Distributed Systems

Lehrstuhl für Informatik IV
RWTH Aachen

Prof. Dr. Otto Spaniol
Dipl.-Inform. Dirk Thißen
Organisation

Exercises

- about all 14 days
- Room AH V, RWTH Aachen
- Teacher-centred exercises

10. May 2004
24. May 2004
14. June 2004
28. June 2004
12. July 2004
26. July 2004

Slide copies and Video records to the lecture

Available on our homepage:
http://www-i4.informatik.rwth-aachen.de/teaching/lectures/sub/vs/vsSS04

Written exam

At the end of the summer term

Contact

Dirk Thißen
Lehrstuhl für Informatik IV, Room 4226 (Building E1)
Phone: 0241 / 80 - 21450
eMail: thissen@informatik.rwth-aachen.de
Available in the library, section DiS


Further literature


Classification of the lecture

**Useful prerequisites:**

→ Lecture on *Data Communication or Internet Technology*

**Following events:**

→ Seminar “Data Communications and Distributed Systems”
→ Diploma Theses at Informatik 4
→ Several lectures, e.g. Multimedia Systems, Security in Communication Networks, …

**Current (related) events:**

→ Lecture "Dependable Distributed Systems“, Prof. Gärtner
What is a Distributed System?

There are several (different) definitions for Distributed Systems.

**Definition:** A Distributed System is a system with spatially distributed components, which do not use shared memory and which are managed in a decentralised manner. For realising common goals, a cooperation of these components is possible. [Popien]
... More Definitions...

or A **Distributed System** is a collection of independent computers that appears to its users as a single coherent system. [*Tanenbaum*]

or A **Distributed System** is a system in which hardware or software components located at networked computers communicate and coordinate their actions only by passing messages. [*Coulouris*]

or A **Distributed System** is a collection of autonomous computers linked by a network and equipped with distributed system software. [...] The distributed system software enables the comprising computers to coordinate their activities and to share system resources. [*Tari*]

Or sometimes: “A distributed system is one in which the failure of a computer which you didn’t even know existed can render your own computer unusable.” (Leslie Lamport)
Contents

1. Introduction: Classification and characteristics of Distributed Systems
   Types of Middleware for Distributed Systems
   Communication in Distributed Systems:
   Client/Server-Model, Remote Procedure Call, Remote Method Invocation, Message-Oriented Systems

2. Processes and Threads, (Mobile) Agents

3. Basic principles: Naming, Concurrency, Synchronisation, Transactions, Replication, Consistency

4. Middleware: DCOM, Java RMI, CORBA

5. The Common Object Request Broker Architecture

6. Services and Components: the Web Services concept
Classification of Distributed Systems

Classification by
- Hardware: computing power (tightly resp. loose coupling)
- Software: operating systems and applications (tightly resp. loose coupling)

<table>
<thead>
<tr>
<th>Hardware</th>
<th>Software</th>
<th>loose coupling</th>
<th>tightly coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>network with independent</td>
<td>network with independent PCs/workstations and shared</td>
<td>distributed application on several homogeneous</td>
</tr>
<tr>
<td></td>
<td></td>
<td>servers</td>
<td>computers</td>
</tr>
<tr>
<td>tightly coupling</td>
<td>-- not suitable --</td>
<td>multi-processor operating systems</td>
<td></td>
</tr>
<tr>
<td></td>
<td>network operating systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loose coupling</td>
<td>distributed operating</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(most used concept: network with independent computers but shared resources)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>distributed operating</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>systems</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(for a user, the network looks like a single computer)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Distributed Operating Systems

Example: Cluster of workstations
Network Operating System

Example: the Internet
Evolution of Distributed Systems

Since the 1980s a general trend to rejection from centralised systems can be seen.

The development of distributed systems was supported by

- **Hardware**: explosion in the performance of semiconductor chips
  - increasing performance while prices and dimensions were decreased
  - usage of more complex software on more and more computers

- **Communication**: development of fast local networks
  - reduction of access times
  - forerunner Ethernet

- **Software engineering**: modules, interfaces, objects
  - Remote Procedure Call, object-oriented modelling

- **Autonomy of organisations**: Decentralisation
  - no more strict hierarchically organisation forms in enterprises
Why Distributed Systems?

Advantages

• *Adaptation of capacities*
  - Adaptation of system size to current requirements

• *Integration of existing solutions*
  - Usage of legacy systems by new system components
    (it is not necessary to develop a new system with the same functionalities)

• *Risk minimisation*
  - By expanding the system, the risk of overloading some system components is minimised

• *Flexibility, adaptability*
  - Lower costs by manageable structures

• *Autonomy*
  - Tolerance of single failures by other components
... but...

Disadvantages and problems

- **Technology**
  - more complexity by distribution and system heterogeneity

- **Complex infrastructures**
  - management of the whole system

- **Lack of software**
  - no experience in developing distributed software

- **Security risks**
  - new sources of failures by new components
  - data protection: easier access as in separated data storage

- **Parallel events**
  - ordering of events occurring on different computers

- **Problems with consistency**
  - access to distributed data
Requirements to Distributed Systems

- **Openness**: interoperability and portability of a system
- **Integration**: handling the heterogeneity
- **Flexibility**: adapt to the evolution
- **Modularity**: for achieving flexibility
- **Federation**: connection of autonomous systems
- **Manageability**: handling the complexity
- **Service quality**: guarantee of user requirements to special services
- **Security**: protection against unauthorised accesses
- **Transparency**: hide implementation details and complexity
Transparencies

*Transparency*: hide implementation details

Important in distributed systems:

- *distribution transparency* (hide the complexity of a distributed system)
  
- eases the usage of distributed systems software
- hides internal events from a user
- supports an application programmer

some transparencies:

- Access transparency
- Location transparency
- Replication transparency
- Concurrency transparency
- Migration transparency
- Failure transparency
- Resource transparency
## Transparencies

<table>
<thead>
<tr>
<th>Transparency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access</td>
<td>Hide differences in data representation and how a resource is accessed</td>
</tr>
<tr>
<td>Location</td>
<td>Hide where a resource is located</td>
</tr>
<tr>
<td>Migration</td>
<td>Hide that a resource may move to another location</td>
</tr>
<tr>
<td>Relocation</td>
<td>Hide that a resource may be moved to another location while in use</td>
</tr>
<tr>
<td>Replication</td>
<td>Hide that a resource may be run in several instances</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Hide that a resource may be shared by several competitive users</td>
</tr>
<tr>
<td>Failure</td>
<td>Hide the failure and recovery of a resource</td>
</tr>
<tr>
<td>Persistence</td>
<td>Hide whether a (software) resource is in memory or on disk</td>
</tr>
</tbody>
</table>
Handling Distribution: Middleware

For handling the large set of transparencies, a special distribution infrastructure is needed.

This infrastructure is called **Distribution Platform** (or: **Middleware**). The functionalities can be compared with the higher layers of the OSI reference model for communication protocols:

- Management of dialogs (layer 5),
- Independent transfer syntax (layer 6),
- Security, transactions, synchronisation, ...

The middleware is build upon different operating systems, networks, and communication protocols (layer 1-4).

The applications can be programmed in different languages.

... and the middleware realises the cooperation between the software components.
Software Concepts

<table>
<thead>
<tr>
<th>System</th>
<th>Description</th>
<th>Main Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distributed Operating System</td>
<td>Tightly-coupled operating system for multi-processors and homogeneous multicomputers</td>
<td>Hide and manage hardware resources</td>
</tr>
<tr>
<td>Network Operating System</td>
<td>Loosely-coupled operating system for heterogeneous multicomputers (LAN and WAN)</td>
<td>Offer local services to remote clients</td>
</tr>
<tr>
<td>Middleware</td>
<td>Additional layer atop of Network Operating Systems implementing general-purpose services</td>
<td>Provide distribution transparency</td>
</tr>
</tbody>
</table>
Relation to OSI reference model

![Diagram of OSI reference model]

- Application
  - Application protocol
- Middleware
  - Middleware protocol
- Transport
  - Transport protocol
- Network
  - Network protocol
- Data link
  - Data link protocol
- Physical
  - Physical protocol

Chapter 1: Introduction
# OSI layers

<table>
<thead>
<tr>
<th>Layer</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application</td>
<td>Protocols that are designed to meet the communication requirements of specific applications, often defining the interface to a service.</td>
<td>HTTP, FTP, SMTP, CORBA IIOP</td>
</tr>
<tr>
<td>Presentation</td>
<td>Protocols at this level transmit data in a network representation that is independent of the representations used in individual computers, which may differ. Encryption is also performed in this layer, if required.</td>
<td>Secure Sockets (SSL), CORBA Data Rep.</td>
</tr>
<tr>
<td>Session</td>
<td>At this level reliability and adaptation are performed, such as detection of failures and automatic recovery.</td>
<td></td>
</tr>
<tr>
<td>Transport</td>
<td>This is the lowest level at which messages (rather than packets) are handled. Messages are addressed to communication ports attached to processes, Protocols in this layer may be connection-oriented or connectionless.</td>
<td>TCP, UDP</td>
</tr>
<tr>
<td>Network</td>
<td>Transfers data packets between computers in a specific network. In a WAN or an internetwork this involves the generation of a route passing through routers. In a single LAN no routing is required.</td>
<td>IP, ATM virtual circuits</td>
</tr>
<tr>
<td>Data link</td>
<td>Responsible for transmission of packets between nodes that are directly connected by a physical link. In a WAN transmission is between pairs of routers or between routers and hosts. In a LAN it is between any pair of hosts.</td>
<td>Ethernet MAC, ATM cell transfer, PPP</td>
</tr>
<tr>
<td>Physical</td>
<td>The circuits and hardware that drive the network. It transmits sequences of binary data by analogue signalling, using amplitude or frequency modulation of electrical signals (on cable circuits), light signals (on fibre optic circuits) or other electromagnetic signals (on radio and microwave circuits).</td>
<td>Ethernet base-band signalling, ISDN</td>
</tr>
</tbody>
</table>

Chapter 1: Introduction
Middleware and Openness

In an open middleware-based distributed system, the protocols used by each middleware layer should be the same, as well as the interfaces they offer to applications.
## Comparison between Systems

<table>
<thead>
<tr>
<th>Item</th>
<th>Distributed OS</th>
<th>Network OS</th>
<th>Middleware-based OS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Multiproc.</td>
<td>Multicom.</td>
<td></td>
</tr>
<tr>
<td>Degree of transparency</td>
<td>Very High</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>Same OS on all nodes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Number of copies of OS</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>Basis for communication</td>
<td>Shared memory</td>
<td>Messages</td>
<td>Files</td>
</tr>
<tr>
<td>Resource management</td>
<td>Global, central</td>
<td>Global, distributed</td>
<td>Per node</td>
</tr>
<tr>
<td>Scalability</td>
<td>No</td>
<td>Moderately</td>
<td>Yes</td>
</tr>
<tr>
<td>Openness</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
</tr>
</tbody>
</table>

Chapter 1: Introduction
Middleware - Approaches

**Distributed file systems**
- Distribution transparencies for *files*
- Popular because of its scalability

**Distributed documents**
- Example World Wide Web
- Information are seen as documents, addressed by an URL

**Remote Procedure Call (RPC)**
- Early middleware approach for transparency in *communication*

**Distributed objects**
- Enhancements of RPC to access to *remote objects*
- Only the interface can be seen from outside, the implementation details are transparent
File System Based Middleware

- Bases on file exchange
- Transfer Models
  - (a) upload/download model
  - (b) remote access model
Document-Based Middleware

- Example World Wide Web (WWW)
  - a big directed graph of documents
  - Transparency for documents
Communication in Distributed Systems

Communication between processes is a central part of each distributed system. Without communication, only a collection of autonomous processes without a possibility of cooperation would exist.

*Problem in distributed systems:*
  - High overhead caused by protocol stack
  - Need for an easy model for efficient communication

*Idea:*
  - Structuring of the operating system as a set of cooperating processes *(Servers)*
  - Servers offer services for users *(Clients)*
  - Communication by using easy primitives

- **Client/Server Model**
- **Remote Procedure Call**
Client/Server Model

- Communication model with low management overhead
- Connection-less request/reply protocol (simplest case: only need for layer 1, 2, and 5)
- Direct addressing of the server by a client:
  - normally, the client knows the server's address
  - simple addressing: machine.process
  - could by written (for process 4226 on host 137.226.12.221):
    - 137.226.12.221.4226 or 4226@137.226.12.221
Communication is made by using two system calls:

send(a, &mp) transmits a message referenced by mp to process a.

The caller is blocked while sending the message.

receive(a, &mp) stores a messages received at address a in a buffer referenced by mp. The caller is blocked while receiving the message.

Needed: synchronisation of send and receive.
Example - Server

Server: creates a message m2 based on the contents of message m1

header.h contains common type definitions for client and server

```c
#include <header.h>

void main(void) {
    struct message ml, m2;
    int r;

    while(TRUE) {
        receive(FILE_SERVER, &ml);
        switch(ml.opcode) {
            case CREATE:  r = do_create(&ml, &m2); break;
            case READ:     r = do_read(&ml, &m2); break;
            case WRITE:    r = do_write(&ml, &m2); break;
            case DELETE:   r = do_delete(&ml, &m2); break;
            default:       r = E_BAD_OPCODE;
        }
        m2.result = r;
        send(ml.source, &m2);
    }
}
```

blocking, wait for incoming messages for FILE_SERVER

Send result to the calling client
Example - Client

```c
#include <header.h>
int copy(char *src, char *dst){
    struct message ml;
    long position;
    long client = 110;

    initialize();
    position = 0;
    do {
        ml.opcode = READ;
        ml.offset = position;
        ml.count = BUF_SIZE;
        strcpy(&ml.name, src);
        send(FILE_SERVER, &ml);
        receive(client, &ml);
        /* procedure to copy file using the server
        * message buffer
        * current file position
        * client's address
        */
        /* prepare for execution
        */
        /* operation is a read
        */
        /* current position in the file
        */
        /* how many bytes to read*/
        /* copy name of file to be read to message*/
        /* send the message to the file server*/
        /* block waiting for the reply*/
        /* Write the data just received to the destination file.
        */
        ml.opcode = WRITE;
        ml.offset = position;
        ml.count = ml.result;
        strcpy(&ml.name, dst);
        send(FILE_SERVER, &ml);
        receive(client, &ml);
        position += ml.result;
    } while( ml.result > 0 );
    return(ml.result >= 0 ? OK : ml.result); /* return OK or error code */
}

Client: uses FILE_SERVER, to copy a file
```

Sending of the request and blocking while waiting for the reply.
Enhancement of Client/Server Model (common method):
Using a **Name server**

1. Request for destination address of searched server
2. Reply with server address
3. Request to server
4. Reply by server
A distinction is made between blocking and non-blocking primitives. The application programmer can choose from them.

**Blocking primitives**
- Blocks a process while sending a message
- Further operations are worked on only after sending the complete message
- In the same way: receiving a message

**Non-blocking primitives**
- The message is copied in a buffer managed by the operating system
- After the copy operation the sender is unblocked
- Speed-up by sending the message and working on the further operations in parallel
- Disadvantage: the sender does not know when the transmission is finished and the buffer can be used again.
Another distinction of primitives can be made by **buffers**

**Primitives without buffers**
- With `receive(a, &mp)` the kernel is informed that the calling process wants to listen on address `a` for a message to be stored at `&mp`.
- Problems:
  1. Message lost for late `receive` (the kernel does not know where to store the message)
  2. Usage of same address by several processes

**Primitives with buffers**
- The kernel stores the received message for a certain time
- Problem: the kernel has to provide and manage own buffers
The separation of client and server can take place at different points in the whole application, considering structure and tasks of the whole system.
Multi-level Systems

The server itself can delegate parts of its tasks to other servers. In doing so, the server becomes a client for the new server.

This separation by areas of responsibilities is called **vertical distribution**

Chapter 1: Introduction
Other architectures

Horizontal distribution arises, when server (or client) are splitted into several equivalent parts (e.g. a web server farm).

... and there are other organisation models for client/server systems.
Remote Procedure Call

Base paradigm for communication (especially for local invocations):

**Input and output** of data

By calling communication primitives **send** and **receive** explicitly:

**Exchange** of data

*Need for another mechanism:*

Let distributed computations look like centralised ones

Solution (Birell and Nelson, 1984): A program calls a subroutine located on another computer.

Known as **Remote Procedure Call, RPC**

*Principle:*

When a program located on host $A$ calls a subroutine located on host $B$, the calling process on host $A$ is suspended while the subroutine on host $B$ is executed. The exchange of parameters and messages is invisible for the user.
Remote Procedure Call

**Mechanism is transparent for the calling client**

**a) Local call to subroutine**

**Stub:**
Acts as a proxy for the subroutine and passes the request to the fitting server

**Marshalling:**
translates data into an independent transport format

**b) Remote Procedure Call**
**RPC - Process**

**Advantage:**
- the distributed execution works without explicitly calling the communication primitives by client resp. server.
- Details are hidden by stubs.
Remote Procedure Call - Example

**Client:**
- Calls the procedure `add(2, 4)`
- The data are pushed onto the stack
- The client library contains a reference to the **client stub** instead referencing a local procedure directly
- Transmission of control to the stub

**Client stub:**
- Generates a message from the request which can be sent to the server
- Marshalling of the data, i.e. the data are transformed in a given message structure
- The `send` primitive initiates the transmission of the message
- The stub calls `receive` and blocks

**Client:**
- \( x = \text{add}(2, 4) \)

**Client stack**
- \( x = \text{add}(2, 4) \)
- Client stack:
  - `add`
  - `2`
  - `4`

**Stub:**
- `mp: add 2 4`
  - `send(ServerAdr, &mp)`
  - `receive(ClientAdr, &mp)`
Remote Procedure Call - Example

Server host:
- The server waits for incoming requests by executing `receive`.
- The kernel passes the message to the server stub.
- The stub extracts the data, pushes them onto the server stack and calls the (local) subroutine.
- When the computation is finished, the server calls its stub. The stub marshals the result and sends a message back to the calling client.
**Client host:**
- The client stub waits for the reply performing the receive
- The message is written in a buffer
- The client stub is unblocked
- The stub extracts the result and pushes it onto the client stack
- The client is given back control and gets – like in the local case – the data from its stack
Problems with RPC

The communication between identical computers causes no problems.

In heterogeneous systems, there can be some problems:

- Different character representations (ASCII - EDCDIC)
- Different representation of integers (little endian - big endian)

Another problem are non-scalar data types:

- There is no shared address space – how to transmit pointers and references?
- Normally: copy/restore, i.e. transmit referenced data directly
- But this works only for simple structures and arrays; for more complex structures which are defined by the programmer, the server cannot understand the structure.

Conclusion: client and server have to agree on a common data format which can be transmitted with the RPC.
Generation of Stubs

For defining common data formats and supporting the implementation of stubs, very often only the interfaces are defined. The stubs are generated from these definition automatically, containing the whole transfer syntax which can be used.

![Diagram of stub generation process]
RPC: *de-facto-Standard for communication in distributed systems*

The RPC can be optimised for special scenarios. Some variants are

- *Lightweight RPC* for inter-process communication in the same address space
- *Asynchronous RPC* for decoupling client and server
Bershad's LRPC

The duration of a RPC depends on several parameters (several copy operations on the parameters, network delay) and can take significant time. For some circumstances, the execution can be speeded up:

→ Use of shared memory for inter-process communication on the same host
  • Only one copy operation for parameter passing (instead of four copy operations like in RPC)
  • The client can access the server code using defined access points
  • Speed-up up to three times

→ Name: Lightweight RPC, LRPC
  (lightweight means easier and faster)
Lightweight Remote Procedure Call

1. Copying the arguments
2. Request
3. Up call
4. Execution of procedure and storing the result
5. Return

Common stack
Asynchronous RPC

RPC: **asynchronous**, i.e. the client blocks till the server request comes in. But in some situations, this waiting is not necessary:

- The client can work on other tasks till the result from the server comes in
- The client gets back no result

In these cases: **asynchronous RPC**, i.e. the client does not block, but is able to work on other tasks, till the server interrupts it with the result:

*one-way RPC*: do not wait for an acknowledgement, but work on directly

*deferred synchronous RPC*: request and reply both are using asynchronous RPC
Remote Method Invocation

Base principle of **RPC**: a client calls a *subroutine/procedure* in a separate server process.

**Remote Method Invocation (RMI)**: builds upon the same principle: an object calls a *method of another object*. The principle is the same, but the underlying model is changed (and enhanced with some more functionality).
Peer-to-Peer computing

- Weakening of client and server roles
- Connection between any pair of computers
- Flexible network of cooperations
- Well known example: File Sharing, e.g. Napster
**Message-based Communication**

**Message-based** (or **event-based**) communication: objects are sending information to the receivers by using messages. These messages are sent **asynchronously** (in contrast to the basic principles of the communication forms shown before). The communication system is responsible for delivering the messages.

Distinction between two kinds of systems:

- **Persistent communication**
  The communication system stores messages till it is able to deliver them to the receiver. Sender and receiver are decoupled completely, the receiving object is allowed to be not available when the message is sent.

- **Transient communication**
  The communication system only stores the message while sender and receiver are up and running. If a message cannot be delivered because the receiver cannot reached, the message is deleted.
Message-queuing Systems

*Persistent, asynchronous communication*: the sender is guaranteed that its message reaches the receiver. Only when the message is delivered (and if the receiver reads it) is unknown.
General Architecture of a Message-Queuing System
Message Broker

For integrating several applications in a distributed information system, some more is necessary: these applications can use different message formats (like in RPC). A so-called **message broker** suits for converting messages in other formats.
Stream-based Communication

The communication model looked at are not considering special requirements in information delivery. But for multimedia applications, time restrictions are very important.
Group Communication

If a message can be sent to a group of receivers, group communication is needed for efficient communication:

- **Fault tolerance/availability**
  A service is replicated, i.e. realised on several hosts. A client can send its request to all members of the server group. All of them are performing the request. Even if some of the servers crash, the service is already available.

- **Performance increase**
  When using replicated servers, e.g. for placing data near to a client and thus improve its access to the data, all databases of the servers have to be in a consistent state. When a data change occurs, the new data have to be transmitted to all members of the server group.

- **Event notification**
  One service type is a distribution service: if a certain event occurs, a certain user group has to be informed. One example for such a service is a news system, where a new incoming news is reported to a user group.

Needed: **multicast communication**