantik-Klassen (3)

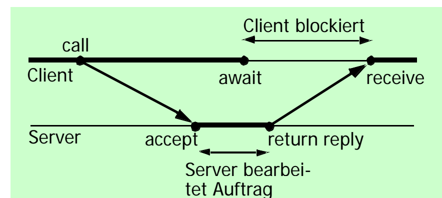
- Maybe → at-least-once → at-most-once → …
  ist zunehmend aufwändiger zu realisieren

Ist „exactly-once“ machbar?

- Man begnügt sich daher, falls es die Anwendung erlaubt, oft mit einer billigeren aber weniger perfekten Fehlersemantik
- Motto: so billig wie möglich, so „perfekt“ wie nötig

Asynchroner RPC

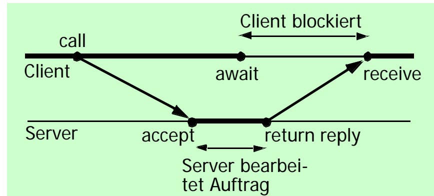
- Andere Bezeichnung: „Remote Service Invocation“
- Auftragsorientiert, d.h. also: Antwortverpflichtung



- Parallelverarbeitung von Client und Server möglich, solange Client noch nicht auf Resultat angewiesen

Asynchroner RPC: Zuordnung Auftrag/Ergebnisempfang

- Unterschiedliche Ausprägung auf Sprachebene möglich
  - „await" könnte z.B. einen bei „call" zurückgelieferten „handle" als Parameter erhalten, also z.B.: `Y = call X(…); … await(Y);`



  - evtl. könnte die Antwort auch asynchron in einem eigens dafür vorgesehenen Anweisungsblock empfangen werden (vgl. Interrupt- oder Exception-Routine)

Asynchroner RPC: Future-Variable

- Spracheinbettung evtl. auch durch „Future-Variablen"
- Future-Variable = „handle" auf das in anderem Thread parallel berechnetes Funktionsergebnis
- Programm blockiert nur dann, wenn der Wert der Variable bei ihrer Nutzung noch nicht feststeht
- Beispiel (Scala):

```
...
val x = future(callRPC())
... // continue local computation
println(x()) // await x iff necessary
```

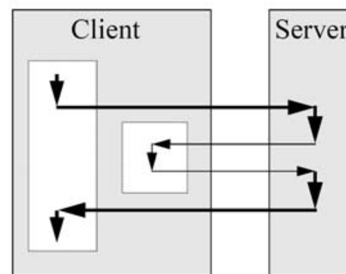
Einige Varianten / Ergänzungen zu RPCs in der Praxis

Wir besprechen nachfolgend 4 ergänzende Aspekte:

1. Rückrufe
2. Context-Handles
3. Broadcast bzw. Multicast
4. Sicherheit

Rückrufe („call back RPC“)

- Temporärer Rollentausch von Client und Server
  - um eventuell bei langen Aktionen Zwischenresultate zurückzumelden
  - um eventuell weitere Daten vom Client anzufordern
- Client muss Rückrufadresse beim call übergeben



Context-Handles

- Struktur mit Kontextinformation zur Verwaltung von Zustandsinformation über mehrere Aufrufe hinweg
  - vgl. „cookies“: kleine Textdatei, die ein Web-Server einem Browser (= Client) schickt und die im Browser gespeichert wird
- Context-Handles werden vom Server dynamisch erzeugt und an den Client (bei „reply“) zurückgegeben
- Client kann diese beim nächsten Aufruf (unverändert) wieder mitsenden
  - Server „erinnert“ sich so an den Kontext
- Vorteil: Server selbst arbeitet „zustandslos“

Weitere Varianten / Ergänzungen

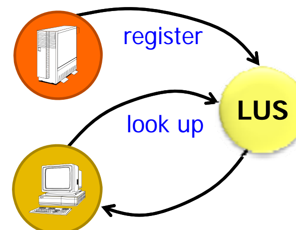
- Broadcast bzw. Multicast
  - Request wird „gleichzeitig“ an mehrere Server geschickt
  - RPC ist beendet mit der ersten empfangenen Antwort oder Client hat die Möglichkeit, nach einer Antwort auf eine weitere Antwort zu warten
- Oft wählbare Sicherheitsstufen, z.B.:
  - Authentifizierung bei Aufbau der Verbindung (“binding”)
  - Authentifizierung pro RPC-Aufruf oder pro Nachricht
  - Verschlüsselung der zugrundeliegenden Nachrichten
  - Schutz gegen Verfälschung (digitale Signatur, verschlüsselte Prüfsumme, MAC)

Beispiel „Secure RPC“
(Teil des „SUN RPC“)

- Client und Server vereinbaren (geheim) einen Session-Key K
  - wird zum Verschlüsseln der Nachrichten genutzt
- Jeder Request enthält einen mit K kodierten Zeitstempel
- Erster Request enthält zusätzlich verschlüsselt eine Zeitfenstergrösse W als zeitliches Toleranzintervall sowie (ebenfalls verschlüsselt) W-1
  - „zufälliges“ Generieren einer ersten Nachricht ist so nahezu unmöglich
  - replay (bei kleinem W) ist ebenfalls erfolglos
  - W ist verschlüsselt, auch um Angreifern keine Information über die Fenstergrösse zu geben
- Server akzeptiert einen Request nur, wenn:
  - Zeitstempel grösser als letzter Zeitstempel und
  - Zeitstempel innerhalb des Zeitfensters
- Zur Authentifizierung enthält die Antwort des Servers (verschlüsselt) den letzten erhaltenen Zeitstempel-1

Lookup-Service

- Problem: Wie finden sich Client und Server?
  - haben i.Allg. verschiedene Lebenszyklen → kein gemeinsames Übersetzen / statisches Binden (fehlende gemeinsame Umgebung)
  - → Lookup Service (LUS) oder „Registry“ fungiert als „Service-Broker“



- Server gibt seinen Service (d.h. RPC-Routine) dem LUS bekannt
  - register: RPC-Schnittstelle „exportieren“ (Name, Parameter, Typen,...)
  - evtl. auch wieder abmelden
- Client erfragt beim LUS die Adresse eines geeigneten Servers
  - Angabe des gewünschten Typs von Service beim look up oder discovery
  - „importieren“ der RPC-Schnittstelle

Lookup-Service (2)

- Vorteile („Mehrwert"): im Prinzip kann LUS
  - mehrere Server für den gleichen Service registrieren (→ Fehlertoleranz; Lastausgleich)
  - Autorisierung etc. überprüfen
  - durch Polling der Server die Verfügbarkeit eines Services testen
  - verschiedene Versionen eines Dienstes verwalten
- Probleme:
  - look up kostet Ausführungszeit (gegenüber statischem Binden)
  - zentraler LUS ist ein potentieller Engpass / „single point of failure" (evtl. LUS geeignet replizieren / verteilen)
  - wie lernen Client oder Server die Adresse des „richtigen" oder „zuständigen" LUS kennen?
    - z.B. feste („well-know") Adresse oder Broadcast in lokaler Umgebung

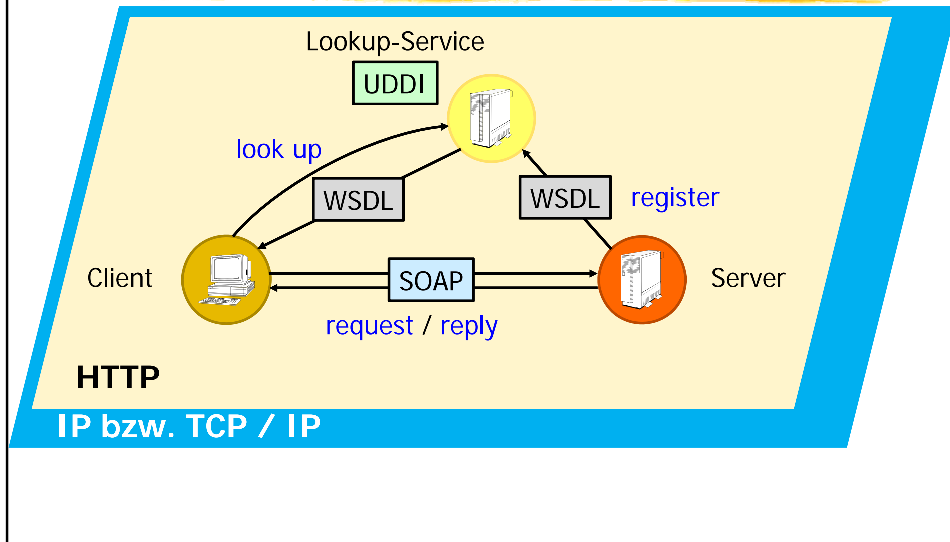
Web Services als Beispiel für das Client-Server-Modell

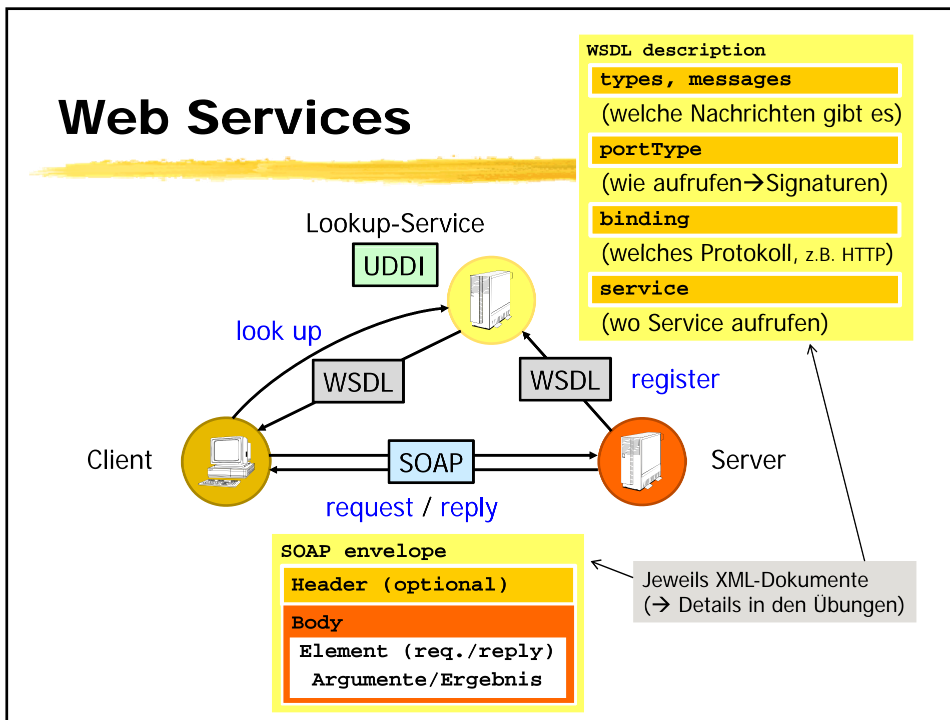
- Problem: Internet ist zu heterogen für eine einheitliche Programmiersprache oder RPC-Laufzeitumgebung
  - „Lösung": Web Services (Weiterentwicklung des XML-RPCs) als offener, plattform- bzw. sprachunabhängiger Standard, dessen Schnittstellen von diversen Plattformen implementiert werden können

---

- HTTP fungiert als Transportschicht
- SOAP als plattformunabhängige Protokollspezifikation (ursprünglich: „Simple Object Access Protocol")
- UDDI als Lookup-Service („Universal Description, Discovery and Integration")
- WSDL als standardisierte Service-Beschreibung („Web Services Description Language")

Alternativ zu HTTP: UDP oder SMTP, z.B. für Mitteilungen ohne Antwort oder wenn Resultatberechnung länger als typischer HTTP-Timeout dauert


Web Services
Lookup-Service
UDDI
look up
WSDL
WSDL
register
Client
SOAP
Server
request / reply
HTTP
IP bzw. TCP / IP


Web Services
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
Lookup-Service
UDDI
look up
WSDL
WSDL
register
Client
SOAP
Server
request / reply
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Jeweils XML-Dokumente
(→ Details in den Übungen)

Web Services

WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)

XML-Dokument, das (bzgl. UDDI bzw. für einen Client) den Service beschreibt

XML (Extensible Markup Language):

- Auszeichnungssprache zur Darstellung hierarchisch strukturierter Daten in Form von Textdaten (z.B. ASCII)
- Unterstrukturen mit Start- (<Tag-Name>) und End-Tag (</Tag-Name>) oder einem Empty-Element-Tag (< *Tag-Name* />)
- Attribute bei einem Tag (Attribut-Name= "Attribut-Wert") für Zusatzinformationen

```
<books>
  <book>
    <author>Karl May</author>
    <title>Winnetou</title>
    <ISBN>3-7802-0170-4</ISBN>
    <price format="EUR"/>
  </book>
  <pubinfo>
    <publisher>
      KM-Verlag
    </publisher>
    <town>Bamberg</town>
  </pubinfo>
</books>
```

WSDL: types

WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)

types

XML-Schema zur Typenbeschreibung

- Definition von Typen bzw. deren Namen als XML-„element"
- Meistens Verweis auf einen „complexType", der aus „elements" der Basistypen (int, float,...) besteht
- Schema definiert auch einen Namens-raum, der als URI angegeben wird
- Schema oft in externe .xsd-Datei ausgelagert

→ nächste 2 slides

WSDL-Namensräume

- ExampleWebServices.wsdl

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<definitions
  name="ExampleWebServices"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:wsam="http://www.w3.org/2007/05/addressing/metadata"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap">

  <types>
    <xsd:schema>
      <xsd:import
        namespace="http://example.org/VS/WebServices/"
        schemaLocation="ExampleSchema.xsd"/>
    </xsd:schema>
  </types>
...
```

Namensraum der beschriebenen Web Services

Weitere Namensräume, aus denen Tags benötigt werden

Importiert die Typendefinitionen; können auch eigenen Namensraum haben

nächste slide

WSDL-Namensräume

- ExampleSchema.xsd

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<xs:schema
  version="1.0"
  targetNamespace="http://example.org/VS/WebServices/"
  xmlns:tns="http://example.org/VS/WebServices/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

<!-- Elementdefinitionen -->

</xs:schema>
```

Hier gleicher Namensraum wie Services

Definiert das Präfix „tns:“, um den targetNamespace anzusprechen

nächste slide

Lässt „xs:“ auf den allgemeinen XML-Schema-Namensraum zeigen

WSDL-Beispiel: types

WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)

types

XML-Schema zur Typenbeschreibung

```
<xs:element name="myArgs" type="tns:myObject"/>

<xs:complexType name="myObject">
  <xs:sequence>
    <xs:element name="i" type="xs:int"/>
    <xs:element name="j" type="xs:float"/>
  </xs:sequence>
</xs:complexType>
```


WSDL: messages

WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)

messages

- Abstrakte Definition der Nachrichten, mit denen ein Dienst angesprochen wird oder antwortet (können unter „bindings“ für spezielle Protokolle konkretisiert werden)
- Daten können in mehrere Teile („part“) gruppiert werden, die jeweils einem «type element» entsprechen
- Je ein Eintrag pro Nachrichtendefinition

WSDL-Beispiel: messages

„myArgs“ wurde oben definiert

WSDL description
- types, messages
  (welche Nachrichten gibt es)
- portType
  (wie aufrufen→Signaturen)
- binding
  (welches Protokoll, z.B. HTTP)
- service
  (wo Service aufrufen)

messages

```
<message name="myRequest">
  <part name="parameters"
        element="tns:myArgs"/>
  <part name="optionalParameters"
        element="tns:myOpt"/>
</message>

<message name="myResponse">
  <part name="result" element="tns:myRet"/>
</message>
```

Die Gliederung in „parts“ ist optional

„myRequest“ hat also einen int und einen float als Parameter

WSDL: portType

WSDL description
- types, messages
  (welche Nachrichten gibt es)
- portType
  (wie aufrufen→Signaturen)
- binding
  (welches Protokoll, z.B. HTTP)
- service
  (wo Service aufrufen)

portType

- Definition der einzelnen Web Services
- Jeder Service entspricht einer „operation“
  - hat typischerweise eine „input“-Nachricht und eine „output“-Nachricht
  - zusätzlich können Fehlerbenachrichtigungen angegeben werden („fault“)
- Services können auch unidirektional sein, (z.B. nur „output“ für Benachrichtigungen)

WSDL-Beispiel: portType

WSDL description
- **types, messages** (welche Nachrichten gibt es)
- **portType** (wie aufrufen→Signaturen)
- **binding** (welches Protokoll, z.B. HTTP)
- **service** (wo Service aufrufen)

portType

```
<portType name="groupOfServices">
  <operation name="myService">
    <input message="tns:myRequest"/>
    <output message="tns:myResponse"/>
    <fault message="tns:someFault"/>
  </operation>
</portType>
```

Hier könnten weitere „operation“ stehen

„myService“ wird also mit einem int und einem float aufgerufen

WSDL: binding

WSDL description
- **types, messages** (welche Nachrichten gibt es)
- **portType** (wie aufrufen→Signaturen)
- **binding** (welches Protokoll, z.B. HTTP)
- **service** (wo Service aufrufen)

binding

- Bindet „portType“ an ein Protokoll (z.B. HTTP)
- Es kann mehrere „binding“-Einträge für verschiedene Protokolle geben
- Im Normalfall genügen die Informationen aus „message“ und „portType“ für die Abbildung der Nachrichten auf ein konkretes Format (d.h. „binding“ enthält kaum Information)

WSDL-Beispiel: binding

WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)

binding

```
<binding name="myBinding"
         type="tns:groupOfServices">
  <soap:binding
transport="http://schemas.xmlsoap.org/soap/http"
         style="document"/>
</binding>
```

Allgemeiner als „rpc“ (veralteter Web Service „style“)

URI definiert HTTP als Transportprotokoll (mit anderen URIs können beliebige Protokolle zwischen Client und Server vereinbart werden)

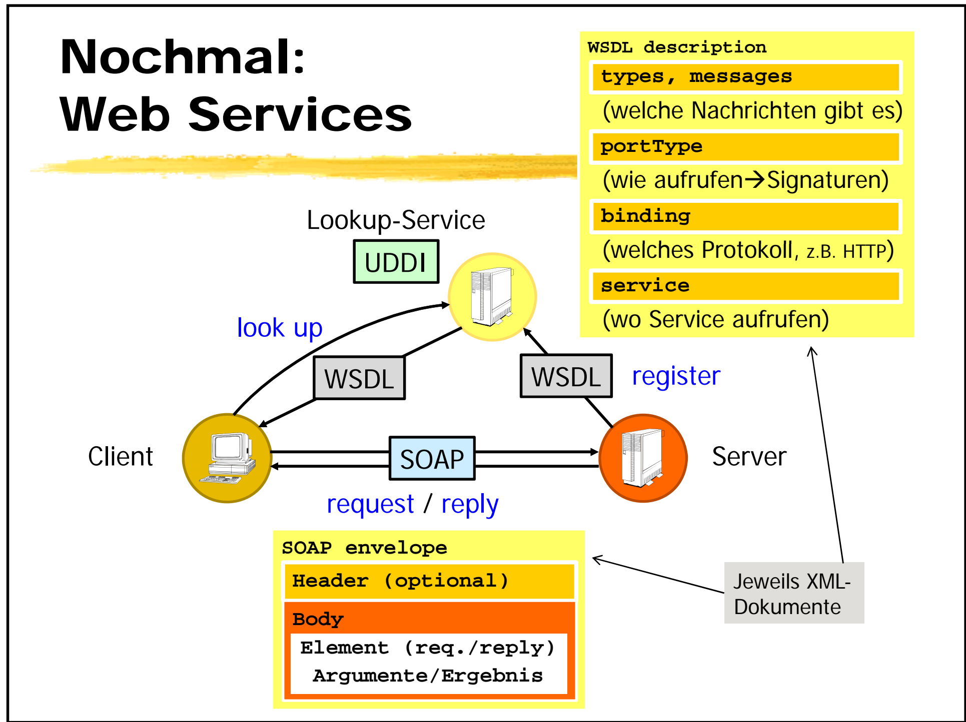
WSDL: service

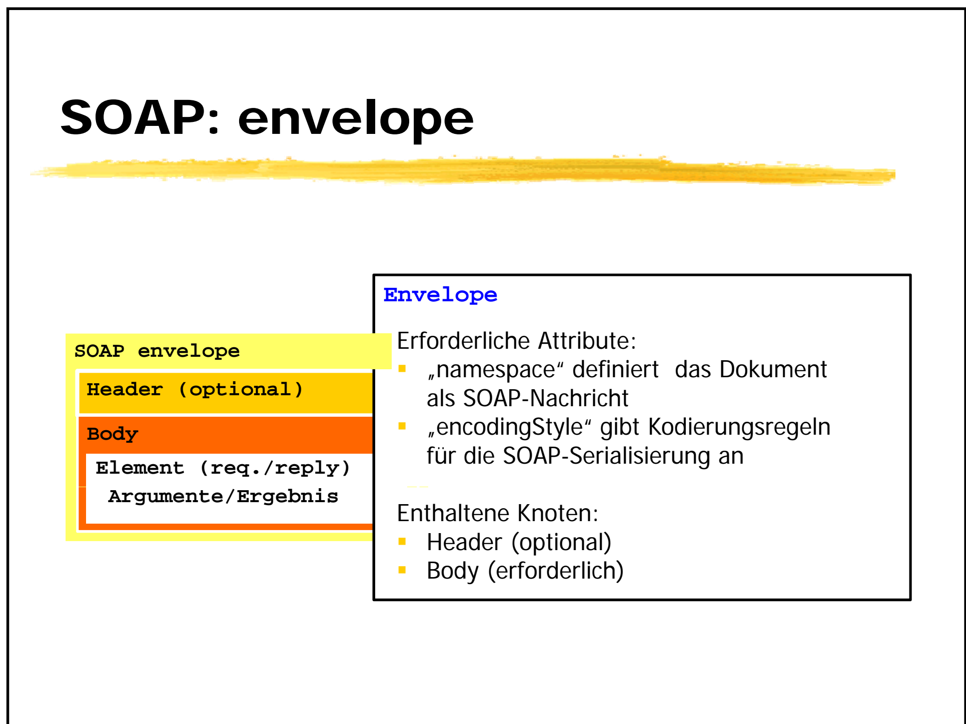
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)

service

- Gibt Adressen der Services an (pro „portType“ und „binding“)
- Kann mehrere definierte „portTypes“ zu einem Service bündeln

```
<service name="specificService">
  <port binding="tns:myBinding">
    <soap:address
 location="http://example.org/VS/service"/>
  </port>
</service>
```


Nochmal:
Web Services
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
Lookup-Service
UDDI
look up
WSDL
WSDL
register
Client
SOAP
Server
request / reply
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Jeweils XML-
Dokumente


SOAP: envelope
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Envelope
Erforderliche Attribute:
„namespace“ definiert das Dokument als SOAP-Nachricht
„encodingStyle“ gibt Kodierungsregeln für die SOAP-Serialisierung an
Enthaltene Knoten:
Header (optional)
Body (erforderlich)

SOAP: header

SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis

Header

Der SOAP-Header muss nicht enthalten sein

Durch ihn könnten zusätzliche Informationen über den Transaktionskontext, z.B. bezüglich Authentifizierung oder Bezahlung, angegeben werden

SOAP: body

SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis

Body

SOAP-Nutzlast:

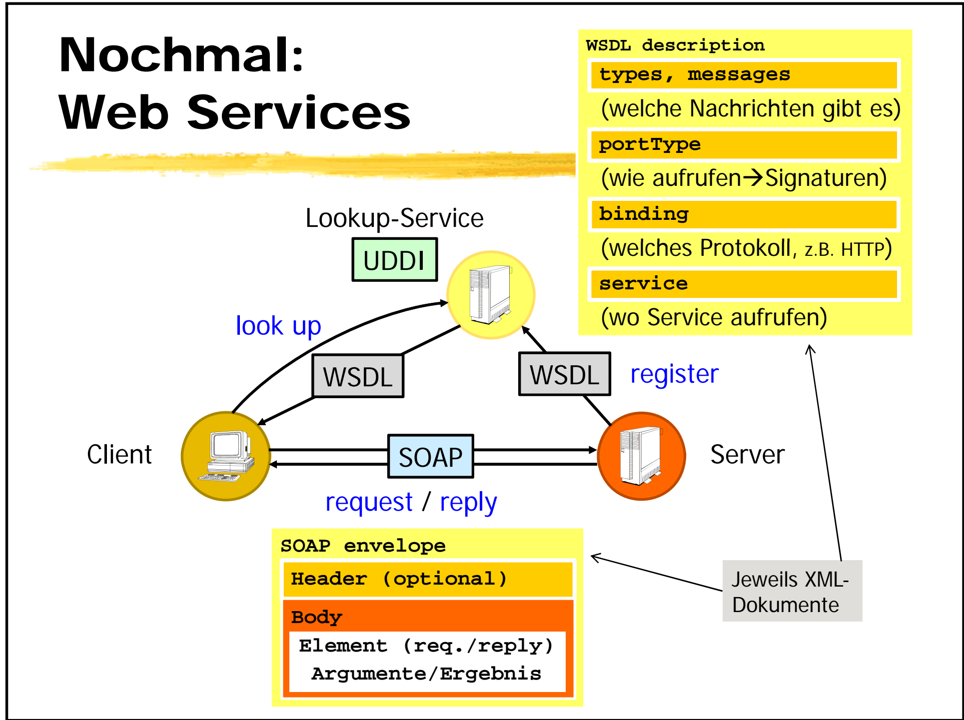
- Enthält die im WSDL definierte „message“ mit ihren Elementen
- Es wird der Namensraum aus dem WSDL angegeben, womit die dort definierten Tags verwendet werden können

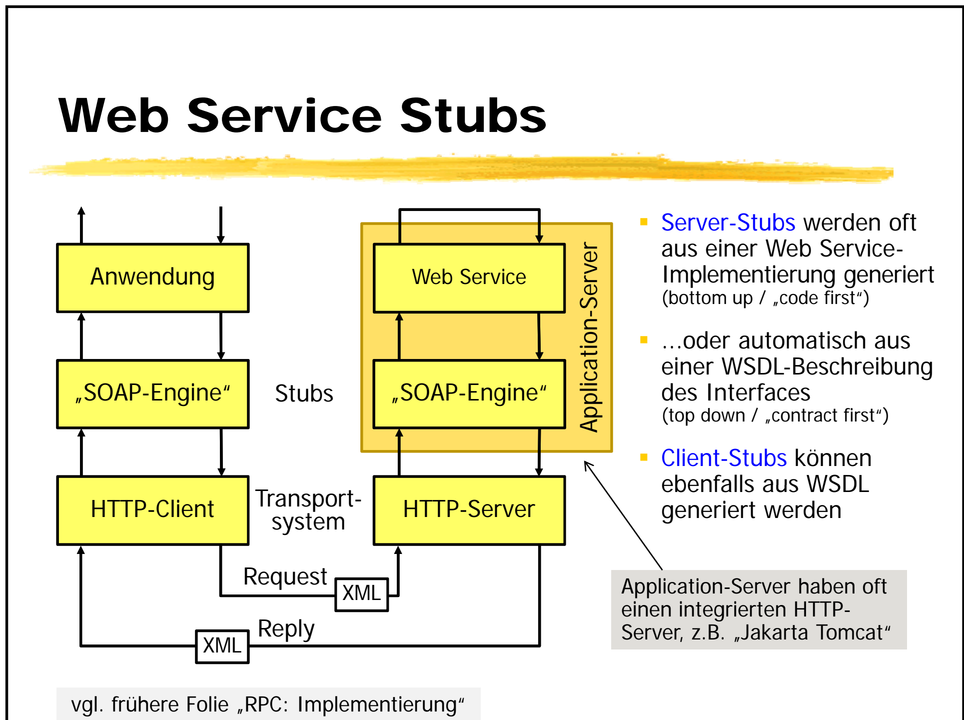
SOAP-Beispiel: Request


```
POST /VS/service HTTP/1.1
Host: example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 355

<?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">

<!-- Kein Header -->

<soap:Body>
  <ns:myRequest xmlns:ns="http://example.org/VS/WebServices/">
    <myArgs><i>23</i><j>4.2</j></myArgs>
  </ns:myRequest>
</soap:Body>

</soap:Envelope>
```

SOAP-Beispiel: Response


```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 340

<?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">

<!-- Kein Header -->

<soap:Body>
  <ns:myResponse xmlns:ns="http://example.org/VS/WebServices/">
    <myRet>27.2</myRet>
  </ns:myResponse>
</soap:Body>

</soap:Envelope>
```


Nochmal:
Web Services
WSDL description
types, messages
(welche Nachrichten gibt es)
portType
(wie aufrufen→Signaturen)
binding
(welches Protokoll, z.B. HTTP)
service
(wo Service aufrufen)
Lookup-Service
UDDI
look up
WSDL
WSDL
register
Client
SOAP
Server
request / reply
SOAP envelope
Header (optional)
Body
Element (req./reply)
Argumente/Ergebnis
Jeweils XML-Dokumente


Web Service Stubs
Anwendung
„SOAP-Engine“
Stubs
HTTP-Client
Transport-system
Web Service
„SOAP-Engine“
HTTP-Server
Application-Server
Request
XML
Reply
XML
Server-Stubs werden oft aus einer Web Service-Implementierung generiert
(bottom up / „code first“)
...oder automatisch aus einer WSDL-Beschreibung des Interfaces
(top down / „contract first“)
Client-Stubs können ebenfalls aus WSDL generiert werden
Application-Server haben oft einen integrierten HTTP-Server, z.B. „Jakarta Tomcat“
vgl. frühere Folie „RPC: Implementierung“

Entwicklung von Web Service-Komponenten mit Java-IDE

- JAX-WS: Java API for XML Web Services (Beispiel: NetBeans)

SimpleService.java

Source Design

```java
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package ch.simple.service;

import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.jws.WebParam;

/**
 * JAX-WS Example
 * "add" Web Service taking two Integers as input and returning an Integer
 */
@WebService(serviceName = "SimpleService")
public class SimpleService {

    /**
     * Web service operation
     */
    @WebMethod(operationName = "add")
    public int add(@WebParam(name = "i") int i, @WebParam(name = "j") int j) {
        return (i + j);
    }
}
```

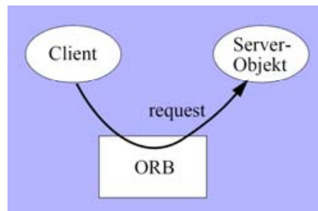
Erweiterung von einfachen Java-Objekten zu Web Services per Java Annotations (bottom up)

Middleware-Entwicklung, die Web Services historisch voranging

1. RPC-Bibliotheken (z.B. von SUN für UNIX)
   - Unterstützung des Client-Server-Paradigmas
   - einfache Schnittstellenbeschreibungssprache, Stubgeneratoren
   - Sicherheitskonzepte: Authentifizierung, Autorisierung, Verschlüsselung
2. Client-Server-Verteilungsplattformen (z.B. DCE)
   - Verzeichnisdienst, globaler Namensraum, globales Dateisystem
   - Programmierhilfen: u.a. Synchronisation, Multithreading
3. Objektbasierte Verteilungsplattformen (z.B. CORBA)
   - Kooperation zwischen verteilten Objekten
   - objektorientierte Schnittstellenbeschreibungssprache
   - „Object Request Broker“ als Vermittlungsinstanz

CORBA (Version 2.0 ca. ab 1997)

- Common Object Request Broker Architecture
  - ORB (Object Request Broker) als Vermittlungsinfrastruktur (Weiterleitung von Methodenaufrufen etc.)
  - IDL (Interface Description Language) mit Stub-Generatoren
  - Systemfunktionen und Basisdienste in Form von Object Services



Methodenaufruf unterschiedlicher Semantik:
synchron (mit Rückgabewerten; analog zu RPC)
verzögert synchron (Aufrufer wartet nicht auf das Ergebnis, sondern holt es sich später ab)
one way (asynchron: Aufrufer wartet nicht; keine Ergebnisrückgabe)

CORBA

- Ab ca. 2000 entstand der Wunsch nach einer wesentlichen Erweiterung der CORBA-Funktionalität aufgrund
  - neuer Anforderungen durch E-Commerce-Anwendungen
  - Ausbreitung des WWW
  - Aufkommen von Java
  - Aufkommen mobiler Geräte

Man lese dazu auch: Michi Henning: *The rise and fall of CORBA. Commun. ACM, Vol. 51, No. 8 (Aug. 2008), 52-57*

- Allerdings geriet die CORBA-Weiterentwicklung ins Stocken
  - zu weitreichende Anforderungen → komplex / ineffizient
  - kommerzielle Implementierungen der Erweiterungen zögerlich
  - fehlende Unterstützung durch Microsoft (eigene Architektur: DCOM und .NET)
  - man versuchte, es jedem Recht zu machen (widersprüchliche Interessen, barocke Konstrukte durch Kompromisse)
  - aufkommende Konkurrenzsysteme (z.B. Web Services), die z.T. direkter und besser an die neuen Anforderungen angepasst waren

Resümee (4)

- RPC-Fehlersemantikklassen
  - maybe, at-least-once, at-most-once
- Weiteres zu RPCs
  - asynchroner RPC, call-back, context-handle, broadcast
  - Sicherheitsstufen (Authentifizierung, Verschlüsselung, Signatur)
- Lookup-Service
- Web Services
  - WSDL-Servicebeschreibung (als XML-Dokument)
  - SOAP-Serviceaufruf
- Middleware: historischer Kurzüberblick
  - CORBA: Architektur (ORB, IDL), Schicksal der Weiterentwicklung

Zustandsändernde / -invariante Aufträge

- Verändern Aufträge den Zustand des Servers?
  - typische zustandsinvariante Aufträge: Anfrage bei Auskunftsdienst (z.B. Namensdienst oder Zeitservice)
  - typische zustandsändernde Aufträge: Schreiben bei Datei-Server
- Idempotente Aufträge
  - Wiederholung eines Auftrags führt zum gleichen Effekt
  - Beispiel: „Schreibe in Position 317 von Datei XYZ den Wert W“ (ist aber nicht zustandsinvariant!)
  - Gegenbeispiel: „Schreibe ans Ende der Datei XYZ den Wert W“
  - Gegenbeispiel: „Wie spät ist es?“ (ist aber zustandsinvariant!)
- Bei Idempotenz oder Zustandsinvarianz kann bei fehlgeschlagenem Auftrag (timeout beim Client) dieser problemlos erneut abgesetzt werden (→ einfache Fehlertoleranz)