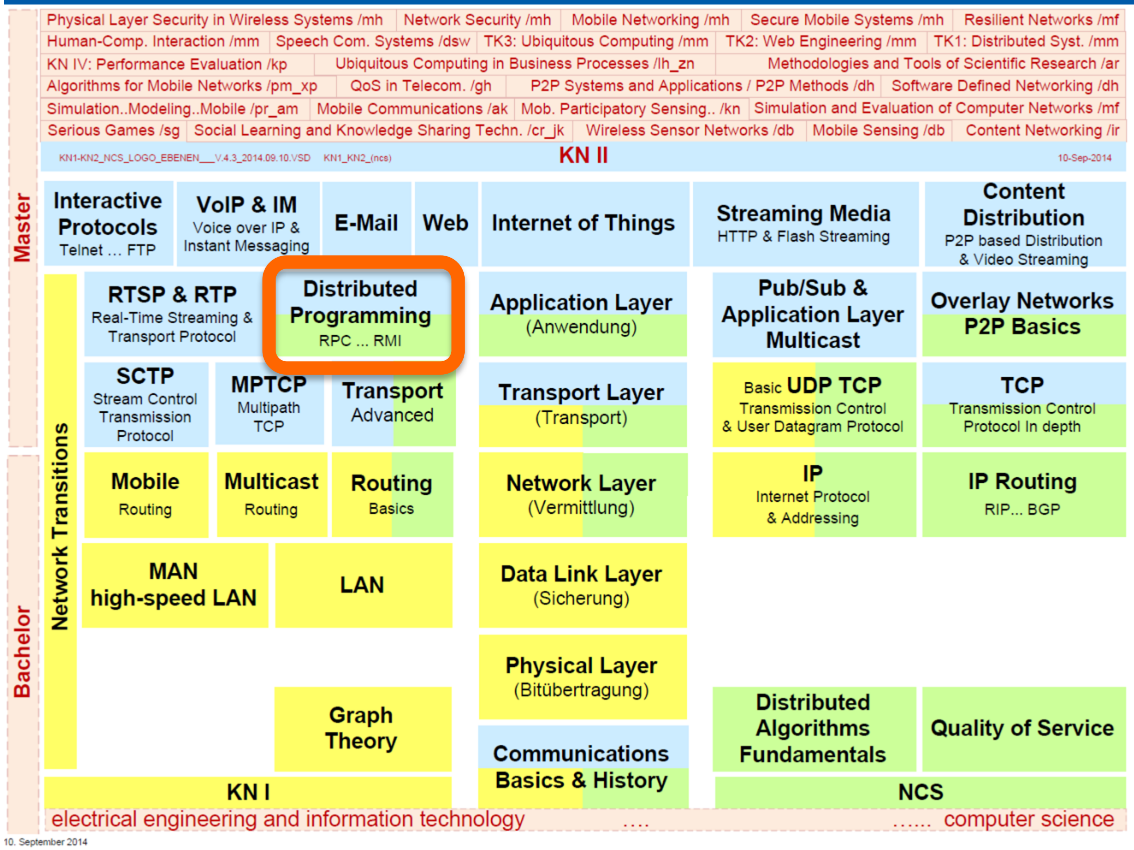
# Communication Networks II

## Distributed Programming

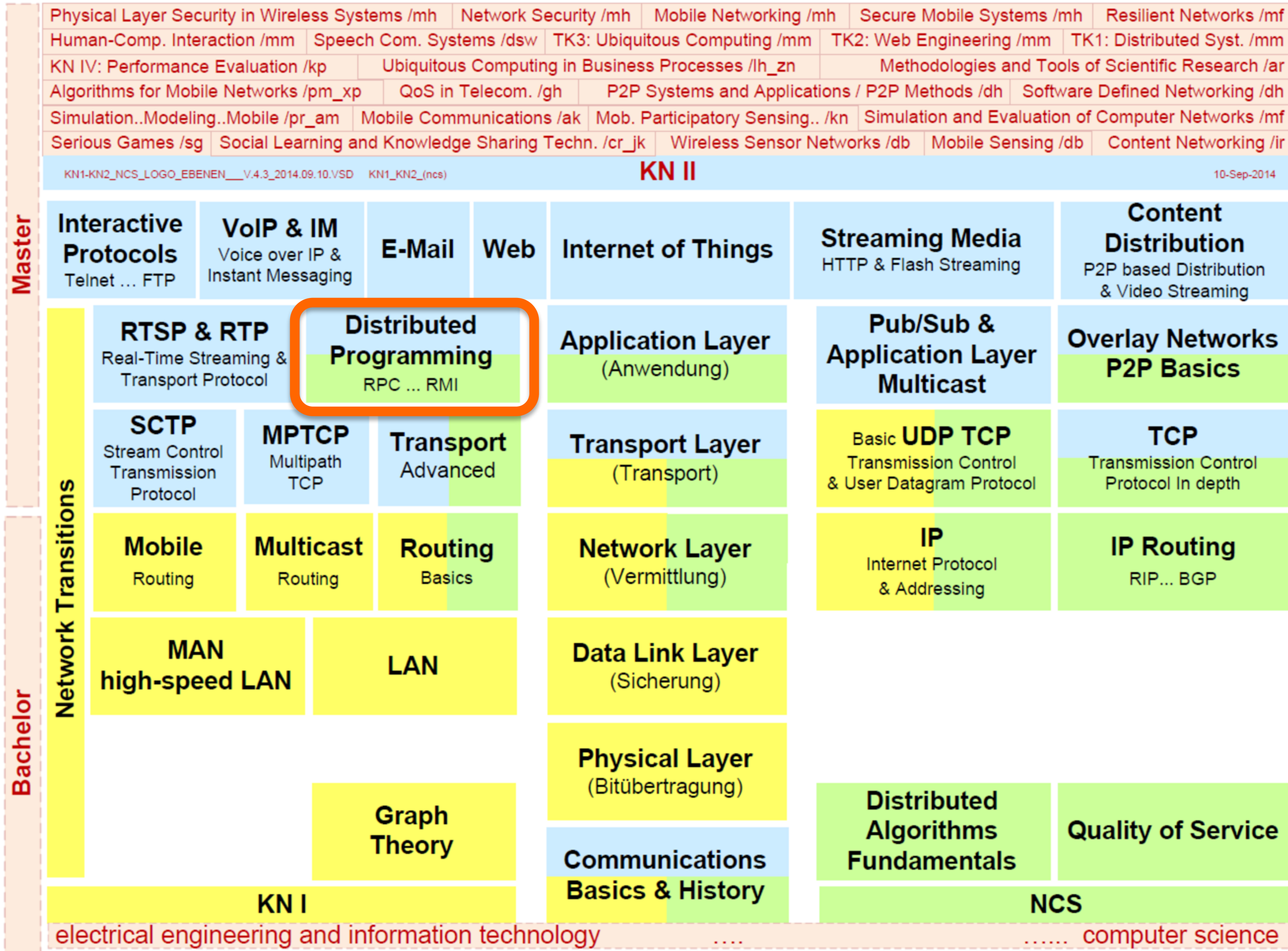




Prof. Dr.-Ing. **Ralf Steinmetz**
KOM - Multimedia Communications Lab

# Overview

TECHNISCHE UNIVERSITÄT DARMSTADT


Physical Layer Security in Wireless Systems /mh
Network Security /mh
Mobile Networking /mh
Secure Mobile Systems /mh
Resilient Networks /mf
Human-Comp. Interaction /mm
Speech Com. Systems /dsw
TK3: Ubiquitous Computing /mm
TK2: Web Engineering /mm
TK1: Distributed Syst. /mm
KN IV: Performance Evaluation /kp
Ubiquitous Computing in Business Processes /lh_zn
Methodologies and Tools of Scientific Research /ar
Algorithms for Mobile Networks /pm_xp
QoS in Telecom. /gh
P2P Systems and Applications / P2P Methods /dh
Software Defined Networking /dh
Simulation..Modeling..Mobile /pr_am
Mobile Communications /ak
Mob. Participatory Sensing.. /kn
Simulation and Evaluation of Computer Networks /mf
Serious Games /sg
Social Learning and Knowledge Sharing Techn. /cr_jk
Wireless Sensor Networks /db
Mobile Sensing /db
Content Networking /ir
KN1-KN2_NCS_LOGO_EBENEN___V.4.3_2014.09.10.VSD
KN1_KN2_(ncs)
KN II
10-Sep-2014
Master
Bachelor
Network Transitions
Interactive Protocols
Telnet … FTP
VoIP & IM
Voice over IP & Instant Messaging
E-Mail
Web
Internet of Things
Streaming Media
HTTP & Flash Streaming
Content Distribution
P2P based Distribution & Video Streaming
RTSP & RTP
Real-Time Streaming & Transport Protocol
Distributed Programming
RPC ... RMI
Application Layer
(Anwendung)
Pub/Sub & Application Layer Multicast
Overlay Networks
P2P Basics
SCTP
Stream Control Transmission Protocol
MPTCP
Multipath TCP
Transport
Advanced
Transport Layer
(Transport)
Basic UDP TCP
Transmission Control & User Datagram Protocol
TCP
Transmission Control Protocol In depth
Mobile
Routing
Multicast
Routing
Routing
Basics
Network Layer
(Vermittlung)
IP
Internet Protocol & Addressing
IP Routing
RIP... BGP
MAN
high-speed LAN
LAN
Data Link Layer
(Sicherung)
Physical Layer
(Bitübertragung)
Graph Theory
Communications Basics & History
Distributed Algorithms Fundamentals
Quality of Service
KN I
NCS
electrical engineering and information technology
….
…… computer science

## Motivation for building distributed systems:

### Scalability

- A single host can handle only a limited load
- If the load gets higher → split the system in components and distribute them

### Openness

- Communication to 3rd party systems is required
- E.g. online shop with Paypal integration

### Heterogeneity

- Systems grow over time
- Old hardware and software components get unavailable and must be replaced with new components

### Fault tolerance

- The question is not if failures happen – the question is when…
- The system should be available even if some components fail

## Our Understanding of "Distributed System"

**"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable". (Leslie Lamport, 1987)**

## Properties

- Composed of several distributed, autonomous components
- Components are linked by a computer network
- Typically, components do not use same storage space
- Components have to communicate by exchanging messages
- Components cooperate to achieve a common goal
- Largest distributed system: World Wide Web

## Goals

- Parallelization (e.g., parallel processing of customer data)
- Load-balancing (e.g., subdivide workload into different tasks)
- Fault tolerance, resilience, availability (e.g., replicate critical components)
- Scalability (e.g., increase capacity by adding additional components)

## Basic Concepts

- Name and Directory Services
- Distributed Transactions
- Security (Authentication, Authorization, Encryption)

- Transparency (specified in reference models: ANSA, ISO)
  - Location
  - Access
  - Failure
  - Concurrency
  - Replication
  - Mobility [ANSA] / Migration [ISO]
  - Scalability
  - Performance

## Typical Approach

- Middleware solutions, which implement the concepts above

## Access Transparency

- Remote methods should look like local methods from the clients point of view
- The interface for local and remote method calls should be equal

## Location Transparency

- A location transparent name contains no information about the named object's physical location
- E.g. visiting www.tu-darmstadt.de via Web-Browser is location transparent
  - (human) visitor does not need to know the IP address or physical location of the hosting server

## Mobility / Migration Transparency

- An object could be moved from one host to another without affecting other components

## Replication Transparency

- Other components(=users) do not need to care whether an accessed object is the original or a replica

# Transparency Concepts

## Concurrency Transparency

- Users and Applications should be able to access shared data or objects without interference between each other

## Failure Transparency

- Enables the concealment of faults, allowing user and application programs to complete their tasks despite the failure of hardware or software components
- This rather abstract criterion is supported by the replication transparency and by the location transparency

## Performance Transparency

- Allows the system to be reconfigured to improve the performance as the load varies
- E.g. any component should respond as fast as possible from the clients point of view

## Scaling Transparency

- A system should be able to grow without affecting application algorithms
- E.g. the Internet scales (nearly) transparent because the addition of new hosts does not affect existing nodes

TECHNISCHE UNIVERSITÄT DARMSTADT

## Distributed operating system

- Support for distributed programming is part of operating system
  - Pro: Quite general solution
  - Con: Needs wide-scale adoption of the same system
    Large systems always heterogeneous

## Distributed database approach

- Same as above, except OS replaced by a database system
  - Pro: Allows for all features of databases (semantics, etc.)
  - Con: Independent applications with shared database
  - Con: Many distributed algorithms hard to realize in this case

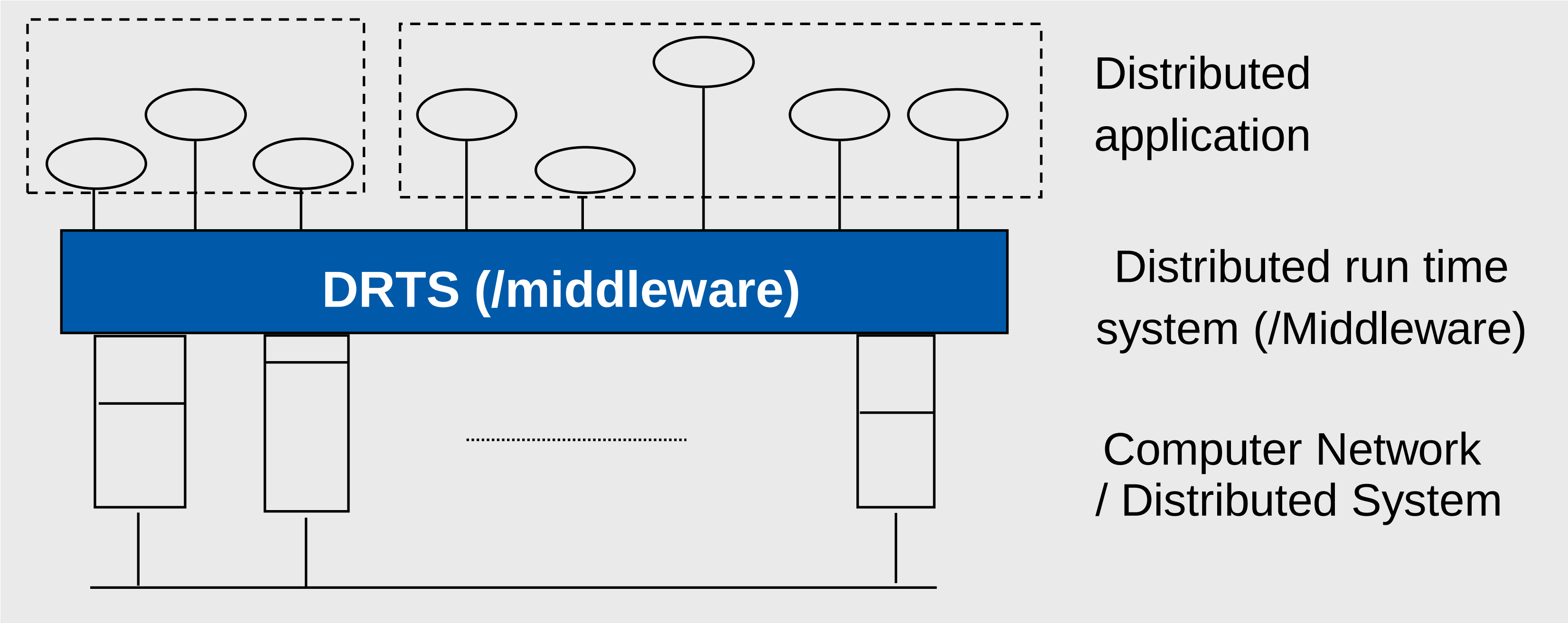
## Protocols

- Standardized APIs for connecting to servers (e.g., HTTP)
  - Pro: Open, global
  - Con: Limited to standard functionalities

# Our Model: Programming Language

## Arbitrary, heterogeneous OSes (and databases)

## Distributed programming language




## Modern approach:

- Sequential programming language + extensions + middleware
- Compiler will not see the distributed program

## Architectural Model defines

- Placement of <u>components</u> across a computer network
- Way of <u>interaction</u> between components

## Classification of Components

- Necessary to
  - Define responsibilities
  - Assess workloads and impact of failures
- Component types: server, client, peer

## Interaction Models

- Synchronous communication
  - Transmitted messages are received within known period of time
  - Requires synchronization between sender and receiver
- Asynchronous communication
  - Requires no assumptions of execution time intervals

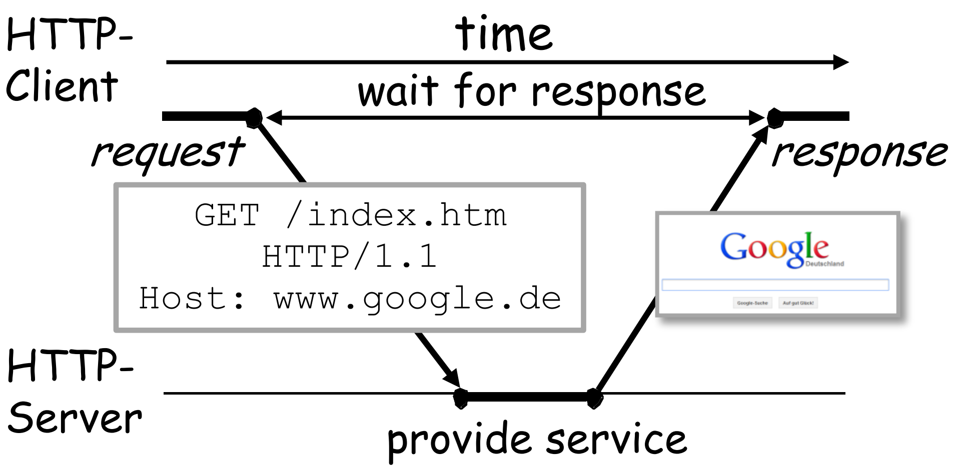
# 2.1 Client/Server Model

## Components

- Client: requests a service
- Server: provides a service

## Typical Interaction

- Synchronous communication
- Client:
  - Sends request
  - Waits for server response
- Server:
  - Receives request
  - Processes request
  - Sends response




## Examples:

- Web Client/Server
- Remote Procedure Call (RPC)

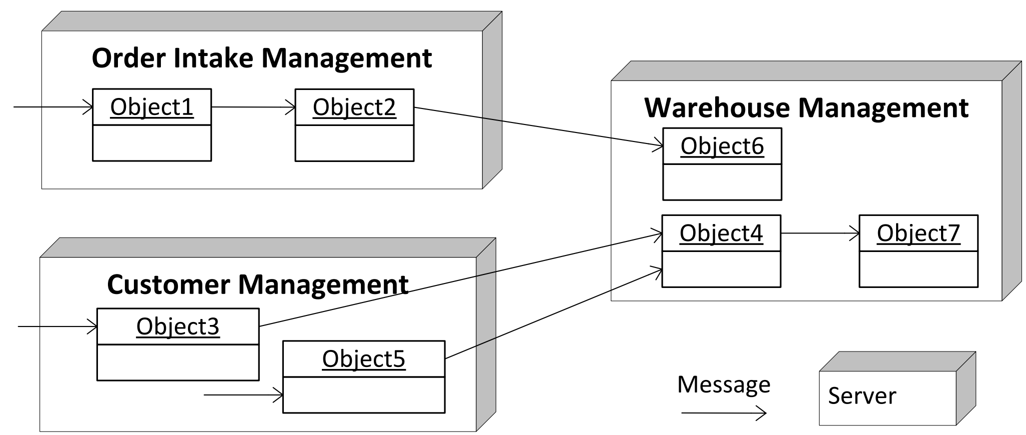
TECHNISCHE UNIVERSITÄT DARMSTADT

## Components

- Based on client/server model
- But objects provide/request services

## Typical Interaction

- Objects exchange messages
- Local and remote object calls
- Objects as input parameters
  - By Value
  - By Reference




## Examples:

- Remote Method Invocation (RMI)
- Common Object Request Broker Architecture (CORBA)

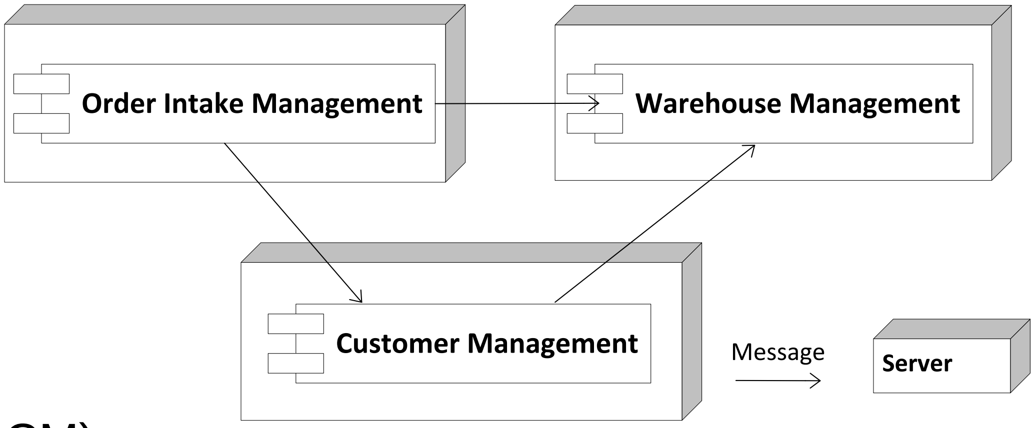
TECHNISCHE UNIVERSITÄT DARMSTADT

## Components

- Idea: separation of concerns
- Decompose application functionality into reusable, functional components
- Add properties for distributed systems at deployment time, such as
  - Persistence
  - Security
- Specific runtime environment realizes deployment properties

## Typical Interaction

- Components expose well-defined interfaces
- Depend on a specific runtime environment




## Examples:

- Enterprise JavaBeans
- CORBA Component Model (CCM)

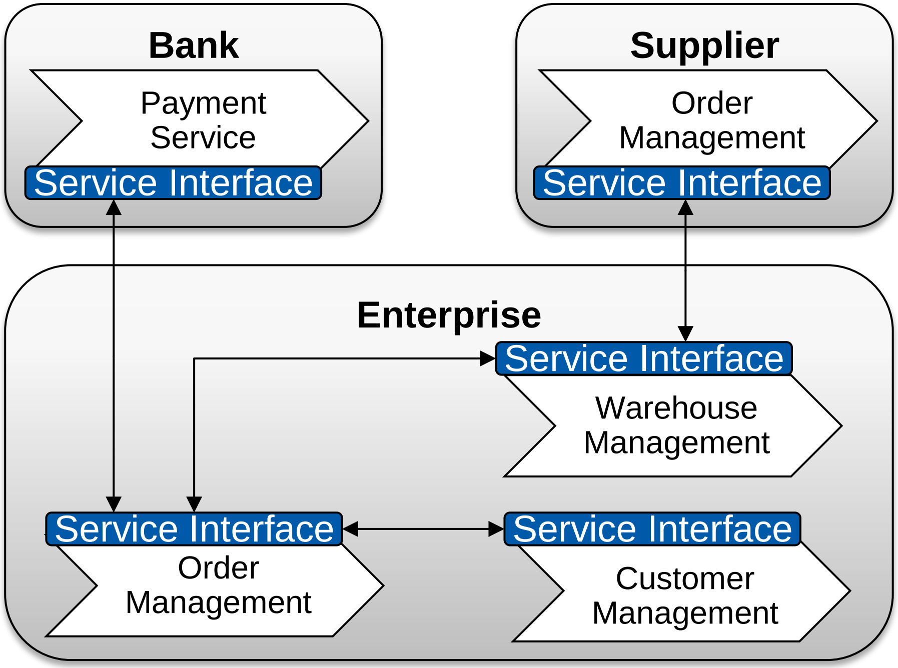
# 2.4 Service-oriented Model

## Components

- Coarse-granular services
- Encapsulate a specific functionality

## Typical Interaction

- Services expose well-defined interfaces
- Loose coupling of services to build complex workflows
- Interoperability across platforms and organizations



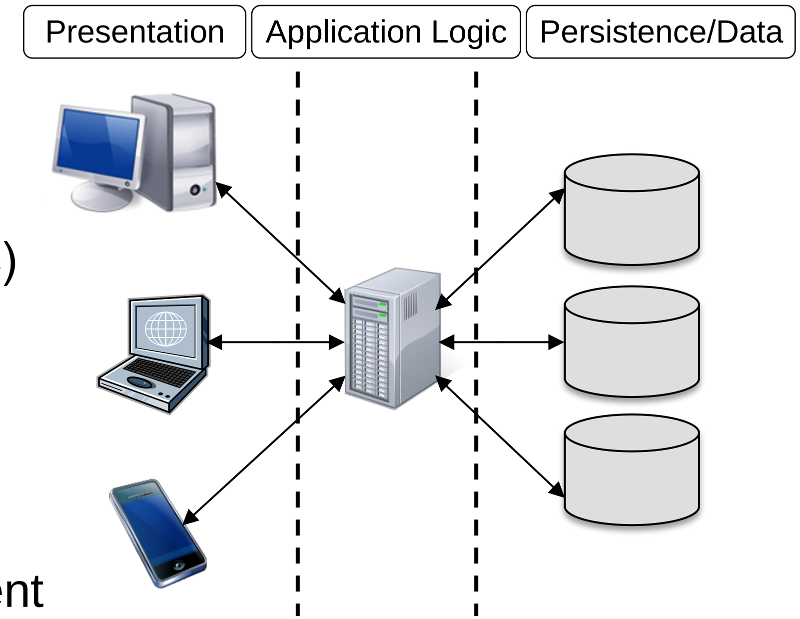

## Examples:

- Web Services
- Service-oriented Architectures (SOA)

# 2.5 Multi-Tier Architectures

## Two-Tier Architecture

- Client/server-based
- Three layers mapped on two components
- Example:
  - “Fat” client (presentation, application logic)
  - Server (persistence/data, e.g. DBMS)
  - → DBMS could not be easily replaced




## Three-Tier Architecture

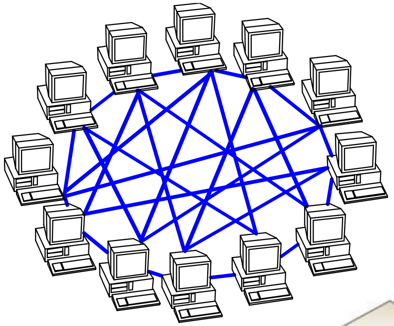
- Each layer mapped onto separate component
- Different layers expose well-defined interfaces
- Functionality of each layer can be modified without affecting another layer

## Real World: N-Tier Architectures

- Typically you will find more tiers (e.g., 2nd layer split into multiple tiers)
- Systems are realized using specific application servers/middleware

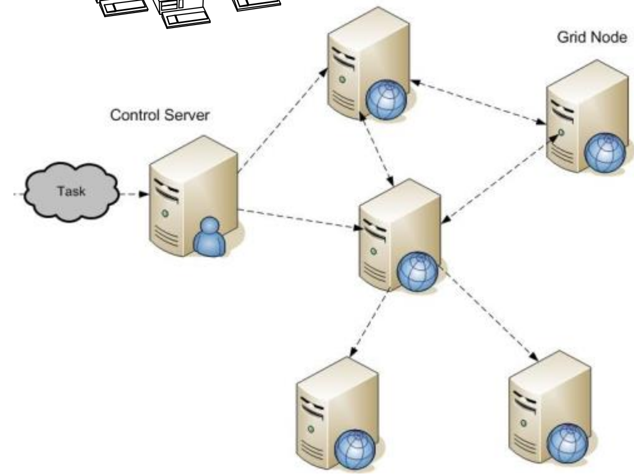
## Peer-to-Peer Computing

- Combined client and server functionality
- Direct interaction between peers
- No centralized usage/provisioning of a service



## Grid Computing

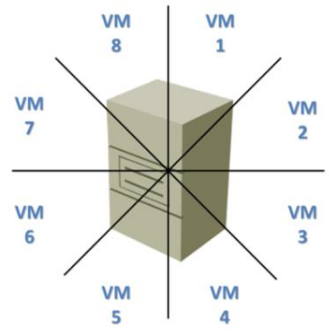
- Aggregation and shared usage of heterogeneous, interconnected resources
- High-end computers with standard OS
- Usually applied to solve large problems
- Multiple owners/decentralized administration




http://dame.dsf.unina.it/images/grid01.jpg

## Cloud Computing

- On-demand service provisioning (“computing as a utility”)
- Shared pool of configurable resources using virtual machines
- Rapidly provisioned with minimal management effort
- Single owners/centralized administration




www.definethecloud.net

## Characteristics

- Two message types for communication between processes: send and receive
- A queue is associated with each message destination
  - Send causes message to be added to remote queue
  - Receive causes message to be removed from local queue
- Communication may be either synchronous or asynchronous

## Synchronous

- Sender and receiver synchronize at every message
- Send and receive are blocking operations
  - Sending process is blocked until corresponding receive is issued
  - Receiving process does not proceed until remote data is arrived

## Asynchronous

- Send is a non-blocking operation, i.e., transmission in parallel with sending
- Receive is blocking or non-blocking (process proceeds after issuing a receive)

## 3.1 Message Destinations

**Messages are sent to (Internet address, local port) pairs**

**Local port**

- Is message destination within a computer (specified by int number)
- Has exactly one receiver (except for multicast ports)
- Can have many senders

**Processes may use multiple ports to receive messages**

**Servers publish their port number for use by clients**

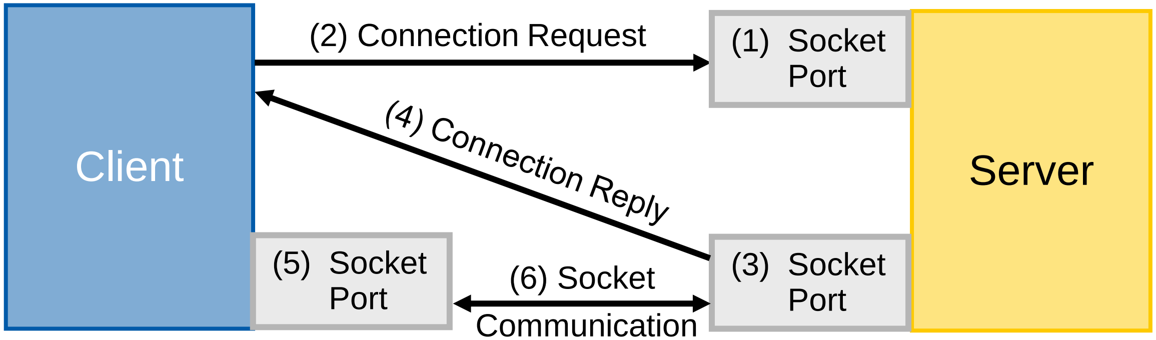
**Some port numbers officially assigned to different services (IANA)**

| Port | Description |
|---|---|
| 20/21 | File Transfer Protocol (FTP) |
| 25 | Simple Mail Transfer Protocol (SMTP) |
| 53 | Domain Name System (DNS) |
| 80 | Hypertext Transfer Protocol (HTTP) |

# 3.2 Sockets

TECHNISCHE UNIVERSITÄT DARMSTADT

- Endpoints of a bidirectional communication channel between client and server
- Different socket types for different underlying protocols (e.g., TCP/IP)
- Foundation for more complex mechanisms (e.g., RPC, RMI, CORBA)




## Typical Interaction

- (1) Server opens a server socket to provide a specific service
- (2) Client sends a connection request to the server socket
- (3) Server accepts the client request, opens a new socket, and sends a response
- (4) Client receives and processes the server response
- (5) Client opens a socket for communication
- (6) Client and server communicate via sockets

## User Datagram Protocol (UDP)

## Transmits datagrams without acknowledgments or retries

## Sockets API for UDP

- Receiver creates socket and binds to it (local address, local port)
- Sender sends from any socket, specifies destination in datagram

- Synchronization
  - Send is non-blocking
  - Receive is blocking, gets messages from local queue

## Reliability of UDP

- Integrity: checksums to detect corrupt packets
- No guaranteed delivery, ordering or duplicate detection

## Usage examples:

- DNS, VoIP, RTP

# Transmission Control Protocol (TCP)

## TCP abstraction: stream of bytes („pipe“);

- To which data may be written and
- From which data may be read

## Stream abstraction hides TCP/IP internals such as

- Lost messages, duplicates, ordering, flow control
- Establishing a connection between communicating processes before they can communicate over a stream

## Reliability of TCP

- Integrity: checksums/seq. numbers to detect corrupt/duplicate packets
- Validity: timeouts and retransmissions to handle lost packets
- Limitation: if a connection breaks before an explicit close operation

## Usage Examples: HTTP, FTP, Telnet, SMTP

# Simple Example: TCP/IP Echo Server

TECHNISCHE UNIVERSITÄT DARMSTADT

## Client

```
Socket sock = new Socket(192.168.2.135,1234);

DataInputStream in = new DataInputStream(sock.getInputStream());
DataOutputStream out = new DataOutputStream(sock.getOutputStream());

out.writeUTF("Hello");
String data = in.readUTF();
System.out.println("Received: " + data);
```

## Server

```
ServerSocket listenSock = new ServerSocket(1234);

for(;;){
 Socket sock = listenSock.accept();

 DataInputStream in = new DataInputStream(sock.getInputStream());
 DataOutputStream out = new DataOutputStream(sock.getOutputStream());

 String data = in.readUTF();
 out.writeUTF(data);
 sock.close();
}
```

TECHNISCHE UNIVERSITÄT DARMSTADT

**One example of above is RPC**

**Fundamental idea behind RPC:**

**Processes can call procedures on other computers**

**Goal: Syntactic and semantic uniformity of local and remote calls in terms of**

- Call mechanism
- “Expressive power“ of language
- Error handling

**→Goal cannot be fully achieved…**

**Regardless of problems, RPC is widely used in the computation world**

## Other goals

- Distribution transparency
- Simplicity (no message conversion/serialization, no packing, no ACKs, ...)
- Type safety (type checking for parameters at client, server)

## Definition (Nelson, 1984):

- Synchronous control flow handshake
- Appears to be at level of programming language
- Separate address spaces
- Connected via “narrow-band“ channel
- Data passing (message exchange): call parameters results

# RPC: “Political“ Issues

## RPC has been successful

## RPC presented/marketed in a good way

- “As easy as what-you-know” (procedural programming)
- Virtually not different from what-you-know
- But: Many subtle differences make it much different and rather difficult

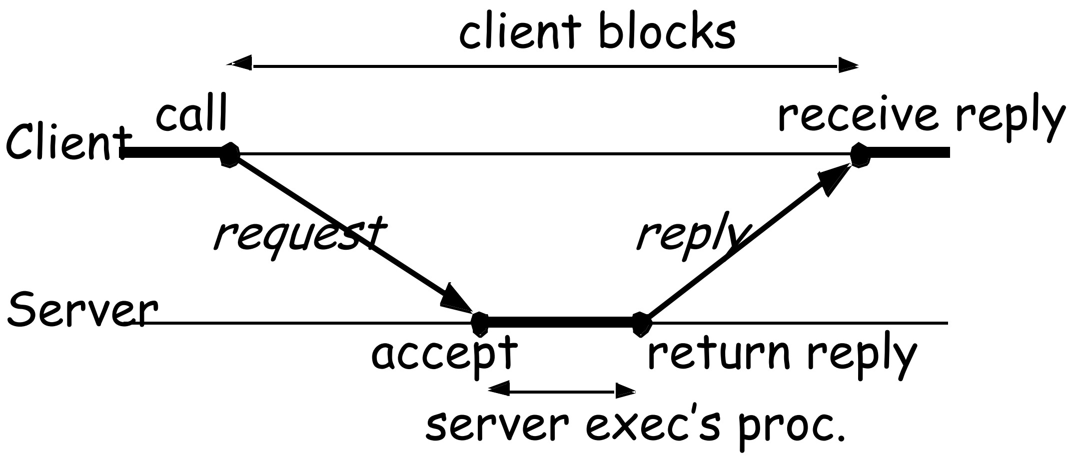
## Basis for client/server computing

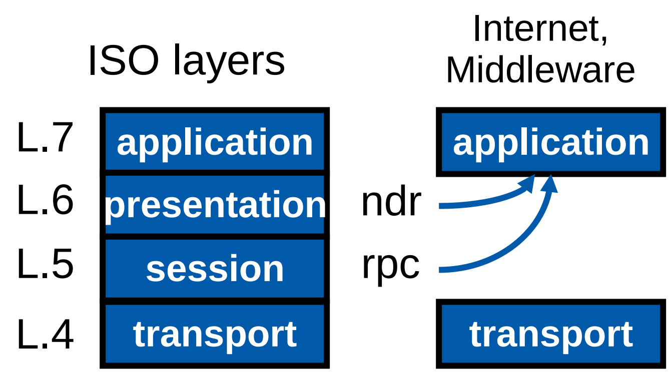
- Popular in commercial context
- Easy to model and understand, suitable for mid-size problems
- But: in many applications today, client/server model is reaching its limits!
- But: Does not (easily) scale

## Sold at right time, together with platforms

- True, but that has nothing to do with RPC in particular

# 4.2 RPC: Principle




ndr: network data representation
(not a common term)
rpc: rpc protocol

**Code in client:**

```
Set out-parameters
Call X(out-parameters,result)
Use result
```

**Code on server:**

```
Proc X(parameters)
Do stuff
Return (result)
```

# RPC: Simple Example

Client

```
int i = 1, j = 2;
int res;

res = RPC_add(i, j);

printf("Result %d\n", res);
```

Server

```
int RPC_add(int a, int b) {
   int res;

   res = a + b;

   return res;
}
```

**Client calls RPC_add like any other function**

- Function RPC_add is implemented in server like any other function
- Client does not see any difference between RPC and local functions (at least in principle)
- Code for communication is hidden in middleware

**Problems:**

- How does the call to RPC_add find the right server?
- How do we know everything works as planned?

# RPC: Basic Properties

**Synchronous communication**

**Only 1 call needed to access remote procedure**

- Other approaches (e.g., IPC) require more

**System takes care of all “small details”**

- Message assembly and disassembly, etc.

**Complexity same as normal procedure call**

- Only one call in progress at a time

**Transparent to distribution**

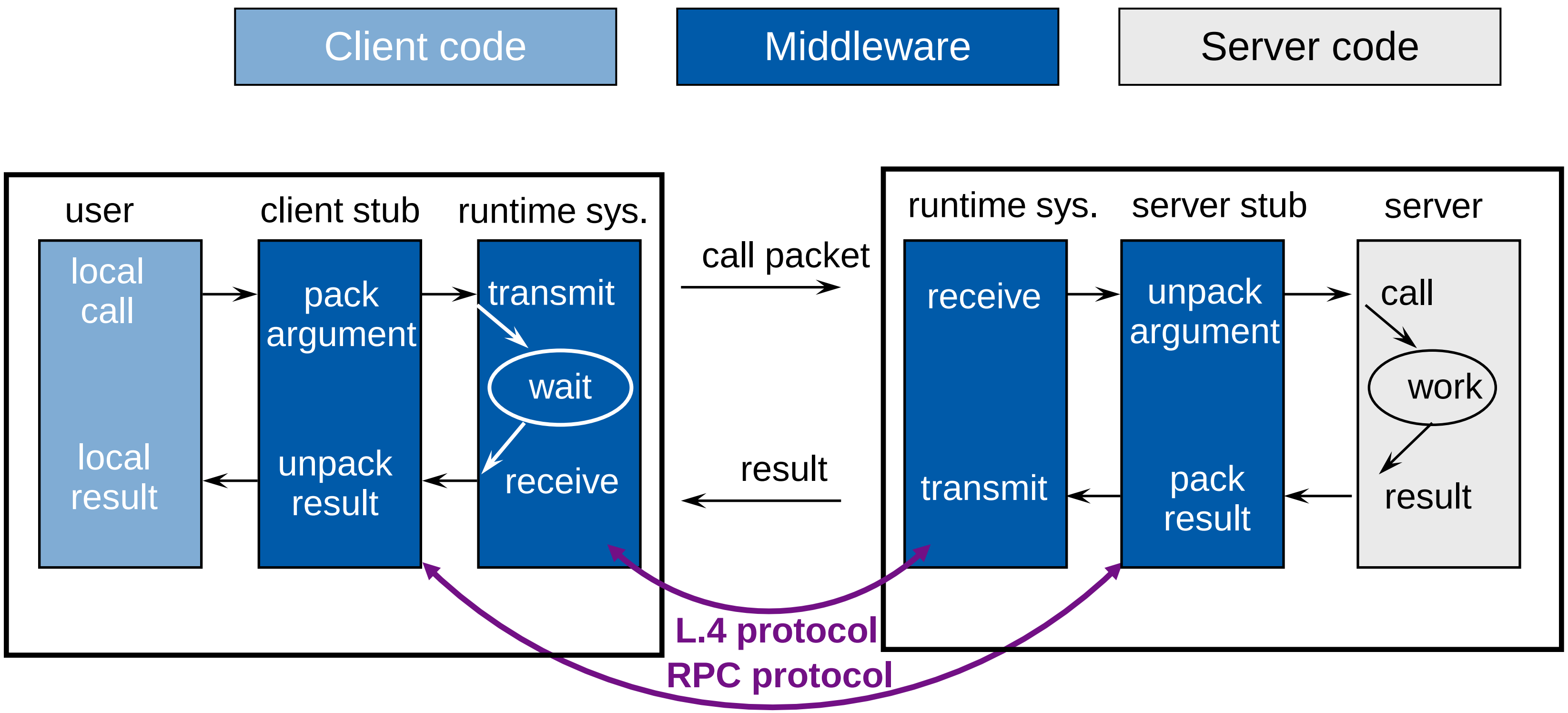
- As long as client can find server, it does not matter where it is

# RPC: Implementation / Steps

## Server must implement "dispatcher":

- Dispatcher waits for incoming requests
- When request comes, invokes/dispatches the correct procedure

## Flow of control:



TECHNISCHE UNIVERSITÄT DARMSTADT

## Stubs:

- Mimic local procedure call, hide "networking"
  - Call remoteserver.X (out: a; in: b); → extended to client-stub code
- Pack / unpack messages (call, reply, ...):

  **→ Marshalling**
- May convert to / from network data representation
- Support mapping from client to server:

  **→Binding**
- Carry out RPC protocol

**Client stub is proxy for server at client side**

**Server stub(skeleton) is proxy for client at server side**

TECHNISCHE UNIVERSITÄT DARMSTADT

## Automatic generation of stub code: stub compiler

## Basis:
→ **IDL: Interface definition language**

## One middleware-specific extension to many programming languages

- Corba-IDL, DCE-IDL, SUN's XDR (external data rep.), Mach's Matchmaker

## Internet-wide: XML

## Stub/IDL compiler knows types

- used in host programming language
  (if strongly typed language)

## May use libraries

- in order to insert code
  for format conversion, marshalling, transmission control, ...

# 4.3 RPC: Marshalling and Presentation

**Marshalling challenge:**
**Flatten (serialize) complex data structures**

- Basic data types *plus* structural info

**Accomodate presentation**
**in different OSes, languages, and hardware architectures:**

- Integer (size?, 1's /2's complement?) / float/real (size?, IEEE?, ...)
- Character (ASCII, unicode?) / string (`\0´ end-flag or byte-count?)
- Arrays (row- /column-based), struct/union/set... (organization?)
- Little-endian, big-endian, bit order (MSB→LSB or inverse)

**Worst-case: n systems → need ~$n^2$ conversions**

**Reality: Often 2 possibilities (see below)**

**OSI model devoted layer 6 to this problem**

- Never widely used, only ASN.1 and BER existed

# RPC: Marshalling and Presentation

**Two possibilities from above:**

**"Receiver makes it right"**

- Mark representation type
- Between client and server with same representation, no need for translation (80% of the cases?)

**Abstract syntax (IDL, or ISO ASN.1) plus standardized network data representation**

- ISO-ASN.1 (abstract syntax notation #1): called BER (basic encoding rules)
- SUN-XDR (same name as for IDL), adopted by DCE
- Corba: CDR (common data representation)
- Java-RMI: "Java serialized form"
- XML: SOAP, XMLP

TECHNISCHE UNIVERSITÄT DARMSTADT

## Corba IDL & CDR example:

```
struct Person {
        string name;
        string place;
        long year;
};
```

| index in sequence of bytes | ← 4 bytes → | notes on representation |
|---|---|---|
| 0–3 | 5 | length of string |
| 4–7 | "Smit" | 'Smith' |
| 8–11 | "h___" | |
| 12–15 | 6 | length of string |
| 16–19 | "Lond" | 'London' |
| 20-23 | "on__" | |
| 24–27 | 1934 | unsigned long |

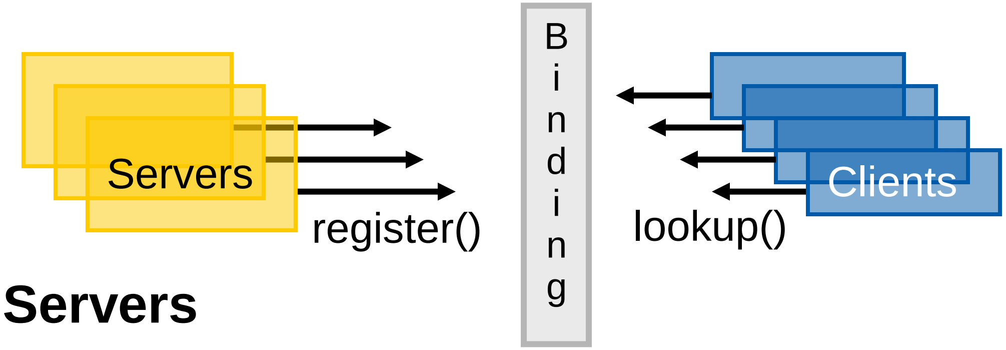
struct with value: {'Smith', 'London', 1934}

```
...
void CreateNewEntry
      (in Person newMember,
              out int MemberNo);
...
```

Note: BER (for ASN.1) more sophisticated:

- Codes basic, constructed types via tags
- Codes user-defined types (cf. Person)
- Considered unnecessary in IDLs since: both sides know IDL → know "what comes next" in marshalled messages
- Corba IDL uses type tags where needed, e.g., for "union"

# 4.4 RPC: Binding




**Binding matches Clients and Servers**

- Clients, servers developed independently

**Binding may distinguish just name of server or up to program, version, protocol**

**Three possibilities**

- Static (at compilation time): fast, no middleware overhead
- Semi-dynamic (at startup time): logical name/ DB/ multicast/ service
- Dynamic (= per call), as semi-dynamic plus: fault tolerance, load balancing

**Binding via intermediate service (“trader”, “broker”):**

- Can become a bottleneck
- Expensive (execution time, triangle communication)
- “Yellow pages” (search via attributes, description)

# RPC: Binding Examples

## Binding for Internet-RPC (e.g., based on SUN-RPC / XDR):

- Port mapper
  rpcbind: name service for RPC servers on server node: port #111
  - Server registers program & version numbers with local port mapper
  - Check out all registered RPC servers with: rpcinfo –p
  - When client calls clnt_create, RPC request is sent to server's port mapper asking info for given program, version, & protocol
  - RPC request returns server's port number to client

## For Java: "you're supposed to know your URLs"

## For Corba:

- May use naming service to translate logical name
- May use trader service for yellow pages

# 4.5 RPC: Protocols

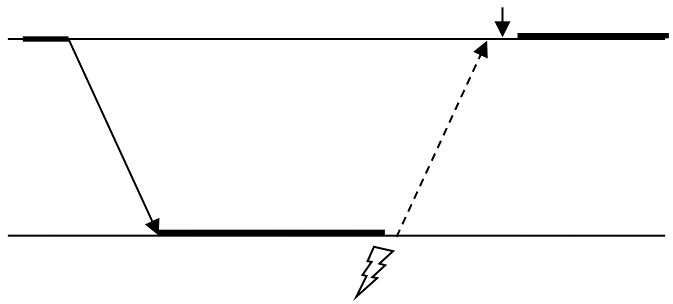
## Underlying Layer 4 protocol:

- Connectionless (e.g., UDP): (normally used in Internet)
  - Pro: Need no L.4 acks since we have reply anyway
  - Con: Max. packet size not good for large objects
- RPC-optimized L.4 protocols may provide **implicit** (dis)connect

## RPC protocol:

- Distinguish two kinds of RPC protocols
  - **RR**: request-reply
  - **RRA**: request-reply-ack (ACK by client)
- Must realize three functions:
  - client: **do_operation** (callee, op_id, *in_args)
  - server: **getRequest**(...) and **sendReply** (caller, *results)
- Possible errors:
  - Request omission, reply omission (both mean lost messages)
  - Server crash, client crash

## Crashes



**Server crash: How far did it get?**

- How to distinguish from omissions (n timeouts?)
- If other server takes over, how should it know?
- For restart, how to cope with locks, dirty states, info recovery?

**Client crash: Server executes "orphan"**

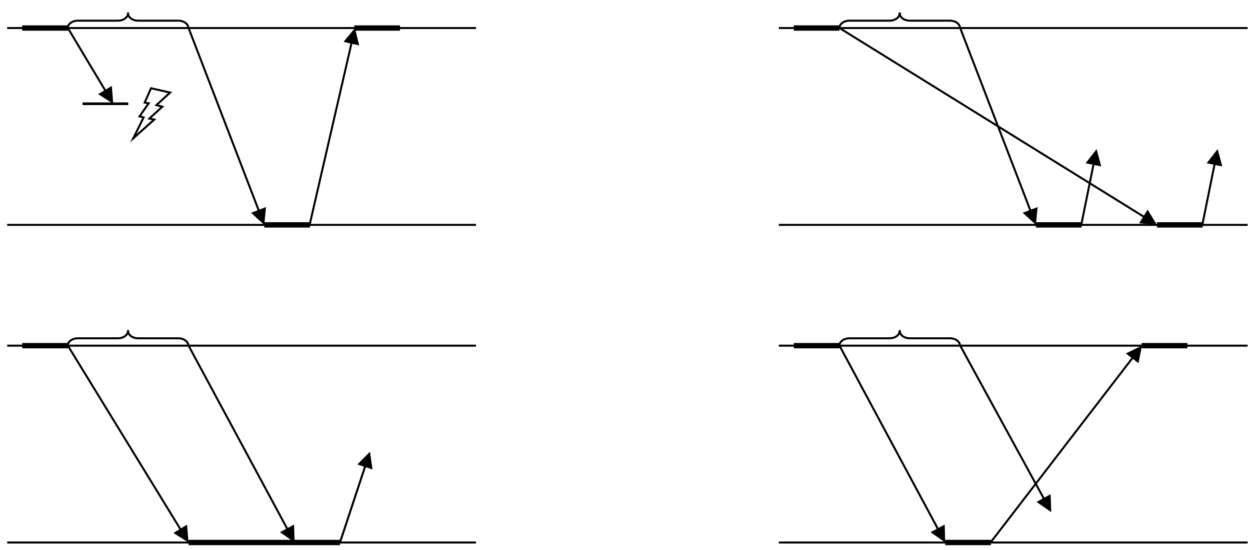
- Procedure execution maybe erroneous, "costly", ...
- Restarted client should not be puzzled by orphan-results
  - Might even tell server to stop them?
- Note: For lengthy procedures, server may poll client

# RPC Errors: Request Omission

**Basic countermeasure: Client sets timeout**

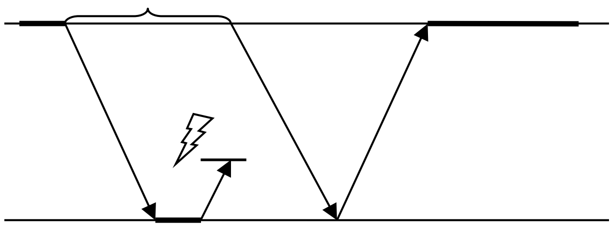
**If no answer within timeout, re-send request**

**Four possible error cases in this case**



- Server must recognize duplicate requests --> Unique ID for requests
- Server must keep state, length of time determines residual error probability

# RPC Errors: Reply Omission



**Same countermeasure as for request omission (timeout)**

**For client, indistinguishable from request omission & delays**

**For server, means to memorize results for potential 2nd reply**

- More states, more memory, especially if parallel calls per client
- Also scalability problem: What if server has thousands of clients?

**Now we see why RRA makes sense**

- If client ACKs reception of reply, server can throw away stored result

## 4.7 RPC: Failure Semantics

TECHNISCHE UNIVERSITÄT DARMSTADT

**Maybe-Semantics: No repeated requests (replies, ...)**

- Simple, fast, efficient but often not sufficient
- Idea: User will try again in case of failure (check eMail, ...)

**At-Least-Once-Semantics: Infinite retry**

- Repeated requests, but stateless servers (no duplicates recognized)
- Restricted to idempotent operations (basically, “read”-operations)

**At-Most-Once-Semantics: Tolerate omissions**

- Repeated requests & replies, duplicates recognized → exec only once
- Server crash → no result (no reply), server may have executed or not

**Exactly-Once: Tolerates crashes**

- For normal commercial RPC systems, this remains a dream
- Transactional systems, fail-safe solutions needed

**Solutions in order or increasing effort**

- Offer the choice to programmer
- Commercial systems usually offer some choice of 1/2/3

**In summary: Forget transparency, local call ≠ RPC**

# RPC: Failure Semantics

| Error Semantic | Absence of errors | In case of omissions | In case of server crash |
|---|---|---|---|
| Maybe | 1 proc-exec.<br>1 result returned | 0\|1 proc-exec.<br>0 results returned | 0\|1 proc-exec.<br>0 results returned |
| At-least-once | 1 proc-exec.<br>1 result returned | ≥1 proc-exec.<br>≥1 result returned | ≥0 proc-exec.<br>≥0 result returned |
| At-most-once | 1 proc-exec.<br>1 result returned | 1 proc-exec.<br>1 result returned | 0\|1 proc-exec.<br>0 results returned |
| Exactly-once | 1 proc-exec.<br>1 result returned | 1 proc-exec.<br>1 result returned | 1 proc-exec.<br>1 result returned |

## Exactly-once remains a dream

- Can approach via redundancy, but this is expensive

## Definition:

- A Web Service is a software application identified by an URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via internet-based protocols (W3C).

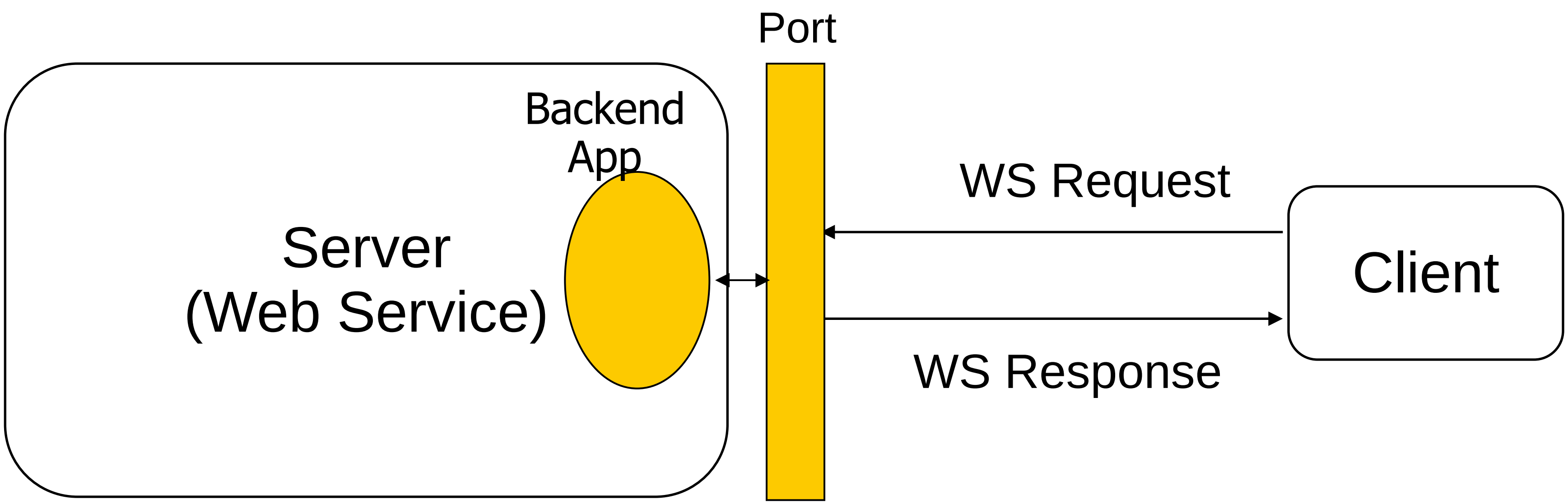
## Web Services

- Are a technology to realize distributed systems.
- Functionalities offered by Web services are similar to traditional Remote Procedure Calls (RPC) apart from the message encoding.

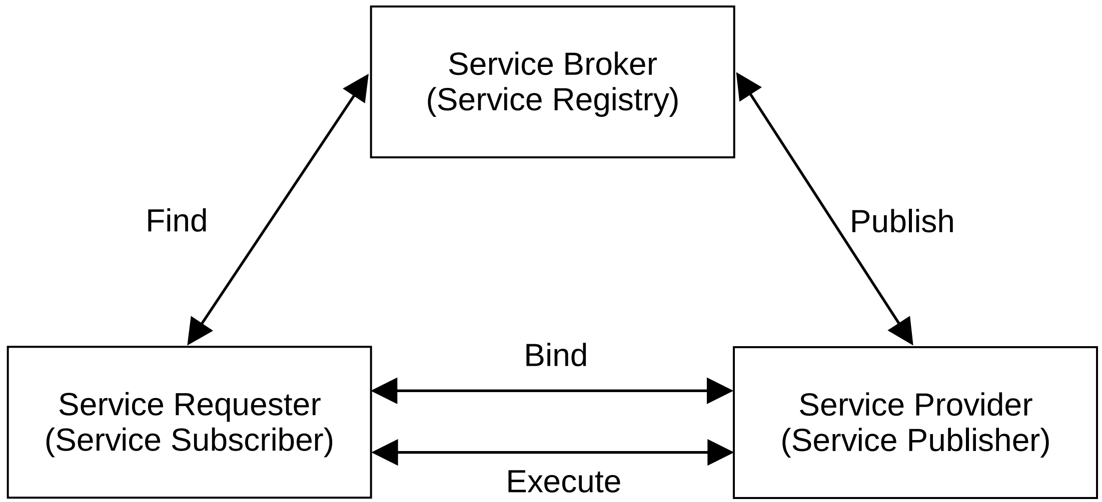
## Web Services use XML

- For data exchange and Remote Procedure Calls:
  - XML allows platform independent description of data
  - But performance drawback because of conversion from/into XML

# 5.1 Web Service Model



# Publish-Find-Bind-Execute Paradigm



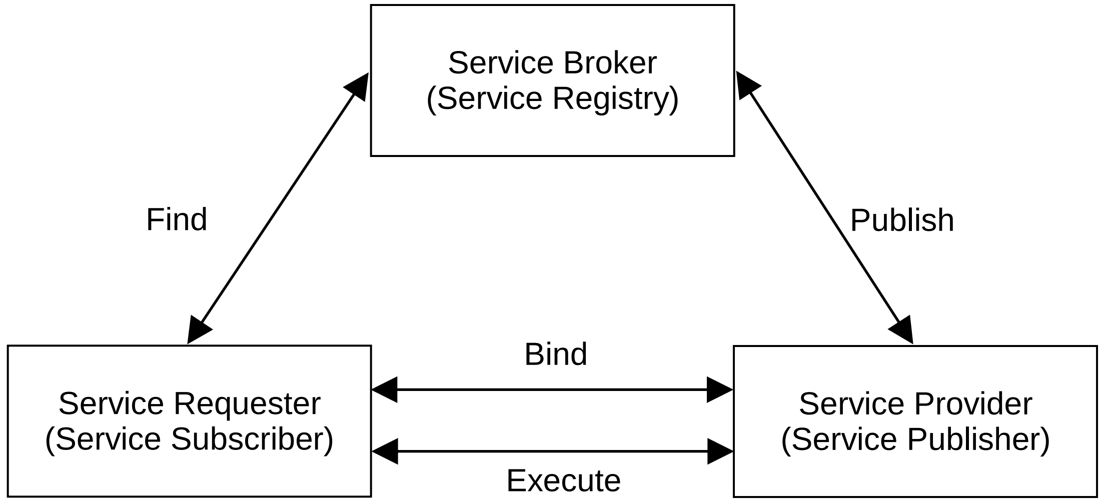

**Roles in a Web Service-based architecture:**

- Service Requestor
- Service Provider
- Service Registry (optional)

**Interaction between roles (including Service Registry):**

- Service Provider registers Web service with Service Broker
- Service Requestor searches for Web service in registry and receives information about where to find and how to use Web service
- Service Requestor is able to bind service based on the information given by the Service Broker
- Service Requestor is able to execute services provided by Service Provider

# Publish-Find-Bind-Execute Paradigm
# .. mentioning UDDI, SOAP, WSDL ..




**Roles in a Web Service-based architecture:**
- Service Requestor
- Service Provider
- Service Registry (optional)

**Interaction between roles (including Service Registry):**
- Service Provider registers Web service with Service Broker (using UDDI).
- Service Requestor searches for Web service in UDDI registry and receives information about where to find and how to use Web service (via UDDI API).
- Service Requestor is able to bind service based on the information given by the Service Broker (based on WSDL).
- Service Requestor is able to execute services provided by Service Provider (using SOAP).

## Features of Web Services technology:

- Reusable logic is divided into services
- Services abstract underlying logic
- Services are composable
- Services are autonomous
- Services share a formal contract
- Services are loosely coupled
- Services are stateless
- Services are discoverable

# Web Service Layer Model

| | | |
|---|---|---|
| **Application Layer** (Anwendung) | **UDDI** | **XML** |
| | **WSDL** | |
| | **SOAP/REST** | |
| | **HTTP, HTTPS, SMTP** | |
| **Transport Layer** (Transport) | **Internet: UDP, TCP** | |
| **Network Layer** (Vermittlung) | **Internet: IP** | |
| **Data Link Layer** (Sicherung) | | |
| **Physical Layer** (Bitübertragung) | | |

**SOAP is an XML-based specification of messages used for communication with Web Services**

**Content of the SOAP standard**

- Contains a syntax for the definition of XML-based messages
- SOAP is a communication model specifying message exchange
- Defines rules for possible content of corresponding messages
- Defines rules for message transport using various L5-protocols (HTTP, SMTP)
- Conventions for Remote Procedure Calls

**Messages**

- Are always exchanged between two participants
- Can be processed by different intermediaries on their way to the final receiver (so called endpoint)

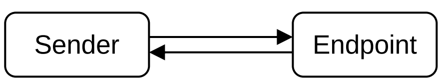
**Every receiver (including intermediaries)**

- Opens the messages and processes the part intended for him
- Is able to be sender again; so message passing is possible

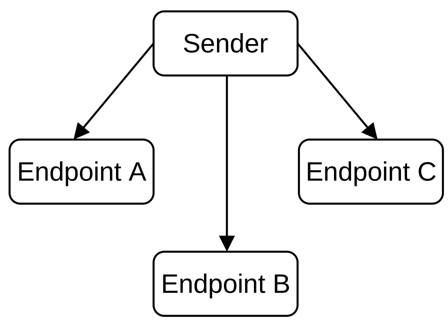
# SOAP – Simple Object Access Protocol

## Possible communication models are:

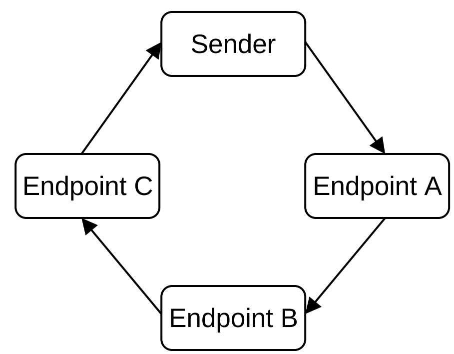
- Request-response: sender sends message, endpoint responds to message
- Broadcast: sender generates messages concurrently transmitted to various receivers
- Workflow: chain of senders and receivers processing a single task forming a circle




Request / Response




Broadcast




Workflow

# 5.3 WSDL – Web Service Description Language

## WSDL

- Is a specification to describe interfaces of Web Services using XML documents
- Defines rules for method invocation (so called contracts)

## Contents of a WSDL document describing a Web Service are:

- Definition of data types
- Declaration of methods
- Combination of methods to Web Services
- Mapping of protocols for method invocation (binding)

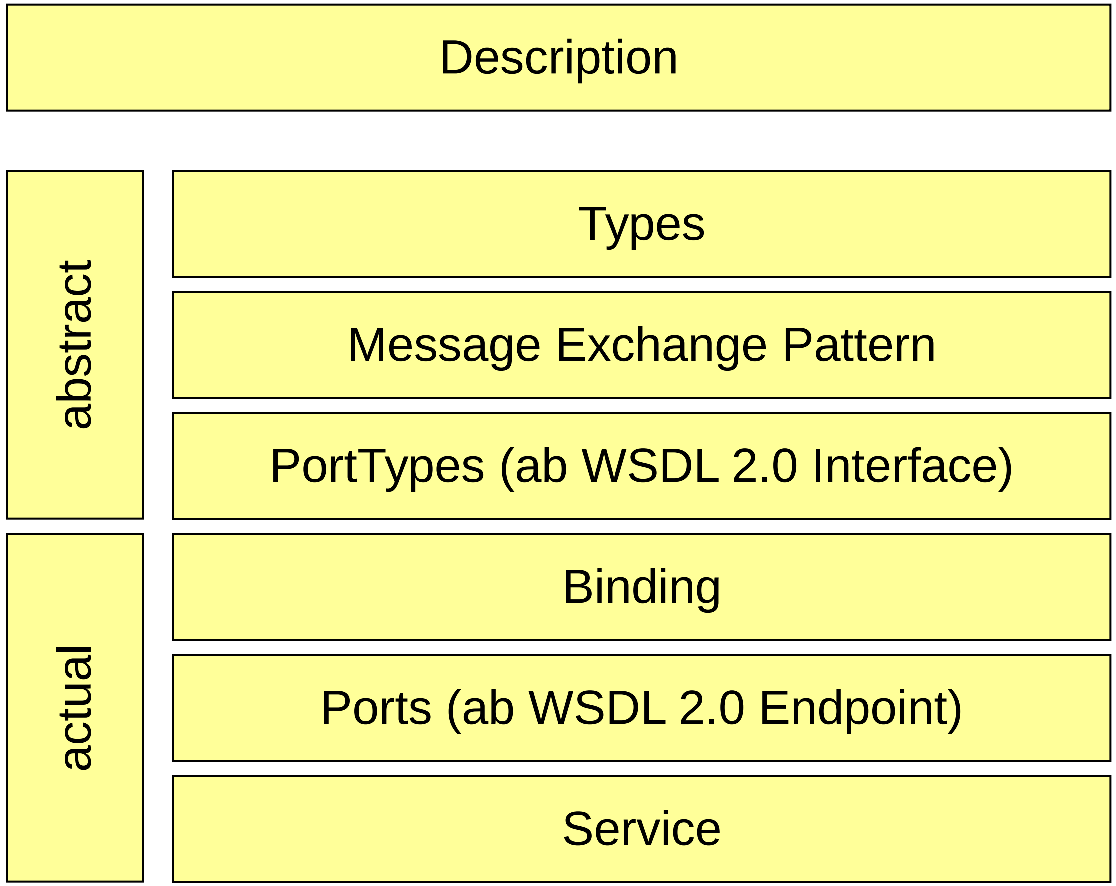
## WSDL allows the following bindings to be used:

- SOAP
- HTTP GET/POST
- MIME

## Possible types of communication defined by WSDL:

- One-Way: clients sends message to server, no response by server
- Request-Response: client sends message to server, server sends response
- Solicit-Response: server sends message to client, response by client
- Notification: server sends message to client, no response by client

# WSDL – Components



# WSDL – Example

```
<?xml version="1.0" ?>
  <definitions name="StockQuote"
      targetNamespace="http://example.com/stockquote.wsdl"
      xmlns:tns="http://example.com/stockquote.wsdl"
      xmlns:xsd1="http://example.com/stockquote.xsd"
      xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
      xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <schema targetNamespace="http://example.com/stockquote.xsd"
        xmlns="http://www.w3.org/2001/XMLSchema">
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
      <element name="TradePrice">
        <complexType>
          <all>
            <element name="price" type="float"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>
```

http://de.wikipedia.org/wiki/Web_Services_Description_Language

# WSDL – Example

```
…
<message name="GetLastTradePriceInput">
      <part name="body" element="xsd1:TradePriceRequest"/>
 </message>
<message name="GetLastTradePriceOutput">
      <part name="body" element="xsd1:TradePrice"/>
 </message>
 <portType name="StockQuotePortType">
      <operation name="GetLastTradePrice">
        <input message="tns:GetLastTradePriceInput"/>
        <output message="tns:GetLastTradePriceOutput"/>
      </operation>
 </portType>
 <binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
      <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
      <operation name="GetLastTradePrice">
        <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
        <input>
           <soap:body use="literal"/>
        </input>
        <output>
           <soap:body use="literal"/>
        </output>
      </operation>
 </binding>
…
```

http://de.wikipedia.org/wiki/Web_Services_Description_Language

# WSDL – Example

```
…
   <service name="StockQuoteService">
      <documentation>My first service</documentation>
      <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
        <soap:address location="http://example.com/stockquote"/>
      </port>
   </service>
</definitions>
```

http://de.wikipedia.org/wiki/Web_Services_Description_Language

# SOAP – Example Request

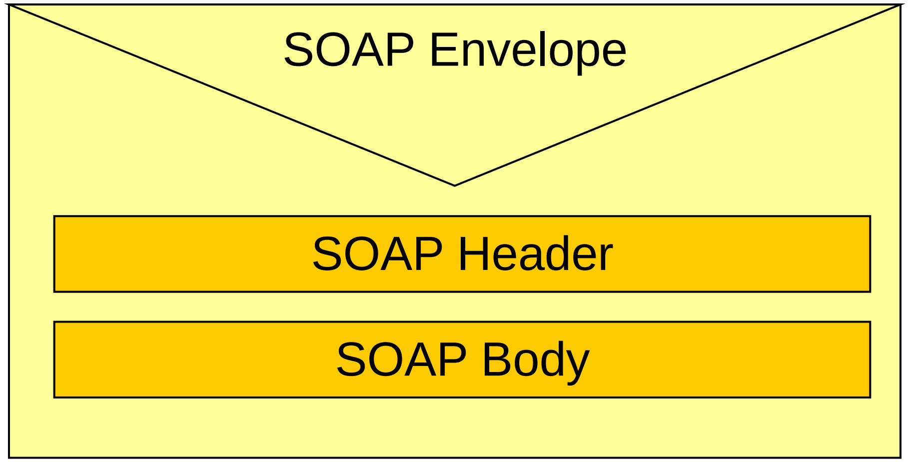


```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```




http://www.w3schools.com/SOAP/soap_example.asp

# SOAP – Example Response

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<soap:Envelope
xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPriceResponse>
      <m:Price>34.5</m:Price>
    </m:GetStockPriceResponse>
  </soap:Body>

</soap:Envelope>
```

http://www.w3schools.com/SOAP/soap_example.asp

**UDDI contains various aspects needed to support dynamic access and usage of Web Services:**

- A logically unique but physically distributed directory service architecture in which Web Services can be published and searched
- Requirements on providers of those directory services
  - A description of an API enabling the publishing and searching of Web Services
  - An XML-based data model to describe companies offering Web Services and the Web Services itself

**Contents of the UDDI data model are (not limited to Web Services):**

- White Pages: information about the company offering the Web Service
- Yellow Pages: classification of the company offering the Web Service
- Green Pages: description of the offered Web Service, containing technical information how to find and use the offered Web Service

# 5.5 Web 2.0 – AJAX

**AJAX – Asynchronous JavaScript and XML**

**Idea:**

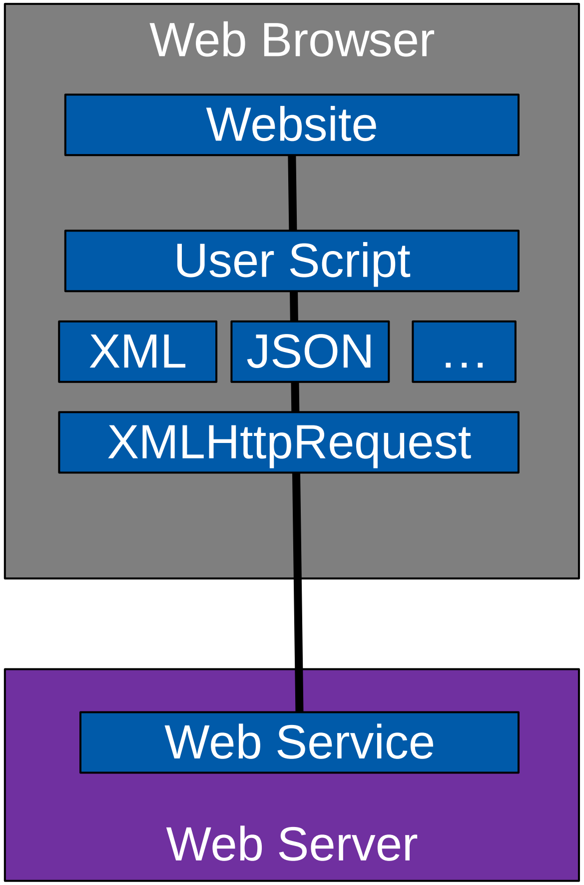
- To build Rich Internet Applications (RIA), i.e., Web applications behaving and looking like "normal" applications

**Core Element is XMLHttpRequest**

- Retrieval of data without loading complete website
- A Web service invocation
- But: Not limited to transmission of XML (JSON etc. more often used)

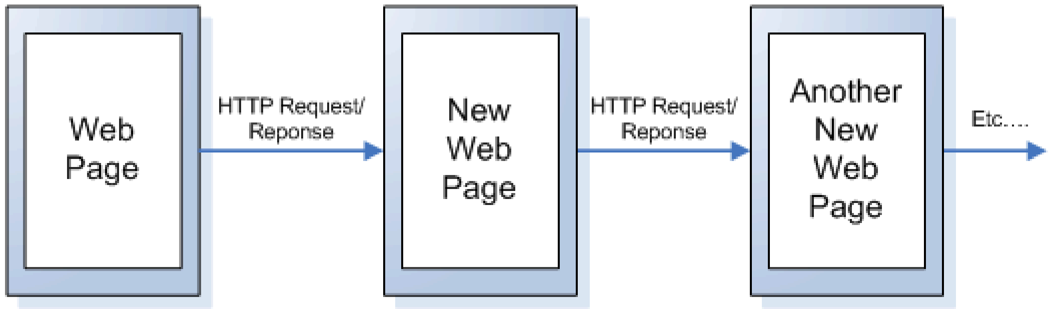
**Sites using AJAX and therefore WS:**

- Gmail
- Facebook
- Yahoo
- …

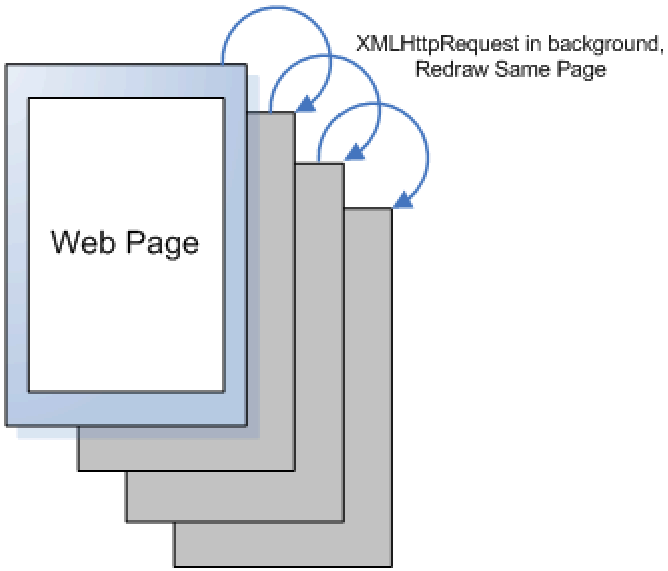


# Web 2.0 – AJAX: In more detail

## Traditional Web interaction




## AJAX based style




http://developer.yahoo.com/javascript/howto-ajax.html

# REST – Representational State Transfer

## Basics

- Introduced by Roy Thomas Fielding in his doctoral dissertation in 2000
- Is an architectural style for distributed Web applications, i.e.,
  - REST is not a protocol (in contrast to SOAP)
  - REST is not a standard (in the sense of the W3C)
- Leverages the existing principles and standards of the WWW (e.g. HTTP, URI)
- An application which adheres to REST architectural style is called „RESTful“
- RESTful applications manage an arbitrary number of resources
- When a client follows a URL within an application, the resulting page represents the next state of the application (→ REpresentational State Transfer)

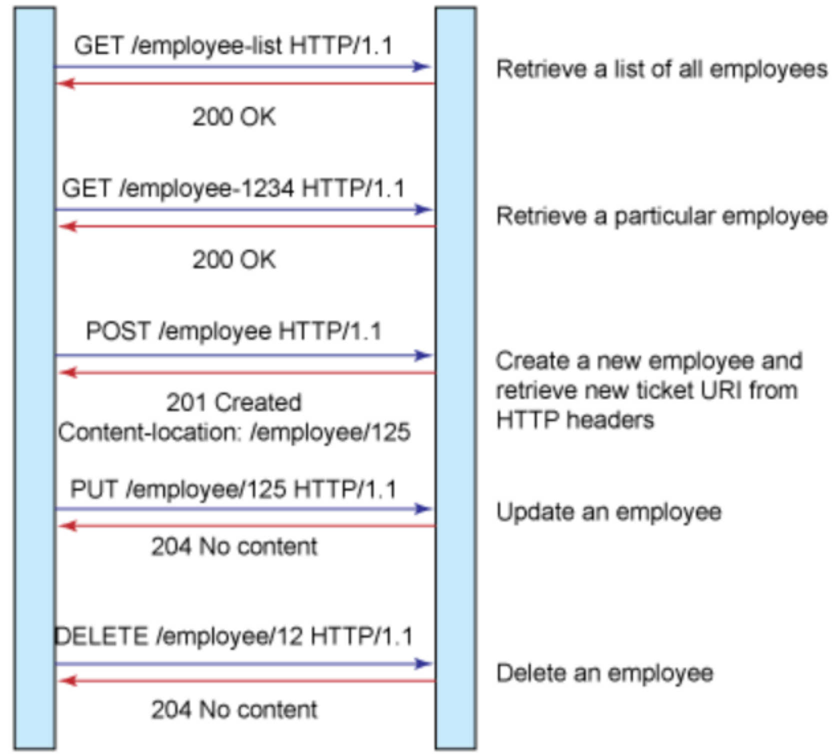
## Resource

- Can be everything that can be referenced by URIs (acc. to W3C)
- Can be identified uniquely
- Has one or more representations (i.e., current state, file format)
- Should be a noun, not a verb
- Example: a song (e.g., three representations: audio file, video file, text file)

# REST – Representational State Transfer (cont'd)

## Communication

- Relies on simple point-to-point communication over HTTP
- HTTP methods (generic interface):
  - GET: get representation of a resource
  - POST: update a resource
  - PUT: create or overwrite a resource
  - DELETE: remove a resource
- All interactions are stateless
- Typically XML or HTML payload




Source: http://www.ibm.com/developerworks/webservices/library/ws-restajax/

## RESTful Web Service Example

- Manage your employees
- http://www.employee-details.com/employee-list...

## AJAX and RESTful Web Services

- XMLHTTPRequest object allows for GET, POST, PUT, DELETE to server
- AJAX-based HTML page can act as client to the above RESTful Web service

# RESTful Web Services

## Benefits

- HTTP widespread
  → standardized implementation; easy to build
- Generic, i.e., uniform interface
  → clients can be reused
- HTTP Caching
  → faster response times
- Stateless
  → improved scalability and load-balancing
- Not a lot of additional markup information
  → lightweight messages

**Does not support the whole Webservice-* (e.g., Webservice-security) advanced protocol stack**

## Limitations

- Relies on HTTP protocol
  → not transport-agnostic
- Point-to-point communication
  → not usable for more complex communication models in distributed environments (cf. SOAP communication models)
- No formal interface description language like WSDL
- Only synchronous message exchange (cf. callbacks using SOAP and SMTP)
- No support for features such as security, reliable messaging, etc.
  → no support for complex user requirements (cf. specific SOAP header)

TECHNISCHE UNIVERSITÄT DARMSTADT

**What's "wrong" with RPC and other similar mechanisms?**

**They are based on Request/Reply approach:**

- Client
  - has initiative, client "pulls" data from server
- Peer
  - "locked" in handshake
- Bad if:
  - Lots of data
  - many receivers which come and go
  - information not generated very often

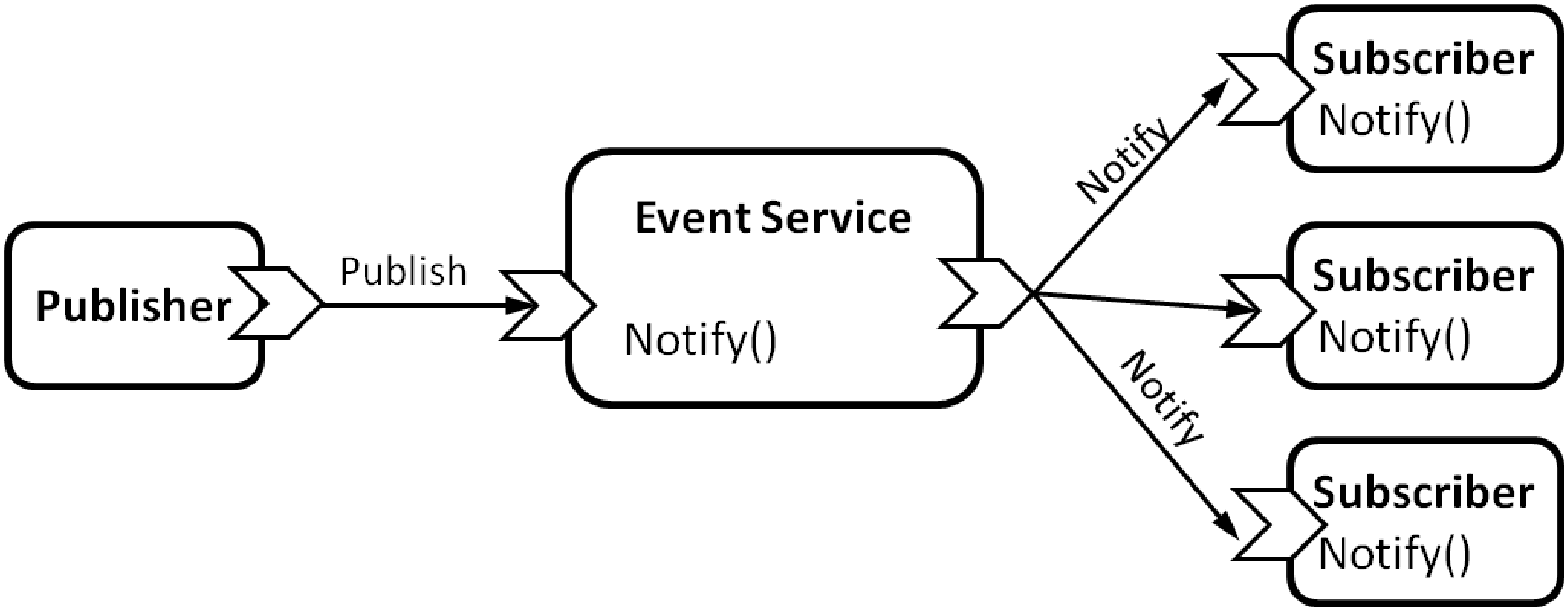
**Push approach**

- Idea:
  - Peer with data "pushes" it to interested parties
- Producer:
  - Immediate information delivery ("publish")
- Consumer:
  - Initial "subscribe" for event-type/channel (= info-category)
- Consumer receives events
  - that match the subscription asynchronously as they are generated by producer

**Such systems are also called publish/subscribe**

**Note: Event systems widely used in many fields, e.g., GUI**

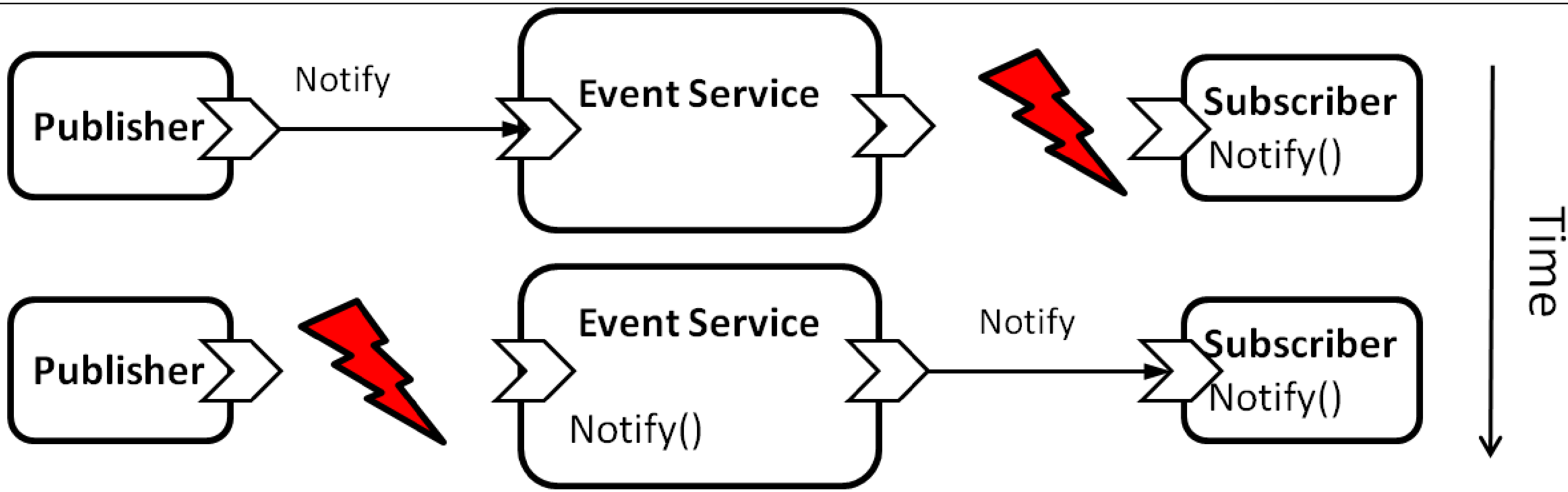
# 6.1 Basic Principles




## Space decoupling

- The interacting parties do not need to know each other
- Producers publish messages through an event service
- Subscribers indirectly receive messages from event service
- One-to-many communication patterns possible

## Basic Principles: Time Decoupling

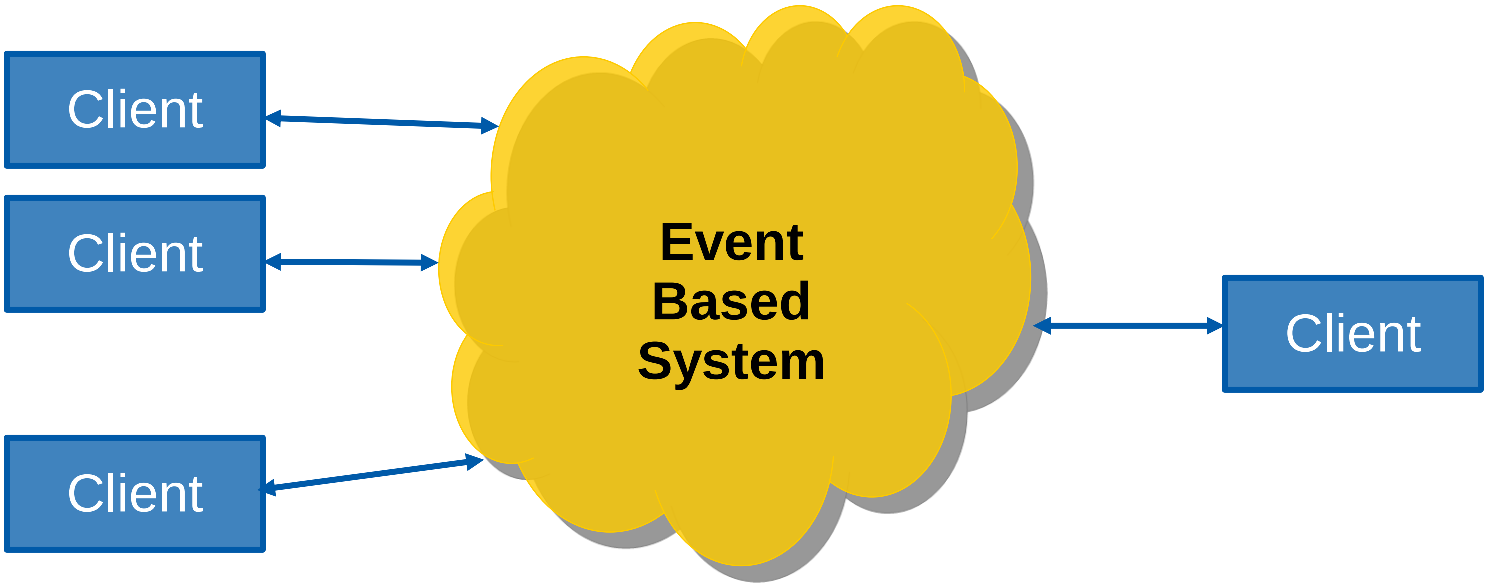



### Time decoupling

- The interacting parties do not need to be actively participating in the interaction at the same time

**Reducing space and time dependencies greatly reduces the need for coordination between systems**

# Distributed Event Systems

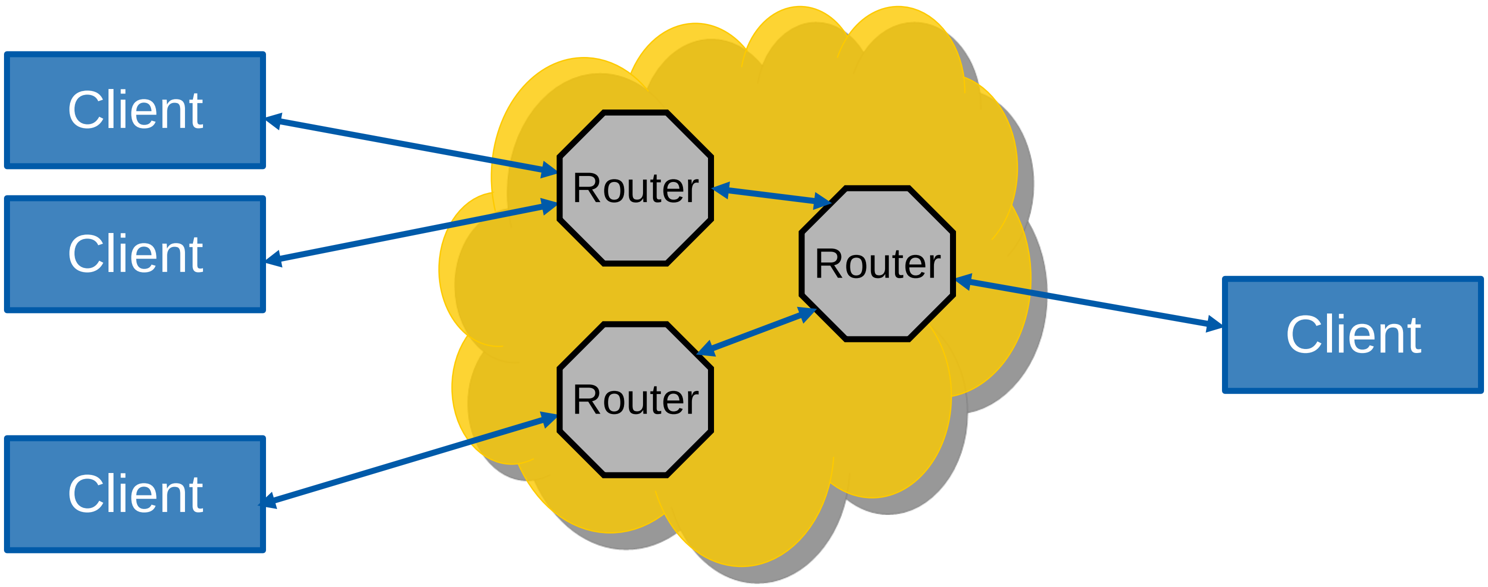



**Permit**
**loosely coupled,**
**asynchronous point-to-multipoint communication patterns**

**Application independent infrastructures**

**Clients communicating via a logically centralized component**

# Distributed Event Systems




**Logically centralized component**

- Single server /
  Network of event routers
- Transparent for application (=Client)

- Router network can be reconfigured
  - independently and
  - without changes to the application

**--> Scalability**

TECHNISCHE UNIVERSITÄT DARMSTADT

## Messaging Domain

- Point-to-Point (Producer -> Consumer)
- Publish/Subscribe
  - Subscription-based
  - Advertisement-based

## Subscription Mechanism

- Channel-based (=Topic-based) Subscription
- Content-based Subscription
- Subject-based Subscription (limited form of Content-based sub.)

# Classification

## Event Data Model

- Un-typed
- Typed
- Object-oriented

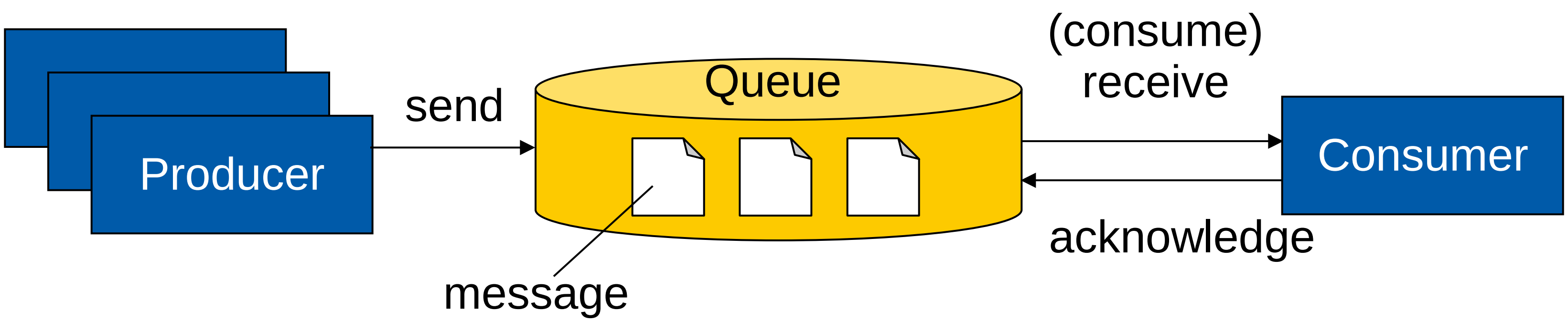
## Event Filters

- Expressiveness and flexibility of subscription language
  - Simple Expressions
  - SQL-like Query Language
  - (Mobile) Code
- Pattern Monitoring: Temporal sequence of events
- Evaluated in router network

## Scalability <-> Expressiveness Tradeoff

- Simple Expressions permit Filter Merging -> better scalability

# Messaging Domain: Point-to-Point




**Each message has only one consumer**

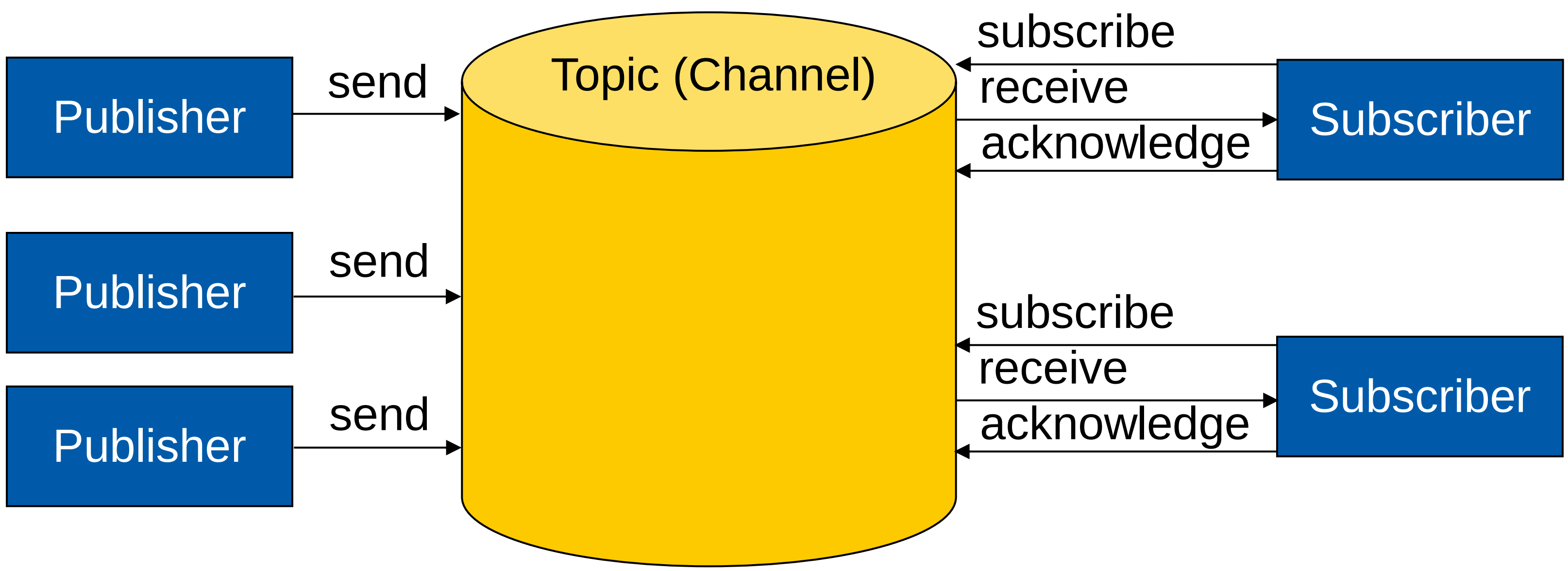
**Receiver acknowledges successful processing of message**

**No timing dependencies between sender and receiver**

**Queue stores message (persistent), until**

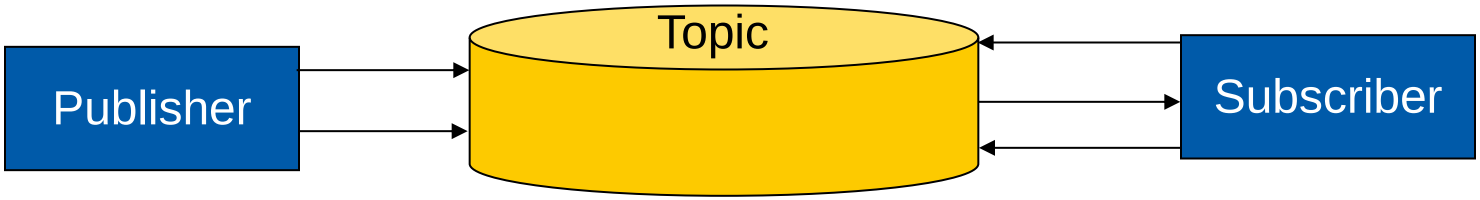
- It is read by a receiver
- The message expires (Leases)

# Messaging Domain: Publish/Subscribe




- Interested parties can subscribe to a channel (topic)
- Applications post messages explicitly to specific channels
- Each message may have multiple receivers
- Timing dependency between publishers and subscribers

# Messaging Domain: Advertisements




- Publisher advertises topic before publishing
- Subscribers can get a list of advertised topics
- Avoids problem of subscriber having to figure out which topics are available for subscription

# Topic-Based vs. Content-Based

## Topic-based subscription

- Messages sent to a well-known topic
- Subscribers subscribe to topics
- Topics are typically expressed as strings

## Content-based subscription

- Subscriptions are matched against the content of the message
- Subscribers describe their interest as filter expressions

## Special case: Subject-based subscription

- Special case of content-based subscription
- Well-known subject in messages
- Subscriptions matched against the subject
- Subject typically strings or key-value-pairs

## Covering Relations

- Attribute Filter: Filter covers Notification (= message)

$$\phi \subset_f^n \alpha :\Leftrightarrow \phi \quad \text{covers} \quad \alpha$$

$$\phi \subset_f^n \alpha :\Leftrightarrow \phi.name = \alpha.name \wedge \phi.type = \alpha.type \wedge \phi.match(\alpha.value, \phi.value)$$

- Subscription: Subscription covers Notification

$$s \subset_S^N n :\Leftrightarrow s \text{ covers } n$$

$$N_S(s) \subseteq N;\; n \in N_S(s) :\Leftrightarrow s \subset_S^N n$$

$$N_S(s) = \{ n \in N : \forall \phi \in s : \exists \alpha \in n : \phi \subset_f^n \alpha \}$$

- Examples:

String event=alarm $\subset_S^N$ String event=alarm<br>Time date=02:40:03

String event=alarm $\not\subset_S^N$ String event=alarm<br>Integer level>3 Time date=02:40:03

## Covering Relations

- Advertisement: Advertisement covers Notification

$$a \subset_A^N n :\Leftrightarrow a \text{ covers } n$$

$$N_A(s) \subseteq N; \quad n \in N_A(a) :\Leftrightarrow a \subset_A^N n$$

$$N_A(a) = \{n \in N : (\forall \alpha \in n : \exists \phi \in a : \phi.name = \alpha.name)$$

$$\wedge (\forall \alpha \in n : \forall \phi \in a : \phi.name = \alpha.name \Rightarrow \phi \subset_f^n \alpha)\}$$

- Advertisement covers Subscription

$$a \subset_A^S s :\Leftrightarrow N_A(a) \cap N_S(s) \neq \emptyset$$

- “a is relevant for s”

# Covering Relations: Examples

| | | |
|---|---|---|
| String event=alarm<br>Time date any<br>Integer level>0 | $\subset_A^S$ | String event=alarm<br>Integer level>3 |
| String event=alarm<br>Time date any<br>Integer level>0 | $\not\subset_A^S$ | String event=alarm<br>Integer level>3<br>String user any |
| String event=alarm<br>Time date any<br>Integer level>0 | $\subset_A^S$ | Integer level>5 |

## Subscription-based event service

- Service delivers notification n to party X iff
  - X subscribes s
  - $s \subset_S^N n$

## Advertisement-based event service

- Service delivers notification n posted by object Y to party X iff
  - Y advertises a
  - X subscribes s
  - $a \subset_A^S s$
  - $s \subset_S^N n$

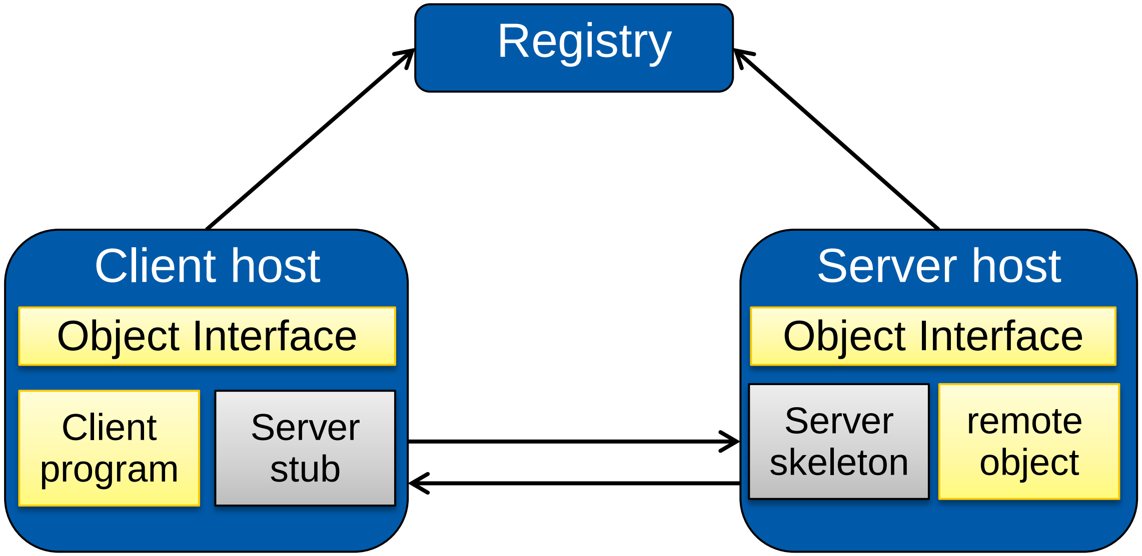
TECHNISCHE UNIVERSITÄT DARMSTADT

## Remote Method Invocation (RMI)

## Common Object Request Broker Architecture (CORBA)

## Enterprise Java Beans (EJB)

**Developed by SUN**
**for the Java Platform**




**Server Object Interface**

- Plain Old Java Interface
- Describes Operations provided by the server object
- Shared between the client and the server

**Remote Object = Distributed Object**

# RMI Components

## Server Stub (automatically generated)

- Provide an interface to call the remote method(access transparently)
- Pack the arguments and transfer them to the server host
- Unpack the result and return it to the caller

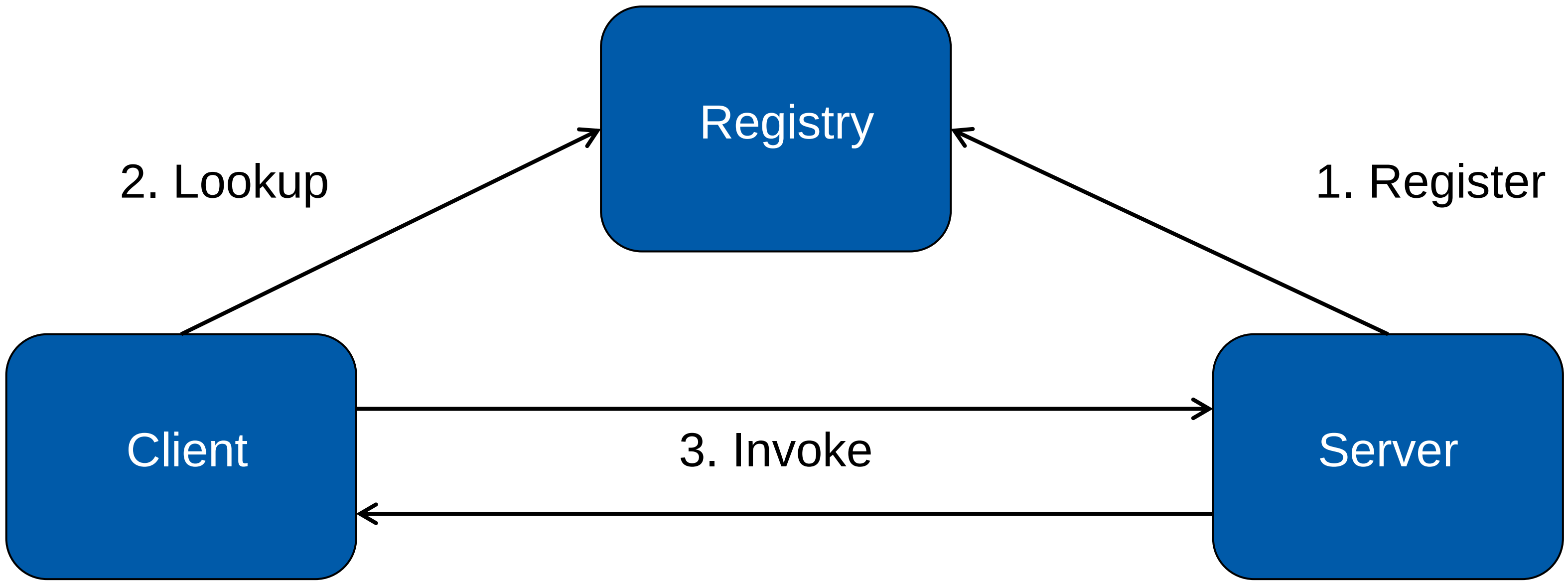
## Server Skeleton (automatically generated)

- Unpack remote method call parameters
- Invoke the method on the server object
- Pack the result of the invocation
- Transfer the result to the client host

## RMI Registry

- Provides a name service for server objects
- The Server Host register server objects
- The Client Host  lookup server objects by their name
- → The client is supposed to know the name of the server object

# RMI Interaction

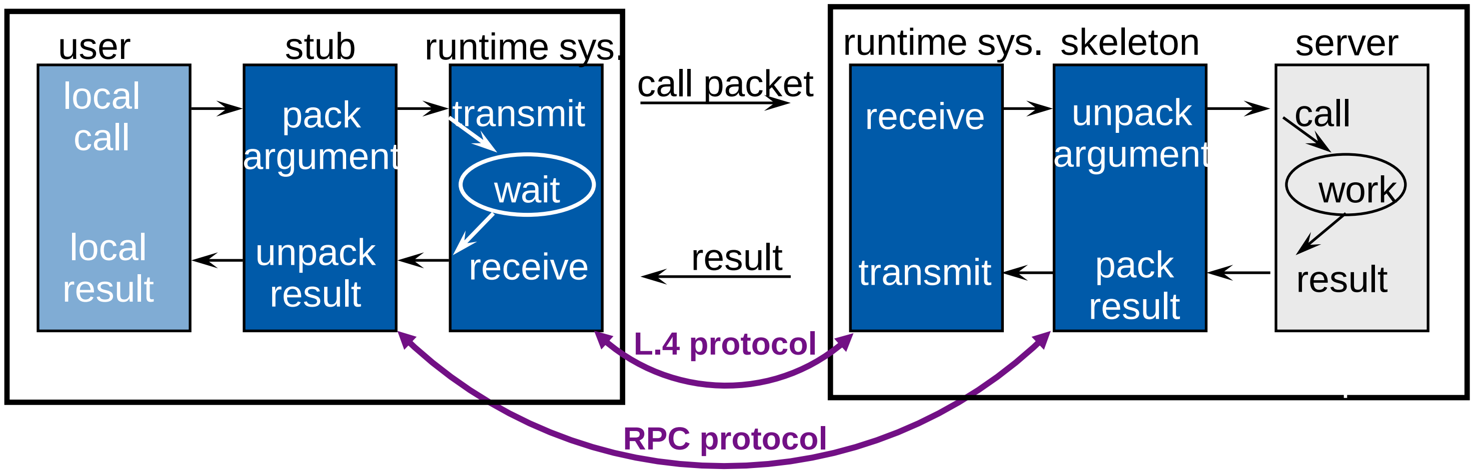



**Workflow (On Bootstrap)**

- Server Object Interface is shared between server and client

  - 1. Server registers the Server object at the RMI Registry
  - 2A. Client looks up the required server object
  - 2B. RMI Registry returns address of the server object
  - 3. client could invoke the server object

# RMI Interaction




## Workflow (Remote Method Call)

- Equal to RPC (see drawing)

1. Client calls the Server stub(which is a local method)
2. server stub serializes the arguments and writes them to the network stream
3. server skeleton reads the network stream, unpacks the arguments and calls the server object method
4. server object method performs the computation
5. server skeleton packs the result and writes them to the network stream
6. server stub reads the response from the network stream unpacks it and returns it to the caller

# RMI Code Example

TECHNISCHE UNIVERSITÄT DARMSTADT

## Remote Object Interface

```
public interface HelloServer extends Remote {
    public String sayHello() throws RemoteException;
}
```

## Server Side(Remote Object)

```
public class HelloImpl extends UnicastRemoteObject
 implements HelloServer {
    public String sayHello() {
        return “Hello from the remote server!";
    }
}
```

## Registration

```
Naming.rebind("server", new HelloImpl());
```

## Client (Usage of remote object)

```
HelloServer server(HelloServer)Naming.lookup(“//localhost/server");
String result = server.sayHello();
System.out.println(result);
```

# Local vs. Distributed Objects

## Differences between local and distributed objects

**Distributed objects have more meta data**

- (object location, unique id, type meta data)
- Depending on the middleware

**Remote method invocations have much higher latency**

- three to four orders of magnitude

**Creation:**

- Objects are either created on another host
- or migrated to another host

**Cleanup:**

- Network communication necessary to decrease reference counter
- For each reference to an object

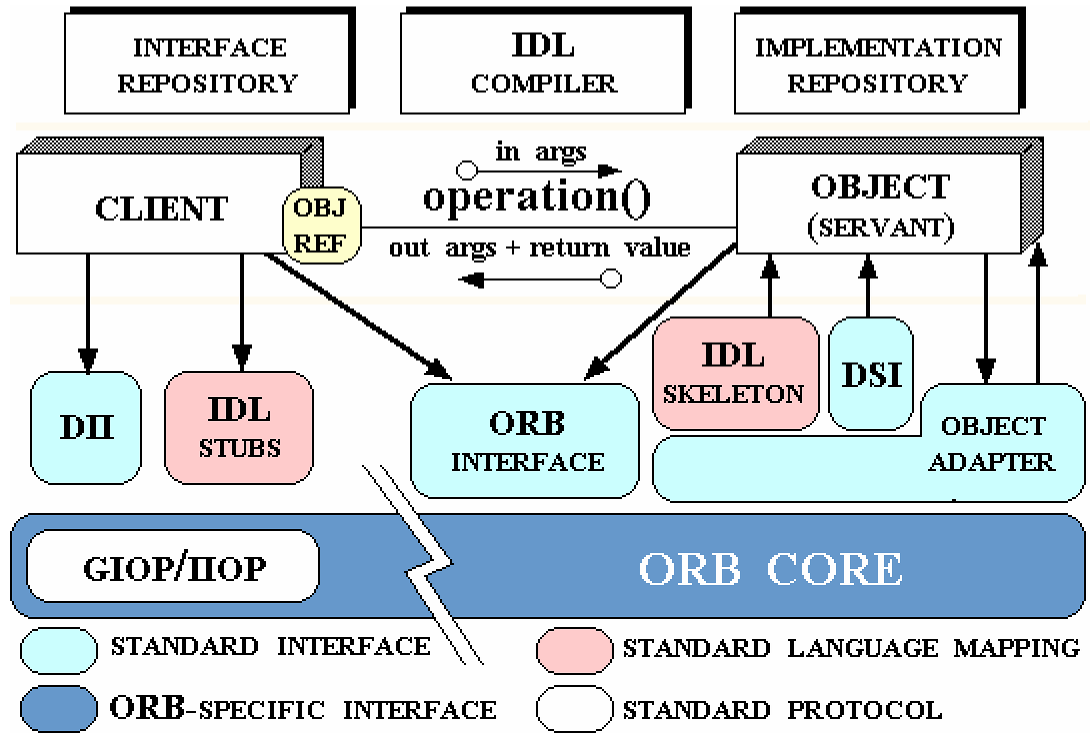
**Referencing:**

- network communication necessary to increase reference counters

# Common Object Request Broker Architecture

## Specified by the Object Management Group

- Defines an interface definition language (IDL)
- Defines a set of language bindings(C, C++, Java, Smalltalk, Ada95, …)
- Stubs and Skeletons are automatically from the IDL




**DII = Dynamic Invocation Interface**
**DSI = Dynamic skeleton interface**
**ORB = Object request broker**
**IDL = Interface Definition Language**
**GIOP = General Inter-ORB Protocol**
**IIOP = Internet Inter-ORB Protocol**

## *The Rise and Fall of CORBA*

**Building dist. Systems in the early 90s was a nightmare**

- because of many different
  - hardware,
  - operating systems and
  - programming languages
- E.g. different byte order, floating point representation or character encoding
- So Programmers wrote an entire protocol stack themselves

**CORBA was designated to solve these problems**

- By specification of a platform independent middleware

**The C++ and Java binding gave developers a tool to build heterogeneous distributed systems**

## *The Rise and Fall of CORBA*

**But technical problems stopped the show:**

- CORBA API was to complex and error prone
- Programming Component models(COM, EJB) were a lot simpler
- During CORBA's growth phase in the mid 90s,
  Java and the web changed the computing landscape
- CORBA did not cooperate with the rapidly expanding web

**Because:**

**The standardization committee consisted of too many members**

- The committee is based on consensus
- Consensus by saying yes is easier than by saying no
- So most request for proposals were added
- Not even a prototypical implementation was required in the request for proposal

**→ The resulting standard was too complex**

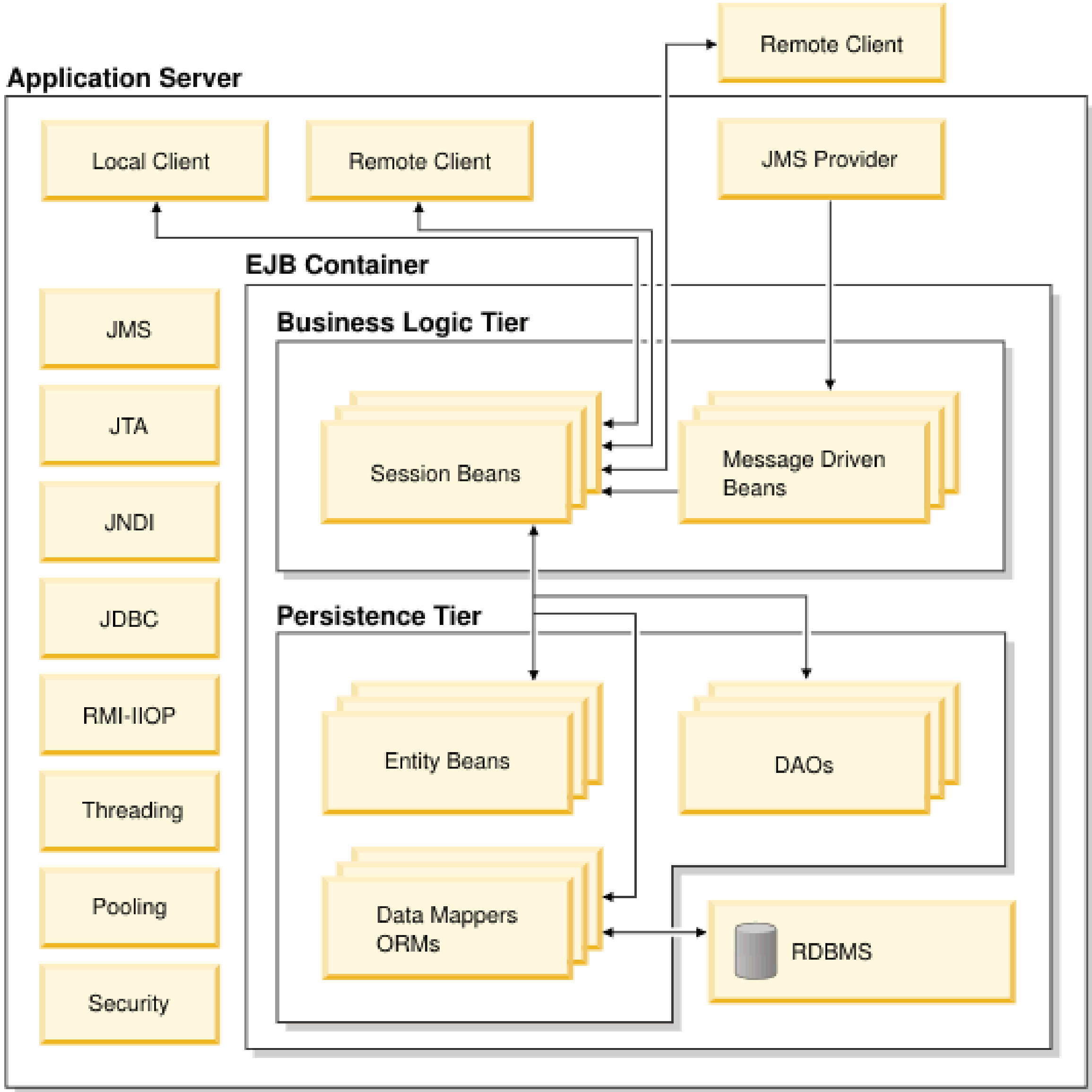
## Enterprise Java Beans (EJB)

**Managed, server-side COMPONENT architecture for modular construction of distributed ENTERPRISE ARCHITECTURE**

**A BEAN is a component**

- Entity bean
  - Contains the data model of the business process
  - Persistence is managed either manually or automatically
  - E.g. all articles in an online store are modeled as Entity bean
- Session Bean
  - Encapsulates operations to transform entities
  - Distinguish between stateless and stateful session beans
  - Stateless session beans could have an web service interface
  - E.g. adding an article to the customers shopping cart of an online shop
- Message Driven Bean
  - Covers asynchronously executed operations
  - E.g. sending confirmation emails to customers

# Enterprise Java Beans: Architecture




Source: public.boulder.ibm.com