EDU/EVS

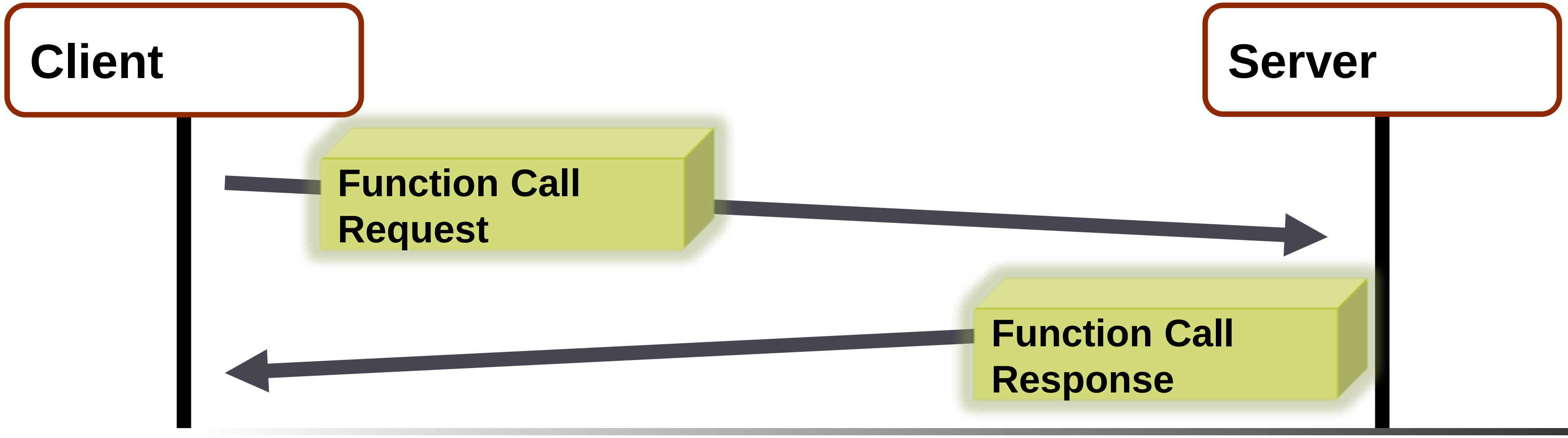
Entwurf Verteilter Systeme

Prof. Dr.-Ing. Martin Gaedke
Technische Universität Chemnitz
Fakultät für Informatik
Professur Verteilte und selbstorganisierende Rechnersysteme

http://vsr.informatik.tu-chemnitz.de

Request/Response Model

- Standard idea of distributed computing
  - Focus on behavior of programming languages
  - Note: In contrast to message exchange models, the Request/Response model is inherently synchronous. Each operation determines a communication relationship



Pull/Push Model

- **Pull-Medium** – Use of the endpoint originates from the user
  - Example: Request/Response approaches like HTTP
- **Push-Medium** – User is notified of specific events / provided data by the endpoint
  - Example: Publish/Subscribe approaches

Part II

PROGRAMMING IN DISTRIBUTED SYSTEMS

Chapter 1

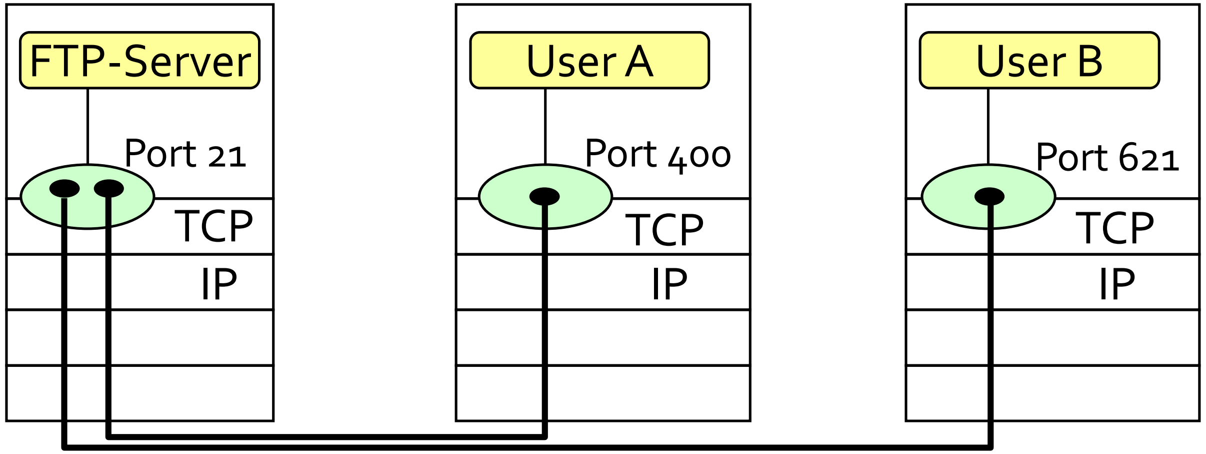
INTRODUCTION

Introduction

- Programming in distributed systems or with distributed systems requires a look at many different aspects
  - cf. the challenges discussed earlier
  - communication aspects between components
  - realisation of address, binding, and contract
  - programming paradigms to take care of
- It all started with TCP/IP and sockets

TCP: Addressing

- Identification of TCP services occurs over ports (TSAPs in OSI terminology)
- Port numbers up to 255 are reserved for frequently used services (for example, 21 for FTP, 23 for TELNET, 80 for HTTP)
- **<u>Socket</u>** consists of computer's internet address and a port.
  - Notation: (IP-Address:Port Number) ⇨ applied internet-wide
- Example: FTP-Server on a computer with IP address (129.13.35.7) can be reached on the 129.13.35.7:21 socket



TCP: Connection Setup (I)

Connections can be setup as active (*connect*) or passive (*listen/accept*) after socket creation.

- **Active mode**: Request of a TCP connection with a specific socket.
- **Passive mode**: User informs TCP that he is waiting for an incoming connection.
  - Specification of a particular socket from which the incoming connection is anticipated (*fully specified passive open*)
  - Accept all connections (*unspecified passive open*)
  - Upon a connection request a new socket is created to become a connection endpoint
- **Remark**: Connection is built up by TCP instances without any further action by the service user.

Well-Known Ports

Many applications use TCP as a protocol; however, the right *Port* has to be chosen in order to communicate to the right application on the other side.

- 13: daytime
- 20: **FTP** Data
- 25: **SMTP**
  (Simple Mail Transfer Protocol)
- 53: **DNS**
  (Domain Name System)
- 80: **HTTP**
  (Hyper Text Transfer Protocol)
- 119: **NNTP**
  (Network News Transfer Protocol)

```
> telnet osiris 13
Trying 129.13.3.121...
Connected to osiris.
Escape character is '^]'.
Mon Aug  4 16:57:19 1997
Connection closed by foreign host
```

```
> telnet sokrates 25
Trying 129.13.3.161...
Connected to sokrates .
Escape character is '^]'.
220 sokrates ESMTP Sendmail 8.8.5/8.8.5;
Mon, 4 Aug 1997 17:02:51 +0200
HELP
214-This is Sendmail version 8.8.5
214-Topics:
214-    HELO    EHLO    MAIL    RCPT    DATA
214-    RSET    NOOP    QUIT    HELP    VRFY
214-    EXPN    VERB    ETRN    DSN
214-For more info use "HELP <topic>".
...
214 End of HELP info
```

Introduction

- Motivation – Development of distributed Internet applications
  - Which service should the application (Server) offer?
  - How do the service user (Client) and Server communicate?
  - How is the application protocol described and implemented?
  - Should the application be based on TCP or UDP?
  - Are there certain reasons for using TCP and UDP?
- Protocol implementation
  - Use of standard protocols – support may already be provided by the programming platform
  - Completely new protocol based on UDP or TCP
  - Requires programming support for internet sockets
- Berkeley Sockets
  - Origin: Developed 1983 at UC Berkeley as a component of BSD Unix
  - Application Programming Interface (API)
  - For further details check – RFC 147

Introduction: Socket – Server in C

http://vsr.informatik.tu-chemnitz.de/edu/old/evs05/ln/evs04/so/tcpsrv.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#define REPLY "HTTP/1.0 200 OK\r\n\r\n"

main(int argc, char **argv) {
  int sock, msgsock, l, buflen;
  struct sockaddr_in sa;
  char buf[1024], *bufptr;

  if (argc != 2) xerr("usage: tcpsrv port");
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock == -1) err("socket");
  /* Eigene Socket-Adresse konstruieren */
  sa.sin_family = AF_INET;
  sa.sin_addr.s_addr = INADDR_ANY;
  sa.sin_port = htons(atoi(argv[1]));
  if (bind(sock,(struct sockaddr*)&sa,sizeof(sa)))
    err("bind");
  if (listen(sock, 5) == -1) err("listen");
```

```
 for(;;) {
   msgsock = accept(sock, 0, 0);
   if (msgsock == -1) err("accept");
   memset(buf, 0, sizeof(buf));
   bufptr = buf; buflen = sizeof(buf);
   while (strstr(buf, "\r\n\r\n") == NULL && buflen
       > 0) {
     l = read(msgsock, bufptr, buflen);
     if (l == -1) err("read");
     if (l == 0) {
       fprintf(stderr, "client closed conn\n");
       break;
     }
     bufptr += l; buflen -= l;
   }
   fprintf(stderr, "request: %s\n", buf);
   if (write(msgsock,REPLY,sizeof(REPLY)) == -1)
     err("write");
   if (close(msgsock) == -1) err("close");
 }
}
```

Introduction: Socket – Client in C

http://vsr.informatik.tu-chemnitz.de/edu/old/evs05/ln/evs04/so/tcpcli.c

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#define REQUEST "GET /index.html HTTP/1.0\r\n\r\n"

main(int argc, char **argv) {
  int sock, l;
  struct sockaddr_in sa;
  struct hostent *hp;
  char buf[1024];

  if (argc != 3) xerr("usage: tcpcli host port");
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock == -1) err("socket");

  sa.sin_family = AF_INET;
  hp = gethostbyname(argv[1]);
  if (hp == NULL) herr("gethostbyname");

  memcpy(&sa.sin_addr, hp->h_addr, hp->h_length);
  sa.sin_port = htons(atoi(argv[2]));
if (connect(sock, (struct sockaddr*)&sa,
   sizeof(sa)) == -1) err("connect");
 if (write(sock,REQUEST,sizeof(REQUEST)) == -1)
   err("write");
 while ((l=read(sock, buf, sizeof(buf))) > 0)
   write(1, buf, l);
 if (l == -1) err("read");
 if (close(sock) == -1) err("close");
}
```

Secure Sockets

- Secure Sockets Layer (SSL)
  - Version 1.0 by Netscape Communications (1994)
- Transport Layer Security (TLS)
  - IETF-standard from the year 1999 (RFC 2246)
- Network protocol for secure data transfer
- Since Version 3.0 SSL is being further developed under the name TLS
  - Minor differences between SSL 3.0 & TLS 1.0
  - TLS 1.0 is presented as SSL 3.1

SSL/TLS – Architecture

- In TCP/IP-model
  - Above the Transport layer (i.e. TCP,…)
  - Below the Application layer (i.e. HTTP,…)
- Basic idea: generic security layer
- Protocol consists of 2 layers:

Handshake Protocol	Change Cipher Spec Protocol	Alert Protocol	Application Data Protocol
Record Protocol			

- Cf. Lecture SVS (Summer Semester)

Secure Socket Client in C

http://vsr.informatik.tu-chemnitz.de/staff/jan/SSL_SOCK/ssl_sock_client.c

```
#include <stdio.h>
#include <unistd.h>
#include <err.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include <openssl/crypto.h>
#include <openssl/ssl.h>

#define HOST "www.deutsche-bank.de"
#define PORT 443
#define BUF "GET /index.htm HTTP/1.1\r\nHost: www.deutsche-bank.de\r\n\r\n"
#define LEN 4096


static char *sslerr(void)  /* Fehlerbehandlung */
{
        static char buf[1024];
        ERR_error_string(ERR_get_error(), buf);
        return buf;
}

 /* Secure Socket-Layer oberhalb der durch “sock” repraesentierten Transportschicht aufbauen: */

void init_ssl(int sock, SSL_CTX **ctx, SSL **ssl)  {
        *ctx = SSL_CTX_new(SSLv23_client_method());              /* Wahl der SSL-Methode (Version) */
        SSL_CTX_set_verify(*ctx, SSL_VERIFY_NONE, NULL);  /* hier: Keine Zertifikatspruefung */
        *ssl = SSL_new(*ctx);                                    /* Erzeugung des Secure Socket-Layers */
        SSL_set_fd(*ssl, sock);                                       /* Aufsetzen des SSL auf vorhanden TCP-Verb. */
        SSL_set_connect_state(*ssl);                             /* SSL-Handshake ausloesen */
        if (SSL_connect(*ssl) <= 0)
                errx(1, "SSL_connect: %s", sslerr());
}
```

Secure- Socket-Example in C

```
int main(int argc, char *argv[]) {
  int sock;
  char retbuf[LEN];
  int len;
  struct sockaddr_in server_addr;
  struct hostent *hp;
  SSL_CTX *ctx;
  SSL *ssl;

  SSL_library_init(); /* SSL-Bibliothek initialisieren */
  SSL_load_error_strings(); /* error-Strings laden */
 /* “normalen” TCP-Socket erzeugen und verbinden: */
  sock = socket(AF_INET, SOCK_STREAM, 0);
  if (sock == -1) errx(0, "socket");

  server_addr.sin_family = AF_INET;
  hp = gethostbyname(HOST);
  if (hp == NULL) err(0, "gethostbyname");

  memcpy(&server_addr.sin_addr, hp->h_addr, hp->h_length);
  server_addr.sin_port = htons(PORT);

  if (connect(sock, (struct sockaddr*)&server_addr,
          sizeof(server_addr)) == -1) err(0, "connect");

  printf("connected\n");
  init_ssl(sock, &ctx, &ssl); /* SSL auf TCP aufsetzen */
```

```
/* GET-Request verschluesslt senden: */
 len = SSL_write(ssl, BUF, strlen(BUF));
  switch (SSL_get_error(ssl, len)) {
    case SSL_ERROR_NONE: break;
    default: err(1, "SSL_write: %s", sslerr());
  }
 /* Antwort vom Server lesen und entschluesseln: */
  do {
    len = SSL_read(ssl, retbuf, LEN - 1);
    switch (SSL_get_error(ssl, len)) {
      case SSL_ERROR_NONE: break;
      case SSL_ERROR_ZERO_RETURN: len = 0; break;
      default: err(1, "SSL_read: %s", sslerr());
    }

    if (len > 0) {
      retbuf[len] = '\0'; /* Antwort ausgeben */
      printf("Antwort: %s\n", retbuf);
    }
  }
  while (len > 0);
  SSL_set_shutdown(ssl,
SSL_SENT_SHUTDOWN|SSL_RECEIVED_SHUTDOWN);
  SSL_free(ssl);
  SSL_CTX_free(ctx);
  ERR_remove_state(0); close(sock);
}
```

```
cc ssl_sock_client.c -o ssl_sock_client -lcrypto -lss
./ssl_sock_client
```

Chapter 2

FUNCTION-ORIENTED APPROACHES

Middleware – What is that?

- Initial situation
  - Middleware germinated in the 1980s as a legacy system connection solution
  - Simplifies Distributed Processing, i.e. goal-oriented connection of numerous applications over a network
- Typical definitions
  - “Glue” between software components and the network
  - “/” (Slash) between Client/Server
  - Software platform bridging the heterogeneity of different systems and networks, which simultaneously provide a number of important system services, such as security policies, transaction mechanisms and directory services. [Schill & Spring]
- Typical forms
  - RPC Middleware
  - MOM (Message Oriented Middleware) via Message Queues
  - EAI (Enterprise Application Integration), such as CRM,ERP, HR Adapter
  - Database Middleware
  - Middleware CORBA, JavaBeans, EnterpriseJavaBeans, Microsoft COM

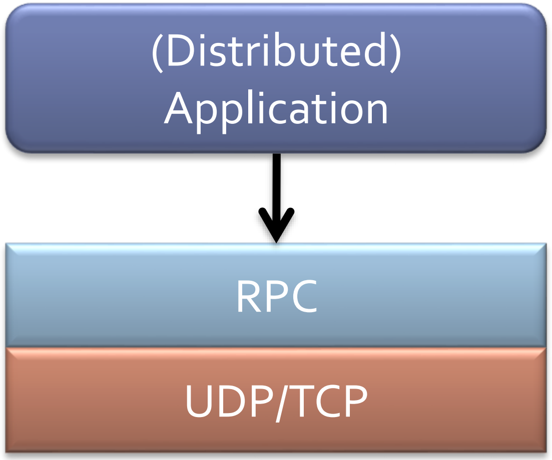
Typical Middleware Tasks

- Communication:
  - Remote Procedure Call, message queuing, peer-to-peer messaging, electronic post, general electronic data exchange

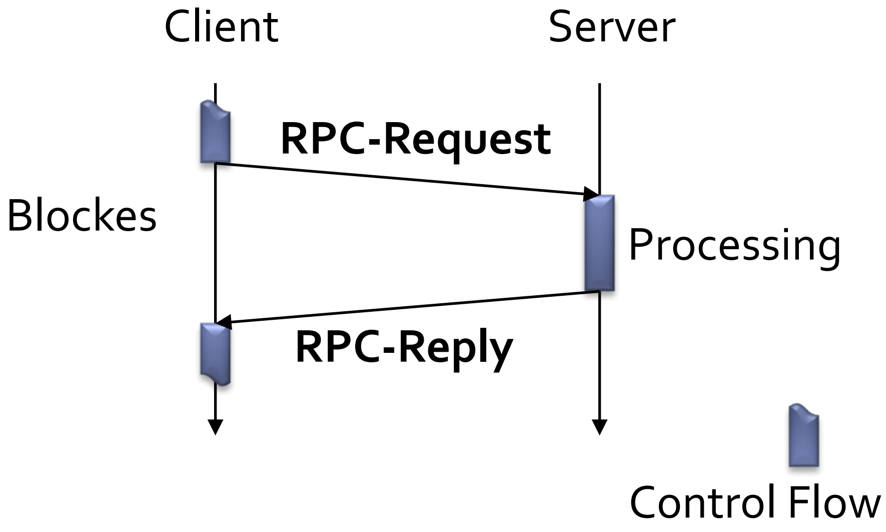
- System services:
  - Event notification, configuration management, software installation, error detection, recovery coordination, authentication, encryption, access control
- Information services:
  - Directory server, log manager, file and record manager, relational database system, object-oriented database system, repository manager
- Flow control services:
  - Mask- and graphic processing, printer management, hypermedia links, multimedia processing
- Computation services:
  - Sorting, mathematical computations, internationalization, data conversion
- Time management

Remote Procedure Call (RPC)

- **Remote Procedure Call – Idea**
  - Programming language embedding
  - Data exchange stays transparent for the programmer
  - RPC is located above UDP or TCP in the protocol stack
  - Is mostly implemented as a part of the actual application

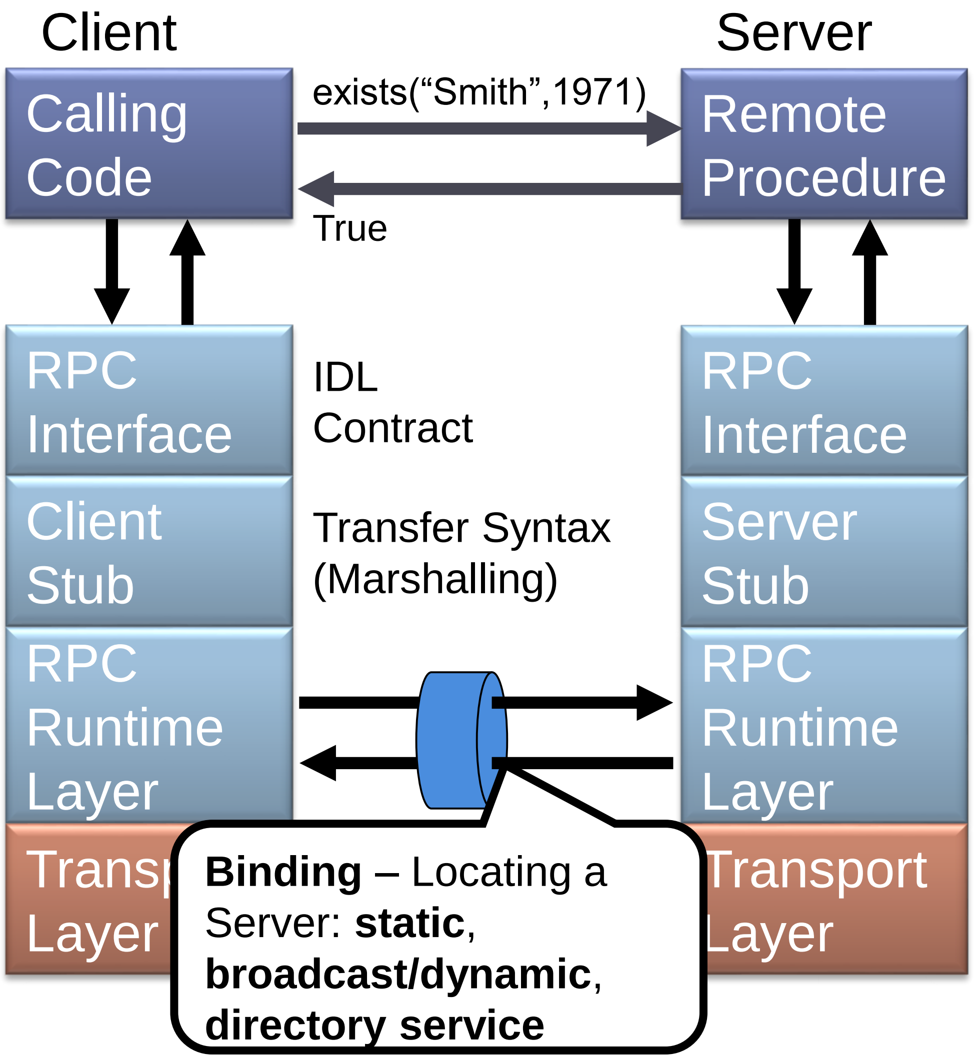


- **Execution**
  - Call in waiting state
  - Parameter- and call transfer to the target system
  - Procedure execution
  - Re-registration
  - Continuation of program execution



Remote Procedure Call (RPC)

- **Remote Procedure Call (RPC)** – enables a **synchronous call** of functionality offered in **separate processes** (possibly, on remote machines), where input and output data is exchanged over a **narrow channel** *(Nelson and Birell, 1984)*
  - Synchronous control flow transfer
  - Separate address spaces
  - Coupling over a narrow channel, which might induce errors not occurring on local systems
  - Data exchange transparency
- Stubs
  - Call encoding / result decoding
- RPC runtime system
  - Call transfer, result receipt



RPC – Execution Sketch

CLIENT PROGRAMM

1. Application program (Client)
   - calls a remote procedure (which is offered by a Client Stub or an RPC Interface) and
   - retreats to an idle state
2. Encoding by the Client Stub
   - Transformation in Transfer-Syntax (Marshaling)
3. RPC runtime system takes over the sending of created messaged to the remote system
   - RPC runtime system uses a transport protocol for message sending

CLIENT→SERVER

1. RPC runtime system forwards received messages to the Server Stub
2. Server Stub decodes the messages (un-marshaling)
3. Server Stub carries out a local procedure call with accordance to the RPC Interface.
   - Passes parameters to the actual server procedure
4. Server procedure passes the result upon termination over the RPC Interface to the Server Stub
5. Call encoding of the result by the Server-Stub
6. Server-Stub sends the created result-message to the client via the RPC runtime system

SERVER→CLIENT

1. RPC runtime system forwards data to the Client Stub
2. Client-Stub decodes the data, activates the application program and passes the result to it

RPC Semantics

- Use of remote procedure calls is prone to a wide range of errors. For instance:
  - Requests or responses are lost in transit or get falsified
  - Client or Server crash during RPC (independently of each other)
- Error handling distinguishes the following error semantics classes:

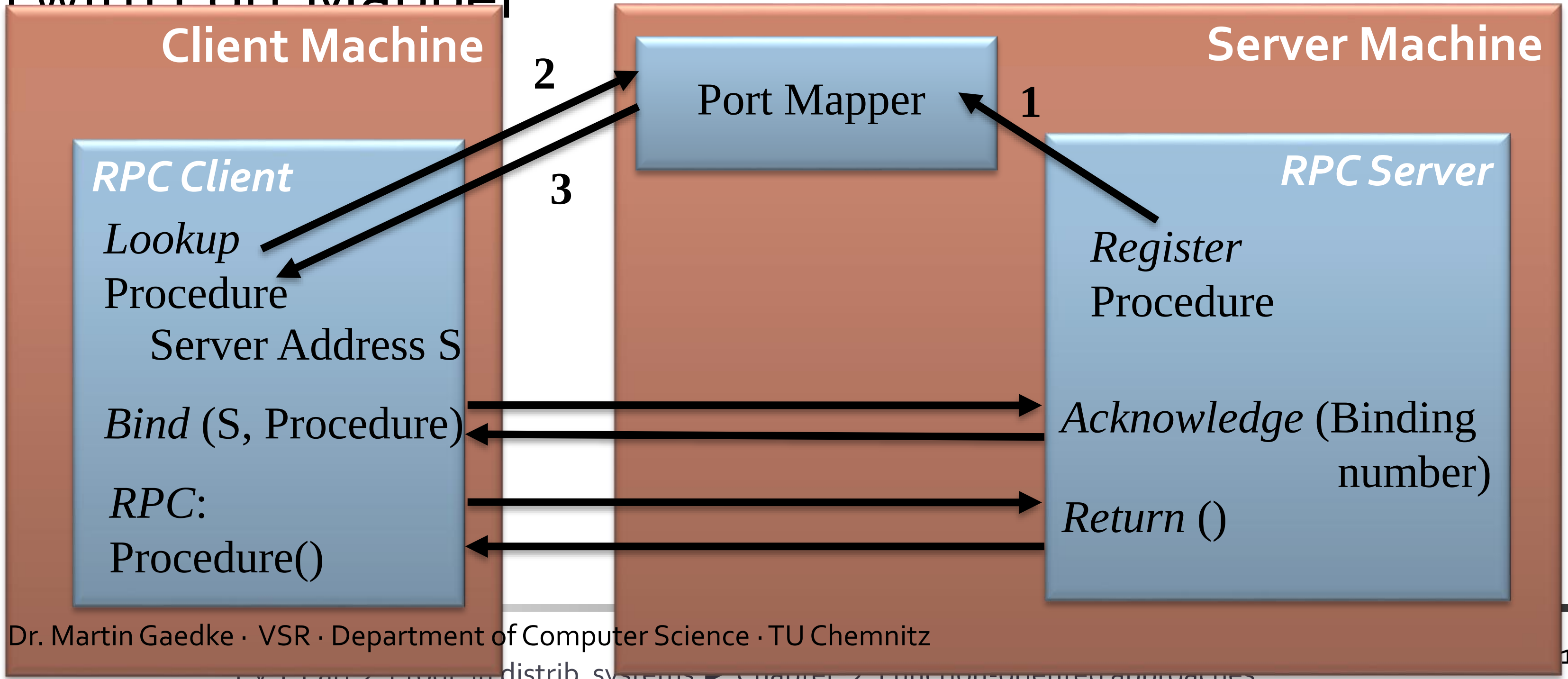
Fehlersemantik	Anfrage einer Wiederholung	Filterung Duplikate	Wiederausführung/ Wiederholung reply
Maybe	No	No	No
At-least-once	Yes	No	Re-execution
At-most-once	Yes	Yes	Reply repeat
Exactly-once	**Yes**	**Yes**	**No**

RPC Binding

- **Problem** – Find and choose an RPC server
- **Solution idea:**
  - **Bind** – Determine the complete address of the communication partner, here - RPC Server
    - Bind a procedure call to a Server
    - Executed at runtime
    - **Static binding** - at compile time, no location transparency
    - **Dynamic binding** – at runtime (via a Broker, Broadcast or other similar approaches)
  - **Execution**
    - **Register (Server)**
      Server or process on a Server makes itself known
    - **Lookup (Client)**
      Client identifies a Server
    - **Bind (Client)**
      Determine context (dynamic, static)
    - **RPC Call (Client)**
      Client engages with the Server

Dyn. Binding: Example SUN-RPC

- Port-Mapper
  - Mapping of RPC program numbers to protocol ports
  - Mapping is stored in a local database
  - Is located on the same machine as the server procedure
  - No location transparency
    - directory service would be required
- Execution with Port Mapper



RPC Interface Definition

- For remote procedure calls a uniform description of procedures and parameters is required
  - Requirement: **Interface Definition Language**
    - Formal language for describing procedures or signatures
    - Signature: Name, in-/output parameters, exception handling
  - Stubs can be automatically generated based on interface definition languages
- Examples
  - **IDL (Interface Definition Language)**
  - ASN.1 (Abstract Syntax Notation Number 1)
  - XDR (eXternal Data Representation)

RPC: Interface Definition Language

- **Interface Definition Language (IDL)** – Interface description in use by many RPC-systems or RPC-based systems
  - Enables Endpoint Definition
  - Programming language-independent approach
  - IDL generator creates the according Client and Server stubs in different languages IDL
- Problem: IDL ***compiled away***!
  - Once the IDL description enters the code, it becomes unreadable by other programs at a later point in execution
- IDL Example here for ONC RPC

- IDL Example **mylist.x**

```
struct node {
  int val;
  struct node *next;
};


typedef struct node *node_p;

program MYLIST {
  version BASVERS {
    void create_nodes(int) = 1;
    node_p get_list(void) = 2;
  } = 1;
} = 0x20000001;
```