Verteilte Systeme
Betriebssysteme II

Kapitel 4.4: Middleware – RPC

Prof. Dr. Wolfgang Küchlin

Dipl.-Inform., Dr. sc. techn. (ETH)

Arbeitsbereich Symbolisches Rechnen
Wilhelm-Schickard-Institut für Informatik
Fakultät für Informations- und Kognitionswissenschaften

Universität Tübingen

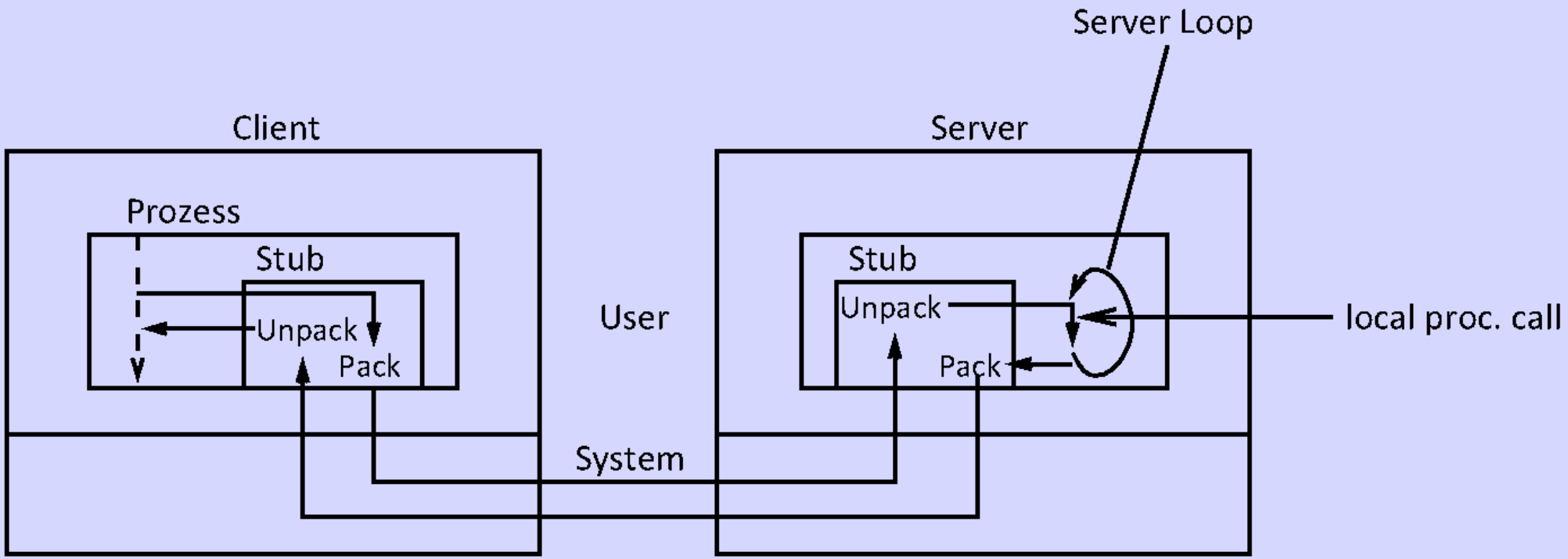
Steinbeis Transferzentrum
Objekt- und Internet-Technologien (OIT)

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

Remote Procedure Call (RPC)

- Entfernter Funktionsaufruf
  - Klient kann Funktion des Servers direkt aufrufen
  - ohne *explizites* Verschicken von Nachrichten auf Programmierebene
  - Implizit:
    - Beim Aufruf: Prozedurname und Parameter werden in Nachricht verpackt *(marshalling)*
    - Nachricht wird an Server geschickt
    - Beim Server: Nachricht wird ausgepackt *(de-marshalling)* und der entsprechende Aufruf wird ausgeführt
    - Ergebnis wird wieder in Nachricht verpackt und zurückgeschickt

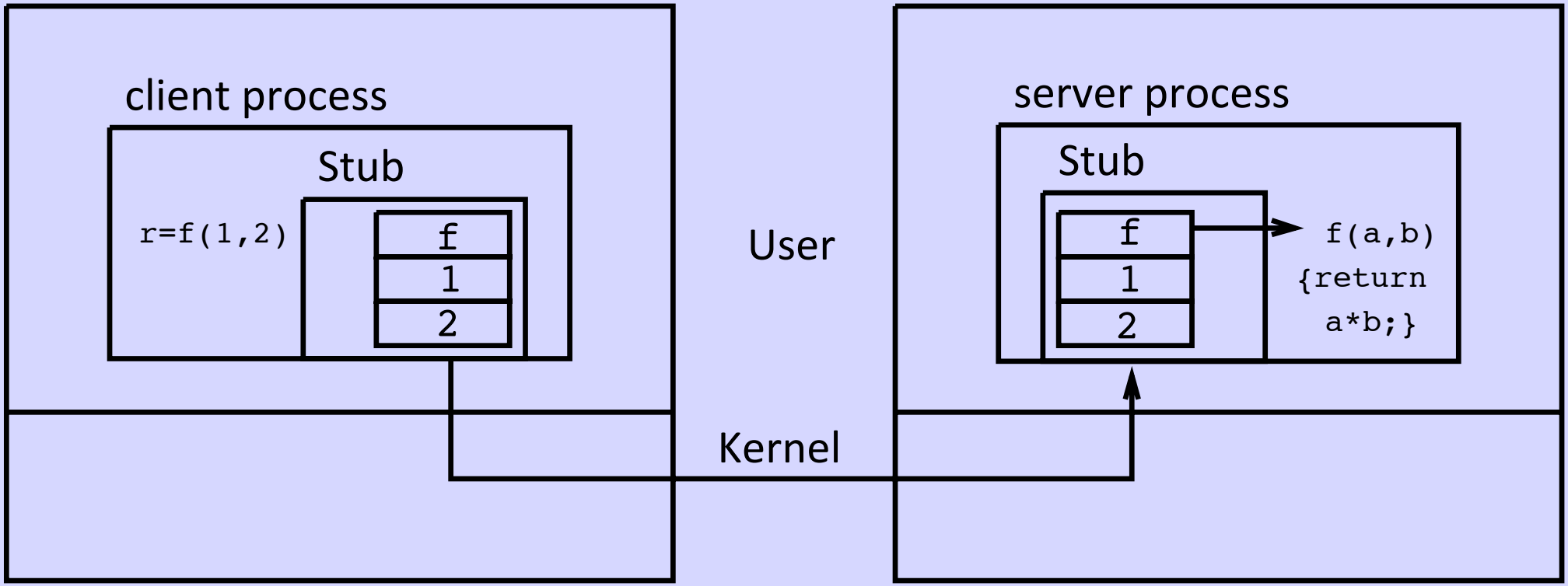
Schema des RPC



Remote Procedure Call (RPC)

- *parameter marshalling* (aufreihen, zusammenstellen)
  - Verpacken von Parametern nebst Konversion
- *unmarshalling* (oder demarshalling)
  - Auspacken
- Server-Loop kann ebenfalls automatisch generiert werden.
  - Server ruft in Endlosschleife den Server-stub immer wieder auf und bearbeitet so einen Auftrag nach dem andern

Schema des Parameter Marshalling



Remote Procedure Call (RPC)

- Bei bekannten Datentypen automatische Erzeugung von Code
  - Erzeugung von zwei Stummel (*stub*) Prozeduren für Ein-/ Auspacken, Versenden und Empfangen
- Ablauf:
  - Klient ruft Client-stub auf, der den Namen der fernen Prozedur trägt
  - Client-stub benachrichtigt Server-stub (und blockiert)
  - Server-stub ruft eigentliche Prozedur auf und schickt Ergebnis zurück
  - Client-stub wird deblockiert, Ergebnis wird ausgepackt und der Client-stub terminiert mit fernem Ergebnis als Ergebnis seines Aufrufs
- Daten müssen in einer Standardrepräsentation verschickt werden
  - Client und Server können auf verschiedenen Architekturen laufen

Remote Procedure Call (RPC)

- Grund-Annahme: Heterogenität
  - Klient und Server können grundverschieden sein
    - verschiedene Prozessoren
    - verschiedene Betriebssysteme
    - verschiedene Programmiersprachen
  - Keine Codeübertragung
    - keine Übertragung von Objekten als Parameter
  - Daten müssen in einer Standardrepräsentation verschickt werden
    - XDR
  - Evtl. Sprach-unabhängige Datendefinitionssprache (IDL)
    - Für jede Sprache erzeugt ein IDL-Compiler Sprach-spezifische Stubs

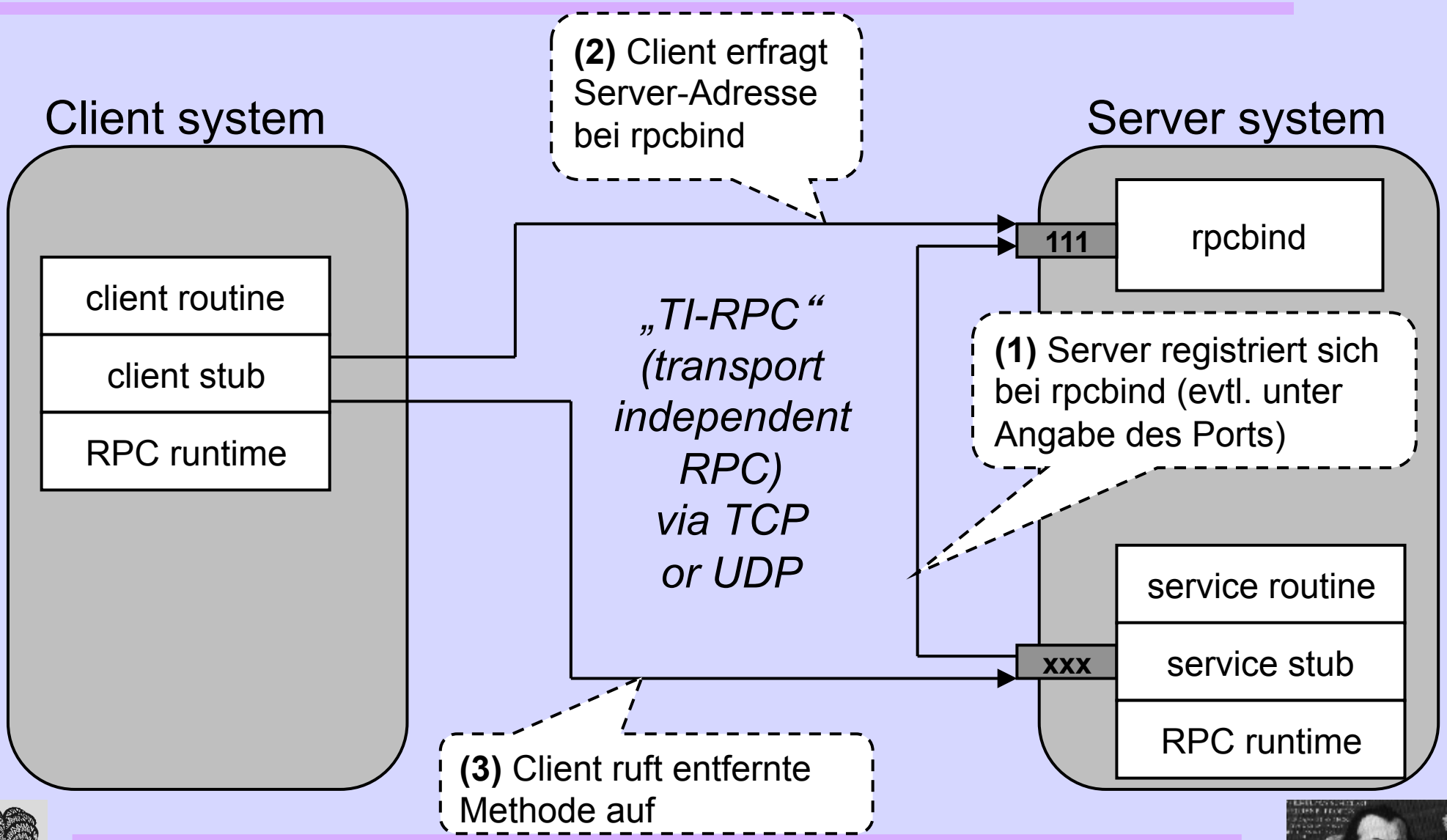
RPC – Dynamisches Binden

- Durch die RPC-Methode sind Aufrufer und gerufene Prozedur entkoppelt:
  - nicht in einem gemeinsamen Programm vereinigt
  - können zu verschiedenen Zeiten gestartet werden

→ **dynamisches Binden (*dynamic binding*)**

- Beispiel: statisch: `mobject.add(int)`
  dynamisch: `invoke(mobject, "add", int)`
- Programmbeginn: Client-stub kennt Partner-Adresse noch nicht
- Bei Aufruf von Client-stub: Anfrage an zentralen *Binder* nach Server, der die Prozedur in der benötigten Version zur Verfügung stellt.
- Server melden sich beim Binder betriebsbereit unter Angabe von Name, Versions-Nr. und Adresse + evtl. id, falls Name nicht eindeutig.
- Zur Laufzeit: Binder reicht dem Client die Server-Information weiter, und der Client-stub wendet sich danach direkt an den Server.

SUN RPC (TI-RPC)



SUN RPC – Server Registration

Interface:

```
rpc_reg ( rpcprog_t prognum /* Server program number */
  rpcvers_t versnum         /* Server version number */
  rpcproc_t procnum         /* server procedure number */
  char *procname            /* Name of remote function */
  xdrproc_t inproc          /* XDR filter to encode arg */
  xdrproc_t outproc         /* XDR filter to decode result */
  char *nettype             /* For transport selection */ );
```

Beispiel:

```
void *rusers();
rpc_reg(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
   rusers, xdr_void, xdr_u_int, "visible");
svc_run();  /* Never returns */
```

Uses the transports chosen with the visible flag (`v') set in their /etc/netconfig entries.
Alternative: "tcp" or "udp"

SUN RPC – Client Call

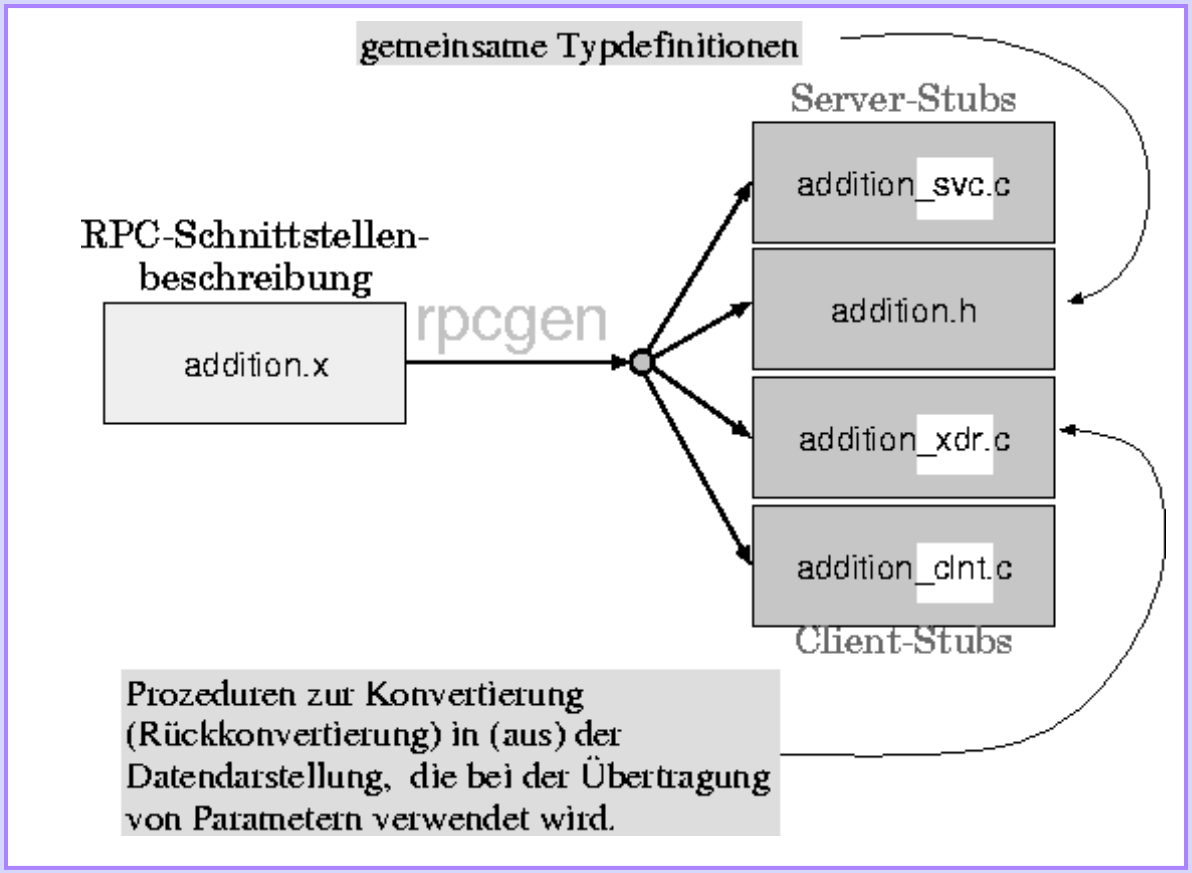
Interface:

```
int 0 or error code
rpc_call ( char *host          /* Name of server host */
       rpcprog_t prognum       /* Server program number */
       rpcvers_t versnum       /* Server version number */
       rpcproc_t procnum       /* Server procedure number */
       xdrproc_t inproc        /* XDR filter to encode arg */
       char* in                /* Pointer to argument */
       xdrproc_t outproc       /* XDR filter to decode result */
       char* out               /* Address to store result */
       char* nettype           /* For transport selection */ );
```

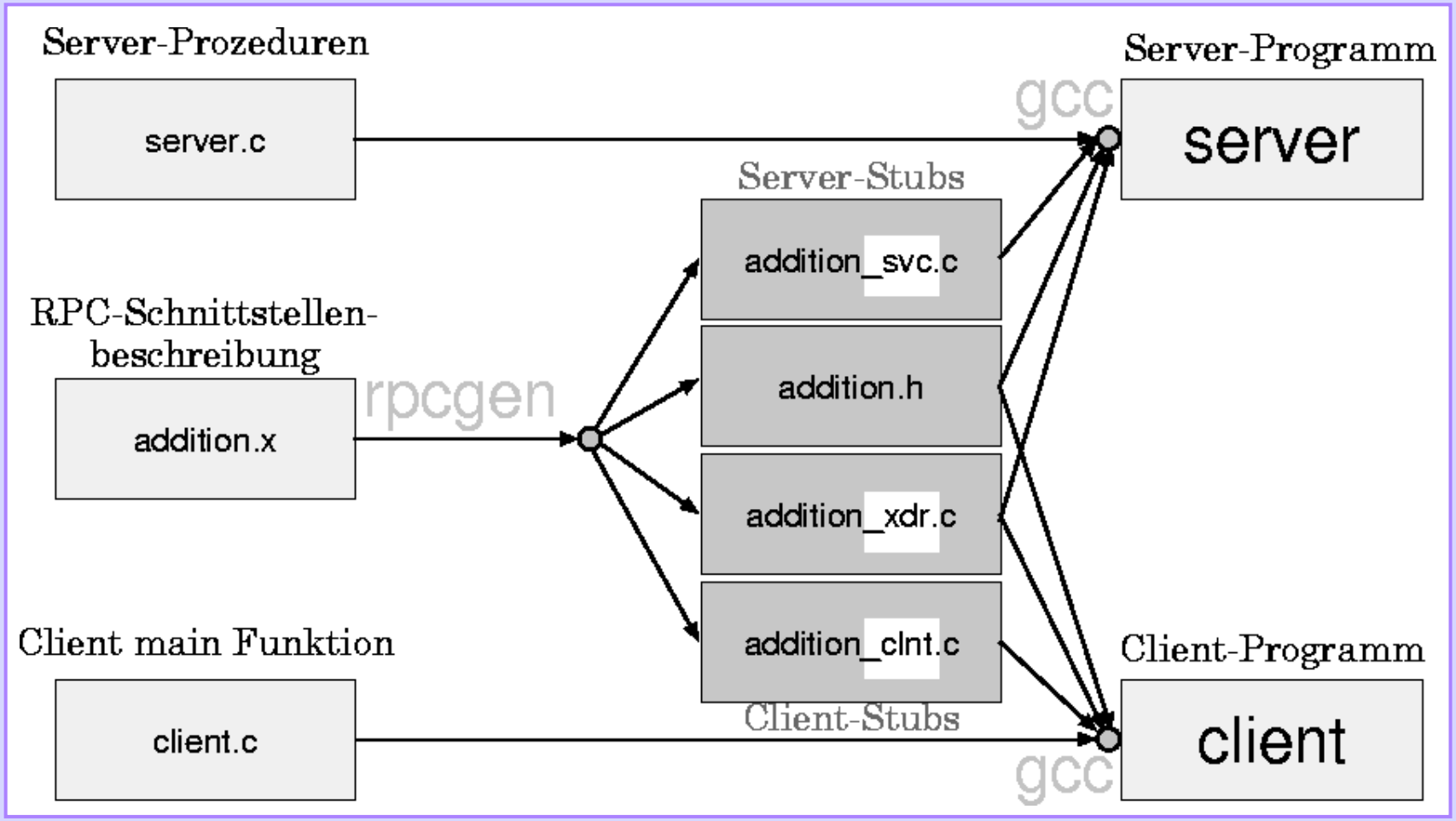
Beispiel:

```
unsigned int nusers;
rpc_call(argv[1], RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
   xdr_void, (char *)0, xdr_u_int, (char *)&nusers, "visible");
```

Automatische Stub Generierung



Gesamtablauf



XDR Schnittstellenbeschreibung: addition.x

```
/* file: addition.x */
struct i_result
{
  int s;
};
struct i_param
{
  int a;
  int b;
};
program ADD_PROG {                    /* Programm */
  version ADD_VERS {                  /* Version */
    i_result ADD (i_param) = 1;       /* Prozedur */
  } = 1;
} = 21111111;
```

File: addition.h

```
#include <rpc/rpc.h>

struct i_result { int s; };
typedef struct i_result i_result;

struct i_param {
  int a;
  int b;
};
typedef struct i_param i_param;

#define ADD_PROG        21111111
#define ADD_VERS        1
#define ADD             1
extern i_result * add_1();

extern bool_t xdr_i_result();
extern bool_t xdr_i_param();
```

File: server.c

```
#include <rpc/rpc.h>
#include <addition.h>

i_result *add_1(i_param *p) {
   static i_result result;

   result.s = p->a + p->b;
   return &result;
}
```

File: client.c

```
#include <rpc/rpc.h>
#include <addition.h>

int main(int argc, char* argv[]) {
  CLIENT* clnt;
  i_result* result;
  i_param parameter;

  clnt = clnt_create(argv[1], ADD_PROG, ADD_VERS, “udp“);
  parameter.a = atoi(argv[2]);
  parameter.b = atoi(argv[3]);

  result = add_1(&parameter, clnt);

  printf(“Ergebnis: %d\n“, result->s);

  return 0;
}
```

Fehlerbehandlung in RPC-Systemen

- Durch die Entkopplung zwischen Klient und Server kann es zu folgenden Fehlern kommen:

1. Der Klient findet den Server nicht.
2. Die Auftragsnachricht Klient/Server geht verloren.
3. Die Antwortnachricht Server/Klient geht verloren.
4. Der Server stürzt nach Auftragserhalt ab.
5. Der Klient stürzt nach Auftragsvergabe ab.

Fehlerbehandlung in RPC-Systemen

Behandlungsmöglichkeiten sind u.a.:

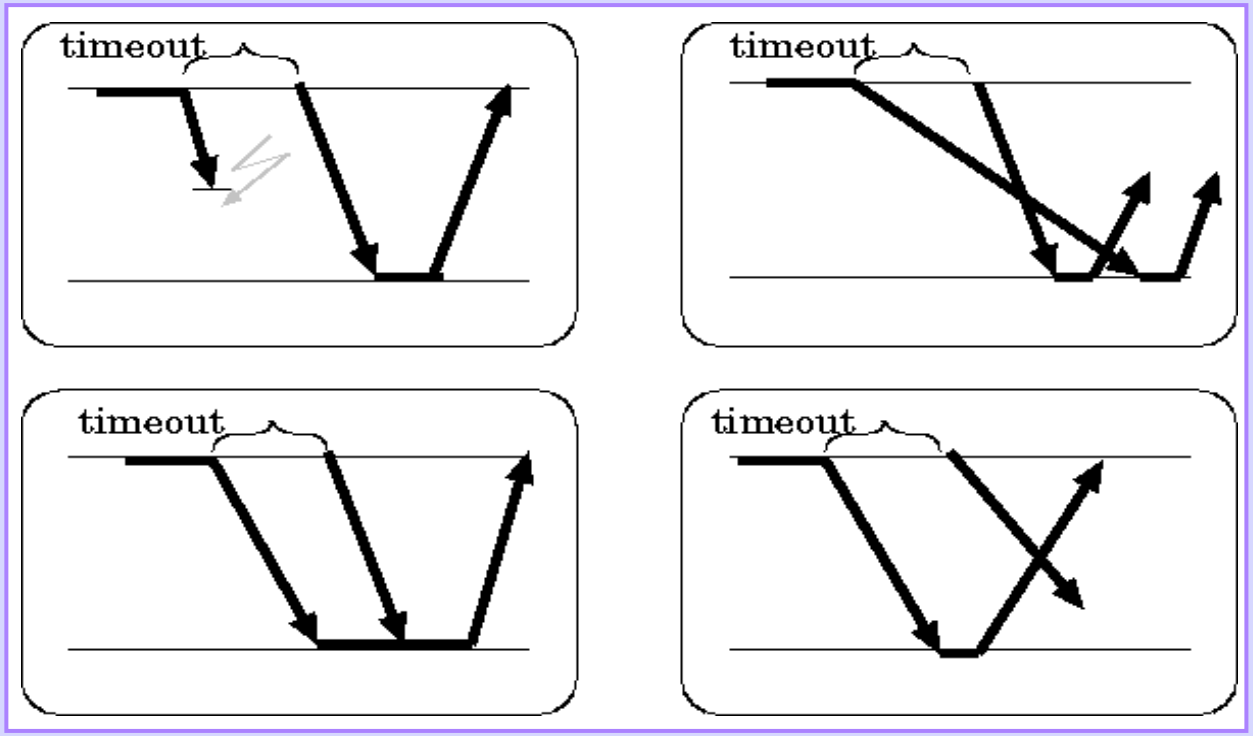
Zu 1: ***No Server.*** Stub-Prozedur gibt eine Fehlermeldung zurück.

- Fehlerwert, z.B. -1, genügt nicht, da er auch ein legales Resultat sein könnte.
- *Exceptions*
- In C: Simulation durch signal handlers, z.B. mit einem neuen Signal SIGNOSERVER

Fehlerbehandlung in RPC-Systemen

Zu 2: ***Lost Request***. Sender startet einen Timer und verschickt den Auftrag nach timeout neu.

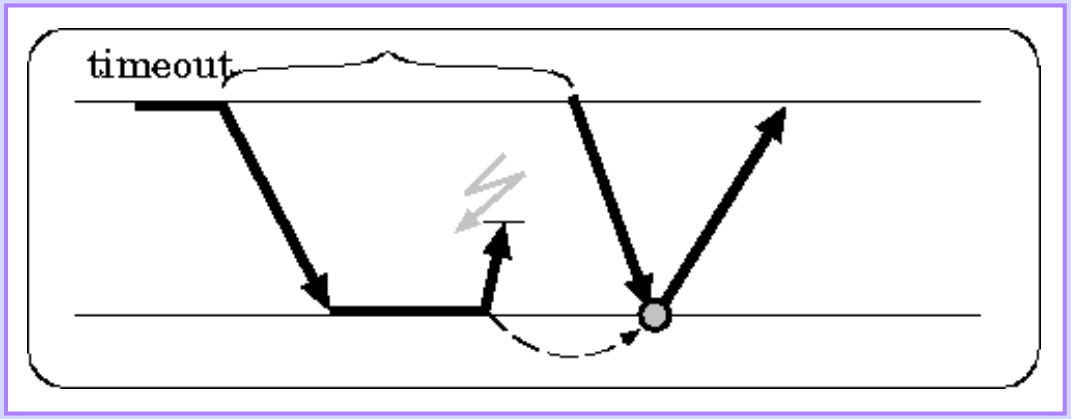
- Kennzeichnung der Aufträge als Original oder Kopie kann verhindern, dass derselbe Auftrag (z.B. Buchung) mehrmals bearbeitet wird.



Fehlerbehandlung in RPC-Systemen

Zu 3: ***Lost Reply***.

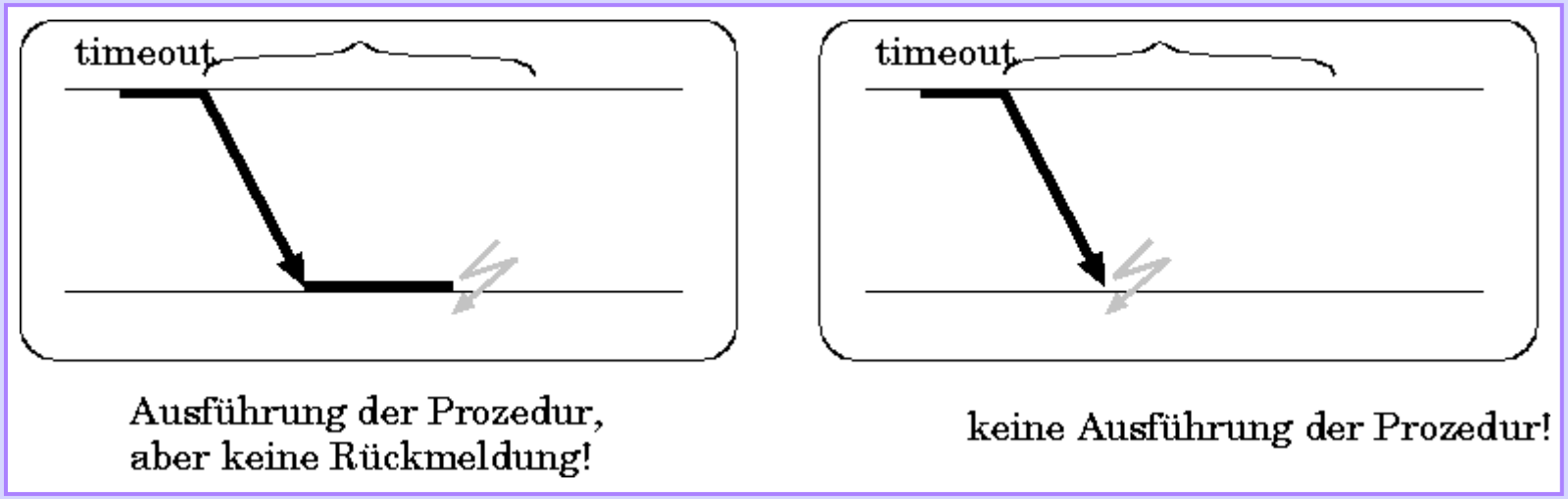
- Manche Aufträge können problemlos wiederholt werden (Lesen eines Datums), andere nicht (*relative update*; Buchung).
- Buchungsaufträge müssen als Originale und Kopien gekennzeichnet werden.



Fehlerbehandlung in RPC-Systemen

Zu 4: ***Server Crashes***.

- Abstürze des Servers vor Auftragseingang fallen in Kategorie 1 (no server)
- Späterer Absturz **vor** Auftragsbearbeitung: Wiederholung des Auftrags (nach reboot)
- Späterer Absturz **nach** Auftragsbearbeitung: Wiederholung nicht unbedingt möglich
- Problem: Klient kennt die Absturzursache nicht.



Fehlerbehandlung in RPC-Systemen

drei Konzepte der RPC-Abwicklung:

1. ***At least once semantics***. Das RPC-System wiederholt den Auftrag so lange, bis er quittiert wurde.
2. ***At most once semantics***. Das RPC-System bricht nach timeout ab mit Fehlermeldung.
3. ***Keine Garantie***. Das RPC-System gibt irgendwann auf. Der Auftrag kann nicht oder auch mehrmals bearbeitet worden sein.

Fehlerbehandlung in RPC-Systemen

Zu 5: ***Client Crashes***.

- Klient-Absturz vor Auftragsvergabe oder nach Auftragsbestätigung hier irrelevant.
- Absturz während der Auftragsverarbeitung führt zu einer Waise (Prozess ohne Eltern; ***orphan***).
- Probleme: Verbrauch von CPU-Ressourcen und Blockierung anderer Prozesse

Fehlerbehandlung in RPC-Systemen

Methoden, um Waisen aus dem System zu entfernen:

1. Extermination

- Notieren jeder Auftragsvergabe beim Klienten auf sicherem Medium
- Nach reboot werden (unquittierte) offene Aufträge storniert (Löschung = extermination)
- Problem: exzessiver Aufwand.

2. Reincarnation

- Zeit wird in Epochen eingeteilt
- Jeder Klient-reboot startet neue Epoche
- Prozesse der alten Epoche werden auf dem Server beendet
- Überlebt doch einer (z.B. durch verlorene Epochen-Meldung), so tragen seine Resultate veralteten Epochen-Stempel

Fehlerbehandlung in RPC-Systemen

3. **Gentle Reincarnation**
   - Server fragen bei Start einer neuen Epoche nach, ob Eltern der Aufträge noch leben
   - Nur wenn sich Eltern nicht melden, werden die Aufträge beendet
4. **Expiration**
   - Auftragsbearbeitung wird mit timeouts versehen
   - Nach timeout müssen sich Eltern melden und dadurch den Timer neu starten
   - ansonsten wird der Prozess beendet