LV8111 - Geschäftsprozessintegration

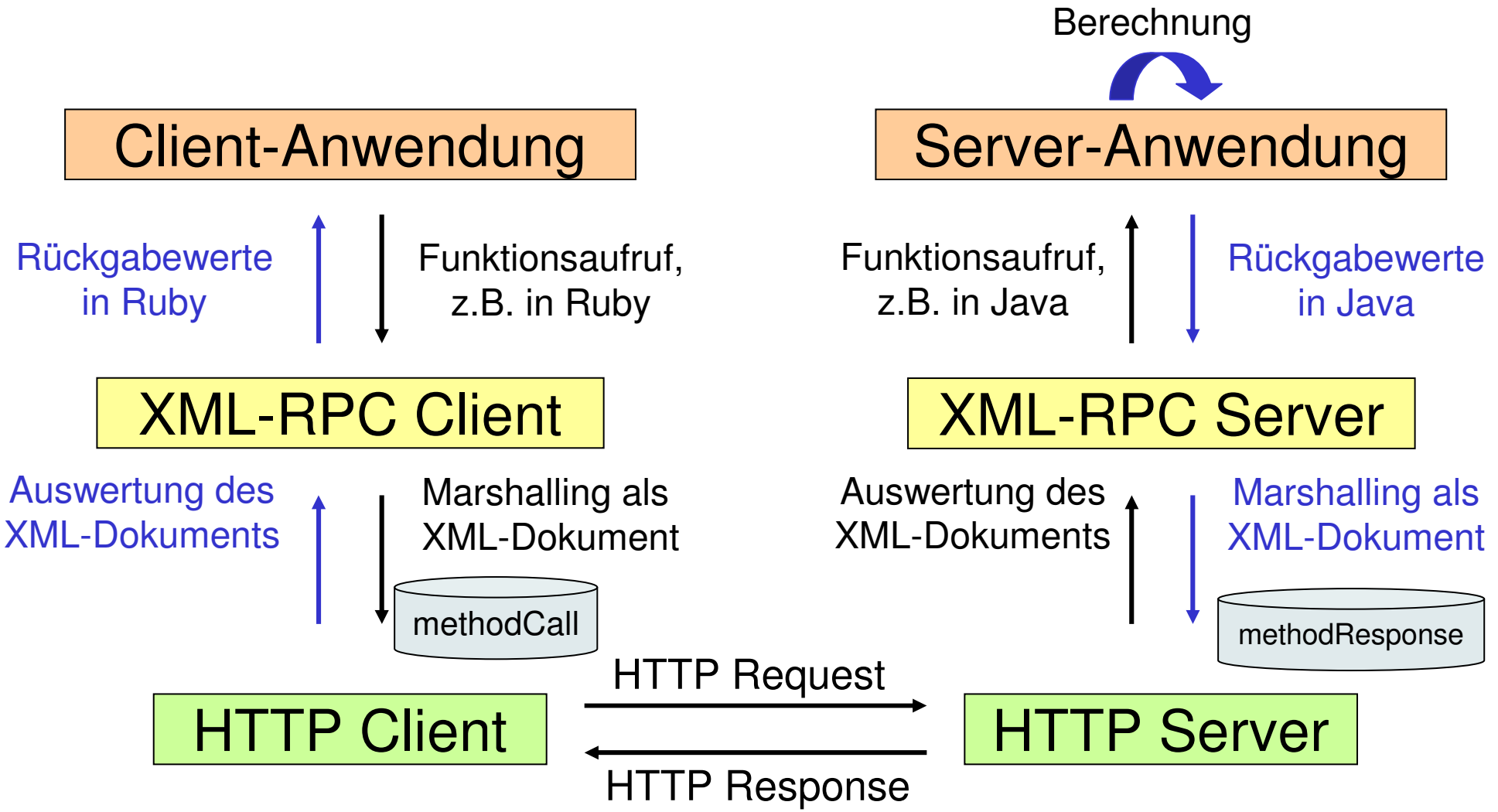
Eine Vertiefungsveranstaltung
im Master-Studiengang Informatik

Web Services:
XML-RPC

XML-RPC

- Entstehung
  - Entwickler der Spezifikation
    - Dave Winer, Userland Software
    - Bob Atkinson, Mohsen Al-Ghosein, Microsoft
    - Don Box, Developmentor
  - Status
    - 1998: Erstes Release. 21.1.1999, 16.10.1999, 30.6.2003: Updates
    - Bem.: XML-RPC ist kein Standard im Sinne des W3C oder der IETF.
  - Ziel
    - RPC über das Internet, unter Verwendung von XML und HTTP.
  - Quellen
    - http://www.xmlrpc.com/spec Spezifikationen
    - http://xmlrpc-c.sourceforge.net/xmlrpc-howto/xmlrpc-howto.html
      How-To, auch für Perl, Python, C/C++, Java, PHP, Ruby, .NET
    - Programming Web Services with XML-RPC, von Simon St. Laurent, Joe Johnston, Edd Dumbhill. O'Reilly, Sebastopol, CA, 2001.
      ISBN 0-596-00119-3

XML-RPC


Berechnung
Client-Anwendung
Server-Anwendung
Rückgabewerte in Ruby
Funktionsaufruf, z.B. in Ruby
Funktionsaufruf, z.B. in Java
Rückgabewerte in Java
XML-RPC Client
XML-RPC Server
Auswertung des XML-Dokuments
Marshalling als XML-Dokument
Auswertung des XML-Dokuments
Marshalling als XML-Dokument
methodCall
methodResponse
HTTP Request
HTTP Client
HTTP Server
HTTP Response

XML-RPC

- Funktionsweise
  - Codierung eines RPC als XML-Nachricht vom Dokumententyp "**methodCall**"
  - Übertragung vom Client an den Server mit einem HTTP Request, Methode "POST"
  - Serverseitige Implementierung der Funktionalität
    - als CGI-Anwendung bzw. mit analogen Techniken
    - als *stand-alone* Anwendung mit eingebautem (einfachen) HTTP Server.
  - Antwort des Servers per HTTP Response mit einem XML-Dokument vom Typ "**methodResponse**"
  - Umwandlung des Antwort-Dokuments in Rückgabewerte einer Funktion, Prozedur oder Methode durch den XML-RPC Client.

XML-RPC

- HTTP Request Header
  - (Methode: Immer "POST")
  - User-Agent
  - Host
  - Content-Type Immer "text/xml"
  - Content-Length

XML-RPC

- Beispiel: Postleitzahlen-Verzeichnis (PLZ → Ortsname)

```
POST /cgi-bin/my-xmlrpc-server.cgi HTTP/1.1
User-Agent: xmlrpc4r (WinNT)
Host: myhost.myorg.xy
Content-Type: text/xml
Content-length: 186

<?xml version="1.0"?>
<methodCall>
  <methodName>beispiel.getCityNameByZIP</methodName>
  <params>
    <param> <value><i4>65197</i4></value> </param>
  </params>
</methodCall>
```

XML-RPC

- Beispiel-Antwort:

```
HTTP/1.1 200 OK
Connection: close
Server: Apache 2.0.49 on Debian Linux ....
Date: Mon, 26 Apr 2004 20:01:30 GMT
Content-Type: text/xml
Content-Length: 183

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><string>Wiesbaden</string></value>
    </param>
  </params>
</methodResponse>
```

XML-RPC

- Beispiel für eine Fehler-Antwort:

```
HTTP/1.1 200 OK
Connection: close
Server: Apache 2.0.49 on Debian Linux ....
Date: Mon, 28 Apr 2004 20:30:52 GMT
Content-Type: text/xml
Content-Length: 231

<?xml version="1.0"?>
<methodResponse>
  <fault><value><struct>
    <member><name>faultCode</name>
      <value><i4>4</i4></value></member>
    <member><name>faultString</name>
      <value><string>Err descr.</string></value></member>
  </struct></value></fault>
</methodResponse>
```

XML-RPC

- Fehlerantworten
  - HTTP-Code: Stets 200, außer bei Problemen mit dem Modul an sich.
  - Fehlerbedingungen aus Prozeduraufrufen in Element "**fault**" mitteilen!

- Konvention zum Inhalt von Element "**fault**"
  - Der Inhalt besteht nur aus einem "value"-Element vom Typ "struct"
  - Dieses spezielle "struct" enthält zwei besondere "member"-Elemente:
    - name = faultCode Ein "int", entspricht dem üblichen Return-Code
    - name = faultString Ein "string" mit beschreibendem Klartext

XML-RPC

- Element "**methodName**"
  - Immer erforderlich!
  - Enthält einen Text aus einer eingeschränkten Zeichenmenge
    - In Regexp-Notation: `[A-Za-z0-9_.:/]+`

- Element "**params**"
  - Nur erforderlich, wenn Parameter übergeben werden müssen
  - Enthält beliebig viele Elemente "**param**"

- Element "**param**"
  - Beschreibt einen einzelnen Übergabeparameter, einfach oder komplex
  - Ein Unter-Element aus folgender Liste:
    - "**array**" für ein Array aus Werten
    - "**struct**" für ein Hash (ungeordnete Liste von *name/value*-Paaren) aus Werten
    - "**value**" "wrapper", für alle Datentypen

XML-RPC

- Einfache Datentypen
  - "**int**" bzw. "**i4**"
    - Entspricht 4-byte signed integer
    - Als Regexp: [+-]?\d+
  - "**boolean**"
    - Enthält ausschließlich die Werte 0 oder 1
    - Als Regexp: [01]
    - Bem.: Die Spez. ist hier widersprüchlich und erwähnt ebenfalls: true|false
  - "**string**"
    - Enthält beliebige Stringwerte. Angeblich sind alle Zeichen zulässig, die XML ermöglicht (*parameter ref* / *entity ref* / CDATA-Konstrukte beachten, etwa für & und <).
    - Ob nur ASCII oder der volle Unicodeumfang unterstützt wird, kann aber vom jeweiligen Server abhängen.
    - Bem.: Die binäre Null sollte (direkt und indirekt) ausgeschlossen bleiben, weil XML 1.0 und 1.1 dies so fordern! Es gibt gegenteilige Aussagen...

XML-RPC

- Einfache Datentypen (Forts.)
  - "**double**"
    - Untermenge von "double" à la "C": Kein Inf, -Inf, NaN, keine Exponenten.
    - Als Regexp: [+-]?\d*\.\d*
    - Der tatsächliche Wertebereich ist implementierungsabhängig
  - "**dateTime.iso8601**"
    - Das Format ist *YYYYMMDD***T***hh*:*mm*:*ss*
    - Beispiel: 20061113T11:35:04
    - Zeitzone? Keine Festlegung, Server entscheidet / dokumentiert.
  - "**base64**"
    - Enthält einen String gemäß base64-Codierung
    - Geeignet zur Übermittlung beliebiger Binärdaten
    - Beispiel: <base64>eW91IGNhbid0IHJlYWQgdGhpcyE=</base64>

- **Default-Regel**
  - Ist kein Datentyp angegeben, wird "**string**" unterstellt!

XML-RPC

- **Arrays**
  - Ein "array"-Element enthält genau ein Element "**data**"
  - Ein "data"-Element enthält beliebige Elemente "**value**"
  - Beispiel:

```
<array>
 <data>
  <value><i4>12</i4></value>
  <value><string>Egypt</string></value>
  <value><boolean>0</boolean></value>
  <value><i4>-31</i4></value>
 </data>
</array>
```

XML-RPC

- **Struct**
  - Ein "struct"-Element besteht aus "**member**"-Elementen
  - Jedes "member"-Element besteht aus einen Element "**name**" und einem Element "**value**".
  - Beispiel:

```
<struct>
  <member>
    <name>lowerBound</name>
    <value><i4>18</i4></value>
  </member>
  <member>
    <name>upperBound</name>
    <value><i4>139</i4></value>
  </member>
</struct>
```

XML-RPC

- **Komplexe Datenstrukturen**
  - Ein Element "value" kann einen beliebigen Datentypen enthalten, incl. "array" und "struct", also lassen sich komplexe Datentypen aufbauen.
  - Arrays von Arrays oder Arrays von Struct sind kein Problem.
- **Structs, Skriptsprachen, XML**
  - Struct-Elemente sind Listen ungeordneter *name/value*-Paare
  - In Skriptsprachen wie Perl oder Ruby entsprechen ihnen die Hashes
  - In XML sind Attribute vergleichbar: Auch hier spielt die Reihenfolge keine Rolle, auch hier liegen Listen von *name/value*-Paaren vor (nur sind diese nicht direkt kaskadierbar).

Konventionen und Erweiterungen

- **Der Nil-Wert**
  - Z.B. bei Datenbankabfragen, Null-Zeigern etc.
  - Für Rubys "nil"-Wert ideal geeignet.
  - NICHT offiziell spezifiziert!
  - Dennoch: Von einigen Tools unterstützt
  - Verwendung ggf. nur als *empty element*:

```
<value><nil/></value>
```

XML-RPC

- DTD-Fragmente, inoffiziell:

```
<!ELEMENT methodCall     (methodName, params?)>
<!ELEMENT params         (param+)>
<!ELEMENT param          (value)>
<!ELEMENT value          (int | i4 | boolean | string |
                          double | dateTime.iso8601 |
                          base64 | array | struct | nil)>
<!ELEMENT array          (data)>
<!ELEMENT data           (value+)>
<!ELEMENT struct         (member+)>
<!ELEMENT member         (name, value)>
<!ELEMENT methodResponse (params|fault)>
<!ELEMENT fault          (value)>
<!ELEMENT nil            #EMPTY > <!-- falls verwendet -->
<!-- (alle anderen:)     #PCDATA -->
```

- Schema?
  - Leider nicht verfügbar. Datentypen und "fault" zu präzisieren!

- **Namenskonventionen für Methodennamen**
  - Keine Standards - nur *de facto*-Konventionen!
  - "camelCase" offenbar verbreitet
  - Namensraum-Konzept verwenden!
  - Trennung Namensraum/Methode mit Punkt. Beispiel:

```
namespace.myMethod
```

XML-RPC

- Selbstauskunft ("Introspektion"):
  - Das Problem:
    - Wie erfahre ich Einzelheiten über die Methoden eines (zunächst) unbekannten XML-RPC Servers?
  - Externe Lösung: Per HTML-Doku auf separatem URL
    - Folgeproblem: Woher URL bekommen?
  - Interne Lösung: Durch spezielle Methoden vom Server selbst!
  - **Die Konvention:**
    - Bestimmte Methoden des **reservierten Namensraums** "system" geben Auskunft über die Methoden eines XML-RPC Servers.

XML-RPC

- Selbstauskunft ("Introspektion"):
  - Pseudo-Prototypen dazu:

```
array  system.listMethods ()
```

  - Liefert ein Array von Strings mit den Methodennamen

```
array  system.methodSignature (string methodName)
```

  - Liefert ein Array von Strings mit den Rückgabe- und Übergabetypen.

```
string system.methodHelp (string methodName)
```

  - Liefert einen Hilfetext zur angegebenen Methode

XML-RPC

- Die Multicall-Konvention
  - Das Problem:
    - Manchmal ist die simultane Bearbeitung mehrerer RPCs innerhalb eines Requests viel effizienter als deren serielle Variante.
    - Wie lässt sich das ohne Änderung der Spezifikationen erreichen?
  - Lösung:
    - Die Multicall-Konvention, auch *boxcarring* genannt.
    - Grundlage: **`array system.multicall (array calls)`**
  - Die Idee:
    - Quelle: http://www.xmlrpc.com/discuss/msgReader$1208
    - Man übergibt ein Array von Methodenaufrufen samt Parametern
      - Jedes Element (call) ist ein struct der Art
        methodName: string, params: array
    - Man erhält ein Array von Antworten (incl. fault-Structs)
      - Antwort i entspricht der Antwort der jeweiligen Methode i

Beispiel: Ein einfacher Ruby-Client

(Quelle: XML-RPC Tutorial, Code von Michael Neumann)

```
require "xmlrpc/client"

# Make an object to represent the XML-RPC server.
server = XMLRPC::Client.new("xmlrpc-c.sourceforge.net",
                            "/api/sample.php")

# Call the remote server and get our result
result = server.call("sample.sumAndDifference", 5, 3)
sum = result["sum"]
difference = result["difference"]

puts "Sum: #{sum}, Difference: #{difference}"
```

XML-RPC

Beispiel: Ein dazu passender Ruby-Server (gleiche Quellen)

```
require "xmlrpc/server"

# Make an object to represent the XML-RPC server.
s = XMLRPC::CGIServer.new
# s = XMLRPC::Server.new( 8080 ) # Standalone version
class MyHandler
  def sumAndDifference(a, b)
    { "sum" => a + b, "difference" => a - b }
  end
end

s.add_handler("sample", MyHandler.new)
s.serve
```

XML-RPC

- On-line Demo
  - XML-RPC Server:
    - **`http://xmlrpc-c.sourceforge.net/api/sample.php`**
  - Proxy:
    - **`www-cache:8080`**
  - Client:
    - Erweiterter Ruby-Client
    - Mit Proxy-Benutzung und Introspektion
  - Erweiterten Versionen aus der Online-Demo:
    - Exception handling, dynamische Ergänzung von Methoden
    - Fileserver: **`~werntges/lv/bpi/01/rpc_*.rb`**