— VS —

Fernaufrufe

Überblick

Aktionsorientierte vs. Datenorientierte Kommunikation

Eine Kommunikation, bei der die bei einem anderen Prozeß beantragte Aktivität im Vordergrund steht, wird **aktionsorientiert** genannt. [2]

Grundschema der Interaktion zw. Auftraggeber (AG) und Auftragnehmer (AN):

1. der AG sendet die Aufforderung zur Diensterbringung, die ein AN empfängt
2. währenddessen wartet der AG auf eine Rückmeldung
3. die Rückmeldung sendet der AN nach erfolgter Diensterbringung
4. mit Zustellung der Rückmeldung kann der AG weiterarbeiten

Eine aktionsorientierte Kommunikation erfordert demnach zwei Vorgänge **datenorientierte Kommunikation**: 1. und 3 (d.h. *send* und *reply*)

Kommunikationsmittel „Prozedur“

Gemeinsamkeit von Prozeduraufruf und aktionsorientierter Kommunikation:

- der AG entspricht der Routine, die den Prozeduraufruf tätigt *Client*
- der AN entspricht der aufgerufenen Prozedur *Server*
- senden der Aufforderung/Rückantwort entspricht dem Aufruf/Rücksprung

Unterschied: beim Prozeduraufruf ist der die Prozedur aufrufende Prozess mit dem die Prozedur ausführenden identisch

- aktionsorientierte Kommunikation ist der Aufruf einer entfernten Prozedur
- „entfernt“ heißt „anderer Faden, Adressraum, Prozess und/oder Rechner“[1]
- aktionsorientierte Kommunikation meint **Prozedurfernaufruf** [6]

[1] Entsprechend einem feder-, leicht-, mittel- bzw. schwergewichtigen Aufurf. In dem Zusammenhang steht dann der Begriff „Prozess“ für „Faden mit eigenem Adressraum“.

Synchrones Kommunikationsmittel

- *remote-invocation send* unterstützt die Werte liefernden Prozedurfernaufrufe:

  send gibt die Aufforderung ab, blockiert den AG und deblockiert ggf. den AN
  receive vom AN nimmt die Aufforderung entgegen
  reply übermittelt die Rückmeldung des AN und deblockiert den AG

- *synchronization send* unterstützt die sonstigen Prozedurfernaufrufe:

  send gibt die Aufforderung ab, blockiert den AG und deblockiert ggf. den AN
  receive vom AN nimmt die Aufforderung entgegen und deblockiert den AG

Asynchrone Variante — „Versprechen“ ($Promise$)

asynchroner Prozedurfernaufruf kehrt sofort zurück und *verspricht* dabei die Resultatslieferung, ohne jedoch den Verfügbarkeitszeitpunkt festzulegen

- ein *promise*-Objekt dient der Aufnahme des ihm zugeordneten Resultats
- der *promise*-Zustand kann (mittels `ready()`) abgefragt werden:
  blockiert der Aufruf läuft, ein Resultat liegt noch nicht vor
  - ein vom Aufrufer ausgeführtes `claim()` wirkt blockierend

  bereit der Aufruf ist beendet, das Resultat liegt vor
  - durch `claim()` ggf. blockierte Aufrufer werden deblockiert
- linguistische Unterstützung in klassenbasiertem Sinn [8] ist angebracht
- mit *no-wait send* wird die *promise*-Implementierung ideal unterstützt

Prozeduraufruf $\Rightarrow$ Nachrichtenaustausch

client stub Prozedurstumpf auf Seite des Auftraggebers:

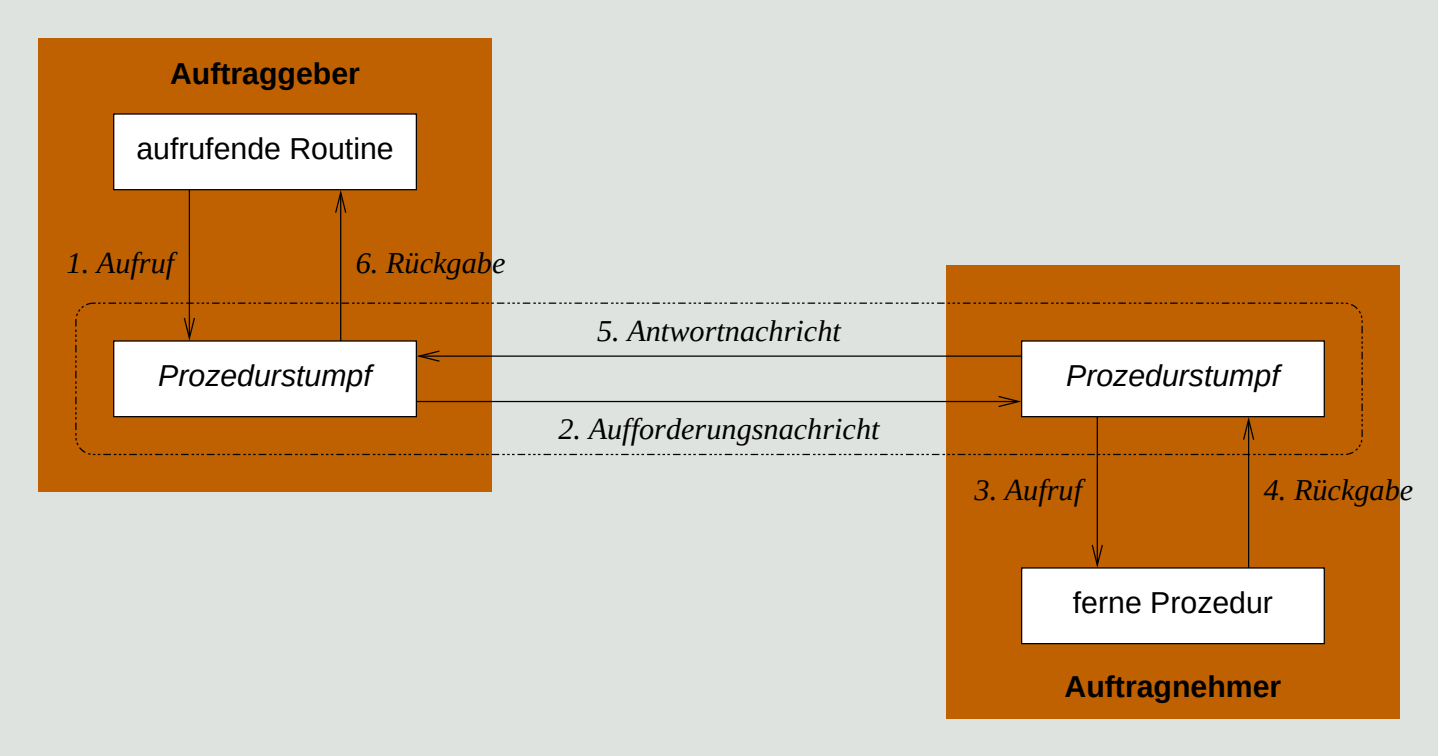
- abstrahiert von der Örtlichkeit der fernen, aufgerufenen Prozedur
- setzt den Prozeduraufruf um in einen Nachrichtenaustausch
- verpackt aktuelle Parameter und entpackt Rückgabewerte

server stub Prozedurstumpf auf Seite des Auftragnehmers:[2]

- abstrahiert von der Örtlichkeit der fernen, aufrufenden Routine
- setzt die Prozedurrückkehr um in einen Nachrichtenaustausch
- entpackt aktuelle Parameter und verpackt Rückgabewerte

[2]Verschiedentlich wird mit *client/server stub* nicht die Örtlichkeit des Prozedurstumpfes zum Ausdruck gebracht, sondern wovon er abstrahiert. Dies trifft den eigentlichen Sachverhalt, dass ein Prozedurstumpf nämlich die jeweils ferne Seite repräsentiert, besser, hat sich jedoch nicht allgemein durchgesetzt.

Prozedurstümpfe beim Prozedurfernaufruf



„Türme von Hanoi“ (1)

Turm/Scheiben verwalten

```
#include <stdlib.h>

extern void versetze (int, char, char, char);

static int n;

int main (int argc, char* argv[]) {
    versetze(n = (argc == 1) ?  1 : strtol(argv[1], 0, 0), 'A', 'B', 'C');
}

int scheiben () {
    return n;
}
```

„Türme von Hanoi“ (2)

Turm versetzen

```
extern void schleppe (int, char, char);

void versetze (int n, char from, char to, char via) {
    if (n == 1) {
        schleppe(1, from, to);
    } else {
        versetze(n - 1, from, via, to);
        schleppe(n, from, to);
        versetze(n - 1, via,  to,  from);
    }
}
```

„Türme von Hanoi“ (3)

Scheibe schleppen

```
#include <iostream.h>

extern int scheiben();

void schleppe (int n, char from, char to) {
    cout << "schleppe Scheibe " << scheiben() - n + 1
         << " von Turm "        << from
         << " nach Turm "       << to
                                << endl;
}
```

„Türme von Hanoi“ (4) Ackern. . .

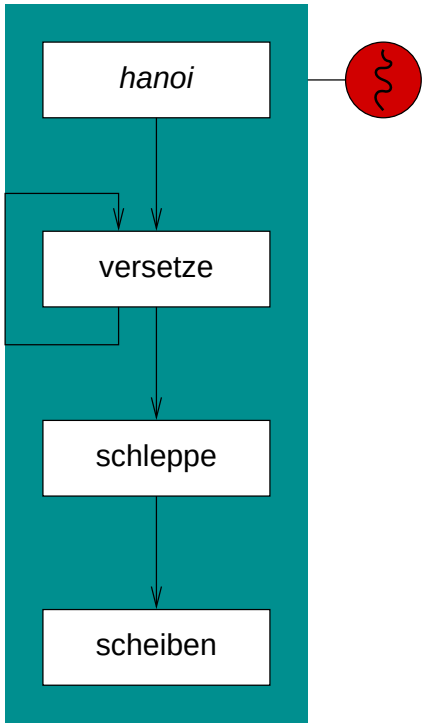
```
wosch@faui42y 13> hanoi 3
schleppe Scheibe 3 von Turm A nach Turm B
schleppe Scheibe 2 von Turm A nach Turm C
schleppe Scheibe 3 von Turm B nach Turm C
schleppe Scheibe 1 von Turm A nach Turm B
schleppe Scheibe 3 von Turm C nach Turm A
schleppe Scheibe 2 von Turm C nach Turm B
schleppe Scheibe 3 von Turm A nach Turm B
```

```
wosch@faui42y 14> hanoi 4711
schleppe Scheibe 4711 von Turm A nach Turm B
schleppe Scheibe 4710 von Turm A nach Turm C
schleppe Scheibe 4711 von Turm B nach Turm C
...
```

Programmauslegung (1)

Einfädig

- Ausgangspunkt: **sequentielles Programm**
  - ein Faden zieht sich durch alle Prozeduren hindurch:
    `main()` → `versetze()` → `schleppe()` → `scheiben()`
  - technisch realisiert als Prozessinkarnation (*thread*)
- alle Programmteile residieren im selben Adressraum
  - die Aufrufe/Rücksprünge sind ELOPs der CPU
  - ebenso die Zugriffe auf den Arbeitsspeicher/RAM



Programmauslegung (2)

Mehrfädig

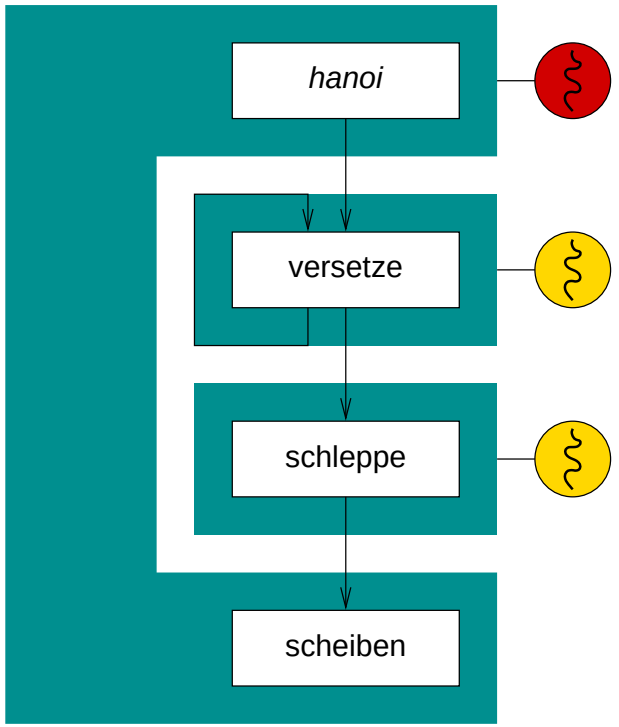
- Variante: prozessorientiertes Programm

  Hilfsfäden verrichten die eigentliche Arbeit
  - ein Faden versetzt Türme
  - ein anderer Faden schleppt Scheiben

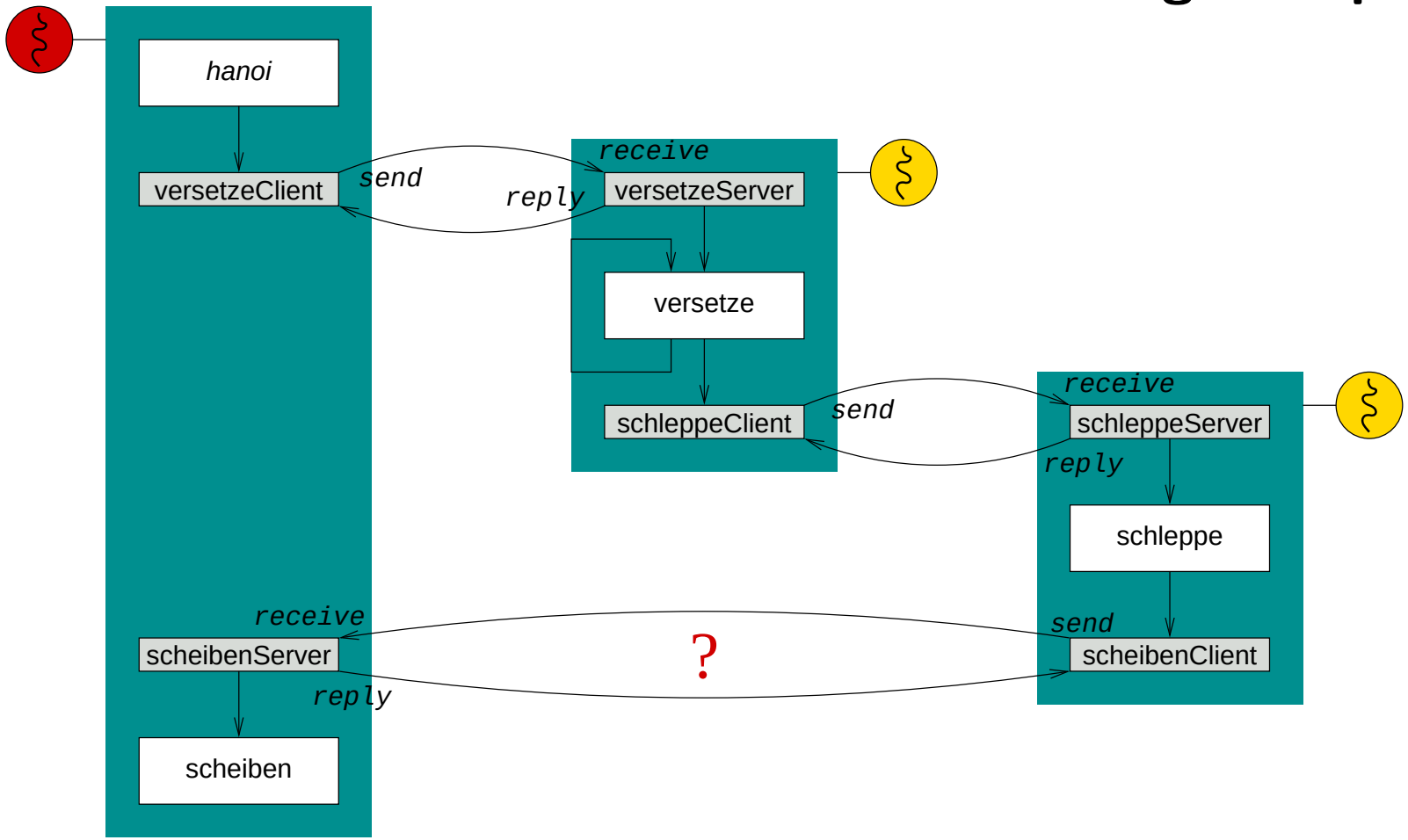
  Hauptfaden verwaltet Turm und Scheiben

- Verteilungstransparenz über Prozedurstümpfe
  - Wiederverwendung der Implementierung
  - Architekturfestlegung zum Bindezeitpunkt

- Problem: Rückruf (`scheiben()`) zum Hauptfaden



Verteilungstransparenz



Prozedurfernaufruf (1)

```
#include "ipcMessage.h"

extern void versetze (int, char, char, char);

namespace RPC {
  class versetzeMessage : private ipcMessage {
    int quantity;
    char from, to, via;
    friend void versetzeServer ();
  public:
    void versetze (int n, char from, char to, char via) {
      quantity = n;
      this->from = from; this->to = to; this->via = via;
      send("versetzeServer");
    }
  };

  void versetze (int n, char from, char to, char via) {
    versetzeMessage().versetze(n, from, to, via);
  }
```

Turm versetzen

```
  void versetzeServer () {
    versetzeMessage msg;
    for (;;) {
      int who = msg.receive();
      ::versetze(msg.quantity,
                 msg.from, msg.to, msg.via);
      msg.reply(who);
    }
  }
}/* RPC */;
```

```
void
versetze (int n, char from, char to, char via) {
  RPC::versetze(n, from, to, via);
}
```

Prozedurfernaufruf (2)

```
#include "ipcMessage.h"

extern void schleppe (int, char, char);

namespace RPC {
  class schleppeMessage : private ipcMessage {
    int slice;
    char from, to;
    friend void schleppeServer ();
  public:
    void schleppe (int n, char from, char to) {
      slice = n;
      this->from = from; this->to = to;
      send("schleppeServer");
    }
  };

  void schleppe (int n, char from, char to) {
    schleppeMessage().schleppe(n, from, to);
  }
```

Scheibe schleppen

```
  void schleppeServer () {
    schleppeMessage msg;
    for (;;) {
      int who = msg.receive();
      ::schleppe(msg.slice, msg.from, msg.to);
      msg.reply(who);
    }
  }
}/* RPC */;
```

```
void schleppe (int n, char from, char to) {
  RPC::schleppe(n, from, to);
}
```

Prozedurfernaufruf (3)

```
#include "ipcMessage.h"

extern int scheiben ();

namespace RPC {
  class scheibenMessage : private ipcMessage {
    int quantity;
    friend void scheibenServer ();
  public:
    int scheiben () {
      send("scheibenServer");
      return quantity;
    }
  };

  int scheiben () {
    return scheibenMessage().scheiben();
  }
```

Turm/Scheiben verwalten

```
  void scheibenServer () {
    scheibenMessage msg;
    for (;;) {
      int who = msg.receive();
      msg.quantity = ::scheiben();
      msg.reply(who);
    }
  }
}/* RPC */;
```

```
int scheiben () {
  return RPC::scheiben();
}
```

Verklemmungsgefahr durch Rückruf

- derselbe Hauptfaden dient zwei (an sich unabhängigen) Funktionen:
  1. er ist `versetze()` Klient ..... `main()`
     - der auf die Beendigung der von ihm aufgerufenen Prozedur wartet
  2. er ist `scheiben()` Dienstanbieter ..... `scheibenServer()`
     - der entfernt gestellte Aufträge annehmen und bearbeiten soll

- der Faden müsste zwischen beiden Funktionen „jederzeit“ multiplexbereit sein
  - was nicht geht: entweder wartet er im *send* oder im *receive*[3]

- alle Bedingungen einer (möglichen) Verklemmung sind erfüllt

[3] Im *send* auf die Beendigung des `versetze()` Fernaufrufs, im *receive* auf das Eintreffen der entfernt gestellten und mit *reply* zu beantwortenden Aufforderung eines `scheiben()` Aufrufs.

Verklemmungssituation

- gegeben sind die drei Fäden F_1, F_2 und F_3, für die jeweils gilt:

  1. F_1 wartet im *send* auf das *reply* von F_2 ..... `RPC::versetze()`
  2. F_2 wartet im *send* auf das *reply* von F_3 ..... `RPC::schleppe()`
  3. F_3 wartet im *send* auf das *receive* von F_1 ..... `RPC::scheiben()`

- in dieser Konstellation ist die Verklemmung beim Rückruf zwingend
  - die *remote-invocation send*-Umsetzung bewirkt **zyklisches Warten**
  - die aktionsorientierte Kommunikation ist nicht Ursache des Problems

- vorbeugende Maßnahmen verhindern die Verklemmung (*deadlock prevention*)

Verklemmungsvorbeugung

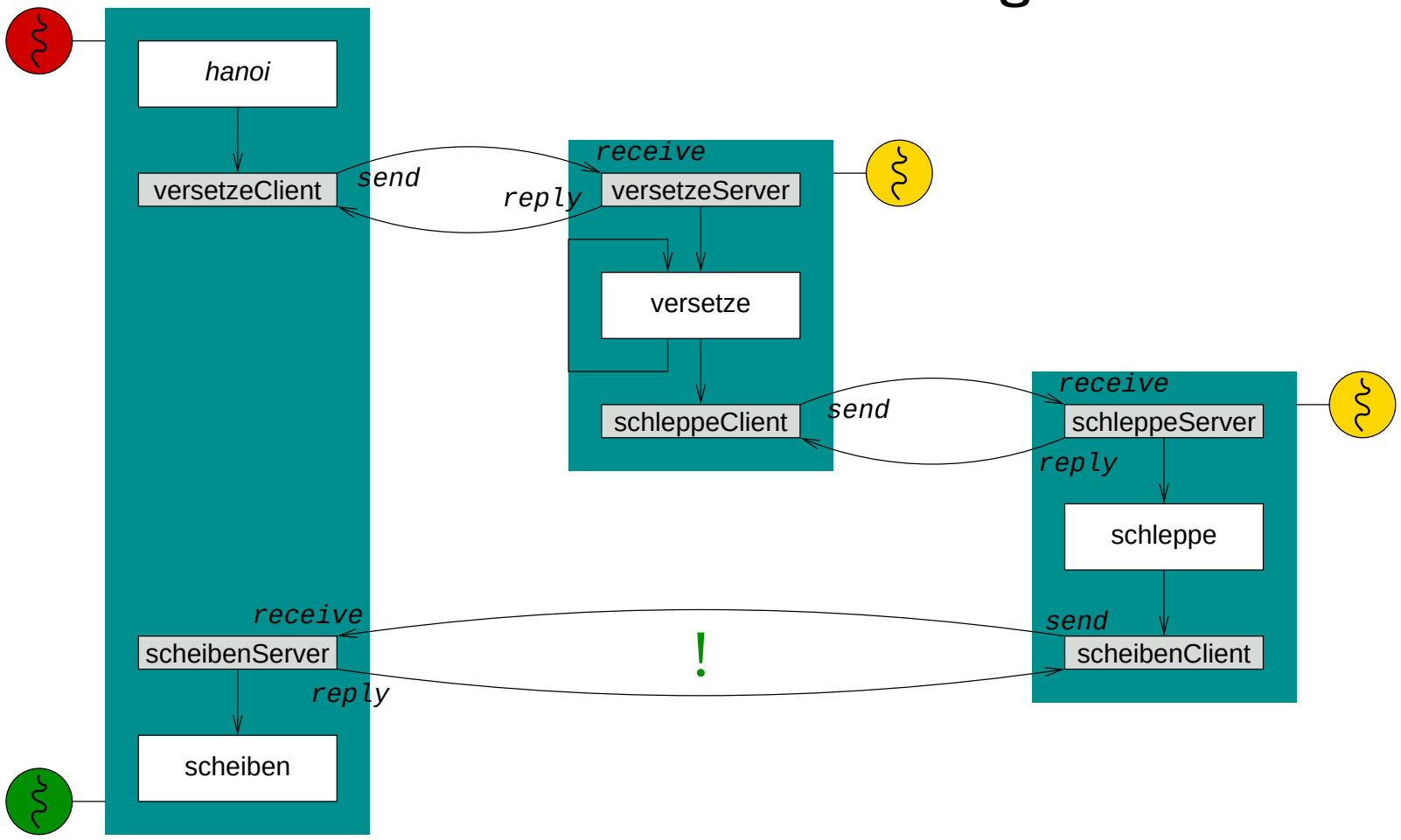
- verklemmungsfreie Fernrückrufe sind auf zwei Arten realisierbar:

  Zustandsmaschine (*state machine*) −
  - Synchronität durch *synchronization send* oder *no-wait send* „lockern“
  - nach dem Senden, mit *receive* auf Antwort oder Rückruf warten
  - der Faden muss sich zwischen verschiedenen Arbeitsphasen multiplexen

  Rückrufanbieter (*back-call server*) +
  - *remote-invocation send*-Modell als Fernaufrufgrundlage beibehalten
  - zur Rückrufverarbeitung einen eigenen (speziellen) Faden vorsehen
  - den Rückruf faktisch als ganz „normale“ Dienstleistung verstehen

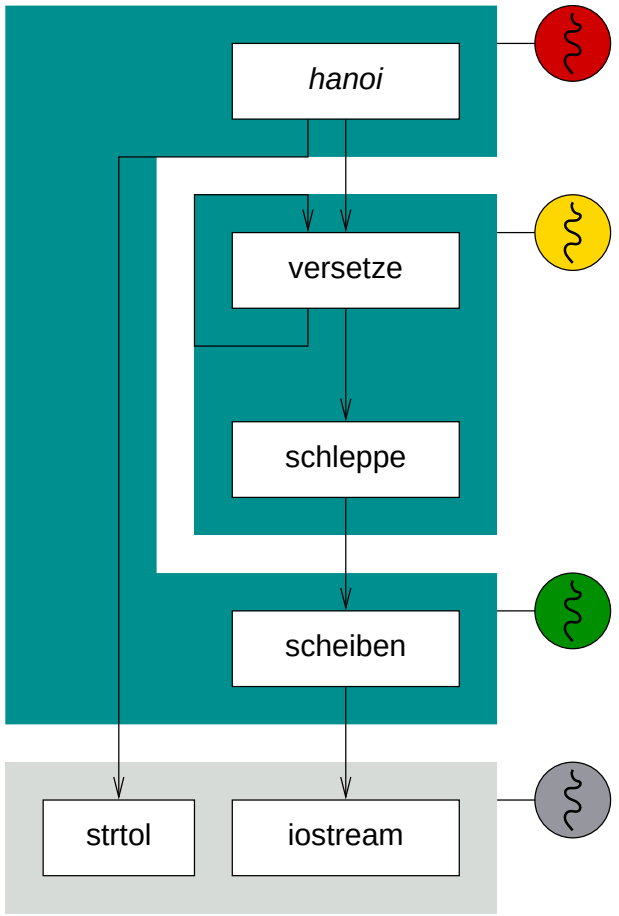
- letztere (orthogonale) Lösung ist strukturfördernd und weniger fehleranfällig

Verklemmungsfreier Fernrückruf



Dienstanbieter ($Server$) „Betriebssystem“

- Systemaufrufe als Prozedurfernaufrufe
  - ein *system-call trap* bewirkt Kontextwechsel
    - Arbeitsmode und ggf. Adressraum
  - der Prozesswechsel ist sinnvolle „Zugabe“
- Betriebssystemdienste könnten verteilt werden
  - $\{$*device*, *file*, *memory*, *thread*, . . . $\}$ *server*
- ein **Mikrokern** realisiert nur noch IPC
  - *remote-invocation send* zwischen *threads*



Dienstanbieter

```
namespace RPC {
  enum hanoiRPC {VERSETZE, SCHLEPPE};

  class verseppeMessage : private ipcMessage {
    hanoiRPC what;
    int quantity;
#define slice quantity;
    char from, to, via;
    friend void verseppeServer ();
  public:
    void versetze (int n, char from, char to, char via) {
      what = VERSETZE;
      ...
      send("verseppeServer");
    }

    void schleppe (int n, char from, char to) {
      what = SCHLEPPE;
      ...
      send("verseppeServer");
    }
  };
```

Turm versetzen/Scheiben schleppen

```
  void verseppeServer () {
    verseppeMessage msg;
    for (;;) {
      int who = msg.receive();
      switch (msg.what) {
      case VERSETZE:
        ::versetze(...);
        break;
      case SCHLEPPE:
        ::schleppe(...);
        break;
      }
      msg.reply(who);
    }
  }
}/* RPC */;
```

Konventioneller Aufruf vs. Fernaufruf

Ziel eines Fernaufrufmechanismus ist es, die bekannte Semantik konventioneller Prozeduraufrufe aufrecht zu erhalten, obwohl sich die Ausführungsumgebung radikal anders gestaltet:

- aufrufende und aufgerufene Seite sind örtlich voneinander getrennt
  - Kode und Daten teilen nicht denselben (physikalischen) Arbeitsspeicher
- beide Seiten arbeiten weitestgehend autonom
  - sie werden von verschiedenen Prozessen/Prozessoren ausgeführt
- beide Seiten können unabhängig voneinander ausfallen

Die Semantik konventioneller, lokaler Prozeduraufrufe ist nur teilweise erreichbar

Semantikaspekte (1)

Parameterarten sind zu unterscheiden, um Mehraufwand zu minimieren

- Eingabe- vs. Ausgabe- vs. Ein-/Ausgabeparameter

Parameterübergabe ist ggf. explizit zu spezifizieren

- *call-by-reference* vs. *call-by-value/result*

Gültigkeitsbereiche schränken sich ggf. massiv ein

- Variablen des fernen umfassenden *scopes* sind nicht zwingend gültig

Speicheradressen sind im Regelfall systemweit uneindeutig

- Zeiger in Nachrichten zu versenden, ist (nahezu) sinnlos

Semantikaspekte (1.1) — Parameterarten

- die Auslegung der (formalen) Parameter bestimmt u.a. den IPC Mehraufwand:

  Eingabeparameter sind nur Bestandteil der *Aufforderungsnachricht*
  Ausgabeparameter sind nur Bestandteil der *Antwortnachricht*
  Ein-/Ausgabeparameter sind Bestandteil beider Nachrichten

- nicht immer liefern Programmiersprachen passende Auslegungshinweise, z.B.

  C/C++ { *Zeiger* `char*`, `struct Foo*` ; *Referenz* `struct Foo&` ; *Feld* `char foo[4]` } welche Parameterart?

- die Art eines jeden Parameters müsste in der Schnittstelle spezifiziert sein

Semantikaspekte (1.2) — Parameterübergabe

call-by-value/result sind „geradlinig“ und einfach zu behandeln

- die aktuellen Parameter werden in die/aus den jeweiligen Nachrichten kopiert

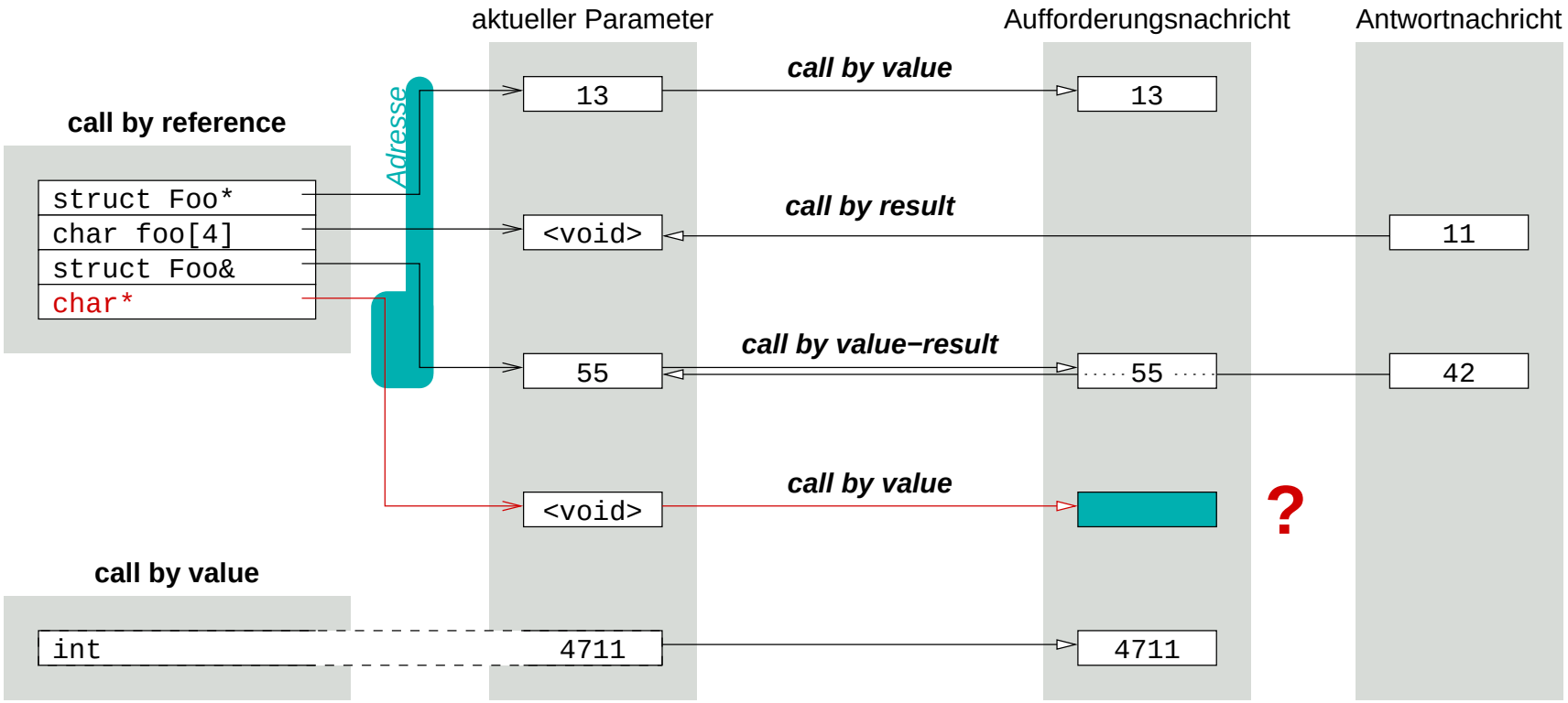
call-by-reference wird je nach Parameterart abgebildet wie folgt:

Einabeparameter	$\longrightarrow$	*call-by-value*	dereferenzieren
Ausgabeparameter	$\longrightarrow$	*call-by-result*	dereferenzieren
Ein-/Ausgabeparameter	$\longrightarrow$	*call-by-value-result*	dereferenzieren
unspezifiziert	$\longrightarrow$	*call-by-value*	Speicheradresse

call-by-name ggf. nur auf Basis von *function shipping* –

- eine Funktion wird mitgeliefert, die den aktuellen Parameter berechnet

call-by-reference vs. *call-by-value*



Semantikaspekte (1.3) — Gültigkeitsbereiche

- jeder Name (einer Variablen) ist mit einem Platzhalter assoziiert
  - die Variable belegt einen Speicherplatz an einer bestimmten Adresse
  - den Namen/Platzhaltern sind Gültigkeitsbereiche („Blöcke“) zugeordnet

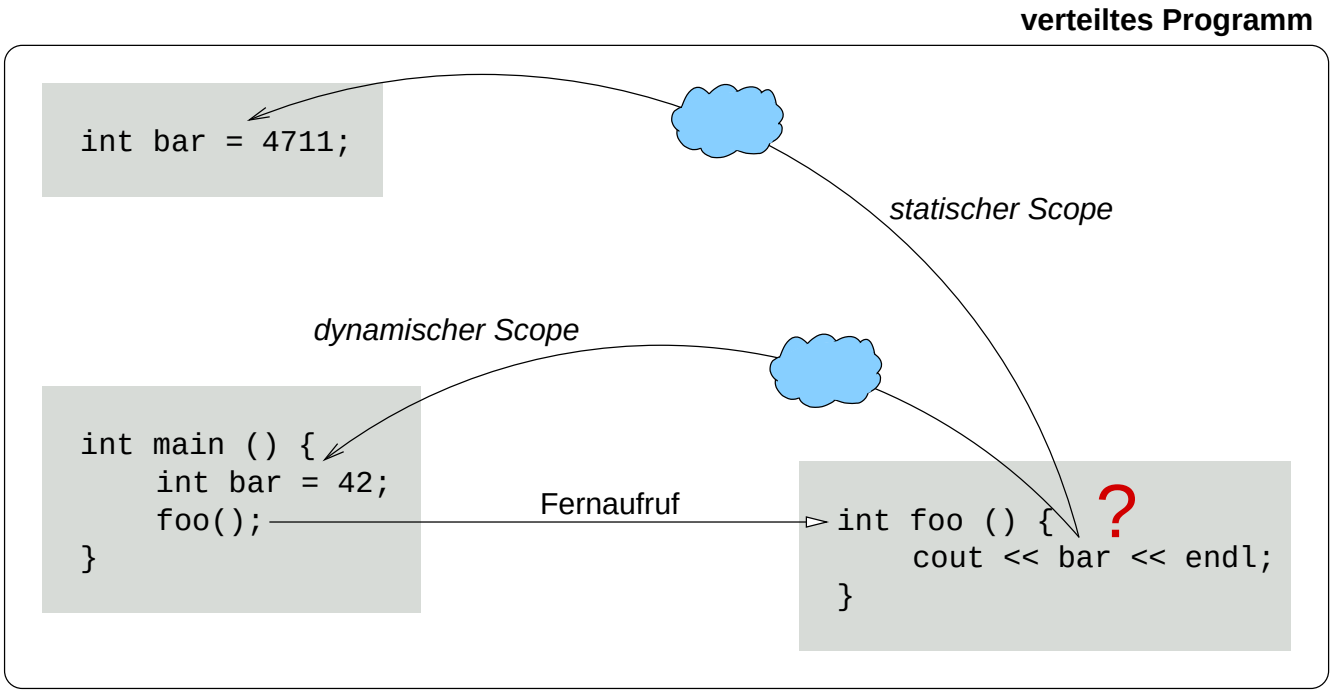
- die Bindung eines Namens an seinen Block (*Scope*) ist statisch oder dynamisch:

  statischer Scope ist bereits zur *Übersetzungszeit* bekannt
  - ändert sich nur bei Quelltextänderungen am Programm

  dynamischer Scope ist erst zur *Laufzeit* bekannt
  - ändert sich mit Eintritt in/Verlassen von Prozeduren bzw. Funktionen

- der „umfassende Block“ ist für eine ferne Prozedur nur bedingt zugänglich

Gültigkeitsbereiche von Programmvariablen



C/C++ bindet statisch, wie die meisten anderen Programmiersprachen auch. Das löst das Problem aber nicht.

Semantikaspekte (1.4) — Speicheradressen

- Programmadressen sind (im Regelfall) nicht systemweit eindeutig:
  - die fragliche Adresse kann beim Dienstanbieter bereits vergeben sein
  - nur im *single programm, multiple data* (SPMD) Modell wäre sie eindeutig[4]
  - im „Normalfall“ ist die Eindeutigkeit einer solchen Adresse eher zufällig

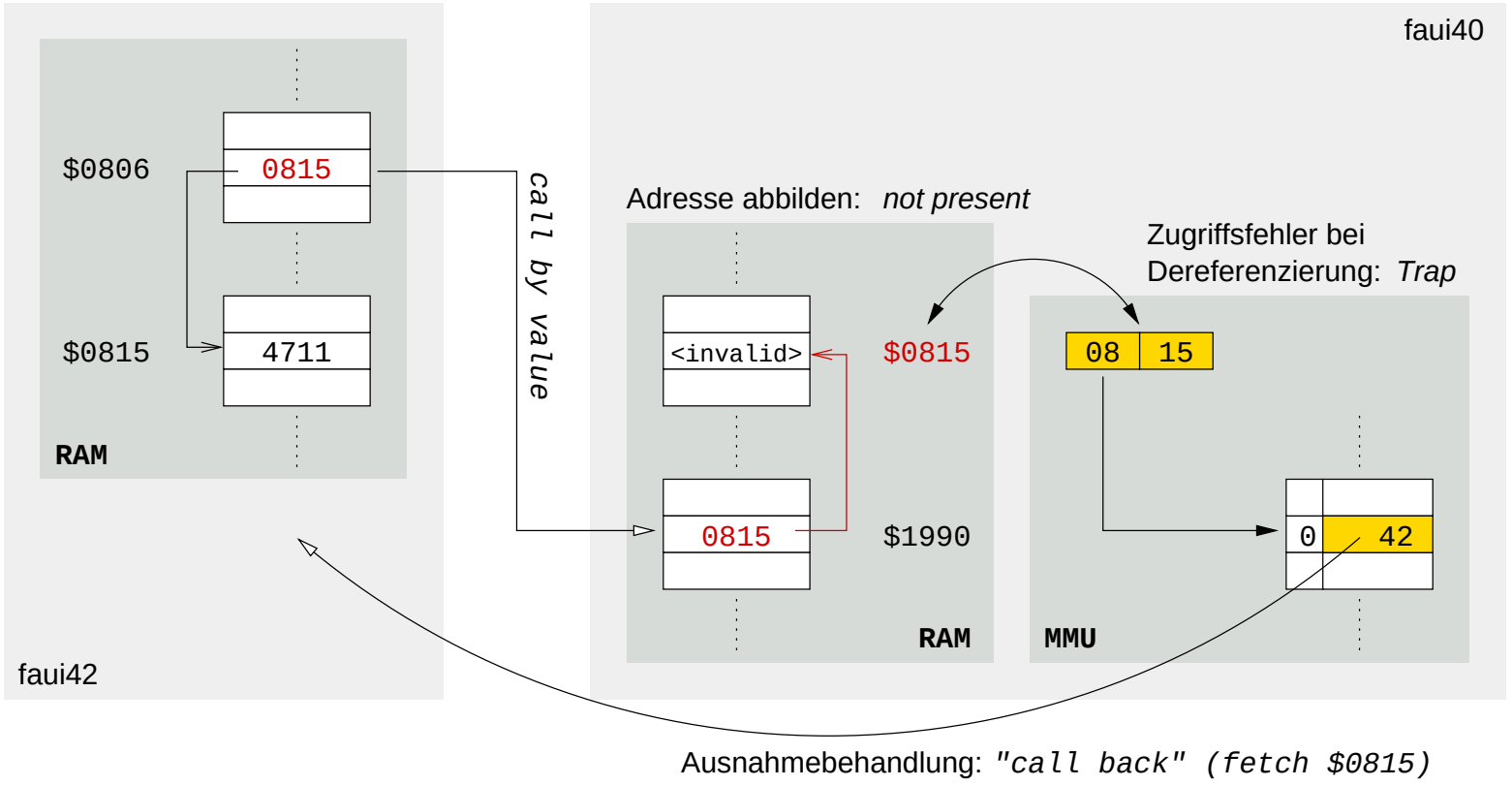
- allgemein sind Adressen abhängig von dem Kontext, in dem sie definiert sind
  - sie beziehen sich auf ein Programm in einem Adressraum auf einem Rechner
  - Adressen verteilter Programme besitzen eine **geographische Komponente**

- ferne Adressen entsprechen logischen Adressen — sie sind (ggf.) abbildbar

[4] In dem Fall liegt auf den Rechnern des verteilten Systems dasselbe Programm. Demzufolge sind auf allen betrachteten Rechnern die Adressen der Variablen aber auch der Prozeduren/Funktionen identisch.

Ferne Adresse

Logische Adresse (1)



Ferne Adresse — Logische Adresse (2)

- entscheidend für die Abbildung ist das jeweilige *Adressraummodell*:

  eindimensionaler Adressraum gekachelter Speicher *paging*
  - \$0815 muss im (logischen) Adressraum des Dienstanbieters frei sein

  zweidimensionaler Adressraum segmentierter Speicher *segmentation*
  - \$0815 ist ein Versatz (*offset*) innerhalb eines Segments
  - mit Empfang der Adresse wird ein neues Segment angelegt
  - in dem neuen Segment ist die Adresse \$0815 definiert d.h. lokal eindeutig[5]
  - der Adressraum des Dienstanbieters verändert sich je nach Fernaufruf

- der Aufwand ist beträchtlich und setzt entsprechende Unterstützung voraus

[5] Je nach Zustand des Segmentdeskriptors wird beim Zugriff ein Fehler (*trap*) ausgelöst oder nicht. Die Ausnahmebehandlung führt einen Rückruf zur (fernen) Dereferenzierung aus, lagert den gelesenen Wert lokal ein und ändert ggf. den Deskriptorzustand, so dass zukünftige Zugriffe keinen derartigen Fehler mehr liefern.

Ferne Adresse — *Smart Pointer* (1)

- eine ferne Adresse als zweiteiligen (getypten) „geschickten Zeiger“ auslegen:

  Kontextbezeichner systemweit eindeutig, z.B. eine Prozessidentifikation
  - durch *Rückruf* wird das referenzierte ferne Objekt herangeholt
  - ein Platzhalter zur Aufnahme der lokalen Kopie wird dynamisch angelegt
  - die Adresse des Platzhalters wird als Zeigerkomponente übernommen

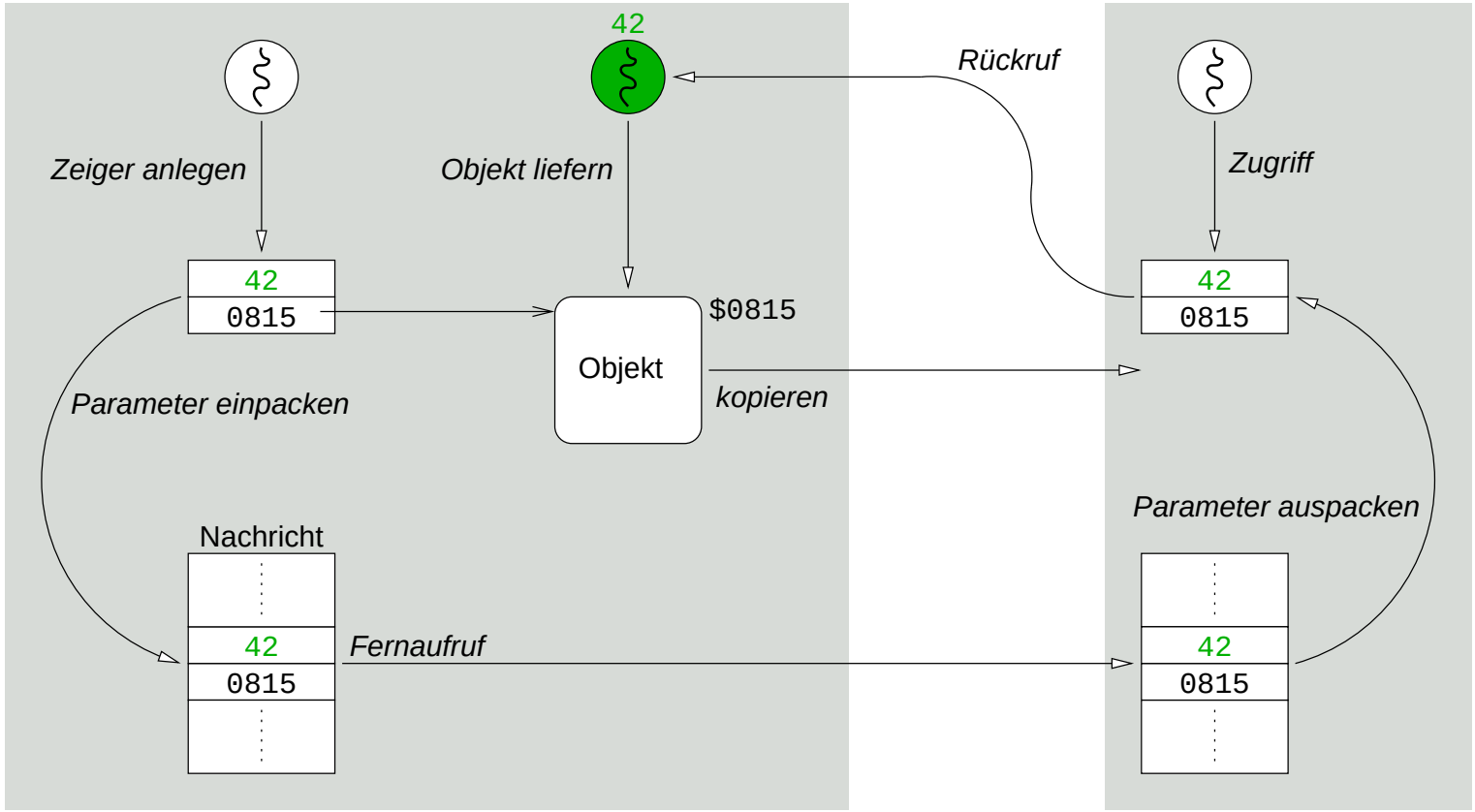
  Zeiger Adresse auf (a) das ferne Objekt, (b) den lokalen Platzhalter

- ohne linguistische Unterstützung ist die Technik wenig praktikabel: C++
  - ferne Adressen z.B. als Instanzen von Klassenschablonen realisieren
  - überlagerte Operatoren setzen bei Bedarf Rückrufe ab und regeln den Zugriff

- ggf. sind lokale Kopien nicht zwingend und alle Zugriffe können Rückrufe sein

Ferne Adresse

Smart Pointer (2)



Ferne Adresse als C++ *Smart Pointer*

```
template<typename CEI, typename T>
class smartPointer {
  CEI home;           // connection endpoint identifier
  T*  item;           // local address at home
  void check () { ... }
public:
  smartPointer (...);

  T& operator [] (int i) { check(); return item[i]; }
  T* operator -> ()      { check(); return item; }
  T* operator =  (T* sp) { check(); return item = sp; }
     operator T* ()      { check(); return item; }
};
```

```
#include "assert.h"
#include "smartMessage.h"

void check () {
  if (home) {
    T* copy = new T;
    assert(copy != 0);
    smartMessage().fetch(home, item, *copy);
    item = copy;
    home = 0;
  }
}
```

Nachrichten zusammenstellen/auseinandernehmen

marshalling *to arrange (troops, things, ideas, etc.) in order; array; dispose*

- die „Datenposten“ in einen Nachrichtenpuffer gepackt anordnen:
  1. für die **Serialisierung** verstreut vorliegender Daten sorgen
  2. die vereinbarte **Repräsentation** der Daten gewährleisten
- je nach Herangehensweise sind Referenzen aufzulösen und umzuwandeln
  – wenn die referenzierten Strukturen *call-by-value* zu übertragen sind
- die so zusammengestellte Nachricht geht per IPC an den Empfangsprozess

unmarshalling die *inverse Funktion* auf Empfangsseite

- Nachricht entpacken und die urpsrünglichen „Datenposten“ wieder herleiten

Serialisierung/Deserialisierung

- geläufig ist, **verzeigerte Datenstrukturen** *call-by-value* (*-result*) zu übergeben
  - beschrieben wird der logische Aufbau strukturierter/dynamischer Daten
  - alle Referenzen werden aufgelöst und die referenzierten Objekte kopiert
  - nach dem Empfang werden die Datenstrukturen wieder rekonstruiert

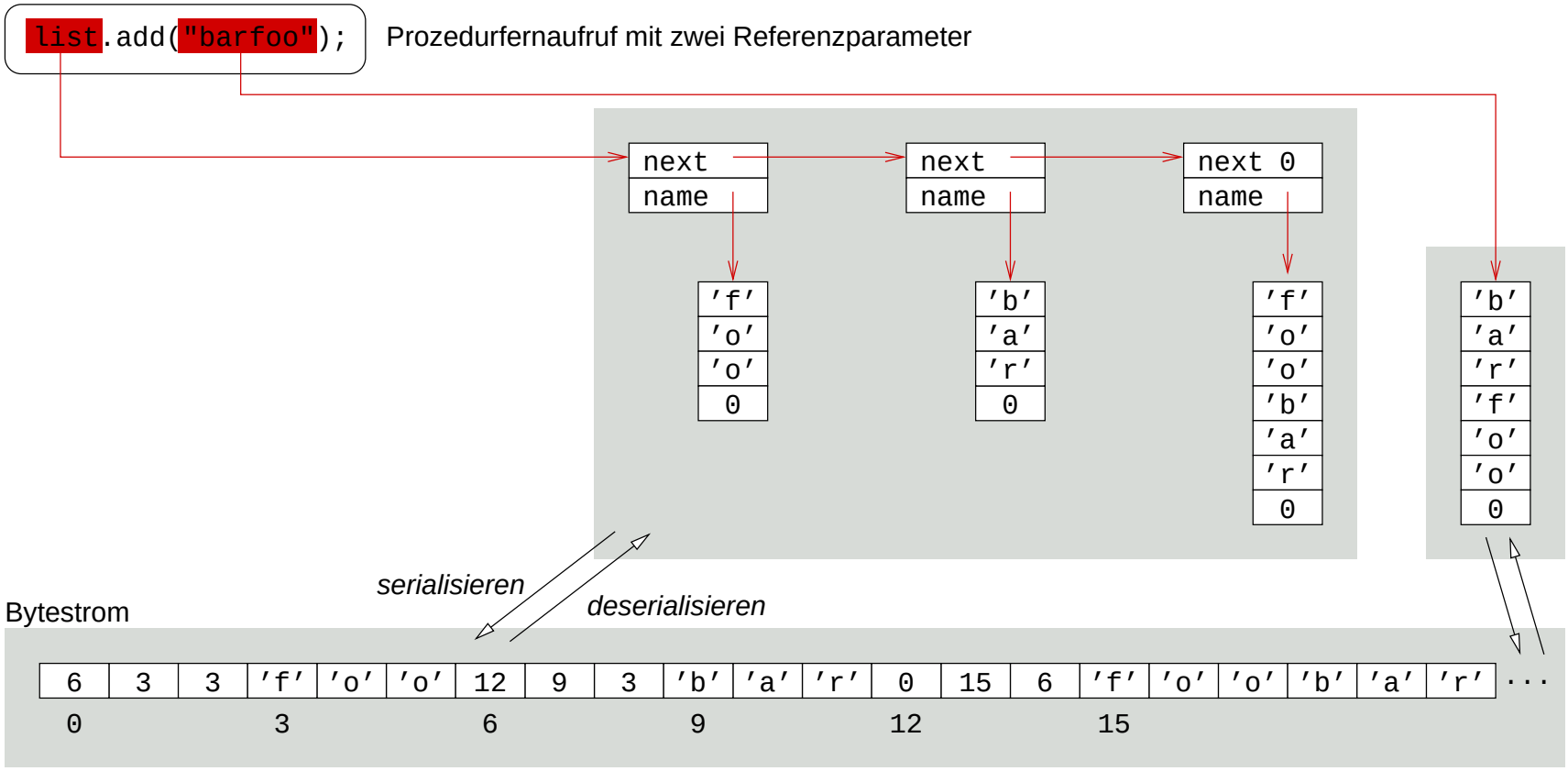
- zwei Sorten von Zeigern sind dabei zu unterscheiden:

  innere Referenzen die Verkettungszeiger rekursiver Datenstrukturen
  - sind vergleichsweise unproblematisch: „logische Zeiger“ vergeben

  äußere Referenzen Zeiger von außen hinein auf einzelne Verbundelemente
  - die relative Position der Verbundelemente kann sich ändern: Relokation

- der Aufwand kann je nach Art/Aufbau der Datenstrukturen beträchtlich sein

Packen/Entpacken strukturierter Daten



Objektorientierung „erschwert“ Automatisierung

- Referenzen auf Oberklassen sind ggf. von den Unterklassen zu behandeln
  - eine *abstrakte Basisklasse* sagt wenig/nichts aus über die Objektstruktur
    - in der Fernaufrufschnittstelle wäre die konkrete Ausprägung unbekannt
  - die spezialisierende(n) Unterklasse(n) bring{t,en} erst Strukturwissen ein

- der *effektive Typ* des aktuellen Parameters ist erst zur Laufzeit bekannt
  - automatische Generierung des Stumpfes zur Übersetzungszeit scheidet aus
  - dies gilt zumindest dann, wenn es heißt, Zugriffstransparenz zu wahren
  - üblich ist die „spezialisierte Zusammenstellung“ der Fernaufrufnachrichten

- nur eine vollständige Analyse des (zu verteilenden) Quellprogramms hilft

Abstrakte Klassen als Fernaufruf-„Handicap“ (1)

```
class listElement {
public:
  virtual listElement* next () const = 0;
  virtual void bind (listElement*)   = 0;
};

class listDescriptor {
  listElement* head;
public:
  void add (listElement& item) {
    if (!head) head = &item;
    else {
      listElement* tail = head;
      while (tail->next()) tail = tail->next();
      tail->bind(&item);
    }
  }
};
```

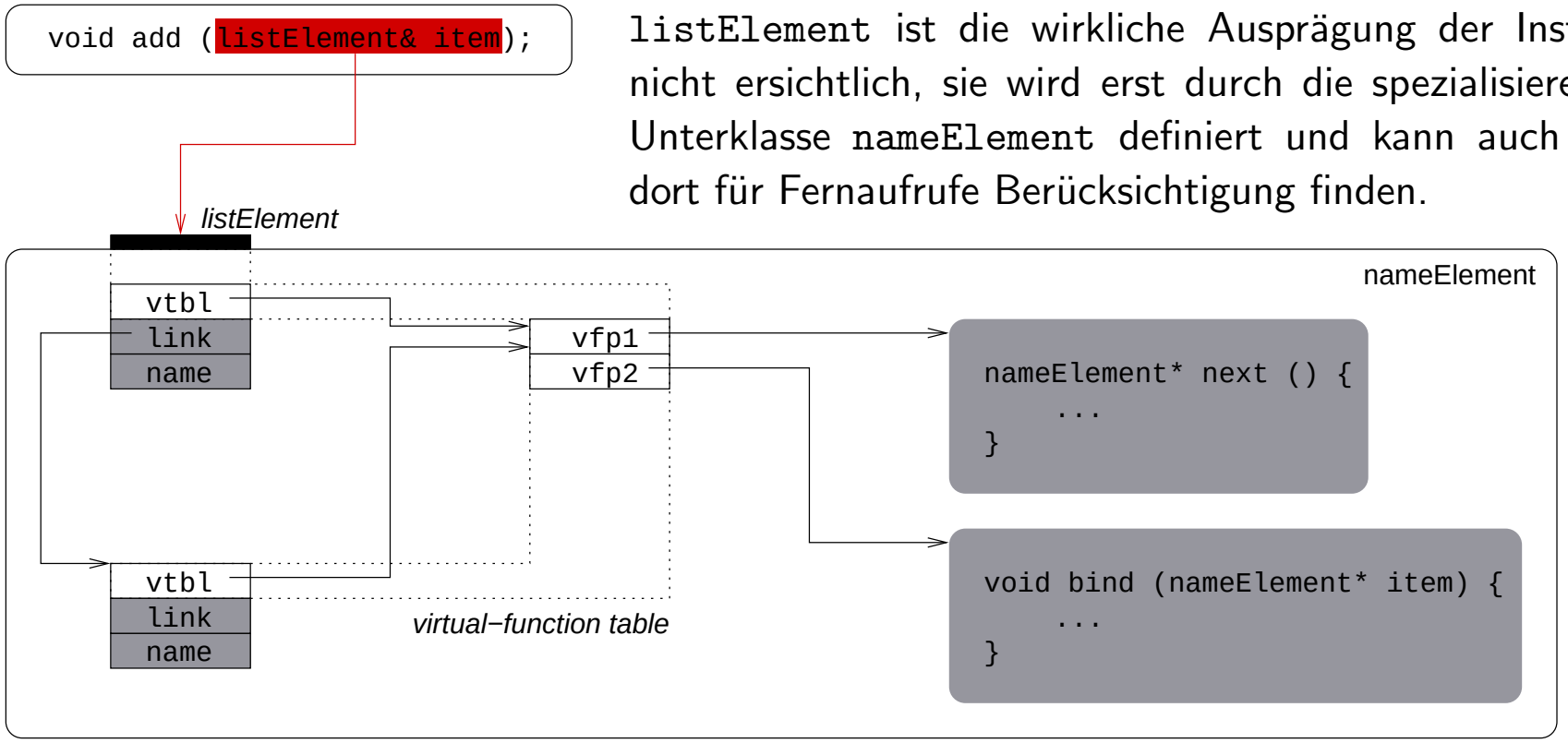
```
class nameElement : public listElement {
  nameElement* link;
  char*        name;
public:
  nameElement* next () const {
    return link;
  }

  void bind (nameElement* item) {
    link = item;
  }
};

listDescriptor list;

void foo (listElement& item) {
  list.add(item);
}
```

Abstrakte Klassen als Fernaufruf-„Handicap“ (2)



In der abstrakten Schnittstelle der Oberklasse `listElement` ist die wirkliche Ausprägung der Instanz nicht ersichtlich, sie wird erst durch die spezialisierende Unterklasse `nameElement` definiert und kann auch nur dort für Fernaufrufe Berücksichtigung finden.

Datenrepräsentation

Heterogenität Die einzelnen in Nachrichten übertragenen Elemente können Werte unterschiedlichster (elementarer) Datentypen repräsentieren. *Speicherung* wie auch *Darstellung* von Instanzen dieser Typen ist nicht in allen Rechnern identisch:

- natürliche/ganze Zahlen: vorzeichenbehaftet, {1,2}er-Komplement, Excess
- Fließkommazahlen: Basis, Mantisse, Exponent
- Zeichensätze: ISO-8859-Familie (ASCII), BCD, EBCDIC, Unicode
- Speicherreihenfolge: *big endian* vs. *little endian*

$\Longrightarrow$ Daten sind ggf. zu konvertieren, damit kooperierende Prozesse funktionieren!

Konvertierungsoptionen

beidseitig *external data representation* (XDR)

- zu sendende Daten in eine **kanonische Darstellung** umkodieren *und*
- empfangene („kanonische“) Daten in die lokale Darstellung umkodieren
- Problem: nutzloser Mehraufwand im Falle „gleichartiger“ Rechner

sendeseitig „*sender makes it right*“

- zu sendende Daten in die empfangsseitige Darstellung ggf. umkodieren
- Problem: Mehrteilnehmerkommunikation, Weiterleiten von Nachrichten

empfangsseitig „*receiver makes it right*“

- empfangene Daten in die lokale Darstellung ggf. umkodieren → *endian tag*

External Data Representation — XDR

- sprachbasierter Standard zur Beschreibung und Kodierung von Daten [7]
  - der ISO/OSI **Präsentationsschicht** (*presentation layer*, 6) zugeordnet
  - **implizite Typung** (*implicit typing*), nicht explizit wie bei ASN.1 [1]
  - Annahme: Bytes bzw. Oktets (d.h. Einheiten von 8 Bits) sind portabel

- Daten werden als „Vielfaches von vier Bytes“ (32 Bits) repräsentiert[6]
  - beim Lesen/Schreiben eines Bytestroms kommt Byte m immer vor $m + 1$
  - Füllbytes (0 – 3) ergänzen den Datenstrom immer zum Vielfachen von 4

- die Reihenfolge (der „Byte-Sex“) ist *big endian*

[6] *Tradeoff* „Vier“ — Groß genug als effiziente Lösung für die meisten Maschinen, mit Ausnahme von 64-Bit Architekturen, und klein genug als vertretbarer Mehraufwand zur Repräsentation der kodierten Daten.

XDR (1)

```
enum hanoiRPC {
  VERSETZE = 0,
  SCHLEPPE = 1
};

union hanoiPile switch (hanoiRPC kind) {
case VERSETZE:
  unsigned int aux;
case SCHLEPPE:
  void;
};

struct hanoiTowers {
  unsigned int from;
  unsigned int to;
  hanoiPile via;
};

struct verseppeMessage {
  unsigned int slices;
  hanoiTowers towers;
};
```

Ungepackte Nachricht

Versatz	Hex	ASCII	Kommentar
0	00 00 00 04	....	Scheibenanzahl
4	00 00 00 41	...A	von Turm
8	00 00 00 42	...B	nach Turm
12	00 00 00 00	....	VERSETZE
16	00 00 00 43	...C	über Turm

XDR (2)

Gepackte Nachrichten

```
...
union hanoiTowers switch (hanoiRPC kind)
case VERSETZE:
  string fromtovia<3>;
case SCHLEPPE:
  string fromto<2>;
;
...
```

Versatz	Hex	ASCII	Kommentar
0	00 00 00 04	....	Scheibenanzahl
4	00 00 00 00	....	VERSETZE
8	00 00 00 03	....	Länge
12	41 42 43 00	ABC.	Türme

```
...
union hanoiTowers switch (hanoiRPC kind)
case VERSETZE:
  opaque fromtovia<3>;
case SCHLEPPE:
  opaque fromto<2>;
;
...
```

Versatz	Hex	ASCII	Kommentar
0	00 00 00 04	....	Scheibenanzahl
4	00 00 00 00	....	VERSETZE
8	41 42 43 00	ABC.	Türme

XDR „*Considered Harmful*“?

Virtualisierung Jede Maschine, deren physikalische Darstellung mit der durch XDR vorgegebenen virtuellen übereinstimmt, wird effizienter arbeiten als jene, bei der die physikalische von der virtuellen Darstellungsform stark abweicht:

- Speicherreihenfolge („Byte-Sex“)

	endian		endian		
Sun	*big*	$\leftrightarrow$	*big*	m68K	„optimal“
IBM 370	*big*	$\leftrightarrow$	*little*	Z80	
Alpha	*little*	$\leftrightarrow$	*big*	R10000	
VAX	*little*	$\leftrightarrow$	*little*	x86	„suboptimal“

- Verschnitt („interne Fragmentierung“)

Sprachintegration von Fernaufrufen

- Fernaufrufsysteme lassen sich in zwei Hauptkategorien einteilen:
  1. in einer Programmierspache **integriertes Konzept** . . . . . Argus
     - internes Wissen über Datentyp- und Laufzeitmodell ist verfügbar
     - der Übersetzer agiert ebenfalls als Stumpfgenerator (*stub generator*)
     - umfassende Analysen, Optimierungen und Automatismen werden möglich
  2. von einer Programmierspache **separiertes Konzept** . . . . . CORBA
     - das Schnittstellenverhalten ferner Prozeduren ist explizit zu beschreiben
       - Programmannotationen, „*Interface Definition Language*“ (IDL)
     - fehlendes internes Wissen schränkt Analysen, Optimierungen etc. ein

- nur das integrierte Konzept ($\rightarrow$ 1.) definiert eine einheitliche Semantik

Entlastung von Routineaufgaben

- die Stümpfe sorgen für eine **lose Kopplung** zweier Ausführungsumgebungen:

  client stub den Ort des Dienstabieters (beim Namensdienst) erfragen
  1. *Marshalling* und versenden der Aufforderungsnachricht
  ⋮ die Durchführung der angeforderten Operation abwarten
  2. empfangen und *Unmarshalling* der Antwortnachricht

  server stub den Ort des Dienstabieters (dem Namensdienst) angeben
  1. empfangen und *Unmarshalling* der Aufforderungsnachricht
  2. durchführen der angeforderten Operation (lokaler Aufruf)
  3. *Marshalling* und versenden der Antwortnachricht

- **Stumpfgeneratoren** erzeugen die dafür notwendigen Programmsequenzen

Transparenz von Fernaufrufen

- Verteilungstransparenz ist nur bedingt erreichbar, trotz integriertem Konzept:
  - syntaktische Unterschiede (in der Schnittstelle) werden vermieden
  - Parameterübergabe, *Marshalling* und *Unmarshalling* wird verborgen
  - Nachrichtenpuffer und Fäden werden erzeugt, verwaltet und entsorgt
  - Stümpfe werden automatisch erzeugt — was ist denn sonst noch nötig?

- Fernaufrufe sind fehleranfälliger als konventionelle lokale Prozeduraufrufe

  fern ein Netzwerk, ein anderer Rechner und ein anderer Prozess
  ¬ **fern** kein Netzwerk, kein anderer Rechner und kein anderer Prozess

- verteilte Systeme unterliegen einem radikal anderem **Fehlermodell**

Zustellungsgarantien

- *request-reply*-Protokolle eröffnen Optionen für Fehlertoleranzmaßnahmen:
  1. Wiederholung der Aufforderungsnachricht . . . . . *request retry*
     – bis die Antwort eintrifft oder ein Anbieterausfall angenommen wird
  2. Filterung der Aufforderungsduplikate . . . . . *duplicate suppression*
     – wenn die Aufforderung empfangen wurde und noch in Arbeit ist
  3. Wiederholung der Antwortnachricht . . . . . *reply retry*
     – bis die nächste Aufforderung oder eine Bestätigung eintrifft

- Kombinationen dieser Optionen begründen verschiedene Aufrufsemantiken

Fehlertoleranzmaßnahme vs. Aufrufsemantik

Option			Semantik	
request retry	*dupl. supr.*	*re-execute reply retry*		
Nein	—	—	„kann sein“	*maybe*
Ja	Nein	*re-execute*	„wenigstens einmal“	*at-least-once*
Ja	Ja	*reply retry*	„höchstens einmal“	*at-most-once*

Aufrufsemantiken (1)

maybe kann sein, der Aufruf wurde ausgeführt oder auch nicht

- der Klient hat im Fehlerfall keine Gewissheit über die korrekte Ausführung

at-least-once mindestens einmal, die mehrfache Ausführung ist möglich

- der Klient erwartet die Antwort innerhalb einer vorgegebenen Zeitspanne
  - jeder Fernaufruf wird mit einem *Timeout* versehen
  - nach der dadurch definierten Pause erfolgt die Aufrufwiederholung
  - die Anzahl der Wiederholungen ist (üblicherweise) begrenzt
  - mit Erreichen der max. Anzahl tritt eine *Ausnahmesitutation* ein
- der Anbieter muss **idempotente Operationen** exportieren

Aufrufsemantiken (2)

at-most-once höchstens einmal, mehrfache Ausführung ist ausgeschlossen

- vergleichsweise aufwendiges, speicherintensives Verfahren:
  - neuen Aufrufen wird eine eindeutige *Aufrufkennung* gegeben
  - Wiederholungen kommen mit derselben Kennung und werden verworfen
    - Möglichkeit der Ausnahmesituation entsprechend *at-least-once*
  - Antworten werden gespeichert und bei Wiederholungen zurückgeliefert
  - ein neuer Aufruf ist Anlass, gespeicherte Antworten zu entsorgen[7]
- die Ausführung erfolgt nur, wenn keine Rechnerausfälle vorliegen

[7] Ansonsten muss der Anbieter hin und wieder beim Klienten nachfragen, ob dieser noch „lebt“, um so ein Kriterium für die Entsorgung gespeicherter Antwortnachrichten zu gewinnen.

Aufrufsemantiken (3)

last-of-many akzeptiert nur die Antwort zum jüngst zurückliegenden Aufruf

- jeder Aufruf, auch der wiederholte, erhält eine eindeutige Kennung
- jede Antwort trägt die Kennung des zugehörigen Aufrufs
  - der Klient verwirft Antworten mit nicht mehr aktueller Aufrufkennung
  - eine Variante von *at-least-once*, falls keine Idempotenz vorliegt
- die wiederholte Ausführung der Operationen ist ggf. weiterhin problematisch
- zumindest können die Klienten aber mit „aktuellen“ Daten weiterarbeiten[8]

[8] Im Gegensatz dazu liefert *at-most-once* ggf. „veraltete“ Daten, die zur ersten Ausführung wiederholt abgesetzter Aufrufe korrespondieren. Dieser Fall kann vorliegen, wenn die vom Anbieter verwalteten Daten von mehreren Klienten be- und verarbeitet werden (*information sharing*).

Aufrufsemantiken (4)

exactly-once genau einmal, entspricht der Semantik lokaler Aufrufe

- erfordert Transaktionskonzepte mit Wiederanlauf von Komponenten
  – gibt damit (eine gewisse) Garantie bei Systemfehlern bzw. -ausfällen
- nach Wiederanlauf sind Aussagen zur Operationsdurchführung „unscharf“:

  Imagine, for example, a chocolate factory, in which vats of liquid chocolate are filled by having a computer set a bit in some device register to open a valve. After recovering from a crash, there is no way for the chocolate server to see if the crash happened one microsecond before or after the bit was set. (Tanenbaum, *Computer Networks*, 1989)

- wünschenswerte, ideale Semantik, die in „Reinform“ jedoch unerreichbar ist

Idempotente Operationen

Idempotenz „idem“ (*lat.*) dasselbe; wenn die wiederholte Ausführung derselben Operation (mit denselben Parameterwerten) durch den Dienstanbieter immer den Effekt einer einmaligen Ausführung besitzt

- wiederholte Ausführungen sind möglich im Falle ungefilterter Duplikate
  - kritisch sind z.B. Schreiboperationen auf Dateien (`write(2)`)
  - ebenso Operationen, die eine Folge (Liste) um Elemente erweitern[9]
- ein **zustandsfreier Dienstanbieter** (*stateless server*) ist gefordert
  - bekannter Vertreter ist das Sun *Network File System* (NFS)
- bei zustandsbehafteten Dienstanbietern ist Duplikatelimination notwendig

[9] Im Gegensatz zu Operationen, die auf Mengen arbeiten.

Fehlersemantiken

Aufrufsemantik	**Fehlerart**							
	fehlerfreier Ablauf		Verlust von Nachrichten		zusätzlicher Ausfall von Anbieter		Klient	
	Aus.	Erg.	Aus.	Erg.	Aus.	Erg.	Aus.	Erg.
maybe	1	1	0/1	0/1	0/1	0/1	0/1	0/1
at-least-once	1	1	≥ 1	≥ 1	≥ 0	≥ 0	≥ 0	0
at-most-once	1	1	1	1	0/1	0/1	0/1	0
exactly-once	1	1	1	1	1	1	1	1

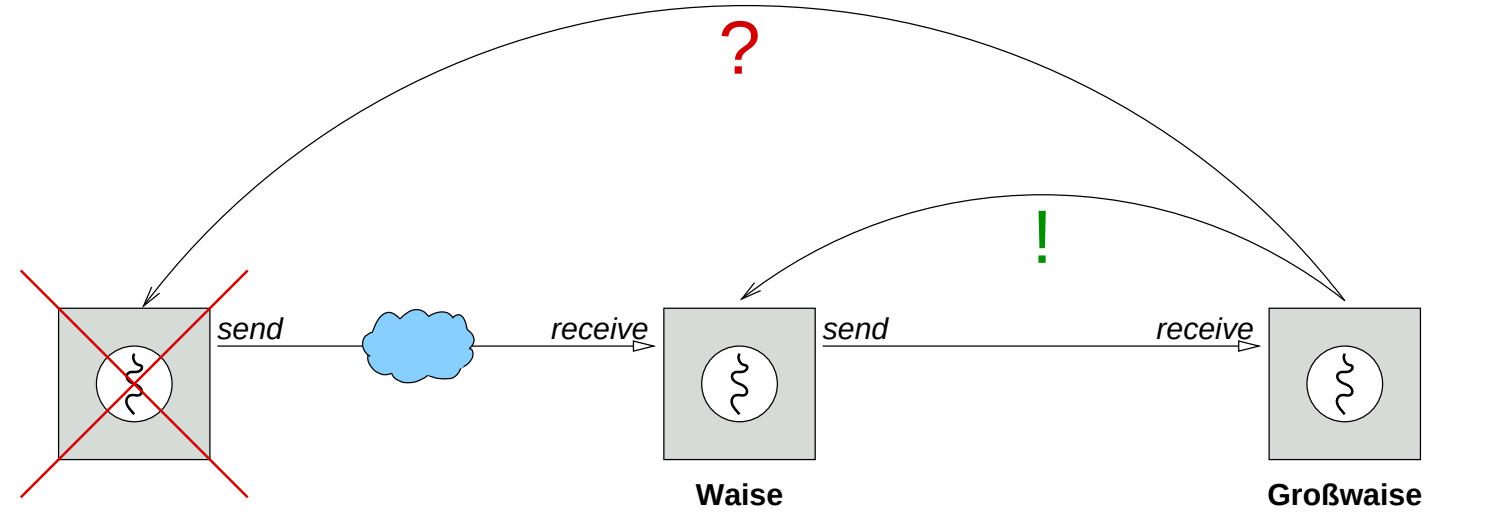
Aus.= Ausführung, Erg.= Ergebnis; die jeweilige Anzahl ist angegeben

Verwaister Aufruf (1)

Orphan ein Klient, der den Aufruf (z.B. wegen eines ggf. zu kurzen *Timeout*) abgebrochen hat und nicht mehr am Ergebnis interessiert ist oder der mittlerweile überhaupt nicht mehr zur Verfügung steht

- Maßnahmen, um unnötige Arbeit des Dienstanbieters zu vermeiden:
  1. zusätzliche Zeitüberwachung des Klienten durch den Dienstanbieter
     - die *Timeout*-Wahl ist klientenabhängig ($T_S \gg T_C$)
  2. pro Klienten einen Ausfallzähler beim Dienstanbieter verwalten
     - Abbruch aller alten Aufrufe nach Ausfall und Wiederanlauf des Klienten
  3. zusätzliche direkte Statusanfragen an den Klienten senden
     - schnelle Waisenerkennung, sofern der Ausfall diagnostizierbar ist
- besondere Schwierigkeiten bereiten „Fernaufrufketten“: **Transitivität**

Verwaister Aufruf (2)



Waisenbehandlung (1)

extermination Vertilgung, Ausrottung, Wegschaffung

- nach Wiederanlauf wird (beim Klienten) auf ausstehende Fernaufrufe geprüft
  - den betreffenden Anbietern gehen sodann Abbruchanforderungen zu
  - rekursives Vorgehen, wegen der Möglichkeit von „Großwaisen“
- Klientenstümpfe müssen „Buch“ führen über alle Fernaufrufe
  - *Log* anlegen vor der Anforderung und zerstören nach Empfang der Antwort

expiration Verscheiden, Ablauf

- der Anbieter gibt dem (Fern-) Aufruf ein *Zeitquantum* zur Ausführung
  - mit Ablauf erbittet der Anbieter ein weiteres Zeitquantum vom Klienten
  - im Fehlerfall bricht der Anbieter den Aufruf ab bzw. terminiert er
- nach Wiederanlauf ist der erste Fernaufruf um ein Zeitquantum zu verzögern

Waisenbehandlung (2)

reincarnation Wiederverkörperung, Wiedergeburt

- scheitert die Ausrottung eines Waisen, wird eine neue *Epoche*[10] eröffnet
  - der Beginn einer Epoche wird allen Maschinen mitgeteilt (*broadcast*)
  - auf den Maschinen werden daraufhin alle Anbieter (Prozesse) terminiert
- jede Anforderungs- und Antwortnachricht enthält eine Epochenkennung
  - damit können unerwartete Antworten von Waisen herausgefiltert werden

gentle reincarnation sanfte –, milde –

- zum Epochenbeginn versucht der Anbieter „seinen“ Klienten zu lokalisieren
- ist der Klient nicht lokalisierbar, wird zur „Donnerbüchse“ (s.o.) gegriffen

[10] Epochen entstehen dadurch, dass die Zeit in sequentiell nummerierte Abschnitte aufgeteilt wird.

Ausnahmebehandlung — *Exception Handling*

- volle Verteilungstransparenz erreichen zu können, ist reine Illusion
  - auch *exactly-once* kann nur die erkennbaren Fehler maskieren
  - in den anderen Fällen ist die Rückkehr vom Aufruf keinesfalls garantiert[11]
- die Stümpfe (auf beiden Seiten) sollten Ausnahmesituationen anzeigen
  - Ausnahmen der Anbieterseite werden als Rückruf zum Klienten propagiert
  - Ausnahmen der Klientenseite werden konventionellerweise „hochgereicht“
- die Anwendungen sollten Maßnahmen zur Fehlertoleranz beinhalten

[11] Es ist der Fernaufrufmechanismus selbst, der einen Klienten z.B. bleibend verklemmen kann. Sicherlich kann auch die Implementierung des per Fernaufruf in Anspruch genommenen Dienstes Verklemmungsursache sein, was aber hier nicht zur Debatte steht.

Zusammenfassung

- Fernaufrufe sind konventionellen Prozeduraufrufen nur ähnlich, nicht gleich:
  - Parameter{arten, übergabe}, Gültigkeitsbereiche und Speicheradressen
  - Fehlermodell, Zustellungsgarantien, Aufrufsemantiken, verwaiste Aufrufe
  - die Latenz entfernter Aufrufe ist um Größenordnungen höher: *Promise*
- Verteilungstransparenz kann nicht wirklich (durchgängig) erzielt werden
  - *exactly-once* ist nicht (bzw. nur mit Abstrichen) zu verwirklichen
  - Ausnahmen sind zu behandeln, die nur in verteilten Systemen auftreten
  - Fernaufrufe machen fehlertolerante Anwendungssoftware keinesfalls obsolet
- der Unterschied zu konventionellen Aufrufen ist (doch) explizit zu machen

Referenzen

[1] ASN.1 Information site. http://asn1.elibel.tm.fr, 2002.

[2] R. G. Herrtwich and G. Hommel. *Kooperation und Konkurrenz — Nebenläufige, verteilte und echtzeitabhängige Programmsysteme*. Springer-Verlag, 1989. ISBN 3-540-51701-4.

[3] H. M. Levy and E. D. Tempero. Modules, Objects, and Distributed Programming: Issues in RPC and Remote Object Invocation. *Software—Practice and Experience*, 21(1):77–90, Jan. 1991.

[4] B. H. Liskov. Primitives for Distributed Computing. In *Proceedings of the Seventh ACM Symposium on Operating System Principles (SOSP)*, volume 13 of *SIGOPS Operating Systems Review*, pages 33–42, Pacific Grove, California, USA, Dec. 1979. ACM.

[5] B. H. Liskov and L. Shrira. Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems. In *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation (PLDI)*, volume 23 of *SIGPLAN Notices*, pages 260–267, Atlanta, Georgia, USA, June 1988. ACM.

[6] B. J. Nelson. Remote Procedure Call. Technical Report CMU–81–119, Carnegie-Mellon University, 1982.

[7] R. Srinivasan. XDR: External Data Representation Standard. http://www.faqs.org/rfcs/rfc1832.html, 1995.

[8] P. Wegner. Classification in Object-Oriented Systems. *ACM, SIGPLAN Notices*, 21(10):173–182, 1986.