

Distributed Operating Systems

4th Chapter
Architectures

Matthias Werner
Chair Operating Systems

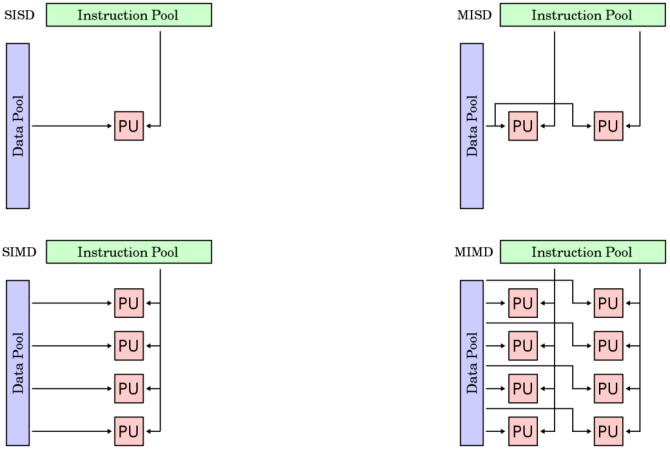
4.1 Hardware

Repetition: Flynn's Taxonomy

- The most famous and simplest classification scheme distinguishes between single and multiple instruction- and data-streams
- Combination leads to four classes:

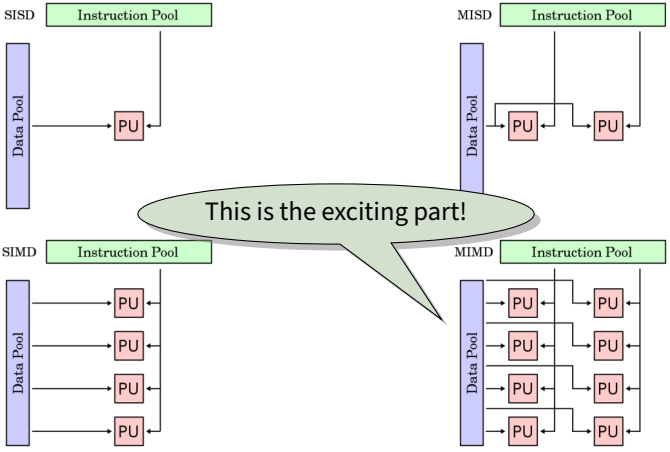
	SI (single instruction)	MI (multiple instruction)
SD (single data)	**SISD** classical Von Neumann Architecture, Harvard Architecture	**MISD** Pipeline, Dataflow Architecture (?)
MD (multiple data)	**SIMD** Vector Processor, Array Processor (Cray-1, CM2, MasPar)	**MIMD** Multi-processor Systems, Distributed Systems (Cray T3E, Cluster, ...)

Repetition: Flynn's Taxonomy (cont.)



Bildquelle: wikipedia

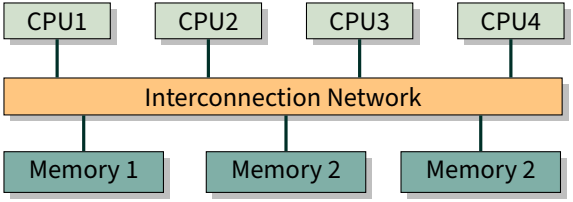
Repetition: Flynn's Taxonomy (cont.)



Bildquelle: wikipedia

MIMD Machines

a) **Multi processor systems** (shared memory systems, tightly coupled systems)
- All processors access shared memory via a joint communication network

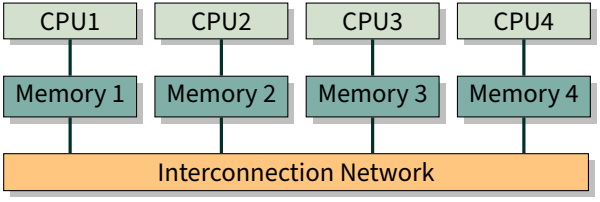


- Dependent on the type of memory we can further distinguish:
  - uniform memory access (UMA)
    ➡ access time to memory is independent from address of processor and memory module
  - non uniform memory access (NUMA)
    ➡ access time to memory depends on address

MIMD Machines (cont.)

b) **Multi-computer system** (message passing systems, loosely coupled systems)

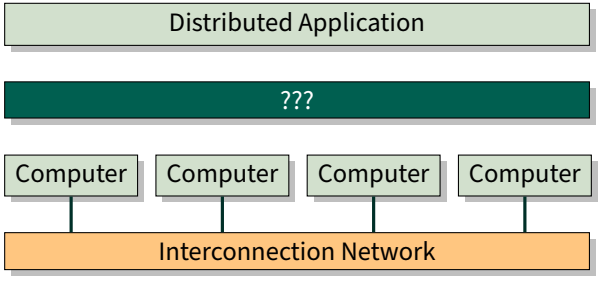
- Each processor can access its own memory only
- Data transfer occurs via message passing over an interconnection network



Unless stated otherwise, we consider multi computer systems in this lecture.

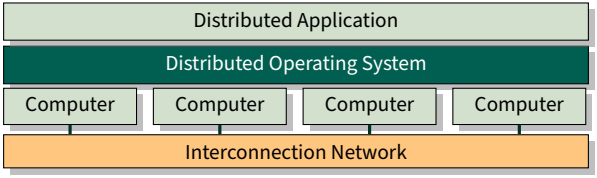
4.2 Architecture of Distributed Systems

- Applications shall be supported in a preferably transparent way; in other words: we need a SW infrastructure that mostly **hides** distribution

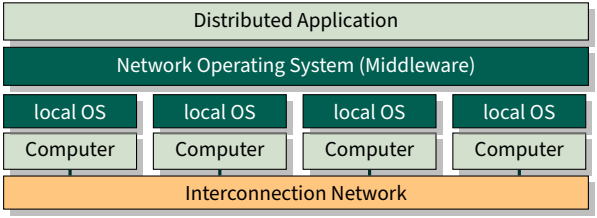


Architecture Approaches for Distributed Systems

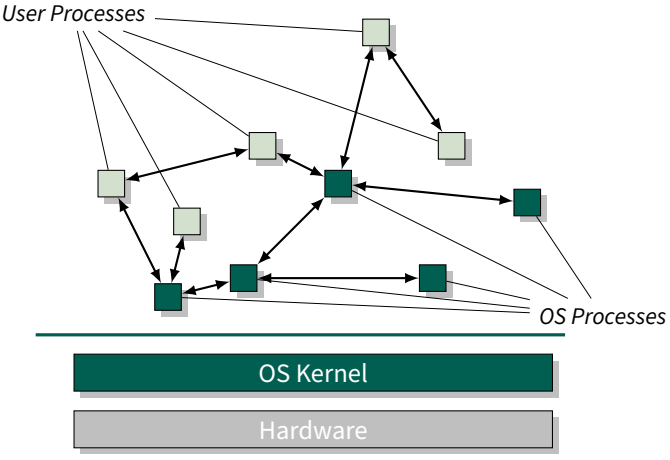
- **Distributed operating system** ➡ Operating system itself provides functionalities relevant for distribution



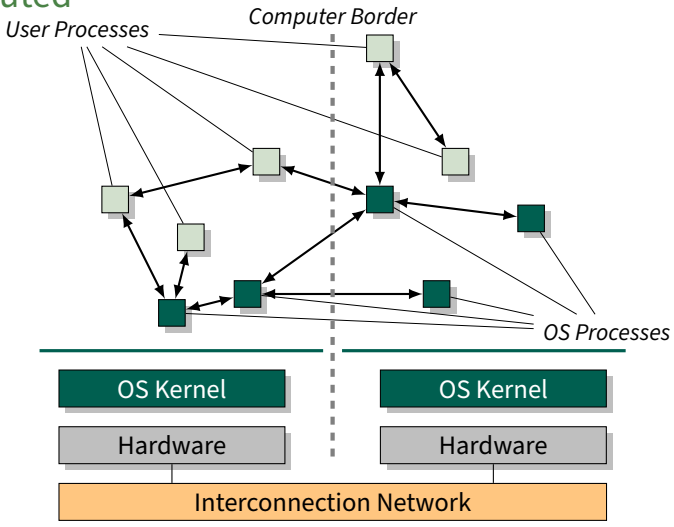
- **Network Operating System** ➡ local operating systems are supplemented with a further layer (middleware) that provides functionalities for distribution



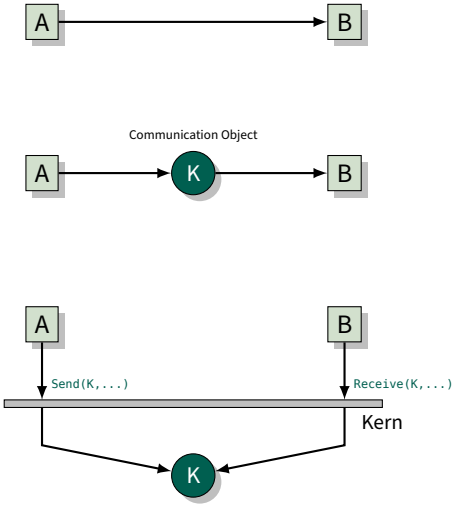
Architecture of Local Operating Systems



Architecture of Distributed Operating Systems



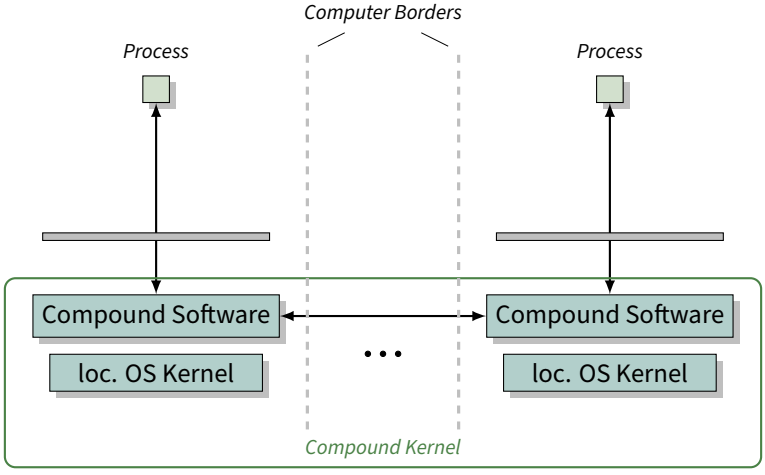
Inter-Process Communication



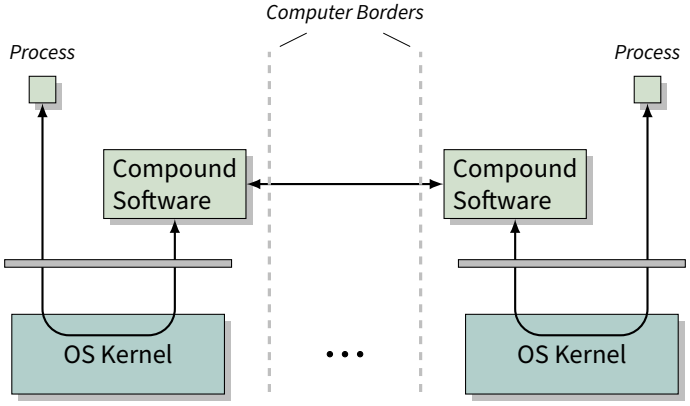
Architecture Variants

- In distributed systems, computer borders need to be crossed for inter-process communication
- Necessary communication software could:
  - be integrated in OS kernel ➡ *kernel federation*
  - exist as extra component outside of the kernel
- Considering kernel federation, distribution is hidden beneath the kernel interface
  - Kernel calls may address arbitrary objects in the network
  - The compound of all kernels is the compound kernel
- Considering *process federation* or *server federation*, the kernel interface stays untouched
  - The local kernel is not “aware” of being part of a distributed system
  - In this way, existing operating systems can be extended to distributed operating systems

Kernel Federation



Process Federation



Remark

The distinction between distributed systems with process federation and network operating systems is fluid.

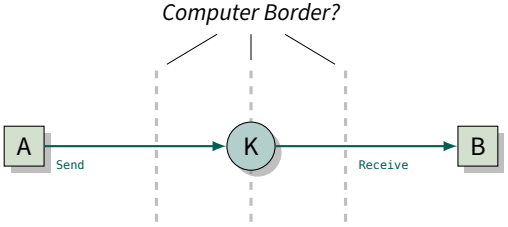
Distinction of Distributed Operating Systems

	Multi processor OS	**Distributed OS**	**Network OS**
All nodes have same OS?	yes	yes	no
# local OS instances	1	n	n
Gemeinsame Prozesswarteschlange?	yes	no	no
Communication	Shared Memory	Message Passing	Message Passing/shared data

4.3 Communication

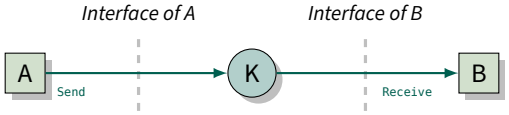
Bridge Principle

- Complex program systems can be constructed by means of **Client-Server-** or **Peer-to-Peer-**Principle as mesh of communicating processes
- Channel approach can easily be transferred to distributed systems
- Pending: How to realise the channel principle when crossing computer borders?

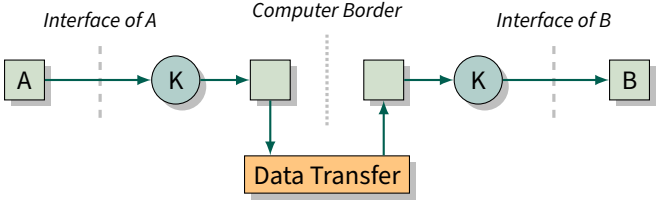


Bridge Principle (cont.)

- Interfaces of the concerned processes should stay unchanged in favor of distribution transparency



- Realization: Make each process “believe” that a local communication partner exists
- This leads to the following construction:

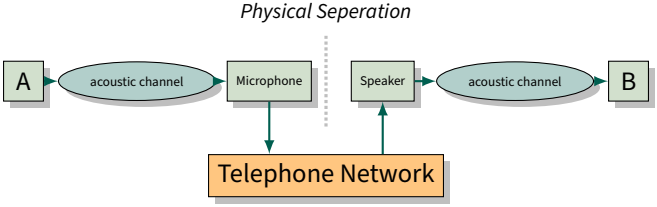


Analogy: Conversation between Persons

- Local communication



- Remote communication via telephone

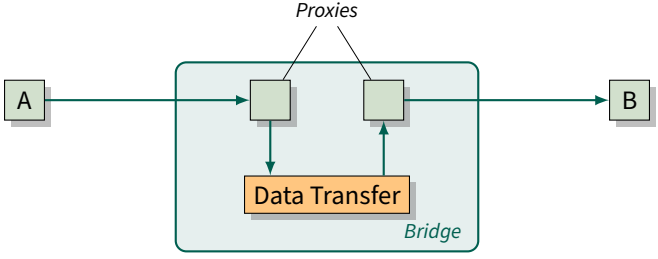


Macrostructure of a Bridge

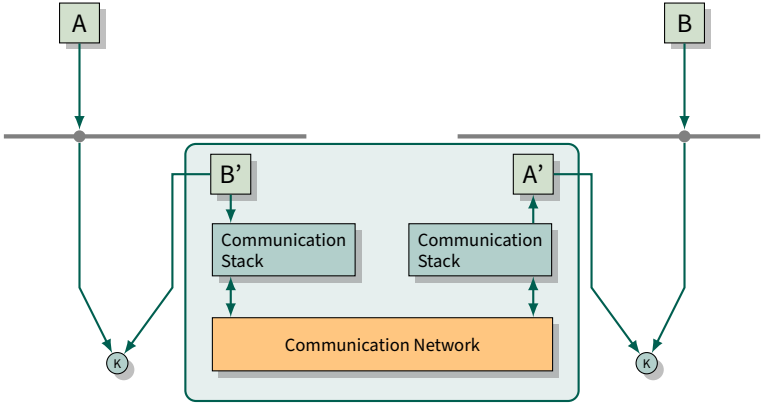
- Local relation between processes/procedures



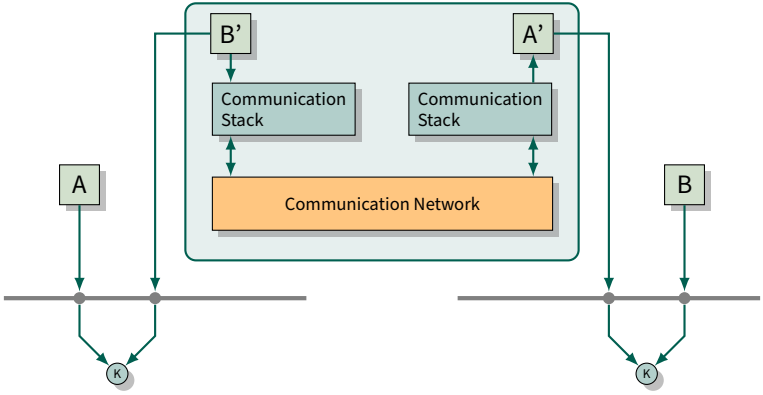
- Relation in distributed case



Bridge in Kernel Federation



Bridge in Process Federation

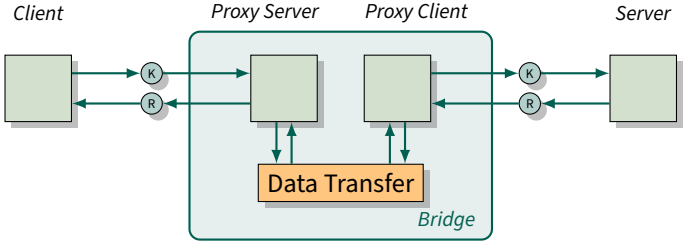


Realization of Bridges

- **Kernel Federation**
  - Bridges realized procedurally in kernel
  - Usually simple, the most important things only
  - Usually not or barely fault tolerant
  - Commonly used in simultaneous computers
- **Process Federation**
  - Bridges realized through processes
  - Usually flexible, powerful and costly
  - Usually fault tolerant
  - Commonly in heterogenous distributed systems

Service Relation in Distributed Case

- Client-Server-Structures usually require a communication in both directions
- A bridge needs to provide proxies that communicate in both directions



Tasks of Proxies

- Precise imitation of interface (syntax and semantics)
- Packing and Unpacking of parameters (Marshaling/Demarshaling)
- Handling of call-by-value and call-by-reference
- Dealing with heterogeneity (e.g. conversion between formats)

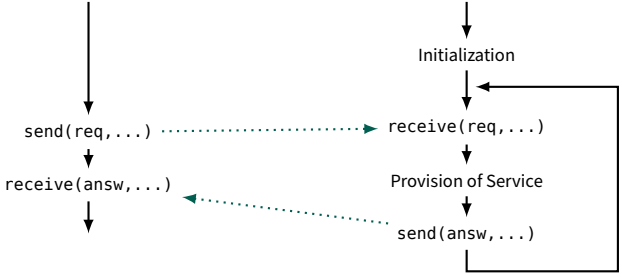
- In certain cases, proxies can be generated automatically (proxy generator)
- In more complex cases, proxies are more extensive (e.g. multiple processes)

4.4 Example: Remote Procedure Call (RPC)

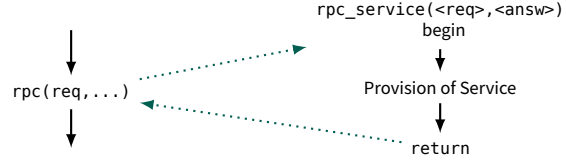
- Realization of service relations through process communication is inconvenient for application programmers
- Regarding the local case, orders to the operating system are usually hidden behind library procedures
- **Idea**:Transfer to distributed case ➡ Service performed by remote computer is provided by procedure call
- Mechanism is known as **Remote Procedure Call** (RPC)
  - Most frequently used realization of service relations in distributed systems
  - Actual inter process communication between computers is hidden behind RPC interface

Client-Server-Relation with Process Communication

- Explicit Communication



- RPC



Parameter Passing

- Call by value
  - Standard case, realization is simple
  - Values are wrapped and send in packages
- Reference Passing
  - Reference passing already complicated in local case ➡ different address spaces
  - Avoid in distributed case
- Substitute by call by copy/restore
  - When calling: transfer to server as in call by value
  - After completion: copying back to client, overriding old values
  - **Attention:** different semantics

Example: Parameter Passing

- Remote Procedure:

```
procedure p(var integer x, var integer y)
   begin
     x := x+1;
     y := y*y;
   end;
```

- Call:

```
i := 4;
p(i,i);
print(i);
```

- Result:
  - Call-by-reference: **25**
  - Call-by-copy/restore: **???**

RPC in Heterogenous Environment

- Computers use different number and character representations (“little endian”/“big endian”, EBCDIC/ASCII, ...)
- Conversion required
- Transfer either...
  - directly to the format of the communication partner
    or
  - to canonical format
- Usually performed automatically

RPC in Heterogenous Environment (cont.)

- Representation of `0x1A2B3C4D` starting from address `10000`
- Smallest address entity is one Byte

	Big Endian		Little Endian	
Adresse	**Hex**	**Binary**	**Hex**	**Binary**
10000	1A	00011010	4D	01001101
10001	2B	00101011	3C	00111100
10002	3C	00111100	2B	00101011
10003	4D	01001101	1A	00011010

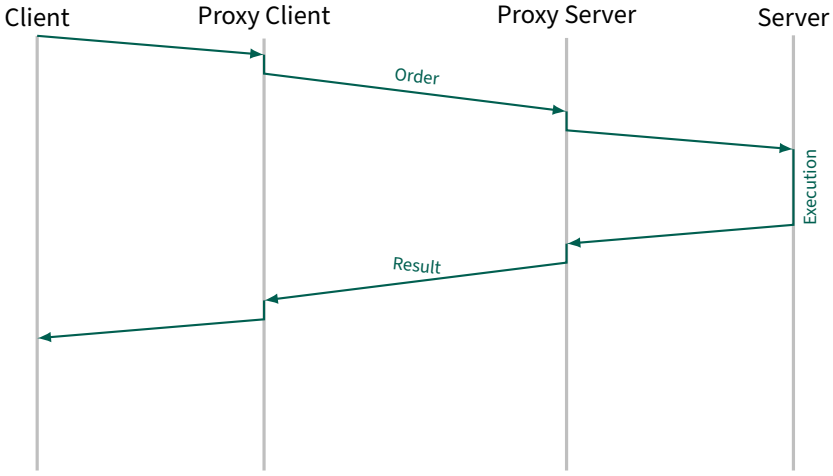
- **Big-Endian-Architectures**: IBM Mainframe, Motorola 680X0, SPARC
- **Little-Endian-Architectures**: Intel x86, VAX, Transputer

RPC – Cases of Error

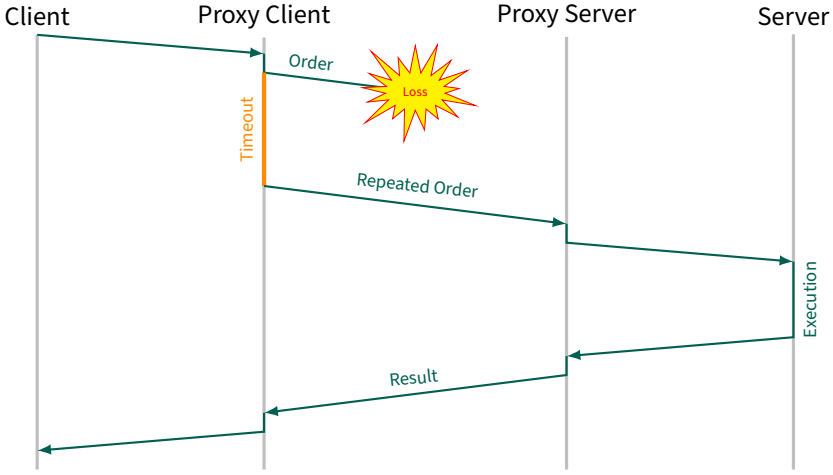
- Loss of Order Messages
- Loss of Result Messages
- Server Failure
  - Before Execution
  - After Execution

- **General solution:** Client proxy uses timeout and repeats order
- **Issues:**
  - Idempotence of server operation: Is multiple execution of the operation tolerable?
- Distinction:
  - **at-most-once-semantics**: order is executed **at most** once
  - **at-least-once-semantics**: order is executed **at least** once
  - **maybe-semantics**: no statement possible

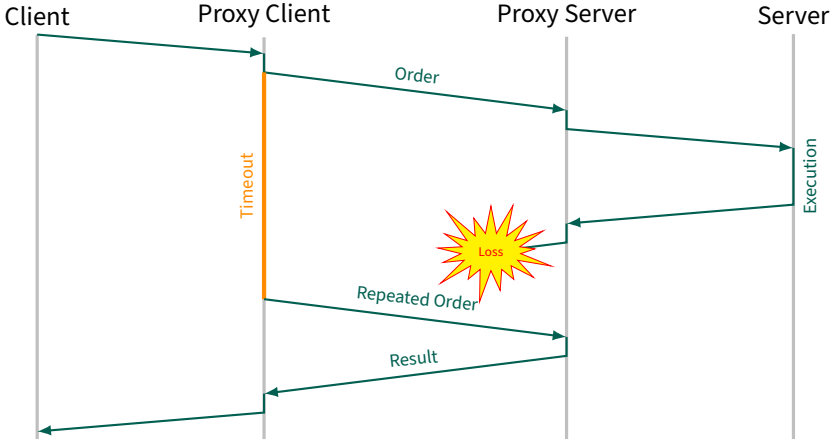
RPC Workflow: Faultless Case



RPC Workflow: Loss of Order Message

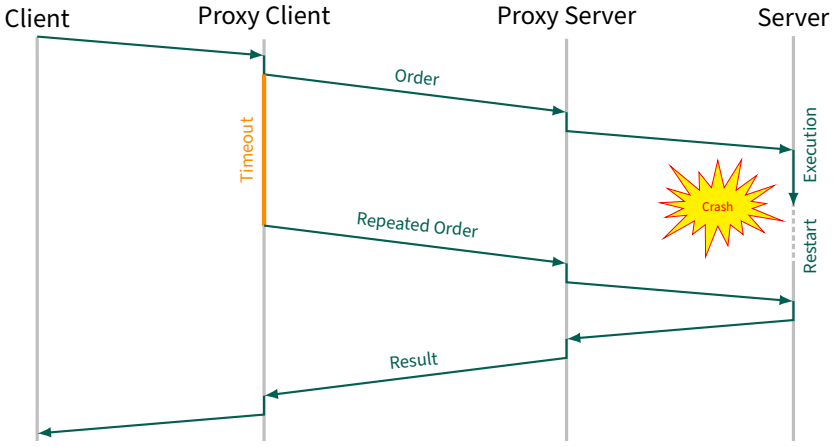


RPC Workflow: Loss of Result Message



- Requires numeration of order messages and persistent logging on harddrive (server-sided)

RPC Workflow: Server Failure after Execution



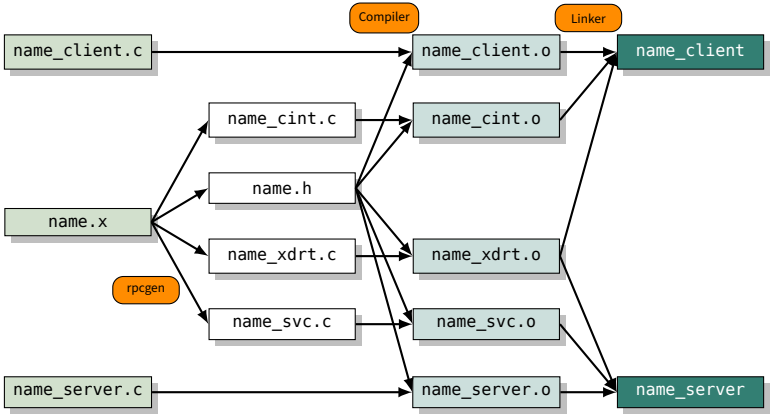
- Requires numeration of order messages and persistent logging on harddrive (server-sided)

Implementation Variant: Sun RPC

OSI Model	Sun RPC
Application	RPC
Presentation	XDR
Session	–
Transport	UDP oder TCP
Network	IP
Data Link	IEEE 802 (z.B. Ethernet)
Physical	–

- By means of `rpcgen` automatic generation of
  - Client Proxy
  - Server Proxy
- Automatic back and forth transformation of data into XDR format
- Choice of timeout value
- Choice of number of repetitions
- Semantics depend on chosen transfer protocol
  - UDP: maybe-Semantik
  - TCP: at-most-once-Semantik

Automatic Generation with rpcgen



- Automatic generation by means of an **Interface Description Language (IDL)** is widely used (Corba, DCE, XCB, …)

Discussion

- RPC provides a simple and comfortable programming interface, but absolutely no parallelism/concurrency between client and server
- More efficient service architectures through general channel concept
- In the distributed case, channel communication needs to be supplemented with bridges
  - Considering process federation, bridges are realized through processes and local channels
  - Considering kernel federation, bridges are located in the kernel: a communication operation accesses a channel on another computer via the bridge

Literature

[ADG90] Autenrieth, K., H. Dappa and M. Grevel: *Technik verteilter Betriebssysteme*. Hüthig, 1990 , Anhang A

[TaSt02] Tanenbaum, A. S. and M. van Steen: *Distributed Systems*. Prentice Hall, 2002 , Chapter 1

[Sin97] Sinha, P. K.: *Distributed Operating Systems*. IEEE Press, 1997 , Chapter 4