

## **Fernaufrufe**

Motivation


Stubs und Skeletons

Marshalling und Unmarshalling

Automatisierung

XML-RPC

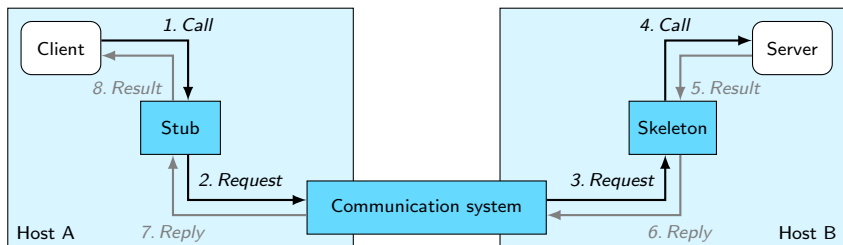


- „Fernaufruf“ als Oberbegriff
  - Prozedurfernaufruf (*Remote Procedure Call*)
  - Methodenfernaufruf an einem entfernten Objekt
- Bereitstellung derselben **Semantik wie bei einem lokalen Aufruf?**
  - Dienstbringung im Normalfall
  - Verhalten im Fehlerfall [Nähere Details hierzu in der Vorlesung zum Thema „Fehlertoleranz“.]
- Herausforderungen
  - Wie lässt sich ein Fernaufruf transparent gestalten?
  - Wie kann ein Aufruf auf Nachrichtenaustausch abgebildet werden?
  - Lassen sich für Aufrufe benötigte **Komponenten automatisch erzeugen?**
- Literatur
  -  Andrew D. Birrell and Bruce Jay Nelson  
**Implementing remote procedure calls**  
*ACM Transactions on Computer Systems*, 2(1):39–59, 1984.



# Stub und Skeleton

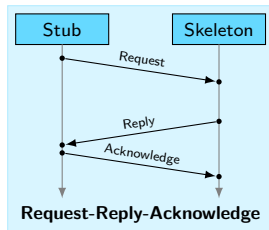
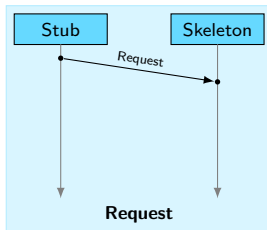
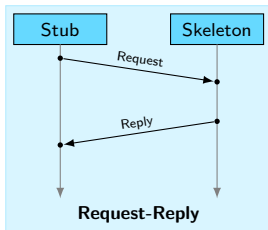
- *Stub (Client Stub)*
  - **Stellvertreter des Servers** auf Client-Seite
  - Umsetzung des Aufrufs in einen Nachrichtenaustausch
  - Verpacken der Aufrufparameter und Entpacken des Rückgabewerts
- *Skeleton (Server Stub)*
  - **Stellvertreter des Clients** auf Server-Seite
  - Umsetzung der Aufrufrückkehr in einen Nachrichtenaustausch
  - Entpacken der Aufrufparameter und Verpacken des Rückgabewerts



- Abbildung auf Kommunikationsprimitiven
  - *No-wait Send*
    - **Explizite Zuordnung** der Antwort zur Anfrage erforderlich
    - Im Normalfall keine zusätzlichen Nachrichten im Kommunikationssystem
  - *Synchronization Send*
    - **Explizite Zuordnung** der Antwort zur Anfrage erforderlich
    - Versand zusätzlicher Bestätigungsnachrichten im Kommunikationssystem
  - *Remote-invocation Send*
    - **Implizite Kopplung** zwischen Anfrage und Antwort
    - Im Normalfall keine zusätzlichen Nachrichten im Kommunikationssystem
- Alternative Implementierungen für *No-wait Send*
  - Umsetzung mittels UDP
    - Ein Paket pro Nachricht
    - Unzuverlässige Nachrichtenzustellung
  - Umsetzung mittels TCP
    - **Nachrichtengrenzen** müssen im Datenstrom kenntlich gemacht werden
    - Zuverlässige Nachrichtenzustellung, solange die Verbindung existiert



- *Request-Reply*
  - **Übliches Verfahren** zwischen Stub und Skeleton
  - Antwort liefert Ergebnis und dient als Ausführungsbestätigung
- *Request*
  - Weglassen der Antwort
  - **Optimierung**, falls Client kein Ergebnis / keine Bestätigung benötigt
- *Request-Reply-Acknowledge*
  - **Zusätzliche Bestätigung** vom Stub nach Erhalt der Antwort
  - Hinweis an den Skeleton, dass die komplette Interaktion erfolgreich war



- Basis für eine **Interaktion per Nachrichtenaustausch**
  - *Marshalling*: Serialisierung aller zu sendender Daten in eine Nachricht
  - *Unmarshalling*: Deserialisierung einer Nachricht nach Empfang
- Nachrichteninhalte bei Fernaufrufen
  - **Anfrage**
    - Bei Methodenfernaufruf: Bezeichner des entfernten Objekts
    - Bezeichner der aufzurufenden Prozedur / Methode
    - Parameter
  - **Antwort**
    - Normalfall: Rückgabewert
    - Fehlerfall: Informationen zur Fehlersituation (z. B. Exception)
- Häufiges Ziel: **Minimierung der Nachrichtengrößen**
  - Grundprinzip: Beschränkung auf Informationen, die der jeweils andere noch nicht hat, aber für die Erfüllung seiner Aufgabe braucht
  - Beispiel: Keine Übermittlung von Prozedur- / Methodensignaturen



- Potenzial zur Reduzierung der zu übermittelnden Datenmenge
  - Eingabeparameter
    - Informationsfluss: Client  $\rightarrow$  Server
    - **Bestandteil der Anfrage** (*Call-by-Value*)
  - Ausgabeparameter
    - Informationsfluss: Client  $\leftarrow$  Server
    - **Bestandteil der Antwort** (*Call-by-Result*)
  - Ein-/Ausgabeparameter
    - Informationsfluss: Client  $\leftrightarrow$  Server
    - **Bestandteil beider Nachrichten** (*Call-by-Value-Result*)
    - Ersetzen des Werts auf Client-Seite durch den Wert in der Antwort
- Problem
  - Die Art eines Parameters ist nicht in allen Programmiersprachen eindeutig
  - Beispiele für Problemfälle aus C/C++
    - Zeiger: `char*`, `struct Foo*`
    - Feld: `char bar[4]`



- Problem: Umsetzung von *Call-by-Reference*
  - Client und Server haben im Regelfall **keinen gemeinsamen Speicher**
  - Konsequenzen
    - Speicheradressen sind nicht systemweit eindeutig
    - Versand von lokalen Zeigern in Anfragen / Antworten nicht praktikabel
- Variante 1: Dereferenzierung und Abbildung auf andere Semantiken
  - **Übermittlung von Kopien**
  - Je nach Parameterart: Einsatz von *Call-by- $\{Value, Result, Value-Result\}$*
- Variante 2: Realisierung mittels *Remote-Referenz*
  - Übermittlung eines **systemweit eindeutigen Zeigers** anstatt des lokalen
  - Zugriff auf Parameter erfolgt per Fernaufruf
- Mögliche Vergleichskriterien für **Effizienzabschätzung**
  - Aufwand für Parameter, die Referenzen auf andere Objekte enthalten
  - Anteil der vom Empfänger tatsächlich benötigten Daten
  - Häufigkeit des Zugriffs auf den Parameter durch den Empfänger





- Integrierte Ansätze (Beispiel: Argus)
  - **Internes Wissen** über Datentyp- und Laufzeitmodell
  - Compiler agiert gleichzeitig als Stub-Generator
- Partiiell integrierte Ansätze (Beispiel: Java RMI)
  - Compiler kennt Konzept eines Fernaufrufs nicht
  - Unterstützung von Fernaufrufen ist **Teil der Laufzeitumgebung**
  - Einsatz von *Reflection*
- Separierte Ansätze (Beispiel: CORBA)
  - **Fehlendes / unvollständiges Wissen** über Datentyp- und Laufzeitmodell
  - Schnittstellenverhalten wird mittels *IDL* explizit beschrieben

## ■ Literatur



Barbara Liskov

**Distributed programming in Argus**

*Communications of the ACM*, 31(3):300–312, 1988.



- Konzepte
  - **Analyse von Systemeigenschaften** zur Laufzeit
  - Dynamische Modifikation von Strukturen oder Verhalten
- Einsatzgebiete im Kontext von Fernaufrufen
  - Zur Laufzeit: Erzeugung von **Proxies aus Schnittstelleninformationen**
    - Abfangen von lokalen Methodenaufrufen
    - Ermöglicht die erforderliche Abbildung auf Nachrichtenaustausch
  - **Dynamischer Aufruf von Methoden**
    - Ermöglicht die Rückabbildung von Anfragenachricht auf Methodenaufruf
    - Beispiel: `invoke`-Methode der Klasse `Method` in Java

```
Method m = PrintStream.class.getMethod("println", String.class);  
m.invoke(System.out, "Hallo Welt!");
```


- Beispiel: Java RMI [Näheres in der Tafelübung.]
  - Dynamische Proxies als Basis für Fernaufruf-Stubs
  - Generischer Skeleton auf Server-Seite



- Sprache zur **Beschreibung von Objektschnittstellen**
  - Entwickelt für CORBA
  - Angelehnt an C++
  - Spezifikation eigener Datentypen
- Sprachabbildungen (*Language Mappings*)
  - **Sprachenspezifische Übersetzung** von IDL-Konstrukten
  - Festlegung für diverse Sprachen (z. B. Ada, C, C++, Java, Lisp, Python)
- Beispiel: Festlegung von Parameterarten (*in*, *out* oder *inout*)

```
module vs {  
    interface VSFileMetadataService {  
        void getFileInfo(  
            in string name,    // Eingabeparameter  
            out long size,     // Ausgabeparameter  
            out string owner   // Ausgabeparameter  
        );  
    };  
};
```



- **Plattformunabhängiges Fernaufrufprotokoll**
  - XML: Nachrichtenformat
  - HTTP: Transportprotokoll
- **Umfang**
  - Spezifikation
    - Festlegung der verfügbaren Datentypen
    - Aufbau von Anfrage- und Antwortnachrichten
  - Implementierungen für **verschiedene Programmiersprachen**
    - Java
    - C++
    - Python
    - ...
- **Literatur**
  -  XML-RPC Specification  
<http://xmlrpc.scripting.com/spec.html>



## ■ Primitive Datentypen (<value>)

| XML-Tag          | Wert                                     |
|------------------|--|
| i4 bzw. int      | 32-Bit-Integer                           |
| double           | Fließkommazahl mit doppelter Genauigkeit |
| boolean          | 0 (false), 1 (true)                      |
| string           | Zeichenkette                             |
| base64           | Base64-codierte Binärdaten               |
| dateTime.iso8601 | Datum und Uhrzeit                        |

## ■ Komplexe Datentypen

### ■ Liste primitiver und/oder komplexer Datentypen (<array>)

```
<array><data>  
  <value>[Wert]</value>  
  [Weitere <value>-Elemente]  
</data></array>
```

### ■ Ungeordnete Menge aus Schlüssel-Wert-Paaren (<struct>)

```
<struct>  
  <member><name>[Schlüssel]</name><value>[Wert]</value></member>  
  [Weitere <member>-Elemente]  
</struct>
```



## ■ HTTP-Header (Ausschnitt)

```
POST [URI: Beliebiger Pfad, potentiell leer] HTTP/1.1
Content-Type: text/xml
User-Agent: [Nutzeranwendung]
Content-Length: [Nachrichtenlänge]
Host: [Zielrechneradresse: z. B. Hostname und Port]
```

## ■ HTTP-Body

```
<?xml version="1.0"?>
<methodCall>
  <methodName>[Methodenname]</methodName>
  <params>
    <param>[Aufrufparameter]</param>[...]
  </params>
</methodCall>
```

- Kapselung der Aufrufinformationen in einzelner `<methodCall>`-Element
- Methodenname: Beliebige Zeichenkette
- Aufrufparameter: Primitive oder komplexe Datentypen



## ■ HTTP-Header

```
HTTP/1.1 200 OK
Server: [Server-Anwendung]
Content-Type: text/xml
Content-Length: [Nachrichtenlänge]
[...]
```

- **Identisch für beide Antworttypen**
- Status-Code bezieht sich auf HTTP, nicht auf den Methodenaufruf

## ■ Reguläre Antwort: **HTTP-Body**

```
<?xml version="1.0"?>
<methodResponse>
  <params><param>[Rückgabewert]</param></params>
</methodResponse>
```

- Kapselung des Rückgabewerts in einzeltem `<methodResponse>`-Element
- Rückgabewert: Primitiver oder komplexer Datentyp in `<param>`-Element



- Fehlermeldung: **HTTP-Body**

```
<?xml version="1.0"?>
<methodResponse>
  <fault>
    <value><struct>
      <member>
        <name>faultCode</name>
        <value><i4>[Fehler-Code]</i4></value>
      </member>
      <member>
        <name>faultString</name>
        <value><string>[Fehlerbeschreibung]</string></value>
      </member>
    </struct></value>
  </fault>
</methodResponse>
```

- Kapselung der Fehlermeldung in einzelнем <fault>-Element
- Semantik des Fehler-Codes **von der Anwendung frei wählbar**

