Verteilte Systeme

R. Kaiser, R. Kröger, O. Hahm

(HTTP: http://www.cs.hs-rm.de/~kaiser
E-Mail: robert.kaiser@hs-rm.de)
Kai Beckmann
Sebastian Flothow

Sommersemester 2022

http://walker.countynewstoday.com/wp-content/uploads/2015/03/logo_marshal.jpg

Inhalt

Einführung und Motivation

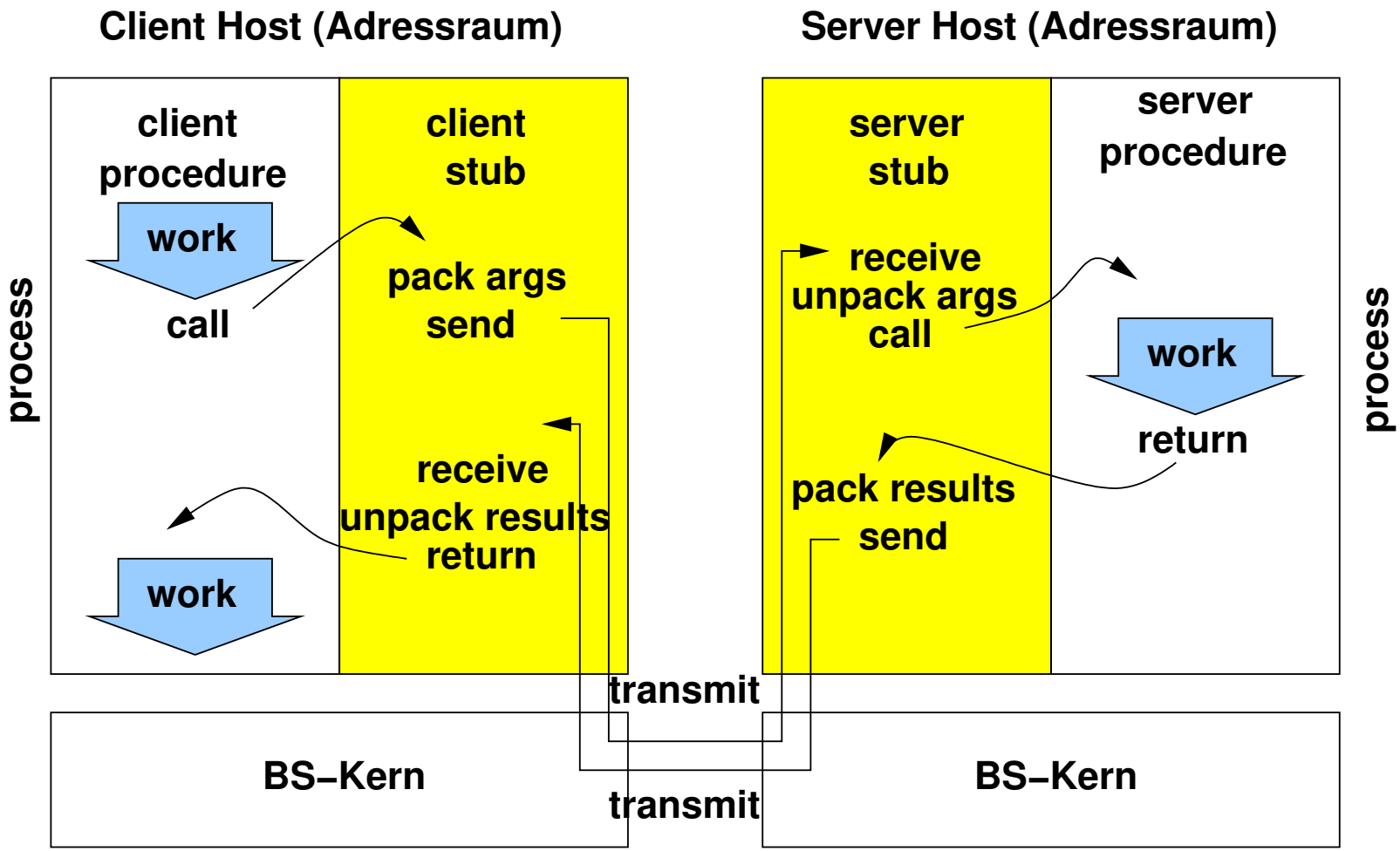
- Nachrichtenorientierte Kommunikation
  - asynchroner Nachrichtenaustausch
  - explizit mit send()/receive()-Operationen
  - Fazit:
    - \+ sehr flexibel, alle Kommunikationsmuster implementierbar
    - \- explizit, I/O-Paradigma
- Ziel des Remote Procedure Call (RPC)
  - deutsch: Fernaufruf (wenig verbreitet)
  - Transparenz der Kommunikation
  - Erscheinungsbild wie üblicher lokaler Prozeduraufruf
- Unterstützung für
  - Dienstorientierung: Dienst = Service = Menge von Funktionen
    RPC für Funktionsaufruf
  - Objektorientierung: RPC genutzt für Methodenaufrufe

Historie

- Erste umfassende Darstellung:
  - Dissertation Nelson (1981, XPARC)
  - abgeleitetes Paper Birrel/Nelson (1984, ACM ToCS)
- Definition:
  - „RPC (Fernaufruf) ist der synchrone Transfer von Kontrolle und Daten zwischen Teilen eines in verschiedenen Adressräumen ablaufenden Programms“
- Nelson's These:
  - RPC ist ein leistungsfähiges Konzept zur Konstruktion verteilter Anwendungen
  - RPC vereinfacht die Programmierung verteilter Systeme
- Heute:
  - Nelson's Sicht allgemein akzeptiert
  - RPC-Systeme in vielen Produkten
  - Typ. Beispiele: SunRPC und NFS, OSF DCE RPC, (aktuell) Apache Thrift, D-Bus

Grundprinzip

Hochschule RheinMain



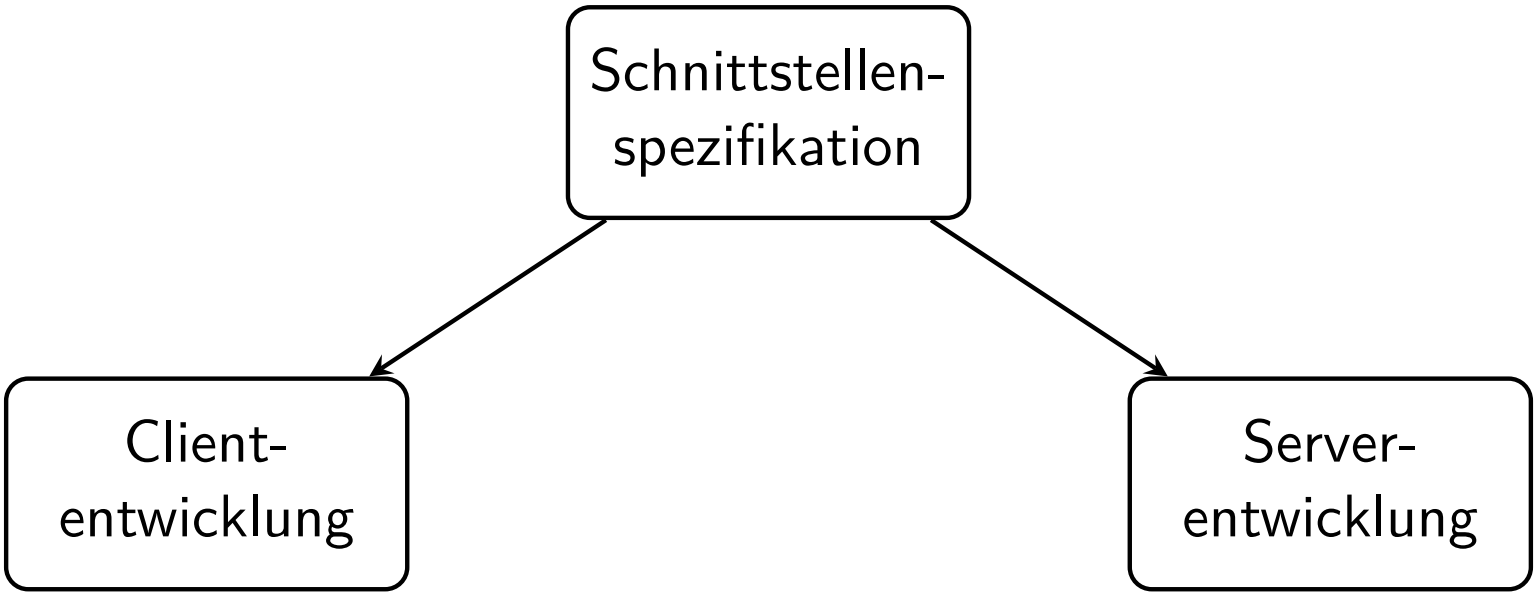
pack/unpack = marshalling/unmarshalling

Stellvertreterkomponenten: stub, proxy, skeleton

Programmentwicklung

Hochschule RheinMain

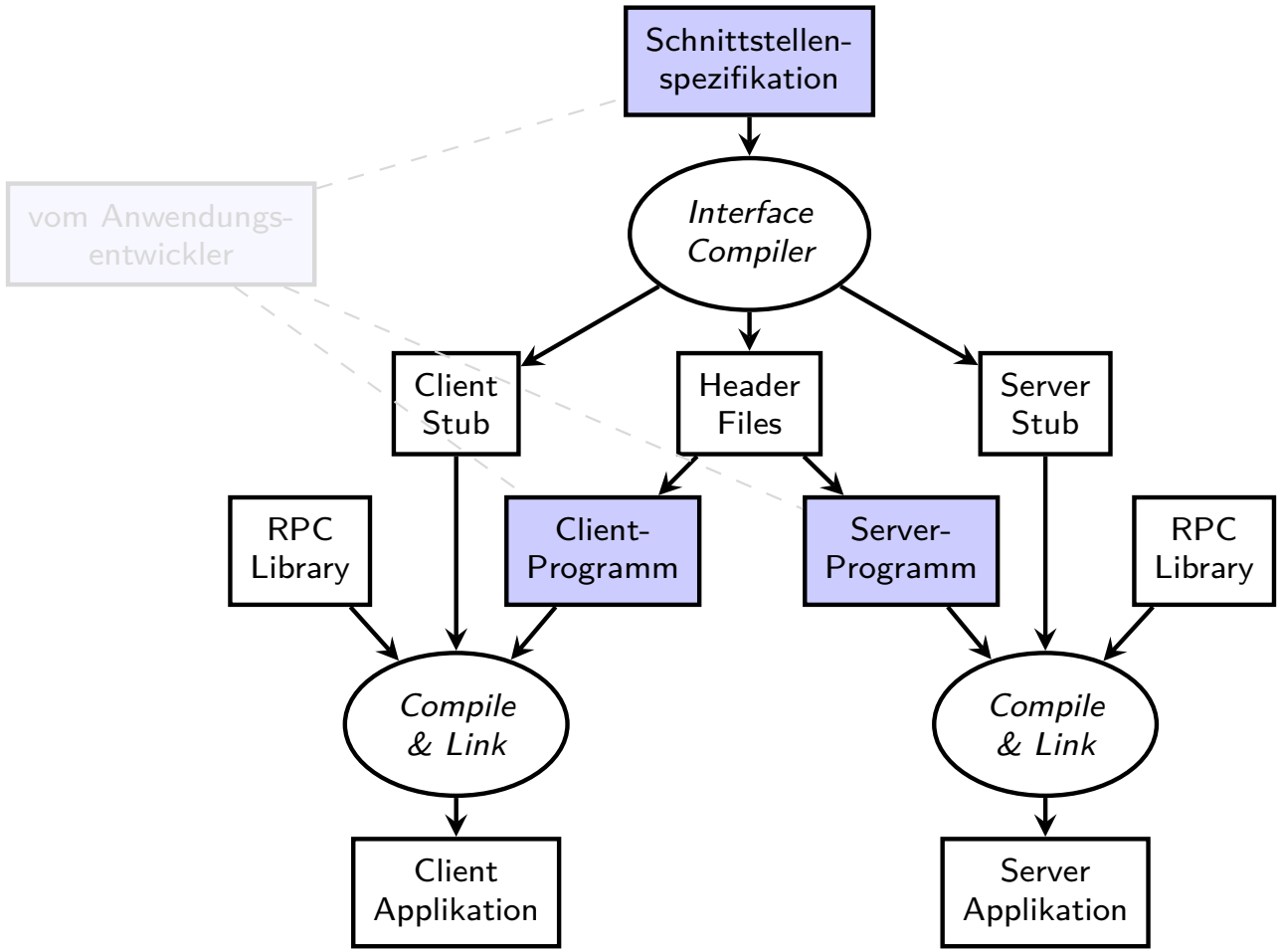
Grobstruktur:



Programmentwicklung (2)

Hochschule RheinMain

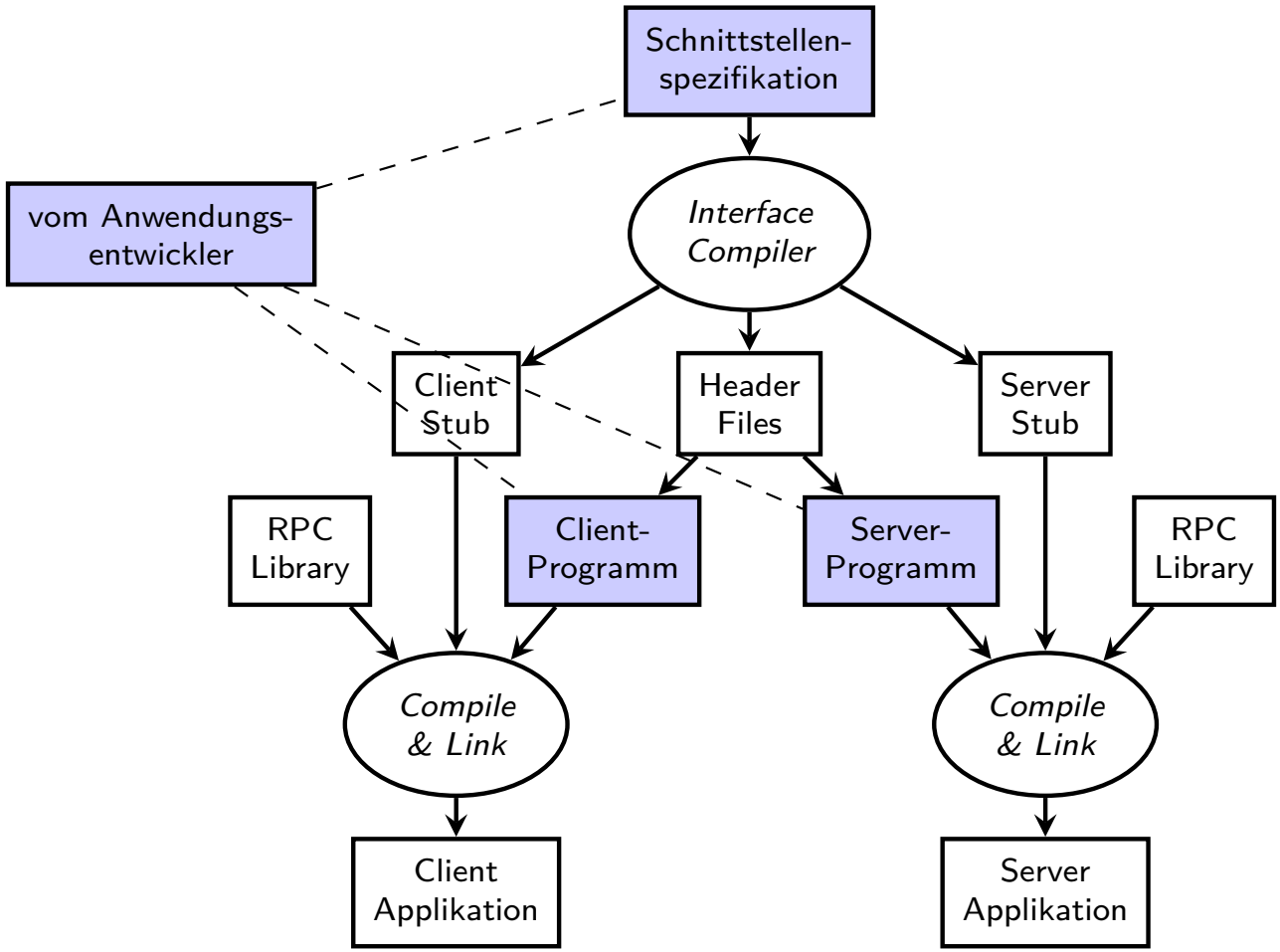
genauer, aber immer noch unabhängig von speziellem RPC-System:



Programmentwicklung (2)

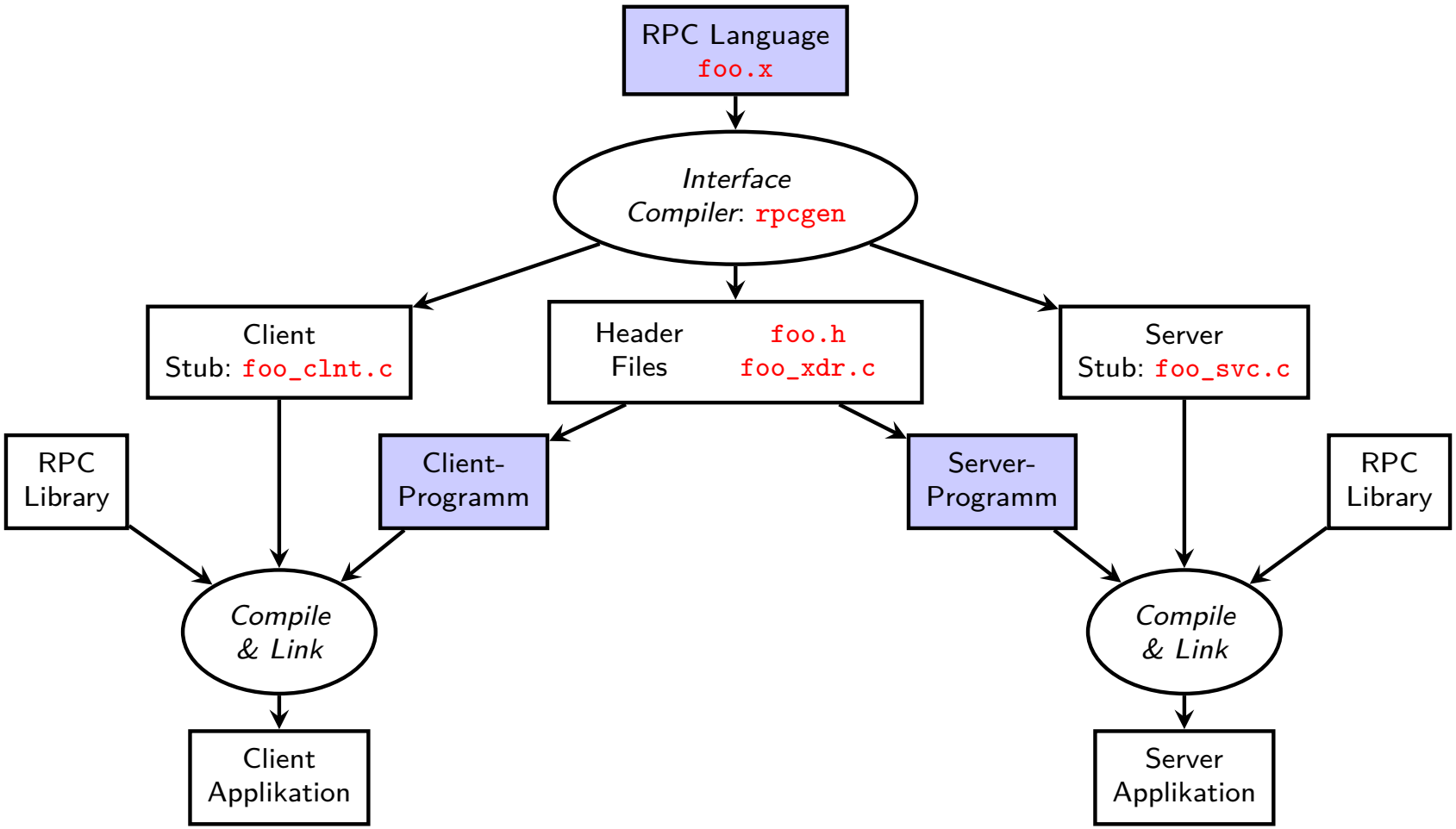
Hochschule RheinMain

genauer, aber immer noch unabhängig von speziellem RPC-System:



Beispiel: SunRPC

Hochschule RheinMain



Beispiel Schnittstellenbeschreibung SunRPC (1)


```
const MAX_FILENAME_LEN = 255;
typedef string t_filename<MAX_FILENAME_LEN>;
const MAX_CONTENT_LEN = 255;
typedef string t_content<MAX_CONTENT_LEN>;
^^I^^I
```

```
struct s_filewrite {
    t_filename filename;
    t_content content;
};
struct s_chmod {
    t_filename filename;
    long mods;
};
^^I^^I
```

```
struct s_fstat {
    long dev;
    long ino;
    long mode;
    long nlink;
    long uid;
    long gid;
    long rdev;
    long size;
    long blksize;
    long blocks;
    long atime;
    long mtime;
    long ctime;
}
^^I^^I
```

Beispiel Schnittstellenbeschreibung SunRPC (2)


```
program fileservice {
    version fsrv {
        int fsrv_mkdir(string) = 1;
        int fsrv_rmdir(string) = 2;
        int fsrv_chdir(string) = 3;
        int fsrv_writefile(s_filewrite) = 4;
        string fsrv_readfile(string) = 5;
        s_fstat fsrv_fileattr(string) = 6;
        int fsrv_chmod(s_chmod) = 7;
    } = 1;
} = 0x30000001;
^^I
```

Beispiel Schnittstellenbeschreibung DCE


```
[ uuid(5ab2e9b4-3d48-11d2-9ea4-80c5140aaa77),
    version(1.0), pointer_default(ptr)
]
interface echo {
    typedef [ptr, string] char * string_t;
    typedef struct {
        unsigned32 argc;
        [size_is(argc)] string_t argv[];
    } args;
    boolean ReverseIt(
        [in] handle_t h,
        [in] args* in_text,
        [out] args** out_text,
        [out,ref] error_status_t* status
    );
}
^^I
```

Beispiel Schnittstellenbeschreibung Thrift


```
typedef i32 MyInteger
enum Operation { ADD = 1,
                 SUBTRACT = 2,
                 MULTIPLY = 3,
                 DIVIDE = 4
}
struct Work {
    1: MyInteger num1 = 0,
    2: MyInteger num2,
    3: Operation op,
    4: optional string comment,
}
exception InvalidOperation { 1: i32 what, 2: string why }
service Calculator {
    void ping(),
    i32 add(1:i32 num1, 2:i32 num2),
    i32 calculate(1:i32 logid, 2:Work w)
         throws (1:InvalidOperation ouch),
    oneway void quit()
}
^^I
```

Binding/Trading

Hochschule RheinMain

- Binding/Trading:
  - Problem: Binden eines Clients an einen Server notwendig
  - Problem gilt analog auch für andere Paradigmen
  - Aspekte: Naming & Locating

⇒ Naming

- Wie benennt der Client, an was er gebunden werden will (Service)
- Interface-Name allgemein aus systemweitem Namensraum
- Trading als Verallgemeinerung: zusätzlich Interface-Attribute
- vgl. Kap. 5: Allg. Namensdienste

⇒ Locating

- Bestimmen der (ortsabhängigen) Adresse eines Servers, der das gewünschte Interface exportiert und zur Diensterbringung benutzt wird
- typisch: (IP-Adresse des Hosts, Portnummer).

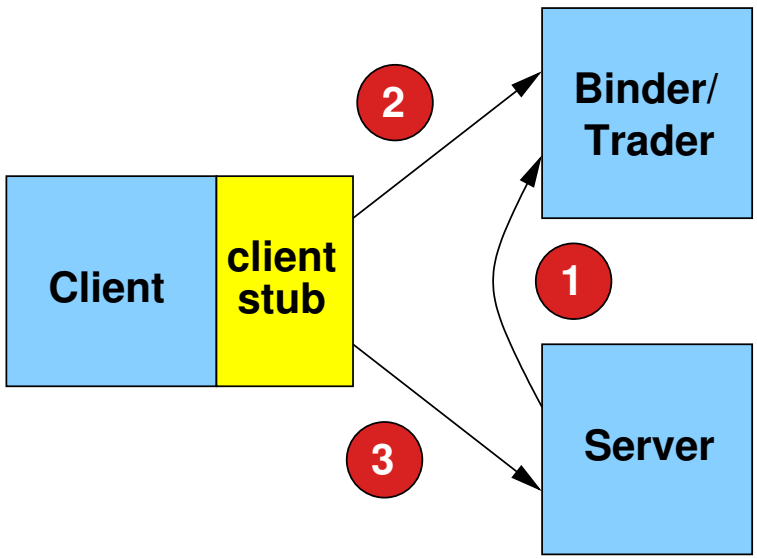
Locating-Alternativen

- Adresse statisch im Anwendungsprogramm
  - kein Suchvorgang erforderlich
  - i.d.R. nicht flexibel genug

  $\Rightarrow$ zu frühes Binden
- Suchen nach Exporteuren zur Laufzeit, z.B. durch Broadcast
  - hoher Laufzeitaufwand
  - Broadcast über Netze hinweg problematisch

  $\Rightarrow$ i.d.R. zu spätes Binden
- Verwaltung von Zuordnunsginformation durch zwischengeschaltete Instanz
  - vermittelnde Instanz wird Binder, Trader oder Broker genannt
  - Exporteur lässt angebotenes Interface (mit allen Attributen) registrieren
  - Bindeanforderung eines Importeurs bewirkt Zuordnung durch Binder/Trader

Prinzipielle Vorgehensweise



1. Exportieren des Interface
   - Registrieren eines Interfaces bei Binder
   - Binder hat bekannte Adresse
2. Importieren
   - bei erster Inanspruchnahme des Dienstes aus stub heraus
   - liefert handle mit Adresse
3. Fernaufruf
   - client stub benutzt Adresse für Aufruf an Server

Binder / Trader

Hochschule RheinMain

Typische Schnittstelle

`Register(` Dienstname, Version, Adresse, evtl. Attribute `)`
`Deregister(` Dienstname, Version, Adresse `)`
`Lookup(` Name, Version, evtl. Attribute `)` $\Rightarrow$ Adresse

- Vorteile:
  - sehr flexibel
  - kann mehrere gleichartige Server berücksichtigen
  - Basis für Lastausgleich zwischen äquivalenten Servern
- Nachteile:
  - zusätzlicher Aufwand beim Exportieren und Importieren eines Interfaces
  - problematisch bei kurzlebigen Servern und Clients

Beispiel: SunRPC

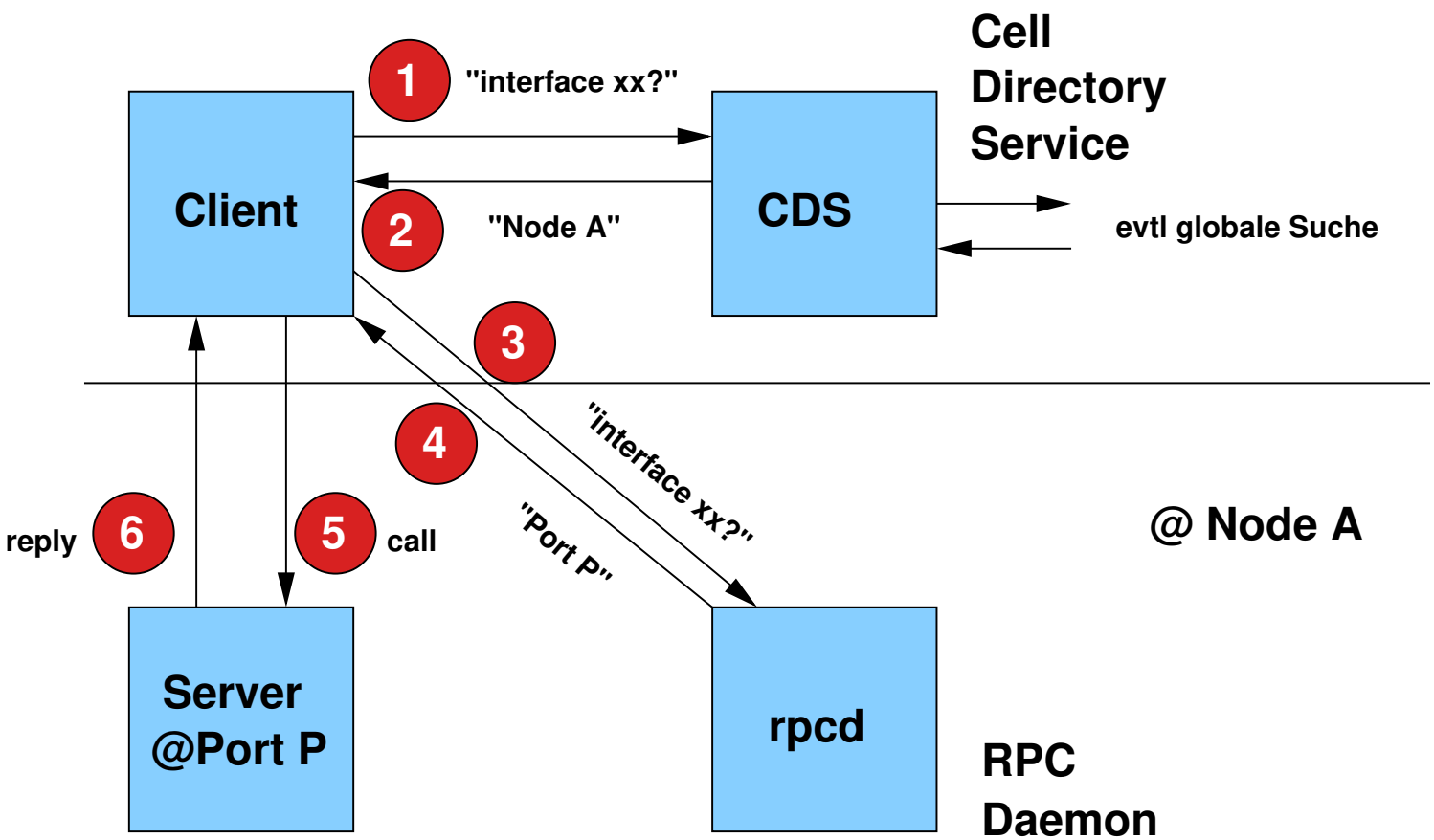
- Namen
  - Paare (Programmnummer, Versionsnummer)
- Adressen
  - Paare (IP-Adresse des Hosts, Portnummer)
- Binder: Portmapper
  - Abbildung von Namen auf Portnummern
  - IP-Adresse des Hosts muss bekannt sein, der dort lokale Portmapper wird befragt
  - Portmapper ist selbst SunRPC-Dienst (Port 111)

Beispiel: DCE RPC

- Namen
  - UUID (Universal Unique Identifier)
  - weltweit eindeutiger String
  - enthält Netzwerk-Adressinformationen (z.B. Ethernet MAC-Adresse) und Zeitmarke
  - generiert durch Tool uuidgen
- Adressen
  - Paare (IP-Adresse des Hosts, Portnummer)
- Binding
  - zweistufig innerhalb einer DCE-Zelle
  - kein zusätzliches Wissen notwendig
  - Binder heisst RPC Daemon

Beispiel: DCE RPC (2)

Hochschule RheinMain



Behandlung der Parameterübergabe

- Heterogenitätsproblem
  - verschiedene Codes (z.B. ASCII - EBCDIC)
  - Little Endian - Big Endian
  - unterschiedliche Zahlenformate
- Lösungsmöglichkeiten
  - Abbildungen zwischen lokalen Datendarstellungen
    - Sender sendet in seiner lokalen Darstellung, Empfänger transformiert
    - erfordert $n \cdot n$ Abbildungen
  - kanonische Netzdatendarstellung für alle Typen
    - erfordert 2n Abbildungen (bei n lokalen Darstellungen)
    - evtl. unnötige Codierung

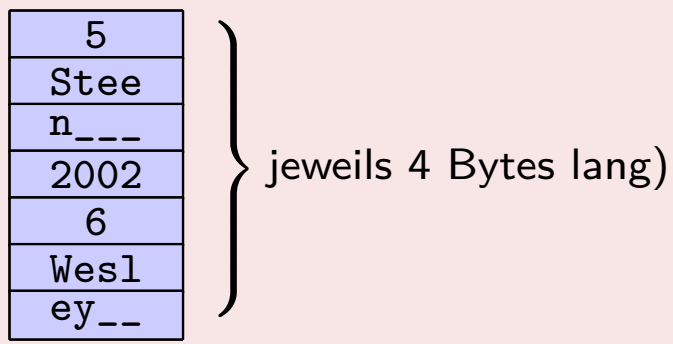
Verbreitete Netzdatendarstellungen

Hochschule RheinMain

- XDR (External Data Representation)
  - definiert durch Sun im Rahmen von SunRPC
  - i.w. Motorola 68000 Datenformate: ASCII; Big-Endian, 2-Komplement; IEEE-Gleitpunktzahlen, ...
  - zusammengesetzte Typen: Arrays, Structures, Unions
  - keine explizite Typisierung der Daten, d.h. keine sich selbst beschreibenden Daten
  - für RPC-Systeme sind die Parameter-Typen aber beim Generieren des Stub Codes für beide Seiten bekannt

Beispiel

```
struct {
    string author<>;
    int year;
    string publisher<>;
}
^^I^^I^^I
```



Verbreitete Netzdatendarstellungen (2)

- ASN.1 BER (ISO Abstract Syntax Notation Number 1, Basic Encoding Rules, ISO 8824, 8825, ITU X.409)
  - explizite Typisierung der übertragenen Daten, d.h. allen Datenfeldern geht die Typinformation voraus.
  - verbreitet: CANopen, LDAP, UMTS/LTE, VoIP, Encryption
  - Standard-Repräsentierung: (Type, Länge, Inhalt)
  - Nachteil: laufzeitaufwändig (Bitzugriffe)

Beispiel

```
count ::=INTEGER
^^I^^I^^I
```

0 2	} Type (Identifier)
0 1	} Länge
1 A	} Inhalt (26_{10})

jeweils 1 Byte (hex)

Type Identifier:

Tag:	1	Boolean
	2	Integer, ...
	16	Sequence
Type:	0	Primitive
	1	Constructed
Class:	00	Universal
	01	Application...

Verbreitete Netzdatendarstellungen (3)

Hochschule RheinMain

- CDR (Common Data Representation)
  - Definition in OMG CORBA 2.0
  - Nutzung im CORBA IIOP-Protokoll
  - Versenden im eigenen Format, „Receiver makes it right“
  - Simple types (`short`, `long`, `float`, `char`, ...)
  - Complex types (`sequence`, `string`, `union`, `struct`, ...)
  - Alignment/Padding entsprechend Mehrfachem der Elementlänge
  - Big-endian

Beispiel

```
struct <string, unsigned long>
^^I^^I
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
5				'S'	'T'	'E'	'E'	'N'				2002			
05	00	00	00	53	54	45	45	4E	00	00	00	00	00	07	D2
← Länge →									← Padding →						

Verbreitete Netzdatendarstellungen (4)

- JSON (JavaScript Object Notation) Data Interchange Format
  - Schlankes, textbasiertes Austauschformat
  - Unabhängig von Programmiersprachen
  - RFC 7159, abgeleitet von ECMAScript
  - leicht zu parsen, viele Parser verfügbar
  - Simple types (string, number, boolean, null)
  - Complex types (object, array)
    - ★ Objekt ist ungeordnete Liste von Name/Wert-Paaren, wobei Name ein String und Wert ein simple Type, ein Object oder ein Array sein kann
    - ★ Array ist geordnete Folge von Werten

Beispiel

```
{
    "AUTHOR" : "Steen",
    "YEAR" : 2002,
    "PUBLISHER" : "Wesley"
}
^^I^^I
```

Probleme

Hochschule RheinMain

- komplexe, zusammengesetzte Parametertypen
  - z.B. structs, arrays, erfordern Regeln zur Serialisierung
- Adressen in Parametern
  - keine Bedeutung im Zieladressraum !
  - einfachste Lösung: Verbieten, nur call-by-value zulassen (i.W. SunRPC)
  - Nutzung eines gemeinsamen globalen Adressraums, falls vorhanden
  - Ersetzen von Zeigern durch Marker, Rekonstruktion zusammengesetzter Datenstrukturen auf Empfängerseite durch dort lokale Zeiger (z.B. DCE RPC)

Sicherheit

Hochschule RheinMain

- Probleme
  - gegenseitige Authentisierung
  - Autorisierung bzgl. ausführbarer Funktionen auf Server-Seite
  - Verschlüsselung der übertragenen Daten
- ausführliche Betrachtung in separatem Kapitel

Semantik des RPC im Fehlerfall

- Fehler-Problematik
  - lokaler Funktionsaufruf:
    Rufer und Gerufener werden gleichzeitig abgebrochen
  - RPC:
    Ausfall einzelner Komponenten in verteilter Umgebung möglich
  - Zusätzlich Fehlerfälle des Nachrichtensystems berücksichtigen
    - Nachrichtenverlust
    - unbekannte Übertragungszeiten
    - Out-of-order-Ankunft von Nachrichten (Überholen)

Semantik des RPC im Fehlerfall (2)

- *at-least-once*-Semantik
  - erfolgreiche Ausführung des RPC
    $\Rightarrow$ aufgerufene Prozedur mindestens einmal ausgeführt,
    d.h. Mehrfachaufruf kann passieren
  - beliebiger Effekt im Fehlerfall möglich
  - i.a. nur für idempotente Operationen geeignet, d.h. mehrfacher Aufruf ändert nicht Zustand und Ergebnis
- Realisierung
  - einfachste Form
  - kommt innerhalb eines Timeouts kein Ergebnis auf Client-Seite an, wird Aufruf vom Stub wiederholt
  - keine Vorkehrungen auf Server-Seite

Semantik des RPC im Fehlerfall (3)

- *at-most-once*-Semantik
  - erfolgreiche Ausführung des RPC
    $\Rightarrow$ aufgerufene Prozedur genau einmal ausgeführt
  - nicht-erfolgreiche Ausführung des RPC
    $\Rightarrow$ aufgerufene Prozedur erscheint als niemals ausgeführt
  - es können keine partiellen Fehlerauswirkungen zurückbleiben
- Realisierung
  - komplexer
  - Duplikaterkennung erforderlich

Semantik des RPC im Fehlerfall (4)

- *exactly-once*-Semantik
  - erfolgreiche Ausführung des RPC
    $\Rightarrow$ aufgerufene Prozedur genau einmal ausgeführt
  - nicht-erfolgreiche Ausführung des RPC
    $\Rightarrow$ aufgerufene Prozedur erscheint als niemals ausgeführt
  - entspricht im Normalfall lokalem Funktionsaufruf
- Realisierung
  - sehr komplex (unmöglich?)

Orphan-Problem

- Orphan = Waise
- Problem: Client stirbt nach Absetzen des RPC
- erzeugter RPC kann weitere Aktivität nach sich ziehen, obwohl niemand darauf wartet
- nach Restart Eintreffen von Antworten aus „früherem Leben“
- Lösungsansätze:
  - *Extermination: gezielter Abbruch verwaister RPCs basierend auf stabilem Speicher (praktisch unbrauchbar)*
  - (Gentle) Reincarnation: Einführung von Epochen auf Client-Seite
  - *Expiration: RPCs werden mit Timeout versehen*

RPC-Protokoll

Hochschule RheinMain

- RPC-Protokoll: Regeln zur Abwicklung von RPCs
- abhängig von unterlagertem Transportdienst
  - Datagrammdienst (z. B. UDP)
    - \+ resourcenschonend, niedrige Latenz
    - \- Duplikate (durch Timeouts), Vertauschungen und Verlust sind möglich
  - zuverlässiger Transportdienst (z. B. TCP)
    - \+ weniger Fehlerfälle auf den höheren Schichten
    - \- ggf. leistungsmindernd

  $\Rightarrow$ die Auswahl erfolgt je nach Dienstanforderung

Beispiel: SunRPC

- auch: Open Network Computing (ONC) RPC
- C-Spracheinbettung
- Unterlagerter Transportdienst
  - TCP oder UDP
  - RPC fügt keine die Zuverlässigkeit steigernde MaSSnahmen hinzu
    - ⇒ UDP und timeouts auf Applikationsebene führen zu „*at-least-once*“-Semantik
    - ⇒ TCP und message transaction ids auf Applikationsebene führen zu „*at-most-once*“-Semantik
- Binding durch Portmapper
  - Portmapper-Protokoll ist selbst RPC-basiert
- Parameter
  - i.W. nur call-by-value
- Sicherheit
  - Authentifizierung: Null, UNIX, DES

Beispiel: SunRPC

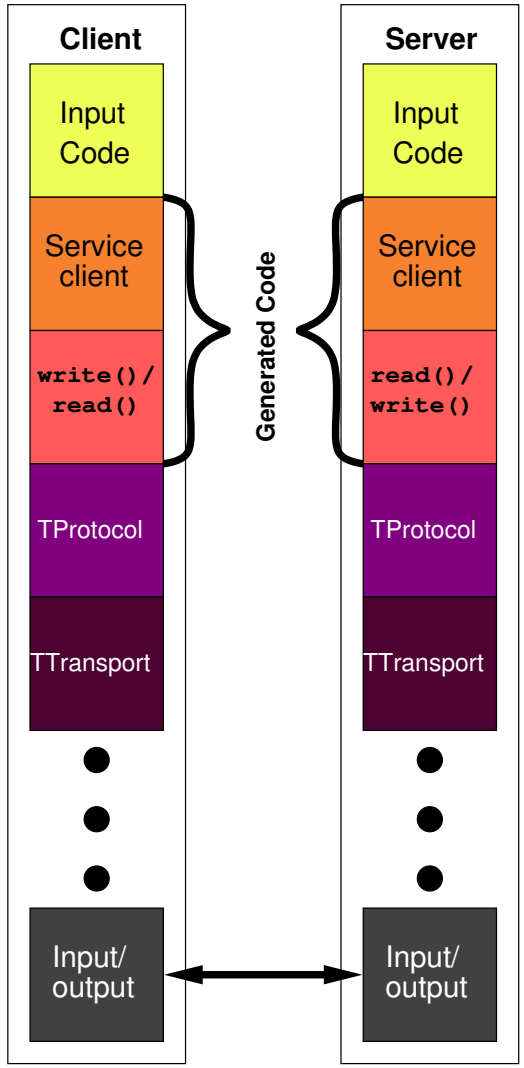
- auch: Open Network Computing (ONC) RPC
- C-Spracheinbettung
- Unterlagerter Transportdienst
  - TCP oder UDP
  - RPC fügt keine die Zuverlässigkeit steigernde MaSSnahmen hinzu
    - ⇒ UDP und timeouts auf Applikationsebene führen zu „*at-least-once*“-Semantik
    - ⇒ TCP und message transaction ids auf Applikationsebene führen zu „*at-most-once*“-Semantik
- Binding durch Portmapper
  - Portmapper-Protokoll ist selbst RPC-basiert
- Parameter
  - i.W. nur call-by-value
- Sicherheit
  - Authentifizierung: ~~Null~~, ~~UNIX~~, ~~DES~~, RPCSEC_GSS

OSF DCE/RPC

- Teil des OSF Distributed Computing Environments
- Grundlage für Microsofts DCOM und ActiveX
- C/C++-Spracheinbettung
- verschiedene Semantiken wählbar mit „*at-most-once*“ als default
- beliebige Parameter-Typen,
  „lange“ Parameter über „Pipe“-Mechanismus
- Sicherheit basierend auf Kerberos-Framework
- Bedeutung stark gesunken

Aktuelles RPC-System: Apache Thrift

- Apache Thrift Projekt (`http://thrift.apache.org/`)
  - Ursprung Facebook, veröffentlicht 2007
  - Unterstützung für alle gängigen Programmiersprachen
  - Einfache Thrift-IDL (vgl. Folie 3-13)
  - IDL Compiler generiert Client- und Server-Hüllen
  - Verschiedene Server-Architekturen nutzbar: TNonBlockingServer, TThreadedServer, TThreadPoolServer, TForkingServer, ...
  - Verschiedene Protokolle und Transports konfigurierbar
  - Protokolle: binäre und textbasierte (u.a. JSON) $\Rightarrow$ geringer Overhead
  - Transports: Tsocket, TMemoryTransport, ...
- Bekannte Nutzer
  - Facebook, last.fm, Pinterest, Uber, NSA



Zusammenfassung

Hochschule RheinMain

- Remote Procedure Calls bieten die Möglichkeit, Funktionen so auf einem entfernten Rechner aufzurufen als würde dies lokal geschehen.
- Wichtige Elemente eines RPC-Systems sind die Schnittstellenbeschreibungssprache (IDL) und deren Compiler, der Binder sowie das Netzdatendarstellungsformat.
- Es existieren verschiedene Fehlersemantiken, die über- oder unterhalb des RPC-Protokolls behandelt werden können.