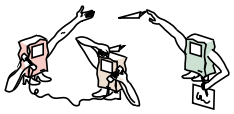
Distributed Applications

Johann Schlichter
Institut für Informatik
TU München, Munich, Germany



March 2002
Material for the Course Distributed Applications
(Student Script[1])

[1]Script generated by Targeteam

Contents

- Prof. J. Schlichter

  Lehrstuhl für Angewandte Informatik / Kooperative Systeme, Fakultät für Informatik, TU München

  Arcisstr. 21, 80290 München

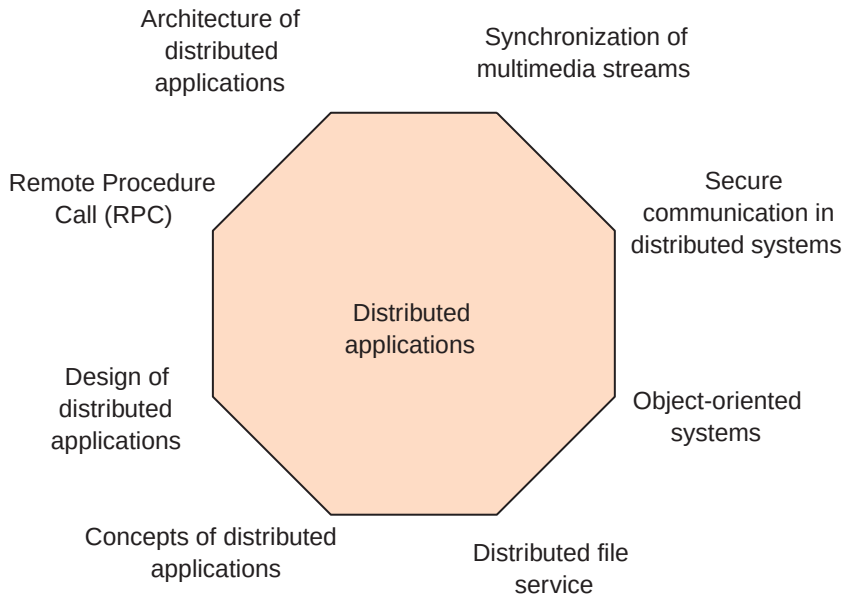
  Email: schlichter@in.tum.de (URL: mailto:schlichter@in.tum.de)
  Tel.: 089-289 25700

  URL: http://www11.informatik.tu-muenchen.de/

Chapter 1

Overview

This lecture aims at introducing basic concepts which play an important role in the design and implementation of distributed applications.



1.1 Lecture Content

This lecture presents various aspects and mechanisms for the design and implementation of distributed applications

- Basic principles for the design of distributed applications.

Terminology, communication mechanisms, client-server model, aspects of Remote Procedure Call.

- Introduction to distributed transactions and group communication.

  2 phase commit, aspects of consistent message delivery ("atomic multicast", virtual synchronization) in groups, group management.

- Information replication and distributed file systems.

  consistency of replicated information, concurrency control.

- Methodology for the design of distributed applications.

  design framework ODP ("Open Distributed Processing"), test support, formal specification of distributed applications (Estelle).

- Object-oriented distributed systems.

  Impact of the object-oriented paradigm on design of distributed applications, especially Corba and RMI ("Remote Method Invocation") in Java environments.

- Secure communication in distributed systems.

  brief introduction to cryptography, authentication of users and systems, and discussion of the Kerberos system.

1.2 Bibliography

Thhe following literature was used to prepare this lecture.

1.2.1 Course Text Books

- Uwe M. Borghoff, Johann Schlichter, "Computer-Supported Cooperative Work - Introduction to Distributed Applications", Springer-Verlag, 2000; especially chapter 1.

- George F. Coulouris, Jean Dollimore, Tim Kindberg, "Distributed Systems: Concepts and Design", Addison-Wesley, 2001

- R. Orfali, D. Harkey, J. Edwards, "The essential distributed objects survival guide", John Wiley & Sons, 1996.

- Andrew S. Tanenbaum, Maarten van Steen, "Distributed Systems - Principles and Paradigms", Prentice Hall, 2002

1.2.2 Further Reading

- S. Allamaraju et al., "Professional Java Server Programming - J2EE Edition", Wrox Press, 2000
- A.L. Ananda, B. Srinivasan (Eds), "Distributed Computing Systems: Concepts and Structures", IEEE Computer Society Press, 1991
- Uwe M. Borghoff, "Catalogue of Distributed File/Operating Systems", Springer-Verlag, 1991
- George A. Champine, "MIT project Athena: a model for distributed campus computing", Digital Press, 1991
- John R. Corbin, "The Art of Distributed Applications", Springer-Verlag, 1991
- A. Goscinski, "Distributed Operating Systems: The Logical Design", Addison-Wesley, 1991
- Tom Lyons, "Network Computing System: Tutorial", Prentice-Hall, 1991
- Horst Langendörfer, Bettina Schnor, "Verteilte Systeme", Carl Hanser Verlag, 1994 (in German)
- Max Mühlhäuser, Alexander Schill, "Software Engineering für verteilte Anwendungen", Springer Verlag, 1992 (in German)
- Sape Mullender (Ed), "Distributed Systems", Addison-Wesley (ACM Press), 1994
- Open Software Foundation, "Introduction to OSF DCE", Prentice Hall, 1992
- Andrew S. Tanenbaum, "Computer Networks", Prentice Hall, 1996
- Andrew S. Tanenbaum, "Modern Operating Systems", Prentice Hall, 2001
- Kenneth J. Turner, "Using formal Description Techniques: an Introduction to Estelle, LOTOS and SDL", John Wiley & Sons, 1993
- Michael Weber, "Verteilte Systeme", Spektrum Akademischer Verlag, 1998 (in German)

1.3 Abbreviations

API	Application Programming Interface
CSCW	Computer Supported Cooperative Work
DCE	Distributed Computing Environment (OSF)
DIT	Directory Information Tree (LDAP)
DME	Distributed Management Environment (OSF)
DNS	Domain Naming Service
EAR	Enterprise Archive
EJB	Enterprise Java Beans
IIOP	Internet Inter-ORB Protocol
IPC	Interprocess communication
ISO	International Standards Organization
J2EE	Java 2 Platform Enterprise Edition
JAF	Java Beans Activation Framework
JAR	Java Archive
JDBC	Java Database Connectivity Extension
JMS	Java Message Service
JNDI	Java Naming and Directory Interface
JSP	Java Server Pages
JTA	Java Transaction API
LDAP	Lightweight Directory Access Protocol
LDIF	LDAP Data Interchange Format
LOTOS	Language of Temporal Ordering Specification
NFS	Network File System (von Sun)
ODP	Open Distributed Processing
OMA	Object Management Architecture
OMG	Open Management Group
ONC	Open Network Computing (von Sun)
ORB	Object Request Broker (Corba)
OSF	Open Software Foundation
QoS	Quality of Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SDL	Specification and Description Language
SSL	Secure Socket Layer
WAR	Web Archive
XDR	eXternal Data Representation

Chapter 2

Introduction

2.1 Issues

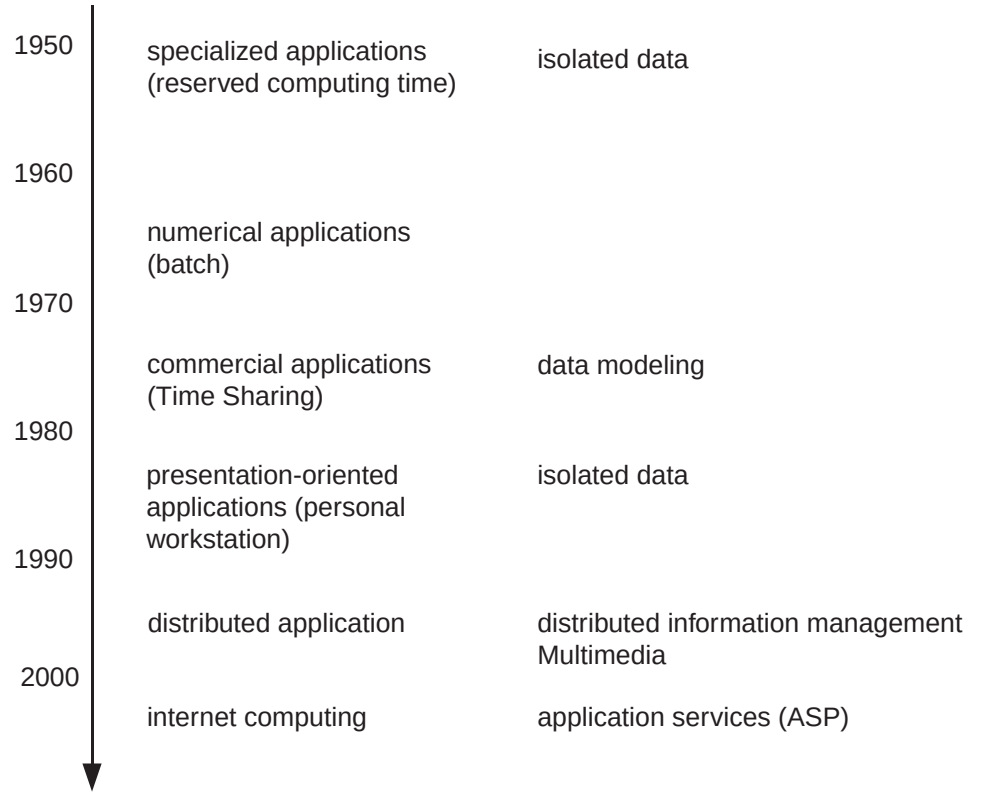
This section focuses on the following issues

- Motivation for distributed systems and distributed applications.
- Basic terminology for distributed systems, e.g. terms like *distributed applications*, and *interface*.
- Introduction to influential distributed systems, such as Amoeba, NFS File system, OSF, Mach and Java 2 Platform Enterprise Edition.

2.2 Background

Applications in a wide variety of relevant domains (e.g., collaborative information spaces, workflow management, telecooperation, autonomous agents) are flourishing. These applications share the concept of distribution.

2.2.1 Development of computer technology



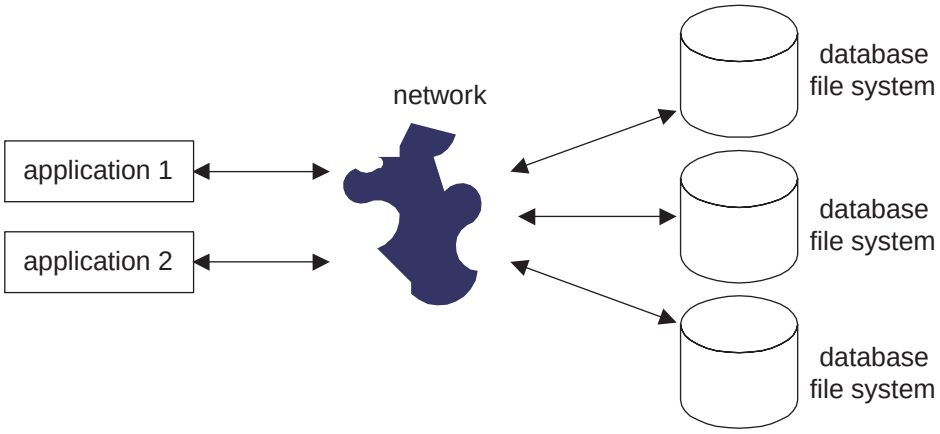
- **Situation in the Future**

  Networks of heterogeneous computers, applications using shared resources which are geographically dispersed, information communication (i.e. improved information flow), and activity coordination.

- Examples:

  – online flight-reservation.
  – distributed money machines.
  – video conferencing applications, for example Mbone (see the application domain "Computer-supported Cooperative Work").
  – World Wide Web.

2.2.2 Enterprise Computing



Enterprise computing systems have close, direct coupling of application programs running on multiple, heterogeneous platforms in a networked environment.

- These systems must be completely integrated and very reliable, in particular

  information consistency, even in case of partial system breakdown.

  security and guaranteed privacy.

  adequate system response times.

  high tolerance in case of input and hardware/user errors (fault tolerance).

  autonomy of the individual system components.

- It is important for the application designer to be familiar with the techniques and mechanisms for the design and implementation of reliable, fault-tolerant and consistent distributed applications.

2.3 Key Characteristics

2.3.1 Motivation

The following factors contribute to the increasing importance of distributed systems:

The costs for processors and storage units have dwindled.

Network technology with high bandwidth is now available.

Large, centralized systems have insufficient and often unpredictable response times; in particular, user interfaces cannot be personalized by the user.

Growing number of applications with complex information management and complex graphical user interfaces.

Growing cooperation and usage of shared resources by geographically dispersed users; caused by the globalization of markets and enterprises,
e.g. applying telecooperation (groupware, CSCW) and mobile communication to improve distributed teamwork.

Informal definition

The term *distributed system* may be defined informally:

1. after *Tanenbaum*: a distributed system is a collection of independent computers which appears to the user as a single computer.
2. after *Lamport*: a distributed system is a system that stops you from getting any work done when a machine you've never heard of crashes.
3. **Definition:** We define a **distributed system** as one in which hardware and software components located at networked computers

   communicate and

   coordinate their actions mainly by passing messages.

Examples of distributed systems

Besides the Internet and intranet also mobile and ubiquitous computing have led to the design of distributed systems.

- The *Internet* is a vast interconnected collection of computer networks and a very large distributed system

  heterogenous components (computers, devices, etc).

  a large number of services, such as World Wide Web, file services, audio/video channels.

- The *intranet* is a portion of the Internet that

  is separately administered by an organization,

  has a boundary that can be configured to enforce local security policies.

Methods of distribution

There are five fundamental methods of distribution:

1. Hardware components.
2. Load.
3. Data.
4. Control, e.g. a distributed operating system.
5. Processing, e.g distributed execution of an application.

In the following sections, we will focus on the latter three types of distribution, in particular on the processing distribution.

2.3.2 Properties of distributed systems

Distributed systems have a number of characteristics, among them are:

1. Existance of multiple functional units (physical, logical).
2. Local distribution of physical and logical functional units.
3. Functional units break down independently.
4. A distributed operating system controls, integrates and homogenizes the functional units, thus providing distributed component control. Distributed operating systems are characterized by their modular structure and extendability.
5. $\rightarrow$ Transparency (see page 28): details irrelevant for the user (for example distribution of data across several computers) remains hidden in order to reduce complexity.
6. Cooperative autonomy during the interaction among the physical and logical functional units $\Rightarrow$ implies concurrency during process execution.

2.3.3 Challenges of distributed systems

The design of distributed systems poses a number of challenges

- *Heterogeneity* applies to networks, computer hardware, operating systems, programming languages and implementations by different programmers.

  Use of *middleware* to provide a programming abstraction masking the heterogeneity of the underlying system.

  middleware provides a uniform computational model for use by the programmers of servers and distributed applications.

  middleware examples are → Corba (see page 164), Java → RMI (see page 156).

- *Openess* requires standardized interfaces between the various resources.
- *Scalability*; adding new resources to the overall system.
- *Security*; for the information resources.

2.3.4 Advantages of distributed systems

Distributed system vs. mainframes

As opposed to a centralized system, a distributed system has the following features:

- Economics (better price-performance than mainframes).
- Shorter response time.
- Inherent distribution.
- Reliability.

  Replication of functional units and information brings about advantages such as increased availability; partial breakdowns rather than total breakdowns.

- Incremental growth.

  extension rather than replacement if additional computing power is required.

Distributed system vs. independent PC's

Compared with isolated personal computers, distributed systems have the following advantages

1. Shared utilization of resources.

   data sharing (e.g. airline reservation system used by travel agencies) or device sharing (hardware components like printers).

2. Communication.

   facilitating human-to-human interaction, e.g. email, tele-conferencing, video-conferencing.

3. Better performance and flexibility than isolated personal computers.

   examples are the computation by chess programs, factorization of a number with 100 digits, computation of a complex image.

   workload balancing; use of specialized computers.

4. Simpler maintenance if compared with individual PC's.

   Software updates, data backup

Disadvantages of distributed systems

Although we have seen a couple of advantages of distributed systems, there are obvious disadvantages.

1. Network performance parameters.

   latency: delay that occurs after a send operation is executed before data starts ro arrive at the destination computer.

   data transfer rate: speed at which data can be transferred between 2 computers once transmission has begun (bits/s).

   total network bandwidth: total volume of traffic that can be transferred across the network in a give time.

2. Dependency on reliability of the underlying network.

3. Higher security risk due to more possible access points for intruders and possible communication with insecure systems.

4. Software complexity

   Consideration of the communication among distributed sites.

   Testing and debugging of such a software is complex, time consuming, and difficult.

2.3.5 Examples for development frameworks

There is a high motivation to use standardized development frameworks for the design and implementation of distributed applications.

- → Sun Network File System (NFS) (see page 20) by SUN

  a distributed file system behaving like a centralized file system.

- Open Network Computing (ONC) by SUN

  platform for distributed application design; it contains library routines for → remote procedure call (RPC) (see page 59) and for → external data representation (XDR) (see page 82).

- Distributed Computing Environment (DCE) by → Open Software Foundation (OSF) (see page 19)

  a set of integrated components providing a coherent environment for the design and implementation of applications in heterogeneous, distributed systems.

- distributed applications in → ODP (Open Distributed Processing) (see page 89) by ISO

  OSI reference model to specify the global characteristics of a distributed system; apart from the specification of the interfaces, it also contains the behavioural description of the components of distributed applications.

  ANSAware is one of the first realizations of this idea.

- → Common Object Request Broker Architecture (CORBA) (see page 164) by OMG

  defines a common architecture model for heterogeneous environments based on the object-oriented paradigm. It aims at interoperability between distributed objects residing on different platforms.

- Java 2 Platform Enterprise Edition (J2EE) by Sun, e.g. → RMI (see page 156)

J2EE is a Java-based framework providing a simple, standardized platform for distributed applications. The framework is component-based. It contains a runtime infrastructure and a set of Java API's for implementing distributed applications.

2.4 Distributed application

A distributed application is an application that consists of a *set of cooperating, interacting functional units*. Reasons to distribute these functional units are potential *parallelism* during the execution, *fault tolerance*, and *inherent distribution* of the application domain (e.g., CSCW applications where the users are geographically dispersed, e-commerce including e-banking, or global supply chain management).

2.4.1 Definition

Definition: The term **distributed application** contains three aspects.

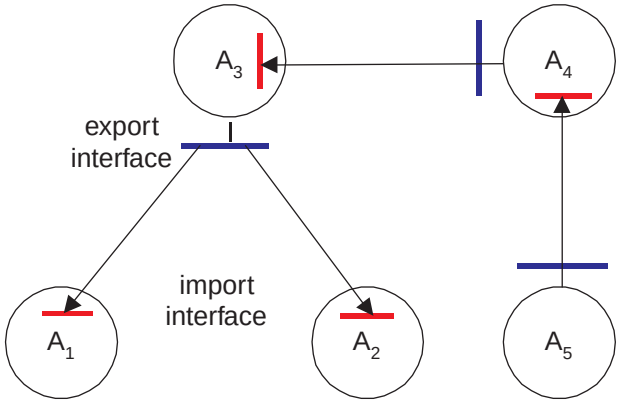
- The application A, whose the functionality is split into a set of cooperating, interacting components A_1, .., A_n, $n \in IN$, $n > 1$; each component has an internal state (data) and operations to manipulate the state.
- The components A_i are autonomous entities which can be assigned to different machines F_i.
- The components A_i exchange information via the network.

2.4.2 Programmer's perspective

Interfaces help to establish well-defined interaction points between the components of a distributed application.

- **Interfaces of a distributed application**

  Consider the distributed application A that consists of the components A_1,..., A_5:



- **Interface specification**

  An interface specifies the operations provided by a component and the communication between components. The specification includes the names and the functionality of the operations:

  required parameters (including their types).

  the results returned by the operation including arity and type.

  visible side-effects caused by the component execution, for example data entry into a database.

  effects of an operation on the results of subsequent operations.

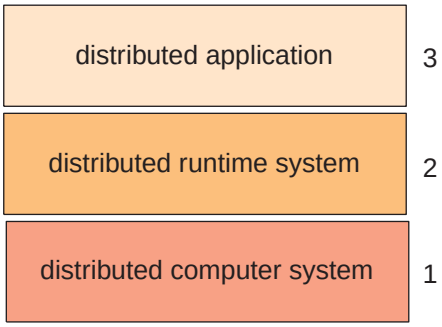
  constraints concerning the sequence of operations.

2.4.3 Distributed application vs. parallel program

Although distributed applications might look similar to parallel programs at first glance, there are still some differences.

	distributed application	**parallel program**
granularity	coarse	fine
data space	private	shared
failure handling	within the communication protocols	not considered

2.4.4 Layers of distributed systems



Level 1: A set of autonomous functional units which are connected with each other; the system consists of computers and links.

Level 2: a distributed virtual machine realized by the distributed computer system and the distributed operating or runtime system.

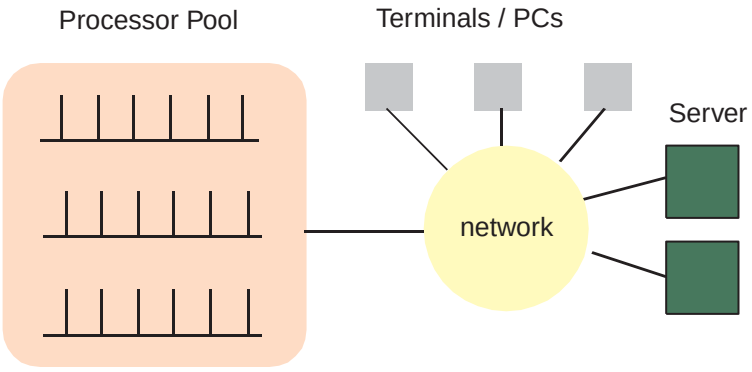
Level 3: set of components of a distributed application; the components are distributed within the distributed computer system.

2.5 Influential distributed systems

In the 1970's at the Xerox Palo Alto Research Center (known as Xerox PARC), researchers designed and constructed with *Alto* a machine that can be seen as the origin of today's personal computers and workstations. The Alto together with the original Ethernet were used as a testbed for distributed applications. The following describes additional important, highly influential distributed systems.

2.5.1 Amoeba

Amoeba has been designed at Uni Amsterdam (Tanenbaum) as an experimental testbed for distributed system studies. Amoeba is characterized through its hybrid architecture consisting of workstations, a processor pool and terminals.



- Definition of a small operating system kernel (microkernel).
- The operating system kernel contains the following major functions:

  creation of processes (organized in *cluster*); a process can contain several control sequences (*threads*).

  support of inter-process communication.
- Amoeba is based on information objects and a set of processes.
- Access of information objects through capabilities.

  (service, object-id, access rights, checksum)
- Client and server subsystems communicate over remote procedure calls (RPC) (see → Client-Server Model (see page 44) and → Remote Procedure Call (RPC) (see page 59) or → Java RMI (see page 156)).

2.5.2 Mach

Mach ist an operating system developed at Carnegie-Mellon University. It is characterized through its small kernel, called microkernel.

Goals of Mach

Mach is especially suitable for multiprocessor applications or applications designed for distributed systems. Major design goals were

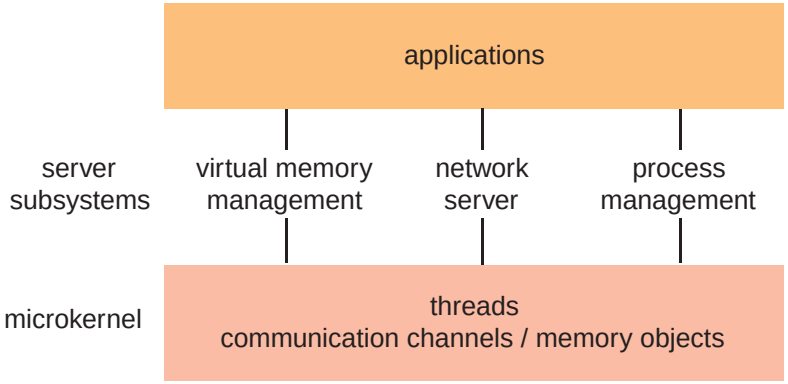
Emulation of Unix.

transparent extension to network operation.

portablility.

Architecture

The process (a task) defines an execution environment that provides secured access to system resources such as virtual memory and communication channels.

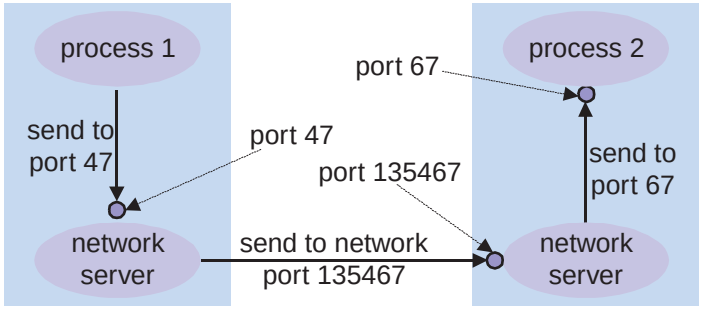


- threads as distribution unit, i.e. only entire threads are assigned to different processors.
- memory objects realize virtual storage units; shared utilization of memory objects by different processes is based on "Copy-on-Write", i.e. the memory object is copied when write operation takes place.

Mach message exchange

Processes communicate through communication channels, called *ports*.

- A port is realized as a message queue to which multiple senders may send messages; there is only one receiving process per queue.
- Ports are protected by capabilities.
- Mach supports network communication through network servers.

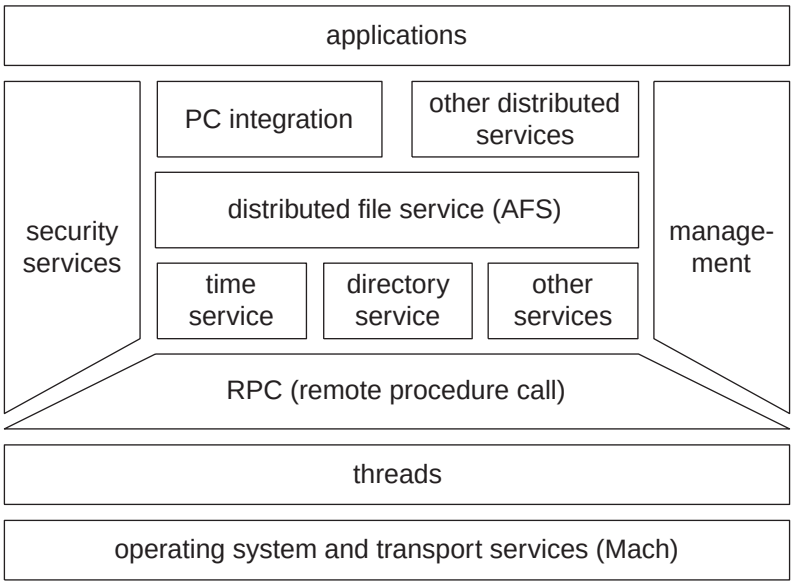


2.5.3 Open Software Foundation (OSF)

The Open Software Foundation (OSF) takes Mach as its basis for a standardized operating system version for Unix. Among others, members of OSF are: HP, IBM and Siemens.

Specification of the Distributed Computing Environment (DCE):

a set of system components that provide a coherent development platform as well as an execution environment for distributed applications.

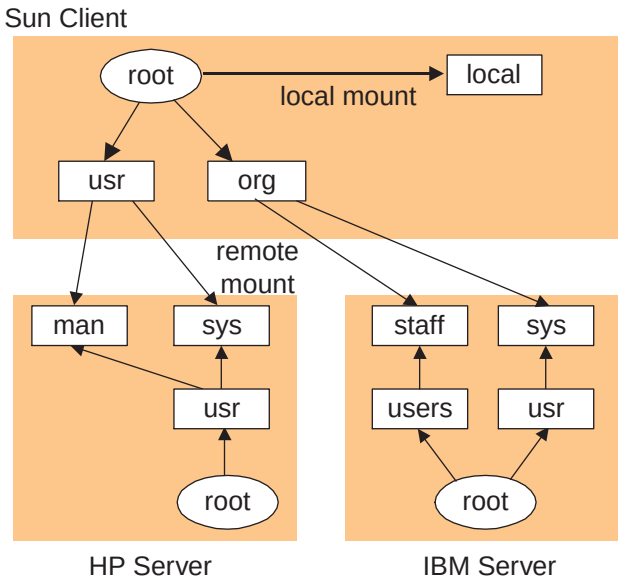


2.5.4 Sun Network File System (NFS)

A well-known networking extension to Unix and other operating systems is obtained through Sun's Network File System (NFS).

Characteristics

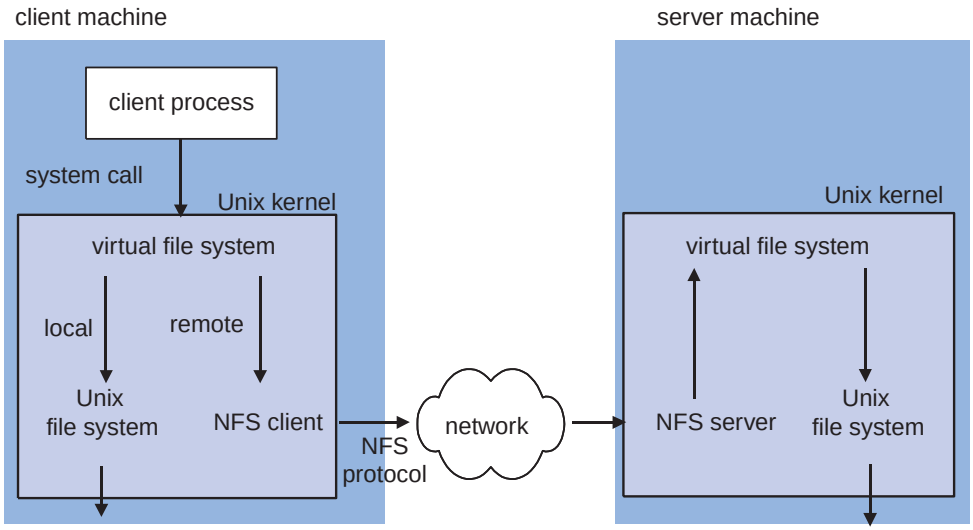
File catalogs are exported (by server subsystems) and mounted (by the client machines).



NFS supports → access transparency (see page 29).

NFS implementation

NFS implementation is based on RPC calls between the involved operating systems. It can be configured to use UDP or TCP.

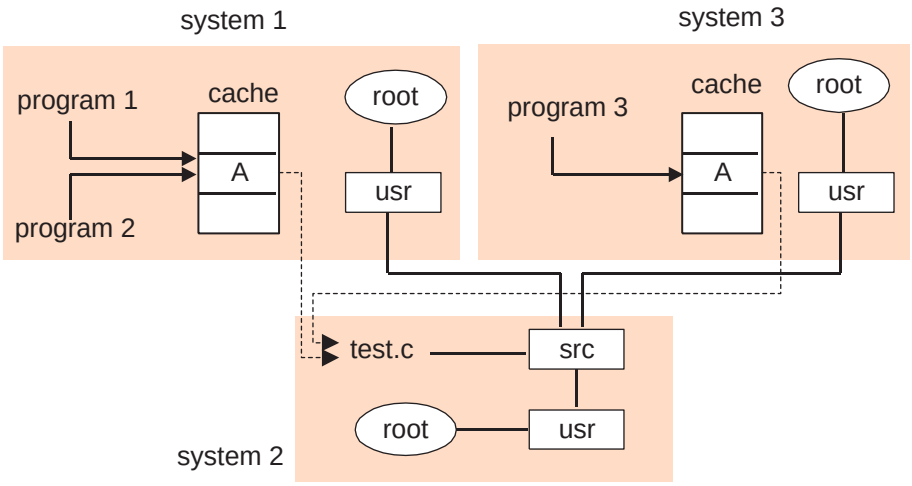


- NFS is a *stateless* file server, i.e. a server subsystem does not store state information about its clients and their past operations.

Caching

Usually there is a *single file cache* per machine, not per program. Write operations cause changes in the file cache. The modified file cache is written into the file system at regular intervals. This may cause problems within a distributed environment if various applications running on different computers *access the same file*.

- **Problem situation**



Approach for solving the problem:

– for each data block d in the file cache, the last modification time $t_{mod/old}(d)$ is stored.

- **Reading a cached data block**

  In the following code fragment we assume that the needed data block is stored in the cache.

```
function read (file d, blockindex k): block;
    /* read data block k of file d */
    if file d is local then return (k-th block);
    else /* file d is remotely located */
    if t_last read of k < (t_now - w)
        /* If the last read was performed more than w seconds
        before, check the validity of the cache entry; t_now be
        the actual time. */
        then
            request latest modification time t_mod/remote(k)
            if t_mod/remote(k) == t_mod/old(k) then return (k-th
            block);
            else
                fetch k from the remote system;
                return (k-th block);
```

2.5.5 Java 2 Platform Enterprise Edition (J2EE)

The J2EE platform is essentially a distributed application server environment. It is a Java environment that provides the following:

a runtime infrastructure for hosting applications.
a set of Java extension APIs to build applications.

Objectives of J2EE

The idea of J2EE is to provide a standardized programming model for the realization of distributed applications at the organisational level.

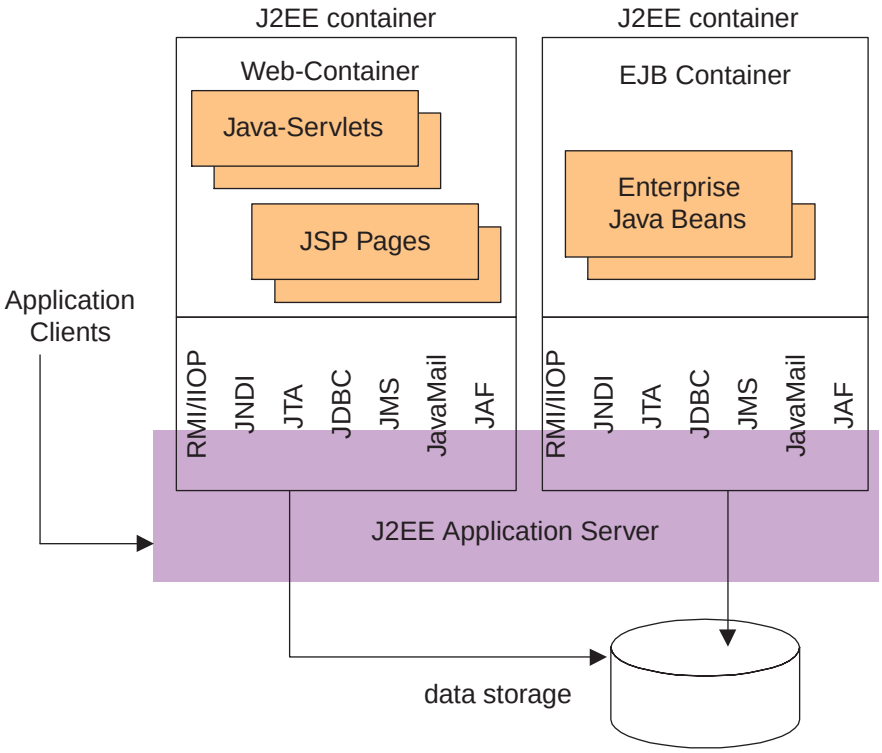
- Java-based, but with interfaces to legacy applications, for example through Corba.
- component-based.
- J2EE consists of 2 components

  a *runtime infrastructure* for applications.

  a set of *Java extension APIs* to build applications. Examples are Enterprise Java Beans (EJB), Java Servlets, JavaServer Pages (JSP), RMI via Internet-Inter-ORB Protokoll (RMI-IIOP), Java Naming and Directory Interface (JNDI), Java Transaction API and Java Mail.

J2EE architecture

A J2EE platform consists of the J2EE application server, one or several J2EE containers, and the data storage.



J2EE container

A typical J2EE platform has one or several containers. A J2EE container has two principal tasks:

runtime environment for managing application components.

to provide access to J2EE APIs:

- available APIs of the J2EE platform

  RMI/IIOP: Remote Method Invocation (via IIOP)

  JNDI: Java Naming and Directory Interface

  JTA: Java Transaction API

  JDBC: Java Database Connectivity Extension

  JMS: Java Message Service

  Java Mail

  JAF: Java Beans Activation Framework.

- Examples for application components: JavaServlets, JavaServer Pages, Enterprise JavaBeans.
- J2EE currently supports the following containers

  Web container: Java Servlets, JSP pages

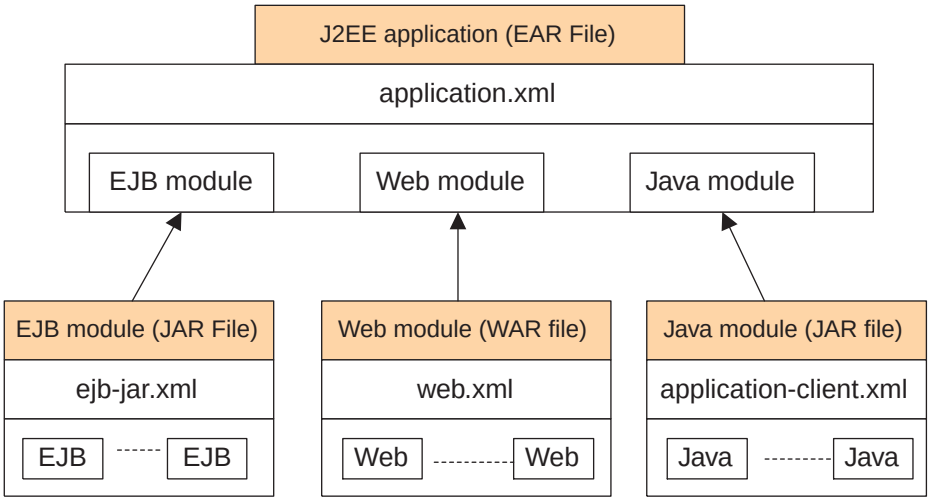
  EJB container : Enterprise Java Bean components

  Applet container : Java applets

  Application container : Standard Java applications

J2EE application

A J2EE application consists of several modules, each of which again contains several application components.



Chapter 3

Architecture of distributed systems

3.1 Issues

This section focuses on the following issues

- Discussion of basic aspects of distributed systems.
- Transparency as a key concept of distributed systems.
- How do distributed components cooperate? Thus, we discuss models of cooperation among components of distributed applications.
- What is the client-server model?

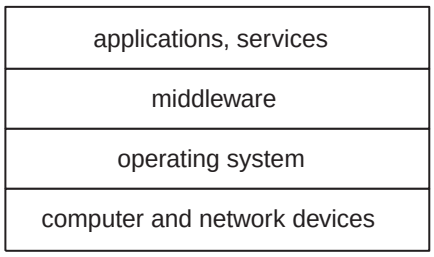
3.2 System Models

A distributed system can be described in form of descriptive models.

3.2.1 Architectural model

defines the interaction between components and the mapping onto the underlying network.

Software layers



- **Middleware**

  is defined as a layer of software whose purpose is to mask heterogeneity and to provide a convenient programming model to application programmers.

  Examples are Corba, Java RMI, DCOM (Microsoft's Distributed Component Object Model).

  Middleware services are: communication facilities, naming of remote entities (objects), persistence (distributed file system), distributed transactions, facilities for security.

System architectures

deals with the placement of components across a network of computers and the functional roles they assume during interaction.

client-server model.

proxy servers.

peer processes.

community of software agents.

3.2.2 Interaction model

The interaction model deals with performance and with the difficulty of setting time limits, e.g. for message delivery.

it is impossible to maintain a single global time $\Rightarrow$ logical clock.

messages do not arrive in the same order at all locations.

consistent ordering of events.

3.2.3 Failure model

The failure model defines the ways in which failures may occur and how they are handled.

distinction between failures of processes and communication channels.

different types of failures:

message loss.

arbitrary failures (Byzantine): a process arbitrarily omits intended processing steps or takes unintended processing steps.

timing failures: a local clock exceeds the bounds on its rate of drift from real time or transmission takes longer than the specified bound.

3.2.4 Security model

The security model defines the possible threats to processes and communication and the ways how they are handled.

secure communication channels, e.g. use of cryptography.

protecting objects against unauthorized access.

authentication of messages; proving the identities of their senders.

The following sections of the course will discuss in more detail various aspects of these system models.

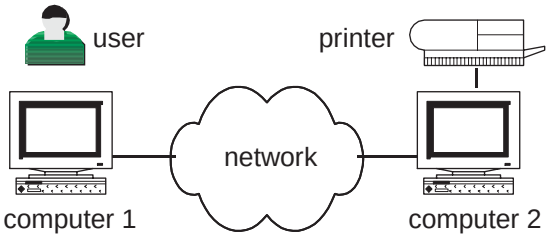
3.3 Transparency

For a better exploitation of resources within a distributed, heterogeneous network a high degree of transparency is necessary. Without a doubt, transparency is seen as one of the key enablers for and one of the key concepts of a successful distributed system implementation.

3.3.1 Location transpareny

Problem: location of an object (resource or service) in a distributed system.

$\Rightarrow$ Location transparency implies that the user need not necessarily know the physical location of the object within the network; the access is realized through a location-independent name of the object.



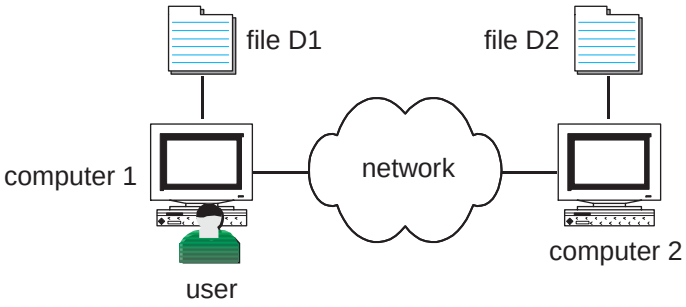
- Important aspect of location transparency:

  Object name contains no information about the current object location.

3.3.2 Access transparency

Problem: How to access objects in a distributed system.

⇒ Access transparency provides access to local and remote objects in exactly the same way.

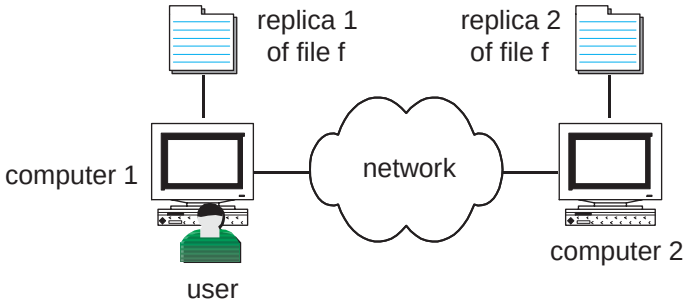


3.3.3 Replication transparency

For reasons of availability or fast access, resources, e.g. objects may be replicated.

Problem: Management of several copies of an object in a distributed system.

⇒ Replication transparency means that the user is unaware of whether an object is replicated or not. The user accesses replicated objects as if they exist only once.



A variety of protocols have been proposed that deal with the problem of consistency among replicated files (→ Update of replicated files) (see page 135).

3.3.4 Failure transparency

Problem: Partial failure in distributed systems, for example computer crashes or network failures.

⇒ Up to a certain degree, failures are masked by the system.

3.3.5 Concurrency transparency

The problem of synchronization of parallel and concurrent accesses to shared resources increases when the resources and the users accessing these resources are geographically dispersed.

Problem: shared access to objects in distributed systems.

⇒ Several users or application programs can access objects simultaneously (for example shared data) without mutual influence.

3.3.6 Migration transparency

Problem: Object relocation in distributed systems.

⇒ Migration transparency provides a solution to the problem of relocation of objects in distributed systems. Objects may migrate from one computer to another without influencing the correct behavior of running applications.

Host migration transparency

Problem: Computer migrates from one subnetwork to another subnetwork, for example if a user connects his laptop computer to different subnetworks. This requires a dynamic generation of the IP address (for example DHCP), a name server, etc.

> ⇒ The computer supports the same environment, the same applications, and the same look-and-feel, no matter where the mobile workers are currently connected to the network.

Types of migration

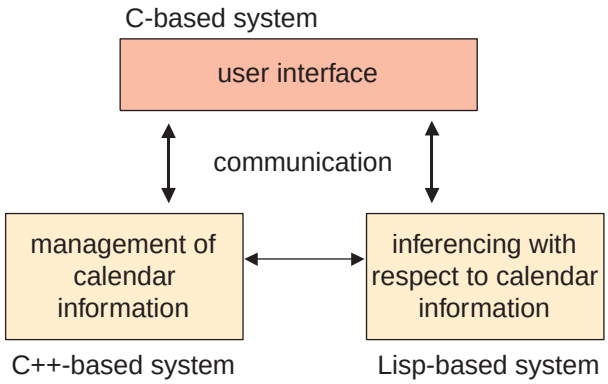
- "off-line" migration.
- "on-line" migration.

3.3.7 Language transparency

Problem: An application's components are realized by different programming languages.

> ⇒ Interactions between individual components is independent from the programming language used for implementing the respective components.

Example: calendar system



3.3.8 Other transparencies

There are a number of other transparencies relevant for distributed systems.

- **Execution transparency**

  Execution transparency implies that processes may be processed on different runtime systems.

- **Performance transparency**

  allows for dynamic reconfiguration of the system to improve the overall system performance when changes in load characteristics are detected.

- **Scalability transparency**

  supports extensions and enhancements of the system or the applications without the need of modifications to the system structure or changes to the application algorithms.

3.3.9 Goal for distributed applications

A major goal of most distributed systems, especially of distributed file or operating systems, is the realization of a rich set of transparency levels. For some applications, however, a high level of transparency can cause problems.

Problem: Computer-supported Cooperative Work

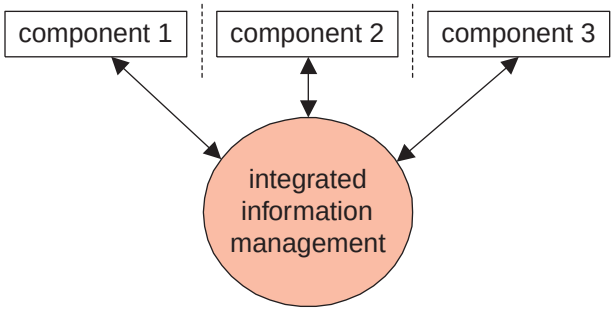
- due to concurrency transparency, the team members are not always aware of their simultaneous activities (there is no "group awareness").
- Hence, selective transparency has been applied for Computer-supported Cooperative Work (CSCW); it supports location and access transparency, but no strict concurrency transparency.

3.4 Models for cooperation

In this chapter, we will discuss mechanisms for cooperation on different levels of abstraction. Communication may take place between entire applications or between components of a distributed application: Information Sharing, message exchange, producer-consumer model (pipe mechanism) and client-server model. The object-oriented cooperation will be a separate section.

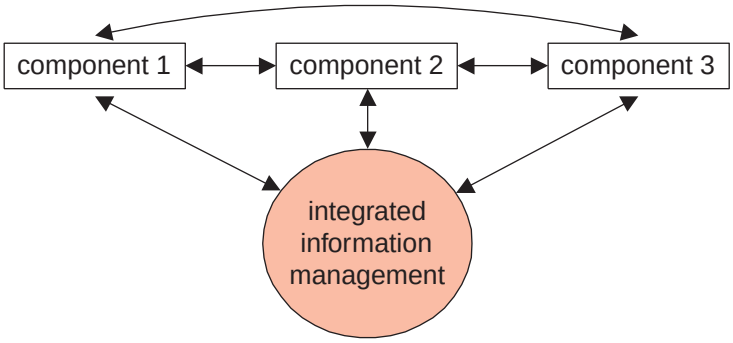
3.4.1 Information Sharing

Components of a distributed application communicate through shared, integrated information management. An example is the BSCW Workspace (URL: http://bscw.gmd.de/).



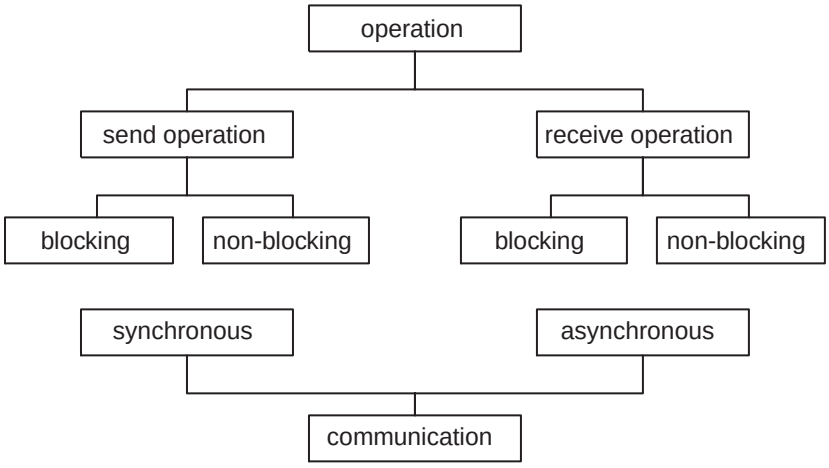
- No direct communication between components.
- **Information sharing plus direct communication**

  Since the purist approach of information sharing might be too restrictive for some of the CSCW applications, many designers provide additional direct communication among the components of the distributed application.



3.4.2 Message exchange

Interprocess communication (IPC) as found in centralized systems can be achieved in distributed systems through message exchange between a sender and a receiver.



Background

Message exchange takes place between a sending and a receiving process.

- **Message structure**

  Message head

  Message body

- **Basic functionality**

  ```
  send(E: receiver, N: message);
  receive(S: sender, B: buffer);
  ```

- **Communication perspectives**

  We can distinguish between different perspectives with respect to the communication among the involved processes:

  the sender's view, and

  the receiver's view

  Assumption: Sender S has invoked the operation `send(E, N)`; receiver E performs the operation `receive(S, B)`.

Sender's view

In the sender's view, we will distinguish between asynchronous and synchronous message exchange, as well as the so-called remote-invocation send.

- **Asynchronous message exchange**

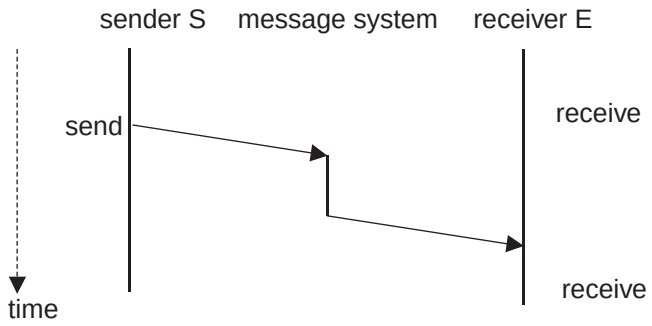
  Sender S can resume its processing immediately after the message N is put forward into the message queue NP (NP is also called message buffer).

  S will *not* wait until the receiver E has received the message N.

  A `receive` operation indicates that the receiver is interested in receiving a message.

  – **Example**

  The receiver E repeats the invocation of the `receive` operation until a message arrives. If the message N is available, it is transferred; otherwise E continues with its normal processing.



  – **Advantages of asynchronous message exchange**

  Advantages

  useful for real-time applications, especially if the sending process should not be blocked.

  supports parallel execution threads at the sender's and the receiver's sites.

  it can be used for event signaling purposes.

  Disadvantages

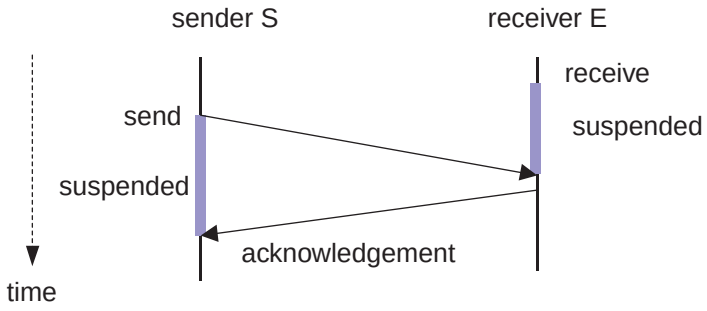
  management of message buffers, handling of buffer overflow, access control problems, and of process crashes (receiver).

  notification of S in case of failures may be a problem, since mostly S has already continued with its regular processing.

design of a correct system is difficult. The failure behavior depends heavily on buffer sizes, buffer contents, and the time behavior of the exchanged messages.

- **Synchronous message exchange**

  Sender S is blocked until recipient E has effectively received message N; similarly, receiver E is blocked until N is transferred, i.e. until N is stored into the receiver's buffer.



  – **Decoupling of sender and receiver**

    In order to avoid endless waiting times, we need some sort of decoupling of sender S and receiver E:

    associate a timeout with every `send` operation, i.e. the sender terminates waiting time after a certain time span has elapsed.

    creation of subprocesses for sending the message, e.g. using "threads".

- **Remote-invocation send**

  Sender S suspends execution until the receiver has received and processed a submitted request that was delivered as part of the message.

Receiver's view

The reception of a message can be classified into three cases, namely the conditional reception, the reception with timeout, and the selective reception.

- **Conditional reception**

```
function receive (S: sender, B: buffer):  errorcode
    if ∃ message N by sender S
    then copy N to buffer B; return (success)
    else return (failure)
```

- **Reception with Timeout**

  In the case of the reception with timeout, the receiver is blocked either until a message is received from a sender or until a timeout occurs.

```
function receive (S:sender, B:buffer, t:timeout):  errorcode
    wait until (∃ message N of sender S or timeout t) do
        if ∃ message N of sender S
        then copy N into buffer B; return (success)
        else return (failure)
```

- **Selective reception**

  In the case of selective reception, the receiver specifies a set of sender names (e.g., using explicit names or wildcards).

  Message selection in case of multiple different message sources:

1. *arbitrary*: random, unspecified selection method.
2. *predefined*: selection according to a predefined strategy (for example according to the sequence of message arrivals).
3. *user-customizable*: selection according to a user-defined criteria (for example according to message priorities).

3.4.3 Naming entities

Names are used to uniquely identify entities and refer to locations. An important issue is name resolution.

- **Names**

  A *name* is a string of characters that is used to refer to an entity (e.g. host, printer, file).

  entities have access points to invoke operations on them $\Rightarrow$ *address* is the name of the access point.

  an identifier is a name which uniquely identifies an entity.

- **Name space**

  Names in distributed systems are organized into a name space.

  Name spaces are organized hierarchically.

  Representation as a labeled directed graph.

  Path along graph edges specifies the entity name; absolute vs relative path names.

- *Name resolution*: a name lookup returns the identifier or the address of an entity, e.g. → LDAP (see page 52) Name Service.

- **Message addressing**

  The sender addresses the receiving process by specifying the appropriate parameter in the `send` operation:

  a) (machine number, process number) or (machine number, socket): not location transparent.

  b) Recipient chooses an arbitrary address out of a large, global address space (e.g. 16-bit number)
  sender localizes receiver by broadcast mechanism; this overhead is only feasible for LAN environments.

  c) Receiver is known under location-independent name (string): use of name servers to map names to addresses.

  – **Buffered and unbuffered sending**

    Problem: the address of the `send` operation refers to the receiving process. Where does the operating system on the receiving side file the message, especially if `send` takes place before `receive`?

    * **Approach 1: unbuffered**
      The operating system saves the message in a system-wide buffer for a certain time span, expecting a suitable `receive` operation of a local process.
    * **Approach 2: buffered**
      The receiving process requests from the operating system to create a mailbox under a certain address. Incoming messages for the receiver are stored in the mailbox.

3.4.4 Bidirectional communication

Usage of the request-answer scheme for message exchange. Communication between sender and receiver is influenced by the following situations

loss of request messages.

loss of answer messages.

sender crashes and is restarted.

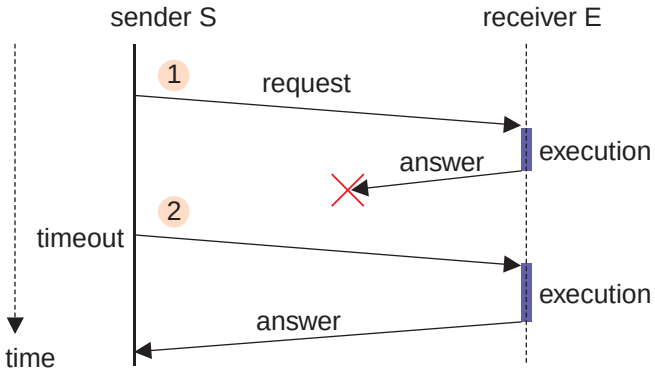
receiver crashes and is restarted.

Different call semantics

Any communication between a sender and a receiver is subject to communication failures. Therefore, we distinguish between different call semantics.

- **at-least-once semantics**

  Under an at-least-once semantics, the requested service operation is processed once or several times.

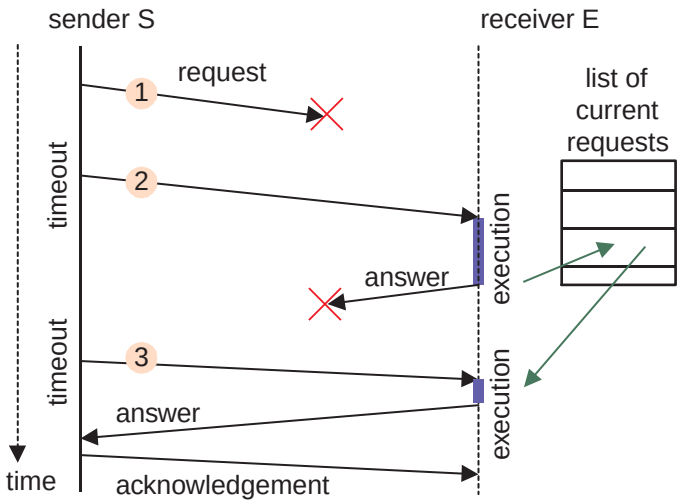


  Animation at least once

  siehe Online Version

- **exactly-once semantics**

  Under an exactly-once semantics, the requested service operation is processed exactly once. That implies, that repeatedly sent requests (due to timeouts at the sending site) have to be detected, and handled, at the receiving site.



Animation exactly once

siehe Online Version

- **last semantics**

  Under a last semantics, the requested service operation is processed once or several times, however, only the last processing produces a result and, potentially, some side-effects.

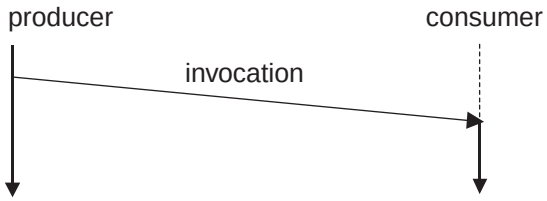
- **at-most-once semantics**

  Under an at-most-once semantics, the requested service operation is processed once or not at all. If the service operation is processed successfully, the at-most-once semantics coincides with the exactly-once semantics.

  Example for providing at-most-once semantics
  After timeout at the sending site the request is not retransmitted.
  The request is transmitted in the context of a transaction.

3.4.5 Producer-consumer interaction

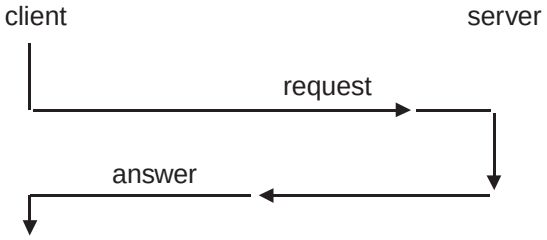
In this interaction type (also called fire & forget interaction) after an invocation of the consumer, the producer resumes its execution immediately (and is not suspended).



- Special case: Pipe mechanism (similar to unix pipes); after information has been provided to the consumer, the producer terminates the execution.

3.4.6 Client-server model

The → client-server model (see page 44) implements a sort of handshaking principle, i.e., a client invokes a server operation, suspends operation (in most of the implementations), and resumes work once the server has fulfilled the requested service.



Examples for servers

- In a distributed environment, a server manages access to shared resources (e.g. a file server).

  Problems:

  server crash ⇒ resource is no longer available in the network.

  server becomes a bottleneck for accessing the resource.

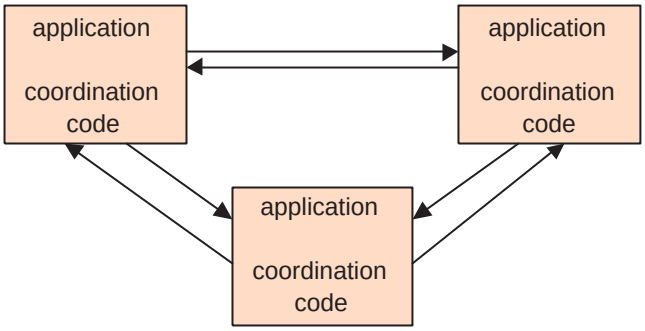
- Internet Explorer, Netscape/Opera browser are examples for clients and httpd-daemon (e.g. Apache Web-Server) is an example for a server.

3.4.7 Peer-to-peer model

All processes play a similar role

interacting cooperatively as peers to perform a distributed computation.

there is no distinction between clients and servers.



3.4.8 Group model

Combining of a set of components (e.g. processes) into a group.

Example: For reasons of fault tolerance, a service is provided by a group of servers .

Important aspects

- Processing of a shared global problem in a shared environment.
- Need to exchange information.
- Group awareness.
- Coordination of communication and actions within the group.

3.4.9 Taxonomy of communication

As the cooperation models show, communication between components of distributed applications is an essential part; it is mostly based on message exchange. Communication can be categorized according different dimensions.

Dimensions for categorization

1. Synchronization of sender and receiver.

   synchronous.

   asynchronous.

2. Connection type.

   individual (1:1) and (1:m) connections.

   collective (n:1) and (n:m) connections.

3. Internal group structure.

   no *pre-defined* structure.

   uniform group: all communicating components are at the same level; there is no hierarchy.

   hierarchical structure: communicating components are, for instance, organized as a tree.

   open vs. closed group.

4. reliability.

   unreliable communication.

   reliable communication.

Message serialization

On the receiving side, messages are delivered according to a certain order and then processed. If messages are sent to several recipients of a group, then the messages may be received in different order, due to different transmission times.

Processing order of messages

- **One sender**

  There are the following ordering schemes:

  according to the message arrival on the recipient's side; different receivers can have different message arrival sequences.

  according to message sequence number generated by the sender; this approach is sender-dominated.

  receiver creates a serialization according its own criteria.

- **Several senders**

  If several senders are involved, the following message ordering schemes may be applied:

1. no serialization.
2. loosely-synchronous.

   There is a loosely synchronized global time which provides a consistent time ordering.
3. virtually-synchronous.

   The message order is determined by $\rightarrow$ causal interdependencies (see page 122) among the messages. For example, a message N has been sent after another message M has been received, i.e. N is potentially dependent on M.
4. totally ordered.

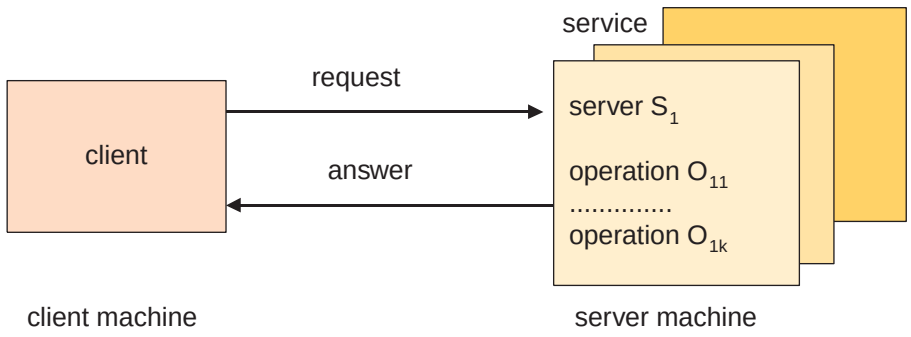
   by token: before a sender can send a message, it must request the send token.

   a selected component (the coordinator) determines the order of message delivery for all recipients.

3.5 Client-server model

The client-server model implements a sort of *handshaking principle*, i.e., a client invokes a server operation, suspends operation (in most of the implementations), and resumes work once the server has fulfilled the requested service.

3.5.1 Terms and definitions



Definitions

It should be clear to the reader by now that we have used so far technical terms at different levels of abstraction.

sender, receiver: pure message exchanging entities.

client, server: entities acting in some specialized protocol.

- **Client**

  Definition: A **client** is a process (some say, an application) that runs on a client machine and that typically initiates requests for service operations.

  Potential clients are a priori unknown.

- **Service**

  Definition: A **service** is a piece of software that provides a well-defined set of services. This piece of software may run on one or multiple (server) machines.

- **Server**

  Definition: A **server** is a subsystem that provides a particular service to a set of a priori unknown clients. A server executes a (piece of) service software on a particular server machine. Obviously, a single server machine can host multiple server subsystems.

  A server provides a set of operations (procedures).

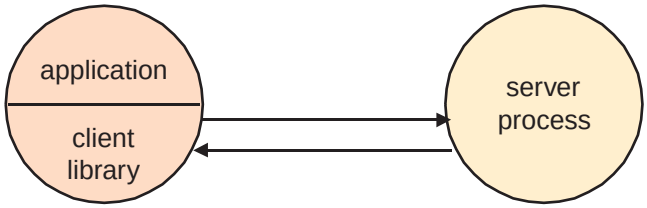
Server loop

The server subsystems can be realized as dedicated processes. Essentially, these permanently running dedicated processes execute the following loop, also called the server loop.

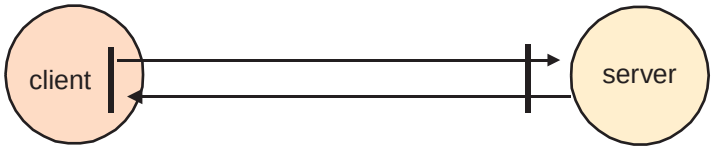
```
while true do
    wait until a service operation is requested by a client
    C;
    execute requested service operation;
    send answer (result) to client C;
```

Embedding of the client code

Typically, a client is embedded into an application. The client is mostly realized via library routines.



Client-server interfaces



1. Client interface (import interface)

   It represents the server within the client;

   It prepares parameters and sends the request messages to the server;

   It prepares the interpretation of the result that is extracted from the answer message submitted by the server.

2. Server interface (export interface)

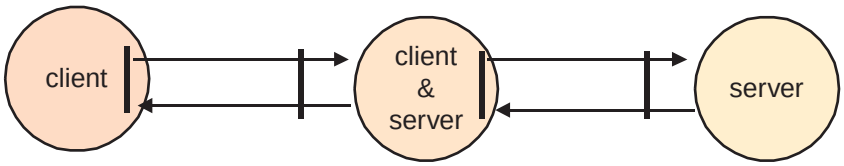
   It represents all potential clients within the server;

   It accepts client requests; interprets the parameters; prepares results;

   It invokes the respective service operation;

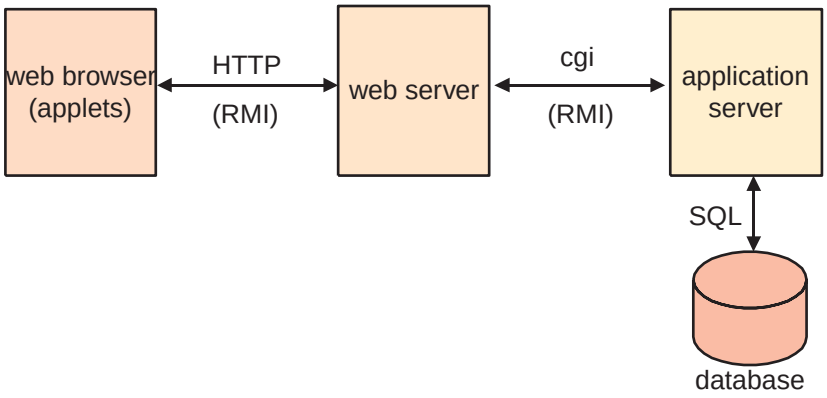
   It prepares and sends the answer message containing the result of the service operation.

Multitier architectures

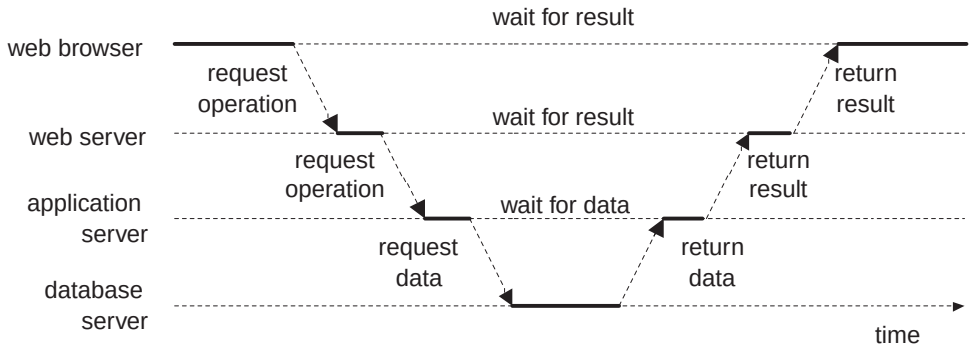


- **Example for a multitier architecture**

  Multitier architectures have become very popular for web environments, with the web server taking the role of the interface between client (browser) and application ⇒ web server is server and client.



  – **Timing process**



Animation Client-Server

siehe Online Version

3.5.2 Concepts for client-server applications

Client					
presentation execution	presentation	presentation	presentation execution	presentation execution (with local database)	presentation execution database
Server					
database	presentation execution database	execution database	execution database	execution (with local database)	database
Case 1	Case 2	Case 2	Case 3	Case 3	Case 4

Different cases

- Case 1: remote data storage. access, for example, via Sun NFS.
- Case 2: remote presentation (for example X window system).
- Case 3: distributed application

  cooperative processing among the individual components of an application.
- Case 4: distributed data storage

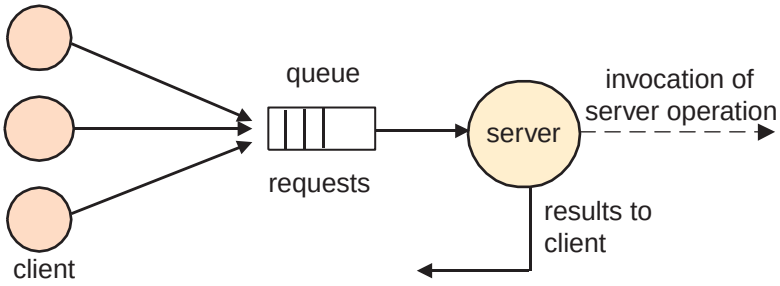
  The information is distributed between client and server; information replication is possible.

3.5.3 Processing of service requests

Since clients and servers have different life spans, and since requests for service operations are not equally distributed among the clients, servers manage these requests in a queue.

Single dedicated server process

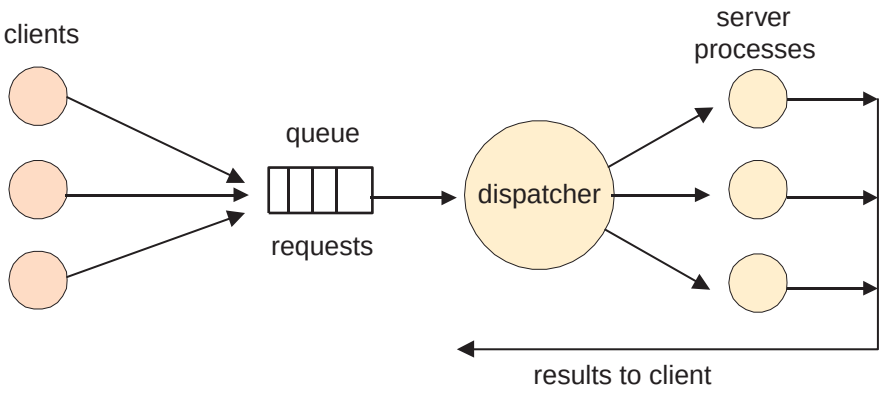
A single dedicated server process is in charge of processing requests for service operations.



- no parallel processing of requests, which results in the following disadvantage:
  approach may be time consuming.
- no interruption of the processing of the current request when a higher prioritized request appears in the queue.
- server becomes bottleneck.

Cloning of new server processes

Every incoming request is handled by a new server process.



- Cloning of new server processes is expensive;
- *Synchronization* of access to shared persistent data;
- *Parallel processing* of several applications is possible;

Parallel request processing through threads

This is a variant of the second approach.

Shared address space, i.e. the approach allows shared utilization of variables;

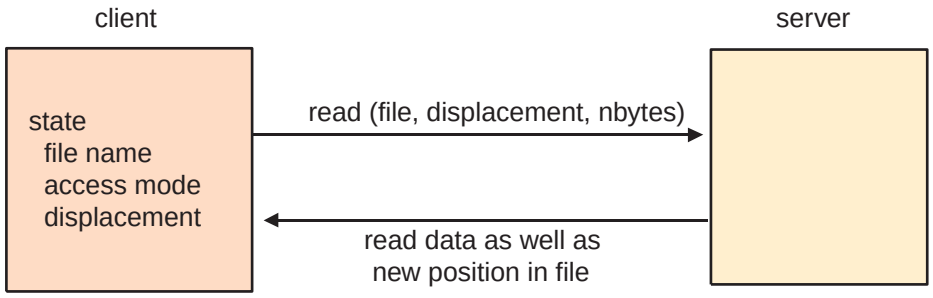
3.5.4 File service

Definition: A **file service** [Svobodova 1984] provides (remote) centralized data storage facilities to clients distributed among a network.

Distinction between a stateless and state-dependent server subsystem.

- **Stateless server**

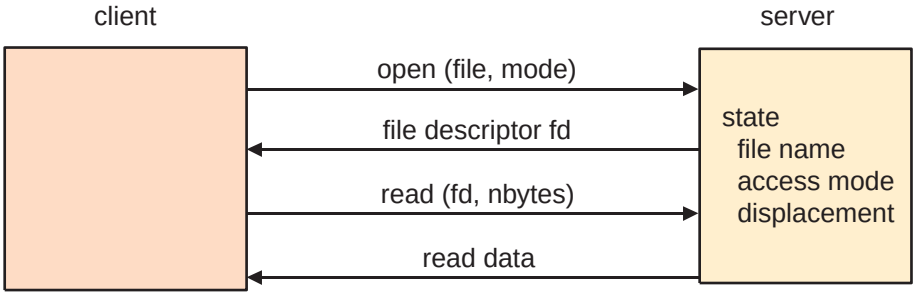
  Stateless server do not manage any state information about their clients; the client must supply all necessary parameters to process the request.



  A crashed server can be restarted without dealing with state reinstallments.

- **State-dependent server**

  State-dependent server subsystems manage state information about their clients.



As a consequence, programming at the client site becomes less complex.

3.5.5 Time service

Definition: A **time service** provides a synchronized system-wide time for all nodes in the network.

3.5.6 Name service

Definition: A **name service**, sometimes called a directory service, provides (remote) centralized name management facilities to clients distributed among a network; names refer to objects; examples are files, other servers, services, personal computers, printers, as well as users.

- Name servers manage a list of names. Such a directory entry might be stored in a data structure

name	/* Name of the object as parameterized in a client request.*/
address	/* Address of the object within the network, e.g., host number concatenated with communication port number. */
access information	/* This access information may limit access to the object for particular clients. */
attributes	/* Additional attributes of the object. */

- **Example for a name service**

  Domain Name System (DNS):

  hierarchical domain-based naming scheme for the Internet.

  distributed database for implementing this naming scheme.

  mapping of host names and email destinations (e.g. www11.in.tum) to their respective IP addresses.

  – top-level organizational domains:

    edu: universities and other educational institutions
    com: commercial organizations
    de: organization in Germany

  – DNS database is distributed across a logical network of name servers.

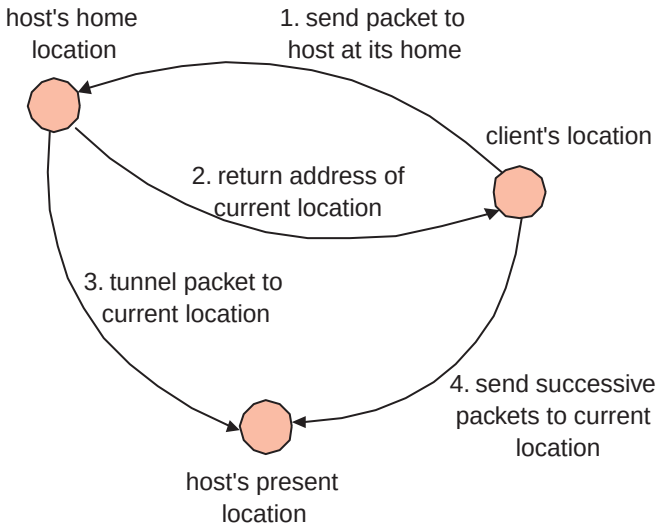
    Each server stores primarily data for the local domain.

Animation Domain Name Service

siehe Online Version

3.5.7 Locating mobile entities

Naming for mobile entities, e.g. laptops must take the changing location of the entity into consideration.



3.5.8 LDAP - Lightweight Directory Access Protocol

In distributed environments, information concerning persons, applications, files, printers, and other resources available via network is often stored in a special database, the so-called directory LDAP is a protocol supporting the access to and update of directory information. It is an open industry standard.

Basics

Definition: A **directory** is a list of objects arranged in some order and with descriptive information (meta-data).

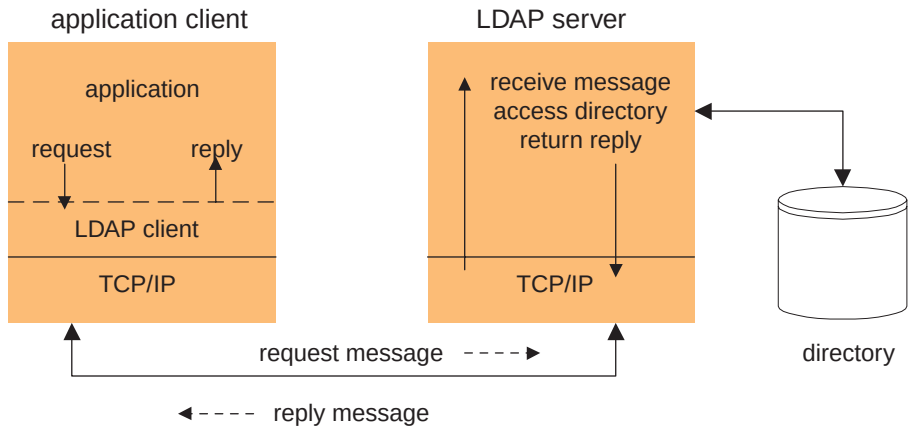
A *directory service* is a name service containing object names and meta-data.

- Queries in directories: based on names *and* meta-data.

- White Pages: object access according to object name.
- Yellow Pages: object access according to object meta-data.
- LDAP is a communication protocol supporting access to/update of directory information.
  - it has been developed as simple alternative to X.500 standard.
  - it is based on TCP/IP rather than the ISO/OSI protocol stack.
  - modern web browsers (for example netscape) support LDAP.

LDAP architecture

The LDAP architecture is based on the client-server model and the TCP/IP protocols.



- LDAP uses strings for data representation.
- General interaction process

1. Client initiates a session with the LDAP server (binding).

   Client specifies a name or an IP address and port (e.g. port 389) of the LDAP server.

   Client specifies user name and password.
2. Client invokes LDAP operations (read, write, seek).
3. Client terminates session (unbinding).

Information model

A directory entry describes an object, for example person, printer, server, organizations etc.

each entry has a *distinguished name* (DN).

each entry has a set of *attributes* with a type and one or several values.

- **Attribute syntax**

Syntax	*Description*
bin	binary information
ces	case exact string, also known as a "directory string", case is significant during comparisons.
cis	case ignore string. Case is not significant during comparisons.
tel	telephone number, numbers are interpreted as text (without blancs or colons)
dn	distinguished name.
generalized time	year, month, day, time represented as printable string
postal address	postal address with lines separated by "$" characters.

- **Examples**

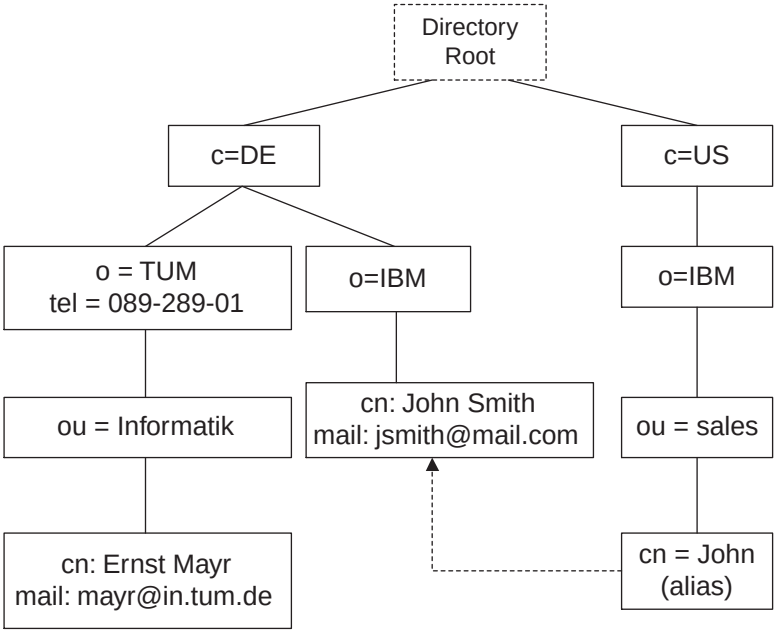
Attribute, Alias	*Syntax*	*Description*	*Example*
commonName, cn	cis	name of entry	John Smith
surname, sn	cis	surname of a person	Smith
telephoneNumber	tel	telephone number	089-289 25700
organizationalUnitName, ou	cis	name of organization	Informatik
organization, o	cis	name of organization	TUM
owner	dn	distinguished name of entry owner	cn=John Smith, o=TUM, c=DE
jpegPhoto	bin	JPEG Photo	photo of John Smith

  - Based on these attributes, schemas for entries can be defined. Examples of schemes:
    - InetOrgPerson: entry for one person
      attributes: commonName (cn), surname (sn)
    - organization: entry for an organization
      attribute: organization (o)

Naming model

The LDAP naming model defines how entries are identified and organized. Any distinguished name (DN) of an object consists of a sequence of parts, so-called relative distinguished names (RDN).

- The entries in an LDAP directory are hierarchically structured as tree (Directory Information Tree, DIT). Entries are arranged within the DIT based on their distinguished name.
- Example of DN: cn=John Smith, o=IBM, c=DE.
- DIT also supports aliases.
- DIT can be distributed across several servers. Reference to entries of other LDAP servers via URLs.



Functional model

The functional model defines operations for accessing and modifying directory entries. Among others LDAP supports the following directory operations:

create a LDAP entry

delete a LDAP entry

update a LDAP entry, e.g. modification of the distinguished name (= move in DIT)

compare LDAP entries

search for LDAP entries which meet certain criteria

- **Search**

  The search operation allows a client to request that an LDAP server search through some portion of the DIT for information meeting user-specified criteria in order to read and list the result(s).

  – Examples

    find the postal address for cn=John Smith,o=IBM,c=DE.

    find all entries which are children of ou=Informatik,o=TUM,c=DE.

  – Search constraints.

    * base object: defines the starting point of the search. The base object is a node within the DIT.
    * scope: specifies how deep within the DIT to search from the base object, e.g.

      baseObject: only the base object is examined.
      singleLevel: only the immediate children of the base object are examined; the base object itself is not examined.
      wholeSubtree: the base object and all of its descendants are examined.

  – **Code example**

```
#define SEARCHBASE "o=TUM,c=DE"
LDAP *ld;
char *User = NULL;
char *Passwd = NULL;
char searchfilter[] = "cn=Mayr";
/* open a connection */
if ((ld = ldap_open("ldapserver.in.tum.de", LDAP_PORT)) ==
NULL) exit(1);
/* authenticate as nobody */
if (ldap_simple_bind_s(ld, User, Passwd) != LDAP_SUCCESS) {
    ldap_perror(ld, "ldap_simple_bind_s");
    exit(1);
}
/* search the database */
if (ldap_search_s(ld, SEARCHBASE, LDAP_SCOPE_SUBTREE,
searchfilter, NULL, 0) != LDAP_SUCCESS) {
    ldap_perror(ld, "ldap_search_s");
    exit(1);
}
.....
/* close and free connection resources */
ldap_unbind(ld);
```

ldif - exchange format

ldif = LDAP Data Interchange Format; it is used to import and export directory information.

```
dn:  cn=Informatik
cn:  Informatik
objectclass:  top
objectclass:  groupOfNames
member:  cn=Baumgarten,
Uwe,mail=baumgaru@informatik.tu-muenchen.de
member:  cn=Schlichter,
Johann,mail=schlicht@informatik.tu-muenchen.de
....
dn:  cn=Baumgarten,
Uwe,mail=baumgaru@informatik.tu-muenchen.de
cn:  Baumgarten, Uwe
modifytimestamp:  20001213084405Z
mail:  baumgaru@informatik.tu-muenchen.de
givenname:  Uwe
sn:  Baumgarten
objectclass:  top
objectclass:  person
....
dn:  cn=Schlichter,
Johann,mail=schlicht@informatik.tu-muenchen.de
cn:  Schlichter, Johann
modifytimestamp:  20001213084406Z
mail:  schlicht@informatik.tu-muenchen.de
givenname:  Johann
sn:  Schlichter
telephonenumber:  +49-89 289-25700
o:  Technische Universität München
streetaddress:  Arcisstr 21
st:  Germany
objectclass:  top
objectclass:  person
```

Chapter 4

Remote Procedure Call (RPC)

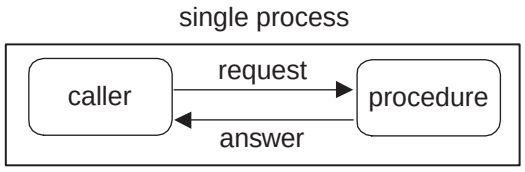
4.1 Issues

After defining the term RPC, we will proceed to introduce various aspects of it , as well as its usage in client-server systems.

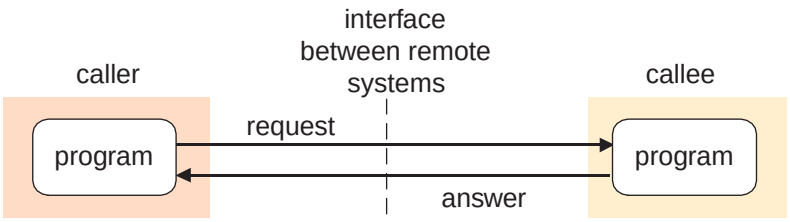
- What is a remote procedure call?
- Which problems arise by its usage?
- What are client- and server stubs?
- What is an RPC generator?
- How do client and server systems find each other?
- What are the characteristics of asynchronous RPC's ?

4.2 Introduction

4.2.1 Local vs. remote procedure call



RPC is an extension of the same type of communication to programs running on different computers; there is still a single thread of execution and the transfer of data between the involved components.



4.2.2 Definition

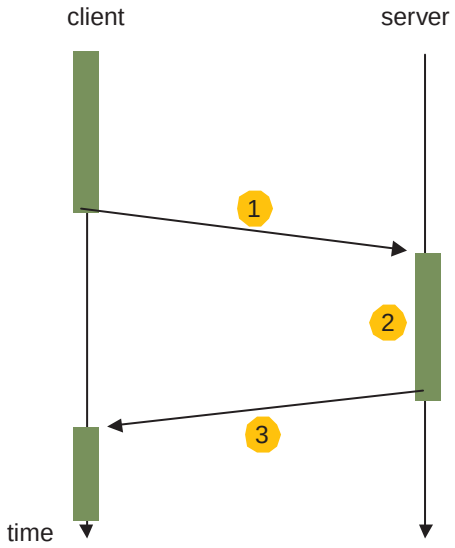
Definition: Birrell and Nelson (1982) define an **RPC** as a synchronous flow of control and data passing scheme achieved through procedure calls between processes running in separate address spaces where the needed communication is via small channels (with respect to bandwidth and duration time).

- *synchronous*: The calling process (client) is blocked until it receives the answer of the called procedure (server); the answer contains the results of the processed request.
- *procedure calls*: the format of an RPC call is defined by the signature of the called procedure.
- *different address spaces*: it cannot be assumed that client and server have network-wide unique memory addresses, i.e. it is necessary to handle pointers during parameter passing different from local procedure calls.
- *small channel*: RPC communication takes place via a special interaction path (for example local network), which does not have the bandwidth of the local interaction path within a single computer.

4.2.3 RPC properties

The beauty and the main success of an RPC lies in the fact that neither the client nor the server assume that the procedure call is performed over a network

Control flow for RPC calls



Differences between RPC and local procedure call

- For an RPC, the caller and the callee run in different processes.
- both processes (caller and callee) have

  no shared address space.

  no common runtime environment.

  different life span of → client and server (see page 44).
- Handle errors occurring during a RPC call, e.g. caused by machine crashes or communication failures

  RPC-based applications must take communication failures into consideration.

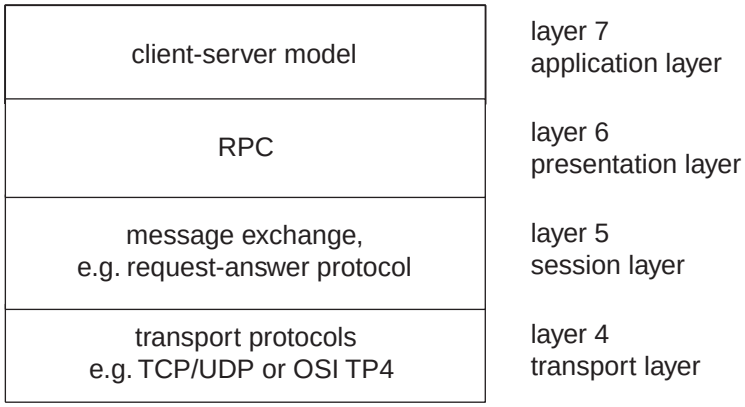
Basic RPC characteristics

An RPC can be characterized as follows

1. uniform call semantics.

2. "type-checking" of parameters and results.
3. full parameter functionality.
4. Optimize response times rather than throughput.
5. Transparency, in particular → transparency (see page 28) concerning exception handling and communication failures (relevant for the programmer).

RPC and OSI



RPC vs message exchange

RPC	**message exchange**
synchronous (generally)	asynchronous
1 primitive operation (RPC call)	2 primitive operation (send, receive)
messages are configured by RPC system	message specification by programmer
one open RPC	several parallel messages possible

The RPC protocol defines only the structure of the request/answer messages; it does not supply a mechanism for secure data transfer.

- **RPC exchange protocols**

  There are different types of RPC exchange protocols

  the request (R) protocol

the request-reply (RR) protocol

the request-reply-acknowledge (RRA) protocol.

4.3 Structure of RPC messages

- **Request message**

```
typedef operation = struct /* defines the requested operation
*/
    cardinal program-number;
    cardinal version-number;
    cardinal procedure-number;

struct request message =
    cardinal identifier;
    cardinal RPC version;
    operation requested service;
    source client;
    list of unspecified parameter list;

struct source =
    struct reference =
        string host;
        string user-name;
        string user-group;
    key verifier;
```

- **Answer message**

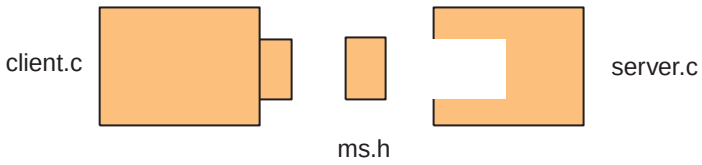
```
struct answer message =
    cardinal identifier;
    status answer status;
    source server;
    list of unspecified result list;
```

The component *server* authenticates the server. Only the *verifier* is set, the rest is not set, since it is assumed that the client - having sent request messages - already knows the server identity.

4.4 Distributed applications based on RPC

We now examine what is needed to implement distributed applications based on remote procedure calls.

4.4.1 Structure of a traditional Unix application



Let a local application consist of the sources client.c, server.c and the header file ms.h. After the sources client.c and server.c have been compiled, the new objects are linked into a main program.

4.4.2 Distributed application

In order to isolate the communication idiosyncrasy of RPCs and to make the network interfaces transparent to the application programmer, so-called *stubs* are introduced.

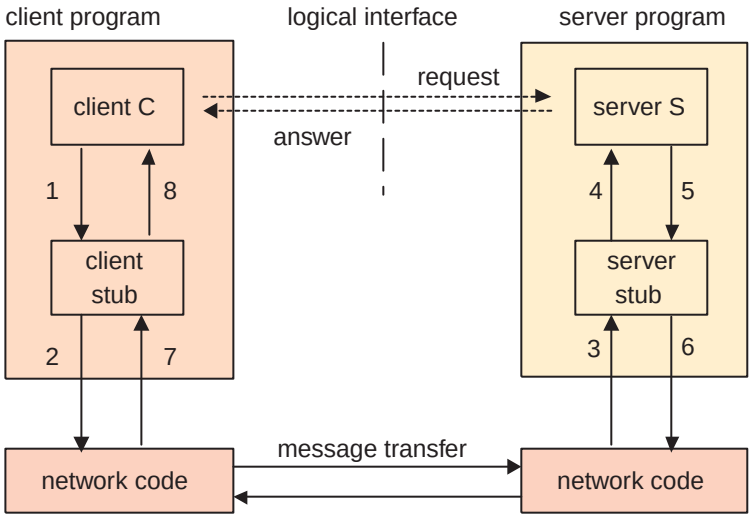
Stubs

Integration of software handling the communication between components of a distributed application.

Stubs encapsulate the distribution specific aspects.

Client Stub: contains the proxy definition of the remote procedure P.

Server Stub: contains the proxy call for the procedure P.



Stub functionality

Tasks of the stubs

1. **Client stub**

   specification of the remote service operation; assigning the call to the correct server; representation of the parameters in the transmission format.

   decoding the results and propagating them to the client application.

   unblocking of the client application.

2. **Server stub**

   decoding the parameter values; determining the call address of the service operation (e.g. a table lookup).

   invoking the service operation.

   prepare the result values in the transmission format and propagate them to the client.

Implementing a distributed application

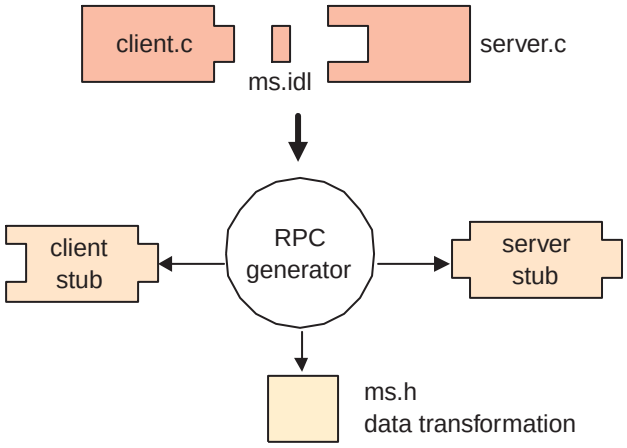
A manual implementation of stubs and of their interfaces to the network is quite error-prone. Therefore, the interfaces between clients and servers should be coded automatically when a declarative specification is given. A so-called *RPC generator* supports the application programmer in this task.

- **RPC generator**

  An RPC generator

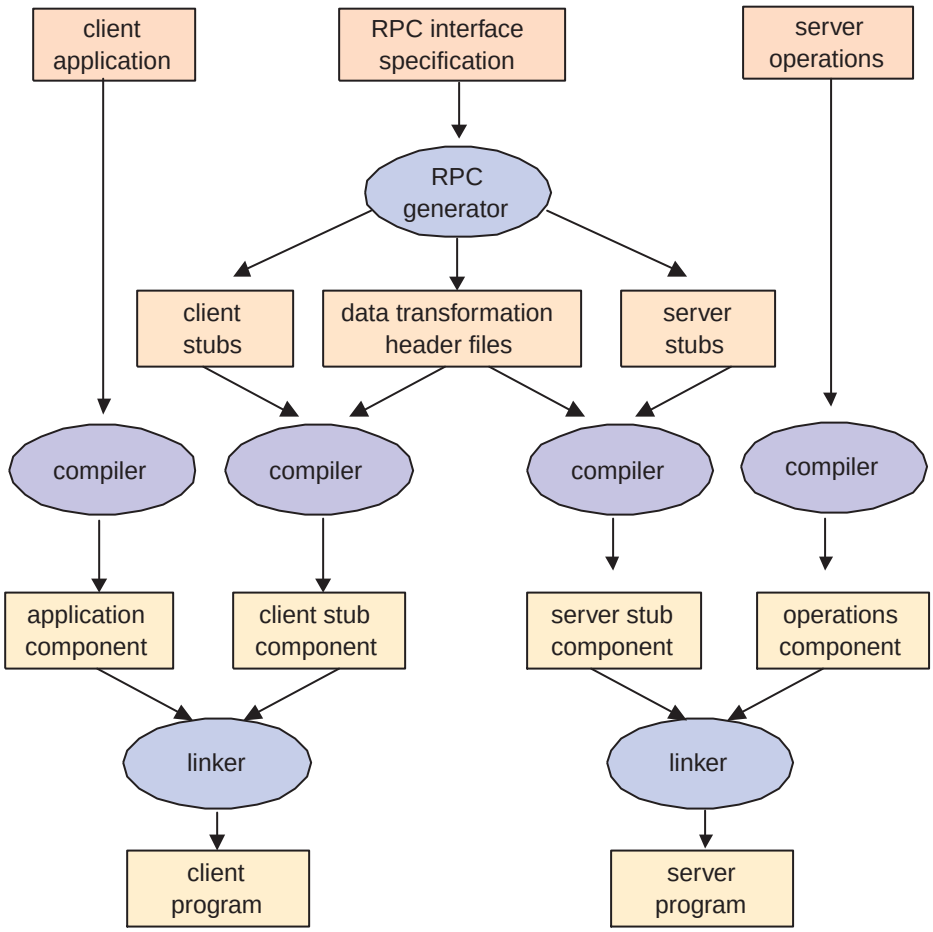
  reduces the time necessary for implementation and management of the components of a distributed application.

  a declarative interface description is easier to modify and therefore less error-prone.



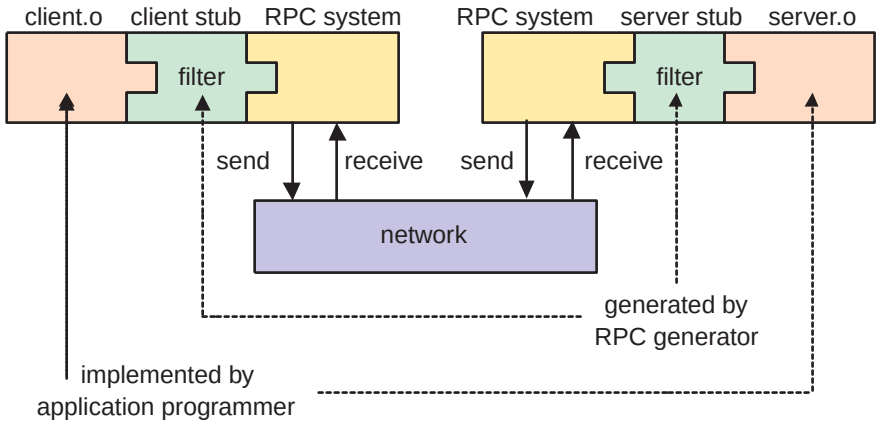
- **Applying the RPC generator**

  The individual steps for generating a distributed application are illustrated in the following figure.



- **Structure of a distributed application**

  The internal structure of a distributed application created using an RPC generator is as follows:



4.4.3 RPC language

Let us now look at the declarative language used to specify the interfaces between clients and servers. Examples of such languages include NIDL by Hewlett Packard and RPCL by Sun.

Structure of the interface description

Use of a declarative language for the specification of the interface between components of a distributed application.

```
[interface attribute list]
    interface identifier
        constant declarations
        type declarations
        operation declarations
```

Purpose of the interface attribute list

identification of the RPC system.

fixed ports through which the server may be invoked.

Constant declaration

Examples are

```
const int MAX_STRING_LENGTH = 128
const char DESCRIPTION = "description"
```

Type declaration

```
typedef [attribute] declaration
```

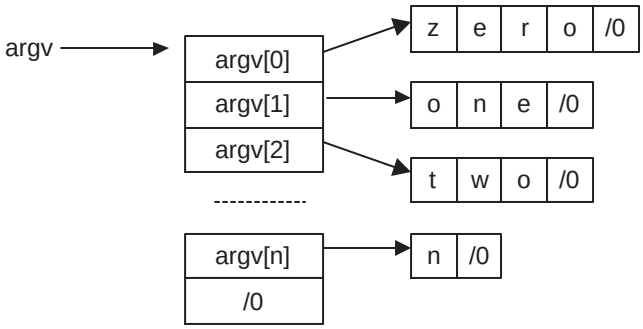
The attribute *transmit_as*: mapping of the internal data representation to the external transmission format.

Example

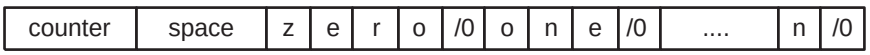
- Type declaration of a data record

```
typedef struct {
    long counter; /* number of arguments */
    long space; /* required storage space */
    char args[]; /* list of arguments */
} external_arguments;
typedef [transmit_as (external_arguments)] char *argv[];
```

- **Internal data representation**



- **External data representation for network transmission**



Operation declaration

General structure of an operation specification

operation :: = [operation_attribute] result_type operation_name (parameter_list);

Each parameter has the following structure:

parameter ::= parameter_type [parameter_attribute] parameter_name

- Among others the following operation attributes are possible:

  idempotent, may-be, broadcast.

- The following parameter attributes are possible: in, out.

- Example

```
[idempotent]
status do_command (
    long [in] argc; /* number of arguments */
    rex_argv [in] argv; /* field of pointers referring to
    arguments */
)
```

4.4.4 Example of an interface description

We aim at a distributed application, where a server manages the calendar entries and clients can create new entries and request existing entries.

- **Interface specification**

```
interface calendar
    const int MAX_STRING_LENGTH = 128
    typedef int status;
    typedef string calendar_entry[MAX_STRING_LENGTH];
    [idempotent] status get_entry (
        string [in] user;
        long [in] time;
        calendar_entry [out] entry; )
    status put_entry (
        string [in] user;
        long [in] time;
        calendar_entry [in] entry; )
```

- **Header file : calendar.h**

```
#define MAX_STRING_LENGTH 128
typedef int status;
typedef char *calendar_entry;
extern status get_entry ();
extern status put_entry ();
```

- **Client stub : calendar_client.c**

```
#include calendar.h
status get_entry (user, time, entry, clnt)
    char *user;
    long time;
    calendar_entry *entry;
    CLIENT clnt; /* refers to the client description;
    generated by the RPC library */
    if (client_call (clnt, get_entry, user, time, entry,
    timeout) != rpc_success)
        return (false);
    else return (true);

status put_entry (user, time, entry, clnt)
    char *user;
    long time;
    calendar_entry entry;
    CLIENT clnt;
    if (client_call (clnt, get_entry, user, time, entry,
    timeout) != rpc_success)
        return (false);
    else return (true);
```

- **Server stub : calendar_server.c**

```
#include calendar.h
main ()
    if (!  svc_register(get_entry, transportprotocol)) return
    (false);
    if (!  svc_register(put_entry, transportprotocol)) return
    (false);
    svc_run();
```

4.4.5 Phases of RPC based distributed applications

We distinguish between 3 phases:

a) design and implementation,

b) binding of components, and

c) invocation: a client invoking a server operation.

Component binding

Problem: How do the components of a distributed application (client and server) find each other?

Variants of client-server binding

- **Static binding**

  Static binding takes place when the client program is generated. In this case, the server address is hard-coded within the client program.

- **Semistatic binding**

  The client determines the server address during the initialization of the client process. The server address remains unchanged within the client code for the whole life span of the client process.

  – Binding can take place via

    entry in a database.

    broadcast or multicast message.

    name service.

    mediation mechanism ("broker" or "trader"); a broker mediates between client and server.

- **Dynamic binding**

  The server address is determined immediately before an RPC is performed.

  – client operation is not hindered by the following situations

    server migration.

    client binding to alternative servers (if the commonly called server is not available).

    dynamic server replacement.

Mediation and brokering

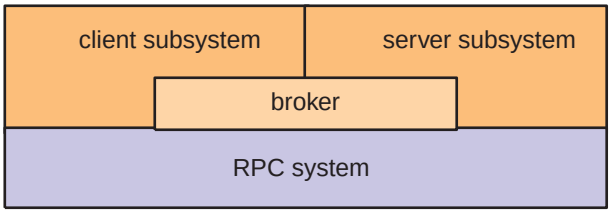
Possible terms for a mediation component are: broker or trader

- **Functionality of a broker**

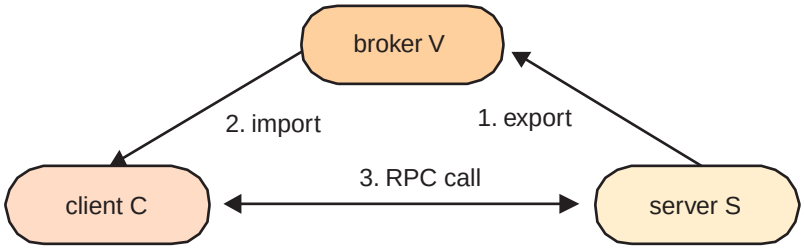
Functions

- servers register their available service interfaces with the broker ("export interface").
- the broker supplies the client with information in order to localize a suitable server and to determine the correct service interface ("import interface").

- **Embedding a broker subsystem**



- **Client-to-server binding**



- **Broker information**

  A broker manages information about the available, exported interfaces.

  server names ("white pages")

  service types ("yellow pages")

  behavioral or functional attributes

  static attributes: functionality of the provided services, cost, required bandwidth.

  dynamic attributes: current server state.

- **Handling client requests**

The embedding of the broker into the client-server interaction depends on the way client RPC requests are handled. The broker may either just provide the service interface to the client or act as a mediator between client and server.

Alternatives for handling client RPC requests

- alternative 1: *direct* communication between C and S.
- alternative 2: *indirect* communication between C and S; communication between C and S is only possible via broker V (or several brokers).

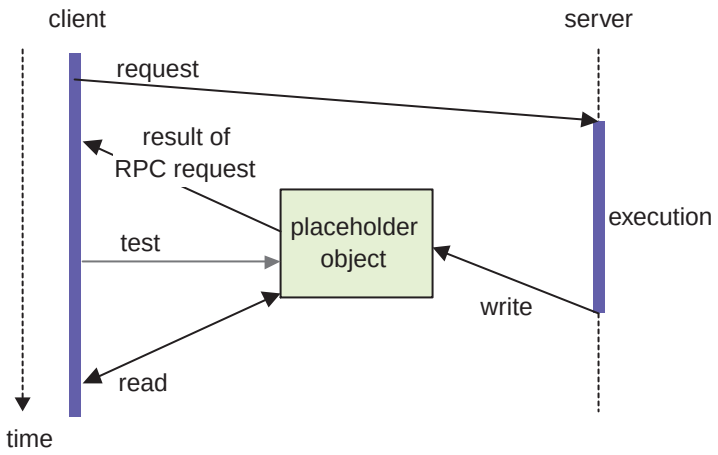
4.5 Asynchronous RPC

Most RPC protocols are synchronous, i.e., client and server are synchronized and their execution becomes sequentialized. When we look at the producer-consumer interaction where a producer does not expect results from the consumer, we see a method of interaction which is particularly easy to achieve with asynchronous RPCs.

4.5.1 General execution model

network-based window systems (like X-11) realize a type of asynchronous RPC between server (X-11 window system) and client (user application).

Problem: returning the results of the RPC request.
asynchronous arrival of server results at the client side.
$\Rightarrow$ use of placeholders to store asynchronously arriving results.



4.5.2 Result of an asynchronous RPC call

In an asynchronous RPC, the process of providing a result starts with the creation of an empty place holder (Futures) data structure.

1. a Future is created (without content).
2. execution of valid operations on Futures.

- Operations on Futures
  - test (with/without waiting).
  - write.
  - read.
- The results of multiple asynchronous RPC requests may be combined into result sets ("FutureSets").
- **Example of an asynchronous RPC request**

```
print_file (string:  file)
boolean:  status;
    future f = FInvokeprint_file (file); /* asynchronous
    procedure call */
    "local operation"
    while IsReady (f, maxwaitingtime)
        FClaimDrucke_Datei (f, &status); "process result"
    Discard (f);
    return (status);
```

4.5.3 Synchronous vs. asynchronous RPC

1. advantages of synchronous RPC.

   easier to comprehend.

   simpler semantics.

   no communication buffer necessary.

   easier error handling.

2. advantages of asynchronous RPC.

   supports parallel processing.

   increases throughput.

   more powerful means of communication.

4.6 Failure semantics of RPCs

Let us now look in more detail at the failure semantics of remote procedure calls. The following failures may impair correct processing of an RPC.

4.6.1 Sources of failures

Computer crashes

Here we have to distinguish between the server machine and the client machine as well as the time when the crash occurs.

- **Target server machine is unavailable**

  The client cannot contact the server, either because the server no longer exists, has crashed, or simply because the network is congested.

  Failure handling
  by transport protocol,
  by exception handling in the client application program,
  by broker; broker selects another server.

- **Server crash**

  A more uncomfortable failure may occur when the server machine crash while processing an RPC.

  There are three possible situations

  – Server breaks down *prior to processing* the RPC request ⇒ Target server machine is unavailable.
  – Server breaks down *in the middle of processing* the RPC request, but before producing results (and, possibly, some side-effects) ⇒ endless waiting of the client.
  – Server breaks down *after processing* the RPC request, but before the answer message could be sent to the client ⇒ notification of the client.

- **Client crash**

  If the client machine crashes after the request for a service operation has been sent, the server subsystem processes an orphan request.

  Solutions to handle client crashes:

- *explicit termination*: after reboot, the client deletes all open requests at the server machines.
- *reincarnation*: the client has a crash counter. Each call contains the current value of the crash counter ⇒ detection of orphaned request, i.e. requests with invalid crash counters.
- status request from server to client.
- expiration time of requests: if the result cannot be delivered before the request expires the request may be of no interest.

Availability of service operation

The requested service operation is no longer supported by the target server subsystem.

Message loss

The request message as well as the answer message can be lost.

In both cases, timers can provide an adequate solution to handle these failures.

4.6.2 Types of RPC call semantics

Any communication between a client and a server is subject to losses of request messages, losses of answer messages as well as crashes (and restarts) of the client or the server machine. Therefore, we distinguish between different RPC call semantics.

- *may-be*: all that is guaranteed is that the request will be executed no more than once;

  may-be is often used in the context of producer-consumer interaction; the result may only be a side-effect on the server site, i.e. the client does not even receive the result (even if no error occurs).

- *at-least-once*: regardless of message loss, the request will be processed at least once.

- *at-most-once*: guarantees atomicity of a request.

Variants of server crash

- before request R has been completely received and locally stored ⇒ after server reboot, there is no processing of R.
- during processing of request R; R has been locally stored as open request ⇒ after server reboot, the execution of R is newly started; a transaction mechanism is necessary to preserve data consistency.
- after execution of R, but before the result could be delivered to client; the result is locally stored ⇒ after server reboot, the saved result is delivered to the client.

- *exactly-once*: the request is processed exactly once despite message loss or machine crash.

4.6.3 RPC call semantics vs. failure type

Depending on the RPC call semantics and the failure type, the number of executions and the number of results returned to the client are given.

	no error	*message loss*	*additionally server crash*	*additionally client crash*
may-be	execution: 1 result: 1	execution: 0/1 result: 0/1	execution: 0/1 result: 0/1	execution: 0/1 result: 0/1
at-least-once	execution: 1 result: 1	execution: ≥ 1 result: ≥ 1	execution: ≥ 0 result: ≥ 0	execution: ≥ 0 result: 0
at-most-once	execution: 1 result: 1	execution: 1 result: 1	execution: 0/1 result: 0/1	execution: 0/1 result: 0
exactly-once	execution: 1 result: 1	execution: 1 result: 1	execution: 1 result: 1	execution: 1 result: 1

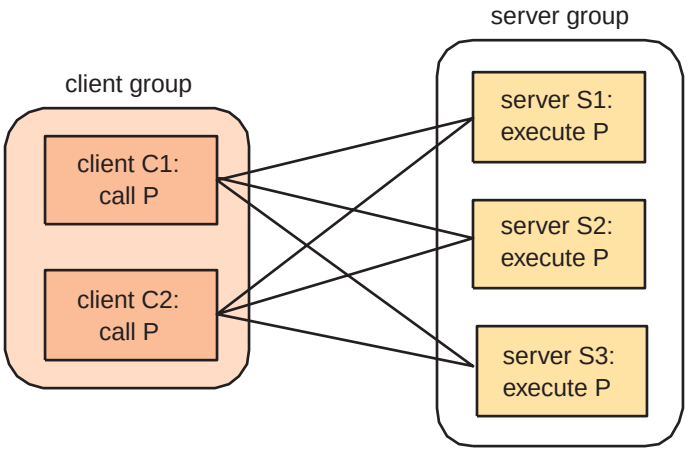
Most RPC systems support either the at-most-once or at-east-once semantics.

4.6.4 Failure tolerant services

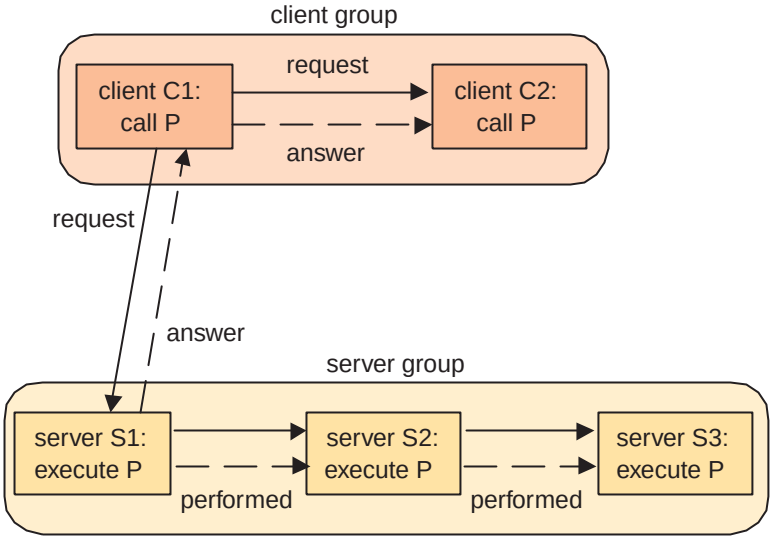
There may exist multiple redundant services as well as replicated RPC calls; server copies and client copies are grouped together into server and client groups.

Modular redundancy

Client requests are sent to and processed by all server replicas (active replication). Each server replica sends its result to all client replicas. The clients vote on the received results.



Primary-standby-approach



At any specific time, there is only one replica acting as master (primary replica); RPC requests are always propagated to the primary replica; at checkpoints the current state is propagated to the secondary replicas.

- in case of an error the master is replaced by a backup replica.

4.7 Comparison between RPC systems

The following table gives an overview of several RPC systems.

	Sun RPC	**HP RPC**	**Modula/V RPC**
request type	blocking; non-blocking/ bundled acknowledgment	blocking; non-blocking/ no acknowledgement	blocking (concurrency by threads)
request semantics	depends on transport protocol	may-be (asynch.); at least-once; at most-once	at least-once; at most-once
binding	static; dynamic (YP)	dynamic (Broker)	dynamic
interface description	RPCL	NIDL	Modula-2
server test	-	test messages	-
transport protocol	TCP/IP, UDP/IP	UDP/IP	VMTP

Animation Remote Procedure Call

siehe Online Version

Chapter 5

Design of distributed applications

In *traditional, nondistributed* applications, procedures or modules help to structure system functionality and data structures. Components of the application use procedures or modules to encapsulate algorithms that logically belong together. This encapsulation may also solve certain reusability issues. The binding of all components into a complete software system is purely static. The following sections discuss the structured design of distributed applications.

5.1 Issues

Software engineering of *distributed applications* raises interesting issues. In particular, the following problems must be considered:

1. Specification of a suitable software structure

   Applications must be decomposed into smaller, distributable components; encapsulation of data and functions.

   Which functionality is provided locally and which remotely?

   How should we test and debug distributed applications?

2. Mechanisms for name resolution

   How can an application localize and make use of a remotely provided service?

   Assignment of names to addresses.

   What should happen if a client cannot contact the localized server subsystem?

3. Communication mechanisms

   Selection of the desired communication model, e.g. client-server model or group communication.

   How does the application (both client and server) handle network communication errors?

4. Consistency

   How can the data be kept consistent, particularly for replicated data?

   If a cache is used for performance improvement, then it must be kept consistent with the stored data.

   User interface consistency for the individual components.

5. User requirements

- Functionality and reconfigurability of the distributed application and its components.
- Service quality, such as security, reliability, fault tolerance and performance.
- What kind of security mechanisms are provided? Is authentication an issue?
- Which actions will be triggered if a client cannot communicate with its server?
- What type of heterogeneity is necessary?
- What efficiency (performance) is expected?

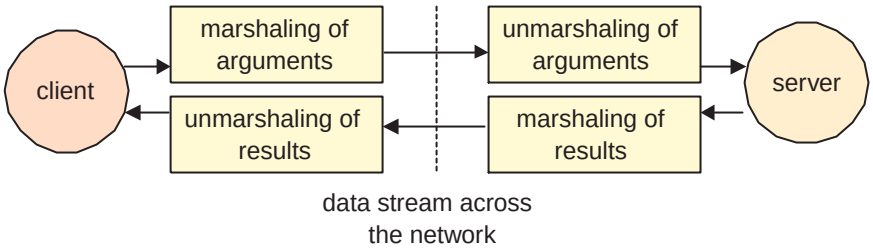
5.2 External data representation

In a heterogeneous environment (PCs, workstations, and mainframes form the hardware basis of the distributed system and installation of different vendor software) different data representations exist. For example, processors of Motorola and Intel represent words in different ways:

$\Rightarrow$ requirement to enable data transformation.

One way to get the desired *independence* from hardware characteristics while exchanging messages can be provided by an external data representation which is independent of any individual method of representing data.

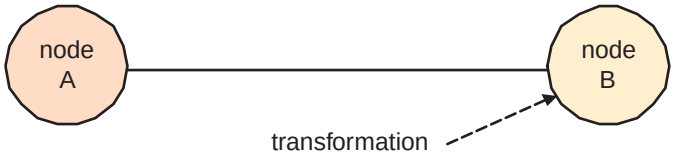
5.2.1 Marshalling and unmarshalling



marshal: parameter serialization to a data stream.

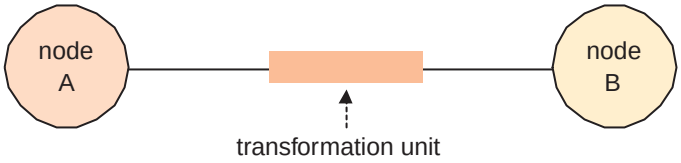
unmarshal: data stream extraction and reassembly of arguments.

5.2.2 Centralized transformation



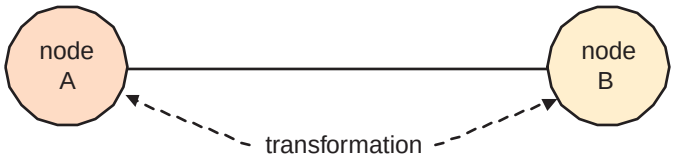
Only B transforms data, both for data to be sent to A and data received from A.

5.2.3 Special transformation unit



This method is often used for linking fast local networks (gateways). It is costly, since special hardware and maintenance are necessary.

5.2.4 Decentralized transformation



All nodes execute data transformations.

Variants

- A transforms data which are then sent to B; B transforms data which are then sent to A.
- A transforms data by B; B transforms data by A.
- A and B transform data in a network-wide standard format; the respective recipients retransform the received data into the local format.

  If new system components are dynamically added to the distributed system, the new system components simply have to “learn” about the network-wide unique standard representation.

  No special hardware is required.

  Example: XDR as part of ONC by Sun.

5.2.5 Common external data representation

Two aspects of a common external data representation are of importance:

a machine-independent format for data representation, and

a language for description of complex data structures.

Examples: XDR ("eXternal Data Representation") by Sun and → ASN.1 (see page 86) (Abstract Syntax Notation) Other formats are

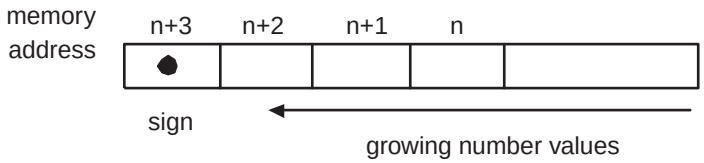
Corba's common data representation: structured and primitive types can be passed as arguments and results.

Java's object serialization: flattening of single objects or tree of objects.

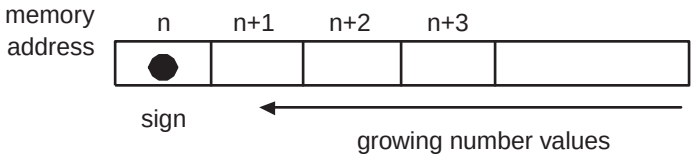
Representation of numbers

For the representation of numbers in main memory, one of the following methods are generally used.

- "little endian" representation: the lower part of a number is stored in the lower memory area, for example the Vax architecture:



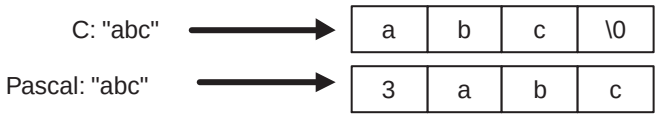
- "big endian" representation: the higher part of a number is stored in the lower memory area, for example the Sun-Sparc architecture:



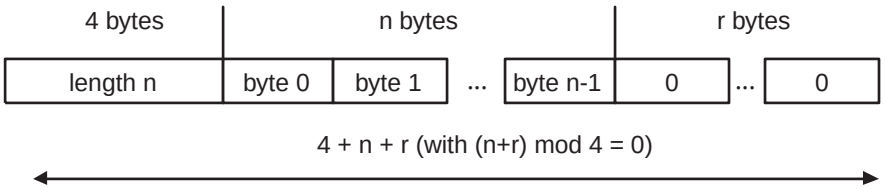
- Convention: for network transfer, numbers which encompass several bytes are structured according to a well-defined representation, such as "big endian".

External representation of strings

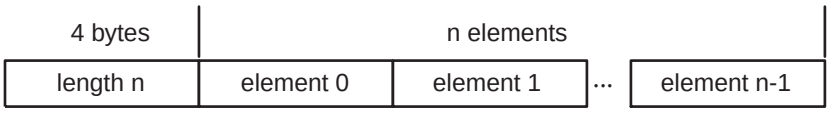
There are different internal representations for strings:



Standardized external representation:



External representation of arrays



Arrays with a variable number of elements are represented as a "counted array."

Transfer of pointers

Since there is no shared address space (for the calling and the called processes), the pointer transfer can be a problem. Approaches for solving this problem:

1. prohibit pointers in a remote procedure call.
2. dereference pointers in a remote procedure call.

   serialize the data structure the pointer is poining to ("marshal"), and transfer the whole data structure.

   - no use of null pointers; instead, we use boolean variables.
   - in heterogeneous environments no use of function pointers.

     However, in a homogenous Java environment function pointers can be dereferenced and the function transferred to the server site.
3. pointer transfer.

Example: ASN.1 (Abstract Syntax Notation One)

As opposed to XDR, the data type is part of the data representation in ASN.1.

- **Data structures**

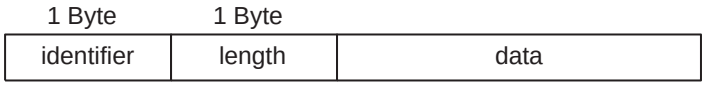
  Standard to specify data to be transmitted (PDU = protocol data unit) in communication protocols;

  defines a number of basic data types: boolean, integer (16 und 32 bit), bitstring, octetstring, IA5string, null or any

  specification of complex data types: sequence, sequenceof, set, setof and choice

  use of type classes: universal, application, context-specific and private

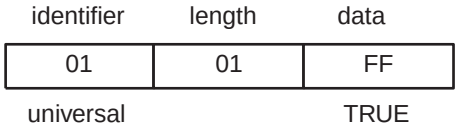
  A data unit has the following structure:



- **Example**

  student := boolean (e.g. assigned the value TRUE).

  ASN.1 Coding



5.3 Steps in the design of distributed applications

Designing a distributed application is a 7-step approach:

1. The repositories of the application data are identified.
2. Data are assigned to individual modules. This is a fundamental step of any software engineering approach.
3. The module interface is defined.
4. Define a network interface.

5. Classify each module as client or server.

6. Registration of servers, i.e. the method in which servers are to be made available to other functional units is determined.

7. A strategy for the binding process of client and server subsystems is defined.

5.4 Integration of management functionality

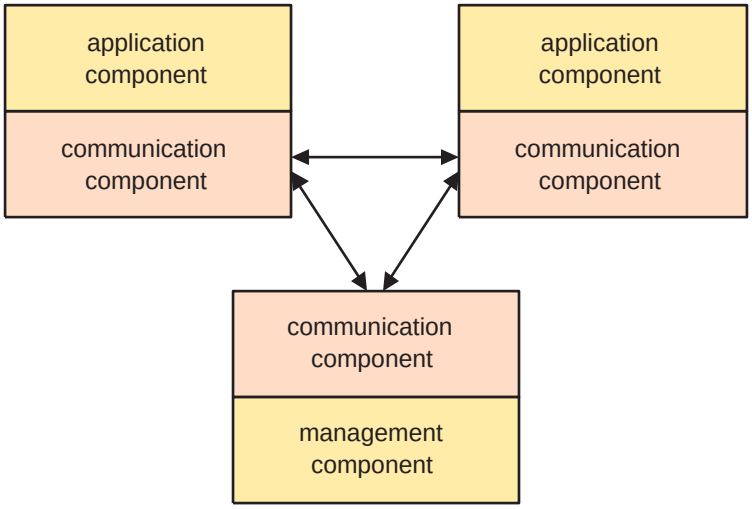
The components of a distributed application are divided into two parts:
application part and
communication part

Definition of a special management component which provides functions to manage the distributed application, e.g. functions for

control and monitoring

evolution of components

easy modification of components at runtime



5.5 Development environment for distributed applications

Distributed applications like CSCW applications are rarely built on top of the bare transport layer. The complexity and the risk of failures is too high. Rather, programmers of such applications work on top of more or less sophisticated development platforms. This is especially useful since distributed applications are typically built by a group of programmers.

5.5.1 Motivation

A motivation to use development platforms comes from insights of the software engineering community.

- Programming-in-the-large.
- Focussing on operational aspects.
- handling the additional complexity due to the distribution of components, i.e. concurrency, nondeterministic behavior caused by transmission delay.
- potentially high maintenance cost.
- **Distributed software environment**

  When a group of programmers has the task to build a distributed application, there is a need for distributed file services and distributed code management.

  the group of programmers work in a distributed environment, e.g. applying a groupware system to support communication, coordination and cooperation.

  a need for tools that help to model concurrency (e.g. based on Petri nets), partial failures, message delays, etc.

5.5.2 Distributed applications in ODP

The Open Distributed Processing (ODP) has been introduced by ISO with the goal of defining a reference model that integrates a wide range of standards for distributed systems.

- ISO/OSI reference model comprises the following standards

  specification of interfaces between heterogeneous system components (physical interfaces).

  communication protocols between connected system components (seven layers).

- ODP (Open Distributed Processing)

  it aims at defining a reference model which integrates a large number of standards for distributed systems and solves the consistency problem within distributed systems.

- Difference between ODP and RPC.

  for ODP, the distribution is the general case; it is the initial position.

  RPC is the extension of a language construct developed for a local application.

- **ODP standard**

  For standardization purposes, ODP identifies four different types of interfaces:

  A programming interface specifies calls to well-defined functions (e.g., data base queries).

  A human-computer interface specifies the user interfaces.

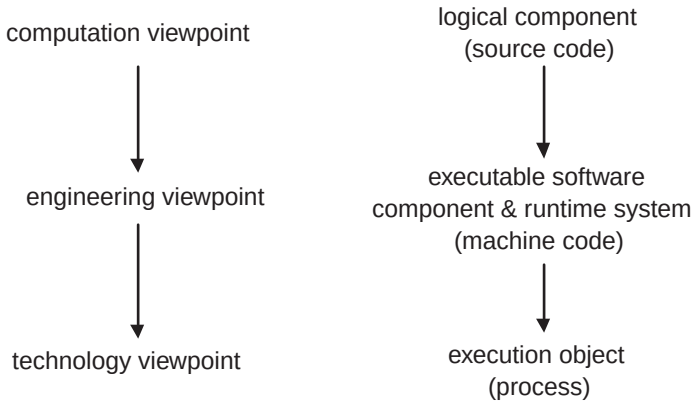
  A communication interface defines a specification compliant to the OSI reference model.

  An interface for external (physical) storage media specifies the modes of information exchange.

5.5.3 Viewpoints in ODP

ODP tries to reduce the inherent complexity of a distributed system by means of introducing five different viewpoints. Each of them focuses on a particular level of abstraction of the distributed system. The viewpoints are: *Enterprise* viewpoint, *Information* viewpoint, *Computation* viewpoint, *Engineering* viewpoint, and *Technology* viewpoint.

Viewpoints and software engineering



Ideally, we would like to *start with the viewpoint* that is most appropriate for the organization in question, and would then regard other viewpoints while the system evolves. Often the computation viewpoint is used as a starting point.

Enterprise viewpoint

This viewpoint focuses on the activities (and aspects of their execution) that optimally pursue the objectives of the target organization.

- deals with the overall goals that the distributed system should reach within the organization.
- considers the organizational structure and the information infrastructure.
- Users of the viewpoint:

  upper management; they consider the organization as an entity.

Information viewpoint

This viewpoint focuses on aspects of the *structure*, the *control* of and the *access to information* that is needed for the key activities of the organization. It identifies information objects and describes the methods for manipulating and managing these objects.

- System specification is functional rather than algorithmical.

- Users of the viewpoint:

  system analysts, system designers themselves, as well as, (upper-level) managers and data modelers of the information.

Computation viewpoint

This viewpoint concentrates on aspects of the logical distribution of data and subsystems. Physical distribution is not considered at this level of abstraction.

- The system requirements refer to cooperating subsystems that as a whole realize the overall system functionality. Structuring and composition of functional units allow the definition of (abstract) data types.
- application structure is independent of the runtime environment.
- Users of the viewpoint: application programmers.

Engineering viewpoint

This viewpoint looks after the support of the distributed applications as far as the physical system environment is concerned. Among others, aspects are:

physical distribution of data and subsystems (also called allocation).

heterogeneity of systems components.

quality of service, e.g. performance, reliability, availability, and where needed, response time for real-time applications.

support different → transparencies (see page 28), e.g. location, access, migration, replication, and concurrency transparency.

- supports component distribution and their isolation from the underlying technology
- Users of the viewpoint: system programmers and network specialists.

Technology viewpoint

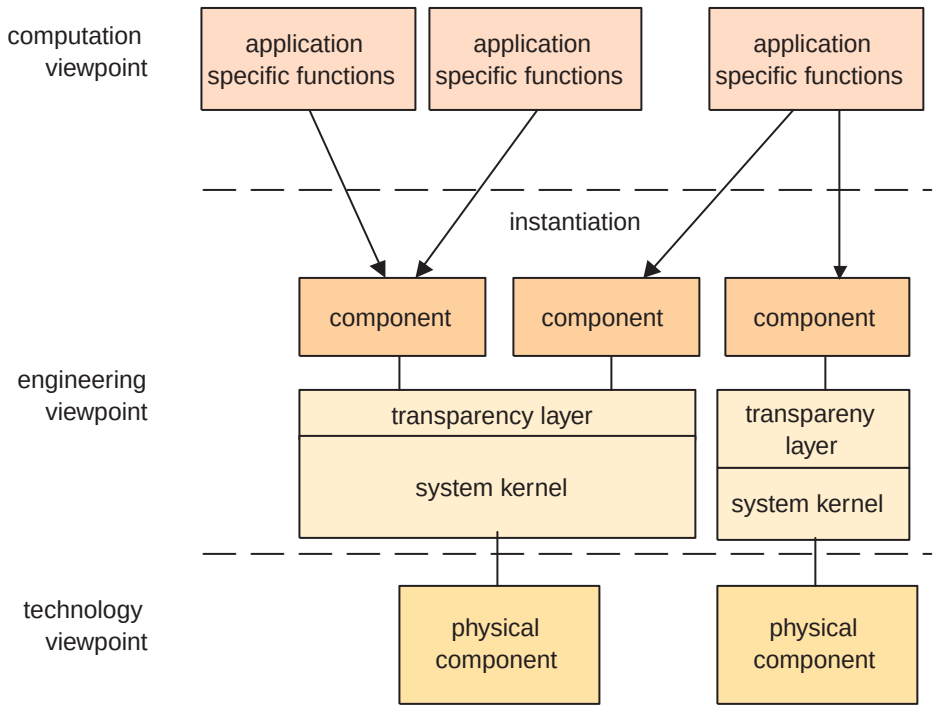
This viewpoint considers the appearance of the different physical and technical subsystems.

focus on technological aspects and real-world components.

real-world components include the *hardware platforms* (including peripheral devices as well as network constituents such as multiplexing units, bridges, routers, etc.) and corresponding *software artifacts* like the operating systems.

- Users of the viewpoint: network and computer support teams.

Relationship between viewpoints



Transparency layer
: provides the desired levels of transparency

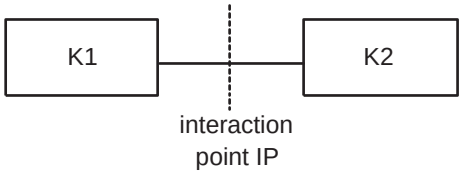
System kernel
: contains the basic functionality the distributed system should provide.

5.6 Distributed application specification

By nature, distributed applications are very complex. Reasons are the autonomous and concurrently acting components, the occurence of communication failures,

heterogeneous processing environments, as well as the size of distributed software system. Rather than modeling the internal processing of the components we will focus on the modeling of the component behavior with respect to its interaction with other components of the distributed application. An open distributed system contains a set of active, autonomous components (agents) executing local activities and interacting with their environment.

⇒ specification of *interaction points* between components.



5.6.1 Formal description

Use of a formal description techniques (FDT) to specify a distributed application.

formal specifications facilitate the proof of correctness and the analysis of distributed applications; this is critical for applications with high security/fault tolerance and quality service demands.

Requirements

1. Expressiveness

   FDT's must be powerful enough to express *the qualitative and quantitative features* of a distributed application.

2. Abstraction

   it must be possible to *hide unnecessary details*.

3. Formal description of semantical aspects.

4. explicit specification of the externally visible behavior, either

   by listing all possible event sequences, or

   by listing the features/rules which constrain the possible event sequences.

5. Composition of already specified building blocks.

In the area of distributed systems (OSI), three standards for FDT's have been developed by ISO.

Estelle

LOTOS

SDL

5.6.2 LOTOS

LOTOS stands for "Language Of Temporal Ordering Specification". The language consists of two parts

a process algebra to specify the system behavior.

specification of abstract data types and values (language based on ACT ONE).

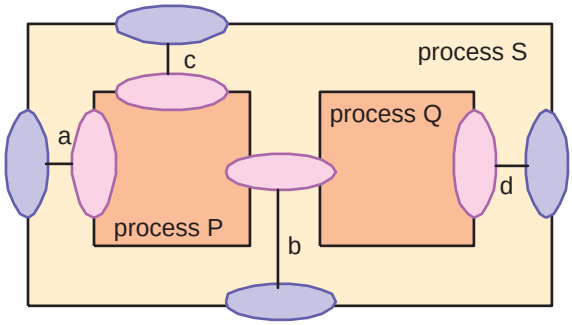
- **Basic concepts**

  Basic concepts of LOTOS are process instances and gates; gates define the interaction points.

  synchronous communication between two or more process instances is carried out through common gates.

- A LOTOS specification describes the externally visible system behavior; it defines the set of all possible interaction sequences between process instances.

- **Example**



Description in LOTOS

```
process S [a, b, c, d] :  noexit :=
    P [a, b, c] |[b]| Q [b, d]
    where
    process P [t, u, v]:  noexit :=
        t; (u; stop [] v; stop)
    endproc
    process Q [x, y]:  noexit :=
        x; y; stop
    endproc
endproc (* S *)
```

5.6.3 SDL

SDL has been developed in the 1970's by ITU. Originally, is was an informal graphical representation method for the description of systems. Later on, however, it was developed into a full FDT; there is also a more recent version, SDL-88. SDL (Specification and Description Language) is also based on the model of a finite state automaton for behavior description, with extensions describing abstract data types.

- SDL contains constructs for representing structures, behavior, interfaces and communication links; in addition, SDL contains constructs for module abstraction, encapsulation and refinement.
- Behavior is specified by processes communicating through messages ("signals").
- Blocks are the compositions of structure (processes or again blocks) and are used to structure the distributed system.
- SDL is based on a complex graphical representation of the system.

5.6.4 Estelle

Originally, Estelle has been developed for the specification of ISO/OSI services and protocols.

Estelle is based on finite state automata.

the distributed application is divided into *modules* and *channels*; Estelle supports a *hierarchical structure* based on modules.

Estelle supports synchronous and asynchronous parallelism of the components' state automata.

Key concepts of Estelle

Estelle has three basic concepts: module, channels, and structure.

- **Module**

  Modules are described by finite state automata; an automaton can also include spontaneous and nondeterministic state transitions.

  1. Use of variables to provide automaton context description

     i.e. the module state is described by the state of the automaton and current values of the context variables.
  2. Executing a transition requires *no* execution time.

- **Channels**

  Modules communicate through channels, i.e. they specify the module interaction points

  for each channel, a set of interactions are defined. These interactions may be initiated or received by the modules associated with the channel end points.

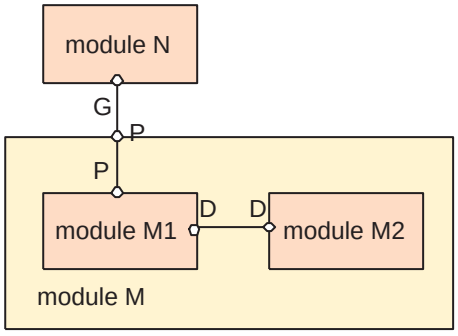
- **System structure**

  Estelle modules are described by the black-box principle. The modules may be hierarchically structured.

  – communication within a module and between parent and child module; two operations are available

  connect: defines external interaction points between children of a module.

  attach: assignment of a parent interaction point to an external interaction point of a child module.



– The respective reverse operations *disconnect* and *detach* remove the communication channel.
– Distinction between 2 module classes:

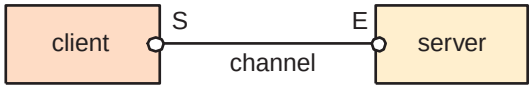
  process: the children of a module are executed in parallel.

  activity: the children of a module are executed sequentially.

  Activities can only be divided into activities, whereas children of processes can be either processes or activities.

Language constructs in Estelle

The following discussion is based on a ("very basic") client-server example; data are transmitted from the client to the server.



- **Channel**

```
channel ch (sender, receiver);
    by sender:
        ConnectionRequest;
        DataRequest (data:  data_type);
        DisconnectionRequest;
    by receiver:
        ConnectionIndication;
        DataIndication (data:  data_type);
        DisconnectionIndication;
```

- **Module**

  A module description contains two parts: the *interface* which is visible outside (the so-called signature) and the *internal* implementation.

  – **Module interfaces**

    Server module interface.

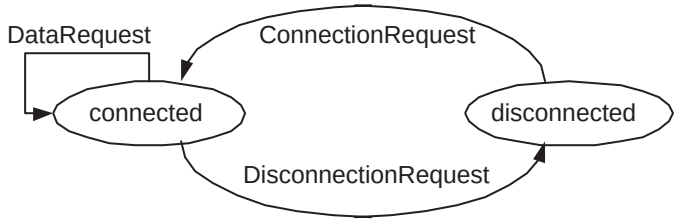
```
module Server process:
    ip E: ch (receiver) individualqueue
end; {Server}
```

    Client module signature.

```
module Client process:
    ip S: ch (sender) individualqueue
end; {Client}
```

– **Implementation of the server module**

```
body ServerBody for Server;
    state Disconnected, Connected;
    initializeto disconnected begin ....  end;
    trans
        when E.ConnectionRequest
            from disconnected to connected
                beginoutput E. ConnectionIndication end;
        when E.DataRequest
            from connected tosame
                beginoutput E. DataIndication end;
        when E.DisconnectionRequest
            from connected to disconnected
                beginoutput E. DisconnectionIndication end;
end; {ServerBody}
```



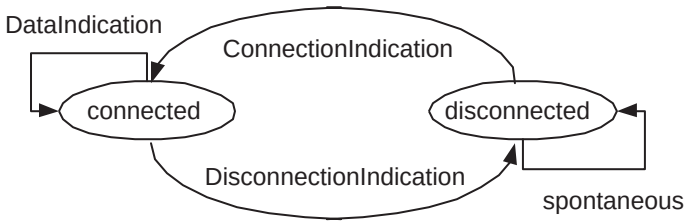
– **Conditioned transitions**

Transitions may have associated additional conditions and priorities. Example:

```
from connected tosame
    provided DataAvailable
    beginoutput S.DataRequest (userdata); end
```

– **Implementation of the client module**

```
body ClientBody for Client;
    state Disconnected, Connected;
    initializeto disconnected begin ....  end;
    trans
        from disconnected tosame /* spontaneous transition */
            provided DataAvailable
            delay (TransmissionPermitted)
            beginoutput S.ConnectionRequest end;
        when S.ConnectionIndication
            from disconnected to connected
            beginoutput S. DataRequest(UserData) end;
        when E.DataIndication
            from connected tosame
            provided DataAvailable
            beginoutput S. DataRequest(UserData) end;
        when E.DataIndication
            from connected tosame
            providednot DataAvailable /* there are no data to be
            transmitted */
            beginoutput S. DisconnectionRequest end;
        when E.DisconnectionIndication
            from connected to disconnected
            begin ...  end;
end; {ClientBody}
```



- **Application structure**

  Structure of the distributed application

```
modvar
    ClientInstance: Client;
    ServerInstance: Server;
    initialize
        begin
            init ClientInstance with ClientBody;
            init ServerInstance with ServerBody;
            connect ClientInstance.S to ServerInstance.E
        end
```

Chapter 6

Basic mechanisms for distributed applications

6.1 Issues

The following section discusses several important basic issues of distributed applications.

- Diskussion of an execution model for distributed applications.
- what is the appropriate error handling?
- What are the characteristics of distributed transactions?
- What are the basic aspects of group communication (e.g. in ISIS) ?
- How are messages propagated and delivered within a process group in order to maintain a consistent state?

6.2 Distributed execution model

6.2.1 Basics

Events

Components of a distributed application communicate through messages causing events the components.

The component execution is characterized by three classes of events:
internal events (e.g. the execution of an operation).
message sending.
message receipt.

- The execution of a component TK creates a sequence of events e_1, ..., e_n,...
- The execution of the component TK_i is defined by (E_i, $\rightarrow_i$) with:

  E_i is the set of events created by TK_i execution

  $\rightarrow_i$ defines a total order of the events of TK_i
- The relation $\rightarrow_{msg}$ defines a causal relationship for the message exchange:

  send(m) $\rightarrow_{msg}$ receive(m), i.e. sending of the message m must take place prior to receiving m.
- There are the following interpretations

  $a \rightarrow b$, i.e. a before b; b causally depends on a.

  a || b, i.e. a and b are concurrent events.

Rules for "happened-before" after Lamport

In order to guarantee consistent states among the communicating components, the messages must be delivered in the correct order. The happened-before relation after Lamport may help to determine a message sequence for a distributed application.

- The following rules apply:

  Events within a component are ordered with respect to the before-relation , i.e. $a \rightarrow b$

  if "a" is a send event of component TK1, and "b" the respective receive event of component TK2, then $a \rightarrow b$;

  if $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$;

  if $\neg (a \rightarrow b)$ and $\neg (b \rightarrow a)$, then a || b; i.e. a and b are concurrent, i.e. they are not ordered.
- Utilization of logical clocks to determine the event sequence.

- Let

  T: a set of timestamps

  C: E $\rightarrow$ T a mapping which assigns a timestamp to each event

  a $\rightarrow$ b $\Rightarrow$ C(a) $<$ C(b)

  If the reverse deduction is valid, too ($\Leftrightarrow$), then the clock is called strictly consistent.

6.2.2 Ordering by logical clocks

Each component manages the following information:

its local logical clock lc; lc determines the local progress with respect to occuring events.

its view on the global logical clock gc; the value of the local clock is determined according to the value of the global clock.

- There exist functions for updating logical clocks in order to maintain consistency; the following two rules apply.

Rules

- Rule R1 specifies the update of the local clock lc when events occur.
- Rule R2 specifies the update of the global clock gc.

1. *Sending event*: determine the current value of the local clock and attach it to the message.
2. *Receipt event*: the received clock value (attached to the message) is used to update the view on the global clock.

6.2.3 Logical clocks based on scalar values

Description

The clock value is specified by positive integer numbers.

the local clock lc and the view on global clock (gc) are both represented by the counter C.

- **Execution of R1**

  prior to event execution, C is updated: C := C + d.

- **Execution of R2**

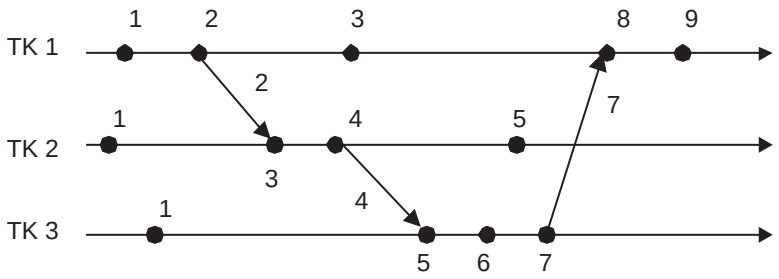
  after receiving a message with timestamp C_{msg} (the timestamp is part of the message), the following actions are performed

  C := max (C, C_{msg})

  execute R1

  deliver message to the application component

Example



- scalar clocks are *not strictly consistent*, i.e.

  the following is not true: $C(a) < C(b) \Leftrightarrow a \rightarrow b$

6.2.4 Logical clocks based on vectors

Description

The time is represented by n-dimensional vectors with positive integers. Each component TK_i manages its own vector $vt_i[1....n]$. The dimension n is determined by the number of components of the distributed application.

$vt_i[i]$ is the local logical clock of TK_i.

$vt_i[k]$ is the view of TK_i on the local logical clock of TK_k; it determines what TK_i knows about the progress of TK_k

Example: $vt_i[k] = y$, i.e. according to the view of TK_i , TK_k has advanced to the state y, i.e. up to the event y.

the vector $vt_i[1....n]$ represents the view of TK_i on the global time (i.e. the global execution progress for all components).

- **Execution of R1**

  $vt_i[i] := vt_i[i] + d$

- **Execution of R2**

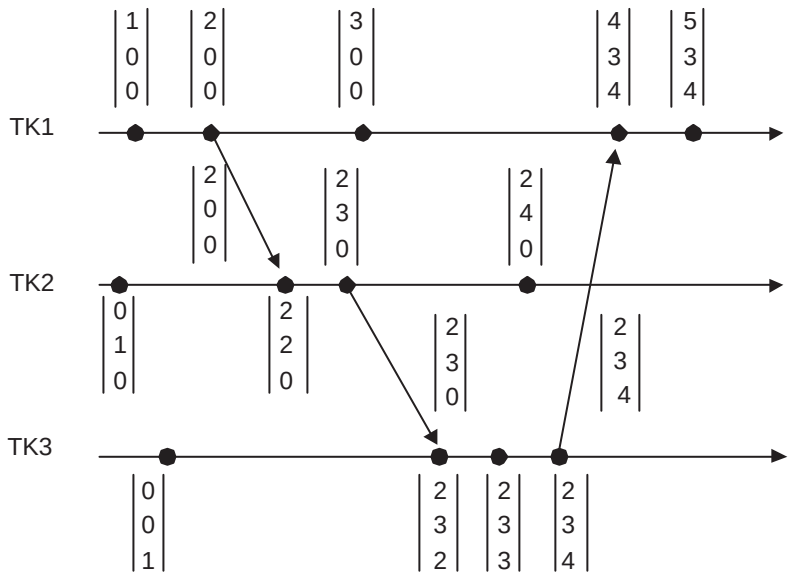
  After receiving a message with vector vt from another component, the following actions are performed at the component TK_i

  update the logical global time: $1 \leq k \leq n$: $vt_i[k] := \max(vt_i[k], vt[k])$

  execute R1

  deliver message to the application process of component TK_i

Example for vector clocks



Animation Vector Clock

siehe Online Version

Characteristics of vector clocks

Comparison of two vector clocks (timestamps) vh[1..n] and vk[1..n]:

$vh \leq vk \quad \Leftrightarrow \quad \forall x: vh[x] \leq vk[x]$
$vh < vk \quad \Leftrightarrow \quad vh \leq vk$ *and* $\exists x: vh[x] < vk[x]$
$vh \parallel vk \quad \Leftrightarrow \quad \neg (vh < vk)$ *and* $\neg (vk < vh)$

- Let a and b be events with timestamps va and vb, then the following is true

  $a \rightarrow b \quad \Leftrightarrow \quad va < vb$
  $a \parallel b \quad \Leftrightarrow \quad va \parallel vb$

- If a of Tk_i and b of Tk_j have been triggered, then the following is true

  $a \rightarrow b \quad \Rightarrow \quad va[i] \leq vb[i]$ *and* $va[j] < vb[j]$
  $a \parallel b \quad \Leftrightarrow \quad va[i] > vb[i]$ *and* $va[j] < vb[j]$

- Vector clocks are strictly consistent.

6.3 Failure handling in distributed applications

6.3.1 Motivation

- Failures in a local application

  handled through a programmer-defined exception-handling routine.

  no handling.

- Failures in a distributed application. Failures may be caused by

  communication link failures.

  crashes of machines hosting individual subsystems of the distributed application.

  failure-prone RPC-interfaces.

  bugs in the distributed subsystems themselves.

- Examples:

  The client crashes $\Rightarrow$ the server waits for RPC calls of the crashed client; server does not free reserved resources.

  The server crashes $\Rightarrow$ client cannot connect to the server.

6.3.2 Approaches for failure detection

In the last two decades, a rich set of failure detection mechanisms has been proposed (prominent examples include the family of fault-tolerant protocols for replicated file services).

- **RPC-info**

  RPC-info is a service that checks and lists the states of all registered servers in the distributed system.

  Clients may request the idempotent service operation `rpc_are_you_there` to verify a server subsystem's state.

  Client C checks the state of server S.

```
callrpc(rpc_are_you_there);
if there is no reply within a specified time then
    C concludes that S is not available;
else
    C concludes that S accepts more RPC calls;
```

- **Monitoring by broker**

  There ar two options

  The broker sends state requests to all registered servers at regular intervals.

  Client C notifies the broker of a potential server crash (e.g. after a timeout occurred during a client request).

6.3.3 Steps for testing a distributed application

Step 1: Test of the distributed application without the communication parts.

enables the test of component functionality. Such aspects as asynchronousness, parallelism and time are not taken into consideration.

Step 2: Test of the distributed application with local communication.

enables time predictions about components without considering network transport times.

Step 3: Test of the distributed application with network-wide communication. In this step, the following problems are identified

time dependencies between component execution.

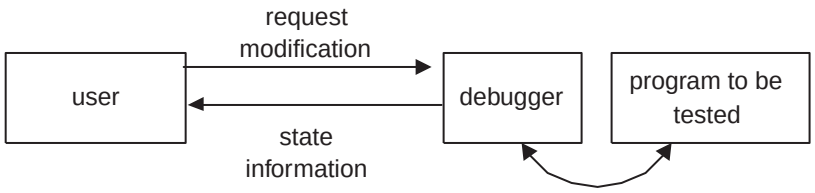
sequential and parallel execution of components.

support of multiple clients.

6.3.4 Debugging of distributed applications

Debugging in distributed systems can be very tricky, for example setting a breakpoint in the server code and inspecting the local variables can cause a timeout in the client process.

Non-distributed debugger



Realization of the debugger and the communication between debugger and testee (program to be tested):

separate processes.

separate processes, but shared memory space.

instrumentation; the program is extended by code which transfers control to the debugger.

Problems with distributed applications

Due to the distribution of the components and the necessary communication between them debugging must handle the following issues.

1. Communication between components.

   Observation and control of the message flow between components.

2. Snapshots.

   no shared memory, no strict clock synchronization.

   state of the entire system.

   the global state of a distributed system consists of the local states of all components, and the messages under way in the network.

3. Breakpoints and single stepping in distributed applications.

4. Nondeterminism.

   In general, message transmission time and delivery sequence is not deterministic.

   $\Rightarrow$ failure situations are difficult to reproduce, if at all.

5. Interference between debugger and distributed application.

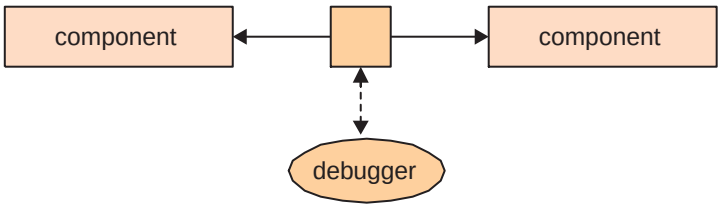
   irregular time delay of component execution when debugging operations are performed.

6.3.5 Approaches of distributed debugging

Debugging of a distributed application might focus on the send/receive events caused by the message exchange and less on the internal operations of the individual components.

Monitoring the communication between components

Only the message flow between components is considered; components are considered as "black boxes"; local test aids are used for the debugging of the individual components.

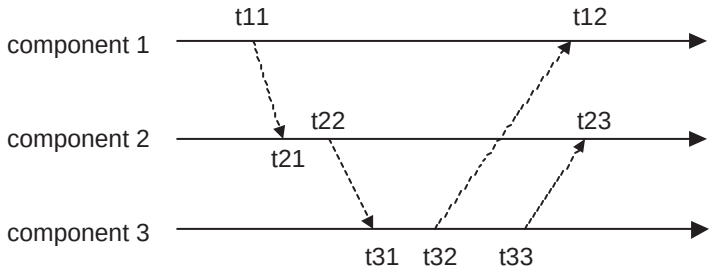


Global breakpoint

- **Approach**

  This approach of global breakpoints is based on the events caused by the message exchange between the components of the distributed application. The events are *partially ordered*.

  use of logical clocks (scalar or vector clock) in order to determine event dependencies.

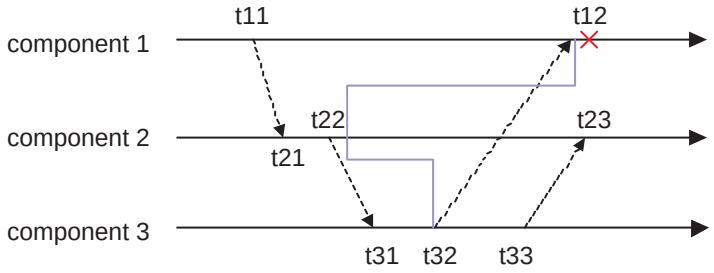


t_{12} and t_{23} are not ordered; t_{11} and t_{33} are ordered;

- **Causally distributed breakpoint**

  In at least one component the breakpoint condition is met.

  all components are rolled back to the earliest possible, consistent state *after* the last event being in a *before*-relationship with the triggering event.



Distributed single stepping

Specification of distributed single stepping:

a distributed single step refers only to interaction points. computing phases are not relevant; they have no impact on the other components of the distributed application.

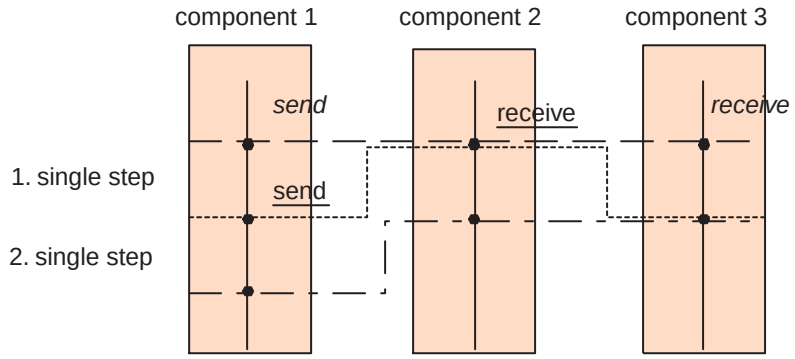
a single step may lead at most from one interaction point to the next interaction point.

all components situated at a sending interaction point are transferred to the next interaction point.

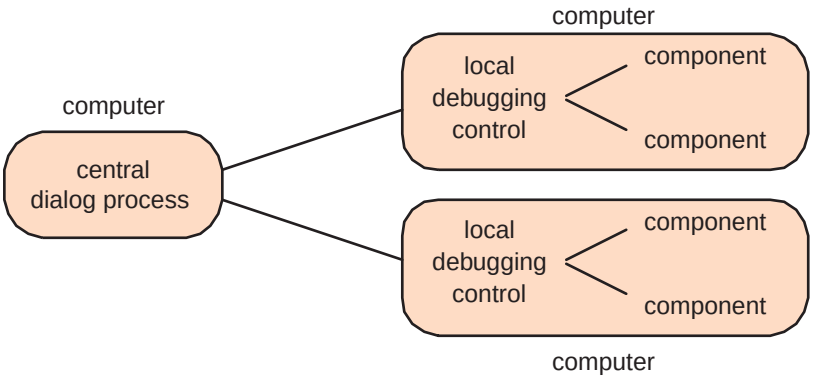
components situated at a receiving interaction point:

if a message is received, then the component is transferred to the next interaction point.

otherwise, the component remains at the current interaction point.



6.3.6 Architecture of a distributed debugger



- a central dialogue process controls the autonomous components which are executed concurrently.
- i.e. the dialogue process must be internally concurrent; it serves as a user interface for debugging the autonomous components (e.g. for user command analysis and output processing).

6.4 Distributed transactions

Distributed transactions are an important paradigm for designing reliable and fault tolerant distributed applications; particularly those distributed applications which

access shared data concurrently.

6.4.1 General observations

Several requests to remote servers (e.g. RPC calls) may be bundled into a transaction. Any transaction whose activities involves multiple servers is a *distributed transaction*.

```
begin-transaction
    callrpc (OP_1, ..., )
    .....
    callrpc (OP_n, ..., )
end-transaction
```

- A distributed transaction involves activities on multiple servers, i.e. within a transaction, services of several servers are utilized.
- Transactions satisfy the *ACID* property: Atomicity, Consistency, Isolation, Durability.

1. *atomicity*: either all operations or no operation of the transaction is executed, i.e. the transaction is a success (commit) or else has no consequence (abort).
2. *durability*: the results of the transaction are persistent, even if afterwards a system failure occurs.
3. *isolation*: a not yet completed transaction does not influence other transactions; the effect of several concurrent transactions looks like as if they have been executed in sequence.
4. *consistency*: a transaction transfers the system from a consistent state to a new consistent state.

6.4.2 Isolation

Isolation refers to the serializability of transactions. All involved servers are responsible for the serialization of distributed transactions. Example:

> let U, T be distributed transactions accessing shared data on the two servers R and S.
>
> if the transactions at server R are successfully executed in the sequence U before T, then the same commit sequence must apply to server S.

Mechanims for handling concurrent distributed transactions are: timestamps, locking, optimistic concurrency control.

Timestamp ordering

In a single server transaction, the server issues a unique timestamp to each transaction when it starts.

In a distributed transaction each server is able to issue globally unique timestamps.

- for distributed transactions, the timestamp is the pair

  (local timestamp, server-ID)

- Assume: timestamp(trans) = t_{trans} and timestamp(obj) = t_{obj}

  transaction trans accesses object obj

  if ($t_{trans} < t_{obj}$) *then* `abort(trans)` *else* `access obj;`

Locking

Each server involved in the distributed transaction maintains locks for its own data items. Prior to each access to an object obj, the transaction trans requests a lock from the managing server, e.g. read or write lock.

- A transaction trans is well-formed if:

  trans locks an object obj before accessing it.

  trans does not lock an object obj which has already been locked by another transaction; except if the locks can coexist, e.g. two read locks.

  prior to termination, trans removes all object locks.

- A transaction is called a *2-phase* transaction if no additional locks are requested after the release of objects ("2-phase locking").

Optimistic concurrency control

Locks and timestamps require additional overhead; therefore, if conflicts are rare, optimistic concurrency control may be useful, since there is no additional coordination necessary during transaction execution.

The check for access conflicts occurs when transactions are ready to "commit";

6.4.3 Atomicity and persistence

These aspects of distributed transactions may be realized by one of the following approaches. Let trans be a transaction.

- **Intention list**

  all object modifications performed by trans are entered into the intention list (log file).

  When trans commits successfully, each server S performs all the modifications specified in AL_S(trans) in order to update the local objects; the intention list AL_S(trans) is deleted.

- **New version**

  When trans accesses the object obj, the server S creates the new version obj_{trans}; the new version is only visible to trans.

  When trans commits successfully, obj_{trans} becomes the new, commonly visible version of obj.

  It trans aborts, obj_{trans} is deleted.

6.4.4 Two-phase commit protocol

This protocol supports the communication between all involved servers of the distributed transaction in order to jointly decide if the transaction should commit or abort.

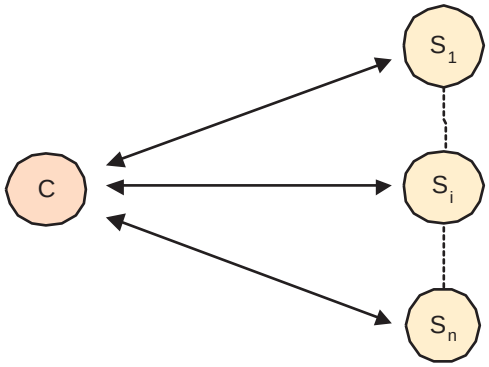
We can distinguish between two phases

Voting phase: the servers submit their vote whether they are prepared to commit their part of the distributed transaction or they abort it.

Completion phase: it is decided whether the transaction can be successfully committed or it has to be aborted; all servers must carry out this decision.

Steps of the two-phase commit protocol

One component (e.g. the client initiating the transaction or the first server in the transaction) becomes the coordinator for the commit process. In the following we assume, client C is the coordinator.



1. Coordinator C contacts all servers S_i of the distributed transaction trans requesting their status for the commit (CanCommit?)

   - if server S_k is not ready, i.e. it votes no, then the transaction part at S_k is aborted;
   - $\exists$i with S_i is not ready

     then trans is aborted; the coordinator sends an abort message to all those servers who have voted with ready (i.e. yes).

2. $\forall$i with S_i is ready, i.e. commit transaction trans. Coordinator sends a commit message to all servers.

3. Servers send an acknowledgement to the coordinator.

Operations

The coordinator communicates with the participants to carry out the two-phase commit protocol by means of the following operations:

canCommit(trans) $\Rightarrow$ Yes/No: call from the coordinator to ask whether the participant can commit a transaction; participant replies with its vote.

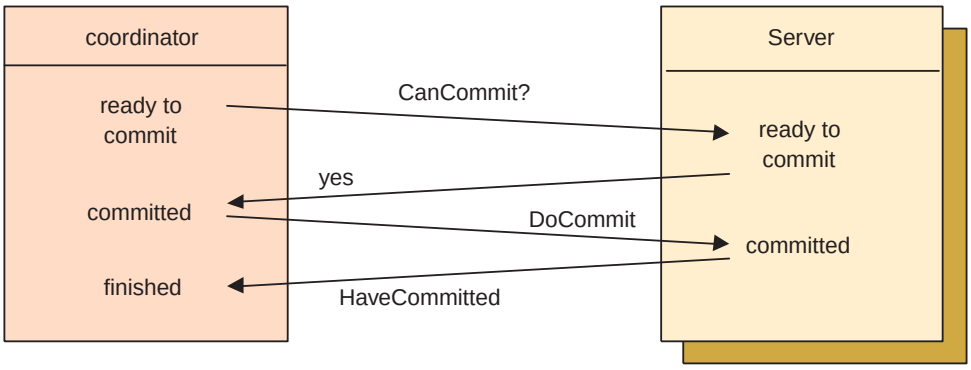
doCommit(trans): call from the coordinator to tell participant to commit its part of a transaction.

doAbort(trans): call from the coordinator to tell participant to abort its part of a transaction.

haveCommitted(trans, participant): call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) ⇒ Yes/No: call from participant to coordinator to ask for the decision on trans.

Communication in the two-phase commit protocol



- Number of messages: 4 * N messages for N servers.

6.5 Group communication

6.5.1 Introduction

Usually, the communication primitives known in operating systems are binary, i.e., an individual sender opens a communication path to a single selected receiver. Group communication facilities the interaction between groups of processes.

Motivation

Many application areas such as CSCW profit immensely if primitives for a group communication are supported properly. Other relevant application areas include fault-tolerant file services and replication-transparent file systems. In both cases, all communication to the "original", primary file server subsystem must also be propagated to the so-called stand-by file service or to the file replicas, respectively. In the first case, a standby file service takes over when the primary site crashes or becomes unavailable for some other reason (e.g., link failures, network partitioning), while in the second case the communication helps to maintain a consistent state among the file replicas.

- typical application for group communication

  fault tolerance using replicated services, e.g. a fault-tolerant file service.

  object localization in distributed systems; request to a group of potential object servers.

  conferencing systems and groupware.

- functional components are composed to a group; a group is considered as a single abstraction.

Important issues

Important issues of group communication are the following:

Group membership: the structural characteristics of the group; composition and management of the group.

Support of group communication: the support refers to group member addressing, error handling for members which are unreachable, and the message delivery sequence.

- Communication within the group

  unicasting, broadcasting, multicasting

- Multicast messages are a useful tool for constructing distributed systems with the following characteristics

  fault tolerance based on replicated services.

  locating objects in distributed services.

  multiple update of distributed, replicated data.

- Synchronization

  the sequence of actions performed by each group member must be consistent.

Conventional approaches

- **Group addressing**

  Central approach: There is a central group server which knows the current state of the group composition.

  Decentralized approach: Each group member is aware of the group structure and its members.

- **Communication services**

  This issue refers to the technology used for the communication between group members.

  Datagrams (for example UDP).

  reliable data stream (for example TCP).

  RPC.

In order to get a consistent global group behavior, even in case of errors, a special group communication support is needed, for example ISIS by Cornell University.

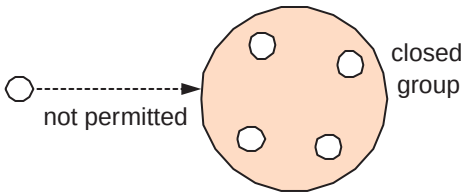
6.5.2 Groups of components

Classification of groups

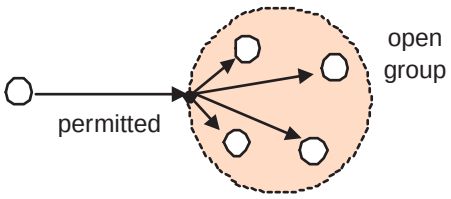
Groups can be categorized according to various criteria.

- **Closed vs. open group**

  Closed group



  Open group



- Distinction between flat and hierarchical group. A flat group may also be called a peer group.

- Distinction between implicit (anonymous) and explicit group.

  In the first case, the group address is implicitly expanded to all group members.

Group addressing

There are mainly three types of group addressing:

a unique group address,

a comprehensive list of addresses of the group members, and

the so-called predicate addressing.

Message M contains a predicate as parameter P which is evaluated by the recipient; if P evaluates to true, then the receiver accepts M; otherwise it deletes M.

6.5.3 Management of groups

Operations for group management

Query for existing group names.

Creation of a new group and deletion of an existing group: groupCreate; groupDelete.

Joining or leaving a group: groupJoin; groupLeave.

Reading and modifying group attributes dynamically.

Read information about group members.

Group management architecture

Again, there are different approaches for providing the group management functionality.

- centralized group managers, realized as an individual group server.
- decentralized approach, i.e. all components perform management tasks.

  requires replication of group membership information, i.e. consistency must be maintained.

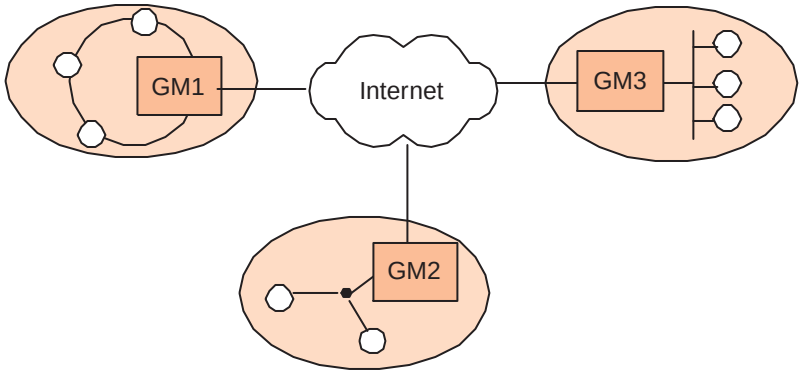
  joining and leaving a group must happen synchronously.

- **Hybrid approach**

  for each LAN cluster, there is a central group manager.

  replication of group membership information and consistency control is limited to the group managers.

  a group manager knows all local components, as well as the remote group managers; on executing a group function (e.g. a modification of the group membership), it contacts the local components and also propagates the information to all other group managers.



6.5.4 Message dissemination

For message dissemination to the group members the following mechanisms are possible options:

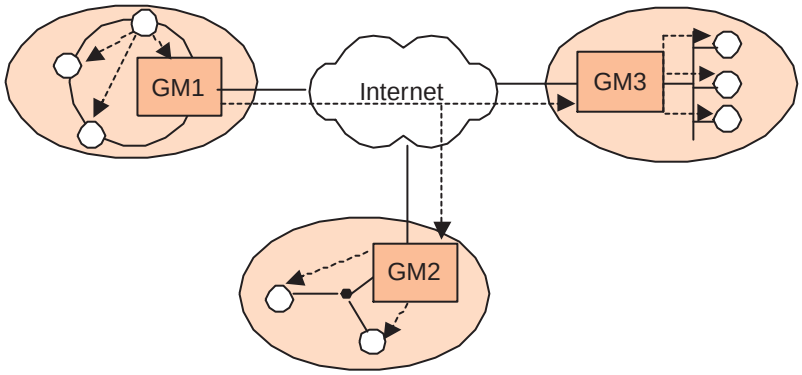
Unicast: send and receive messages addressed to individual group members.

Group multicast: send and receive messages addressed to the group as a whole.

Inter-group multicast: send and receive messages addressed to several groups.

Broadcast: send and receive messages addressed to all components (requires filtering).

Hybrid approach for wide-area networks



6.5.5 Message delivery

Message delivery is an important issue of group communication; two aspects are relevant:

a) *who* gets the message, and

b) *when* is the message delivered.

Atomicity

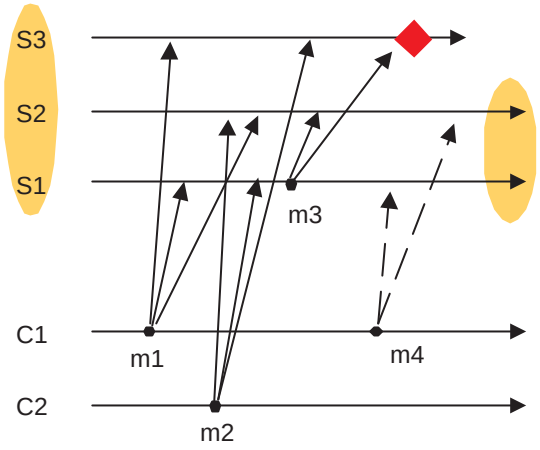
Atomicity specifies *who* receives a message.

- in the absense of errors, we have the "exactly-once" semantics, i.e. messages to the group are delivered exactly once to all group members.
- "all-or-nothing" semantics for messages to the group ("atomic broadcast"), i.e. a message is either delivered *to all group members or to none*.
- atomicity facilitates distributed application programming.

Sequence of message delivery

It is desired to deliver all messages sent to the group G to all group members of G in the *same sequence*, because otherwise we might get non-deterministic system behavior.

Example for group reconfiguration



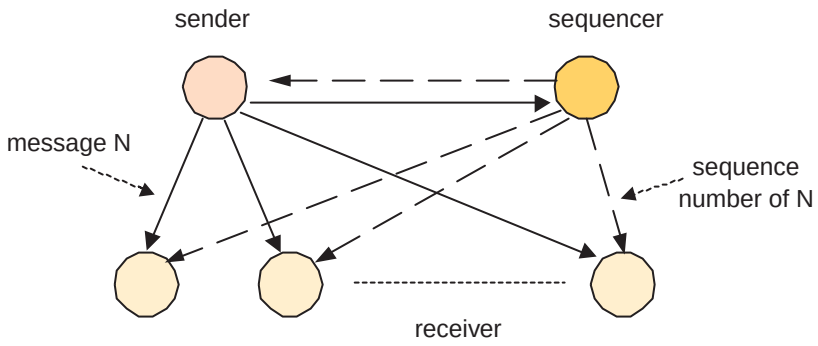
m4 is sent by C1 before the group composition is modified. However, in order to guarantee atomicity, m4 should not be delivered to S1 and S2 (since, due to the crash, it is no longer possible to deliver m4 to S3).

Ordering for message delivery

In an ideal world, all messages could be delivered without delay in the same sequence in which they were sent to the group as a whole. In a distributed system this is unachievable. Therefore, we discuss several ordering methods for message delivery:

- synchronously, i.e. there is a system-wide global time ordering.
- loosely synchronous, i.e. consistent time ordering, but no system-wide global (absolute) time.
- **Total ordering by sequencer**

  An selected group member serializes all the messages sent to the group.



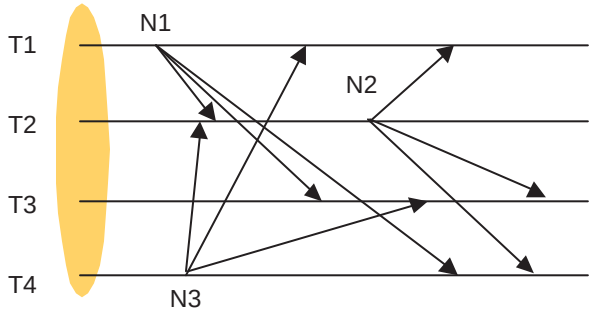
1st step: the sender distributes the message N to *all* group members;

2nd step: sequencer (serializer) determines a sequence number for N and distributes it to all group members; delivery of N to the application processes takes place according to this number.

- **Virtually synchronous ordering**

  The determination of a correct sequence during message propagation and delivery is based on the *before* relation between two events modeling their causal dependency; the approach is similar to the $\rightarrow$ causally distributed breakpoints (see page 111).

  Example
  1. T_1 sends N_1, and T_2 sends N_2 with N_2 dependent on N_1
  2. T_4 sends N_3 with N_1 and N_3 concurrent
  3. at T_2 : N_3 is received before N_1
  4. at T_3 : N_3 is received after N_1



- **sync-ordering**

This approach for message delivery introduces synchronization points. Synchronously ordered messages are delivered to all group members *in-sync*.

let N_i be a synchronously ordered message

all other messages N_k are delivered either before or after N_i has been delivered to all group members.

The ordering method enables the group to synchronize their local states (at synchronization points the group members have a common consistent state).

6.5.6 Taxonomy of multicast

Multicast messages are a very useful tool for constructing distributed systems based on group communication. Multicast communication semantics vary according to reliability guarantees and also according to message ordering guarantees.

Multicast classes

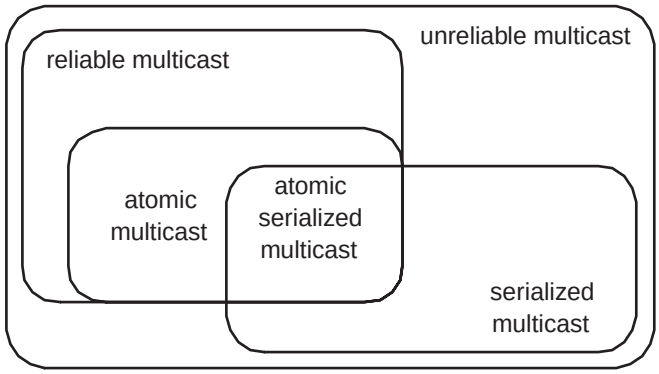
Depending on the message delivery guarantee, five classes of multicast services can be distinguished.

1. *unreliable multicast*: an attempt is made to transmit the message to all members without acknowledgement; at-most-once semantics with respect to available members; message ordering is not guaranteed.
2. *reliable multicast*: the system transmits the messages according to "best-effort", i.e. the "at-least-once" semantics is applied.
3. *serialized multicast*: consistent sequence for message delivery; distinction between

   totally ordered

   causally ordered (i.e. virtually synchronous)
4. *atomic multicast*: a reliable multicast which guarantees that either all operational group members receive a message, or none of them do.
5. *atomic, serialized multicast*: atomic message delivery with consistent delivery sequence

Relationship between multicast classes



Multicasting in overlapping groups

Let G_1 and G_2 be two groups with $G_1 \cap G_2$ not empty; let N be a multicast message to G_1 and L a multicast message to G_2;

1. Delivery according to the total ordering

   $\forall$ m $\in G_1 \cap G_2$: N is delivered to all m before L ; or N is delivered to all m after L.

2. Delivery according to causal ordering

   $\forall$ m $\in G_1 \cap G_2$: if N $\rightarrow$ L, then N is delivered to all m before L.

3. Delivery according to sync-ordering

   let N be ordered synchronously and L arbitrarily ordered (e.g. causal or total);

   $\forall$ m $\in G_1 \cap G_2$: N is delivered to all m before L, or N is delivered to all m after L.

Multicasting can be realized by using IP multicast which is built on top of the Internet protocol IP.

Java API provides a datagram interface to IP multicast through the class `MulticastSocket`.

6.5.7 Group communication in ISIS

The ISIS system developed at Cornell University is a framework for reliable distributed computing based upon process groups. It specifically supports group communication. Successor of ISIS is Horus (URL: http://www.cs.cornell.edu/Info/Projects/HORUS).

> ISIS is a toolkit whose basic functions include process group management and ordered multicast primitives for communication with the members of the process group.
>
> abcast: totally ordered multicast.
>
> cbcast: causallly ordered multicast.

Other systems with similar orientation as ISIS are Totem (URL: http://beta.ece.ucsb.edu/totem.html) and Transis (URL: http://www.cs.huji.ac.il/~transis/publications.html).

abcast protocol

atomic broadcast supports a total ordering for message delivery, i.e. all messages to the group G are delivered to all group members of G in the same sequence.

> abcast realizes a serialized multicast
>
> abcast is based on a *2-phase commit* protocol; message serialization is supported by a distributed algorithm and logical timestamps.

- **Phase 1**

  Sender S sends the message N with logical timestamp $T_S(N)$ to all group members of G (e.g. by multicast).

  each $g \in G$ determines a new logical timestamp $T_g(N)$ for the received message N and returns it to S.

- **Phase 2**

  S determines a new logical timestamp for N; it is derived from all proposed timestamps $T_g(N)$ of the group members g.

  $T_{S,new}(N) = \max (T_g(N)) + j/|G|$, with j being a unique identifier of sender S.

  S sends a commit to all $g \in G$ with $T_{S,new}(N)$.

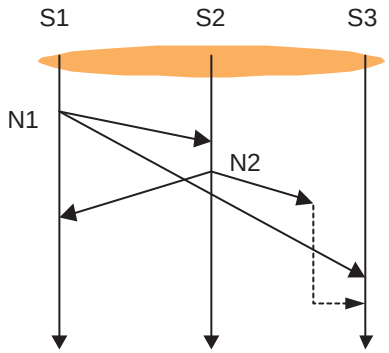
  Each $g \in G$ delivers the message according to the logical timestamp to its associated application process.

cbcast protocol

causal broadcast guarantees the correct sequence of message delivery for causally related messages. Concurrent messages can be delivered in any sequence; this approach minimizes message delay.

- **Introduction**

  The cbcast protocol uses vector timestamps to implement causally ordered message exchange between the members of a peer group.



- **Algorithm of the cbcast protocol**

  Let n be the number of group members of G. Each g $\in$ G has a unique number of {1, ..., n} and a state vector z which stores information about the received group messages. The state vector represents a $\rightarrow$ vector clock (see page 105).

  Each message N of sender S has a unique number; message numbers are linearly ordered with increasing numbers.

  – Let j be a group member of the group G.

    the state vector $z_j = (z_{ji})_{i \in \{1,...,n\}}$ specifies the number of messages received in sequence from group member i.

    Example: $z_{ji} = k$; k is the number of the last message sent by member i $\in$ G and received in correct sequence by the group member j.

    at group initialization all state vectors are reset (all components are 0).

  – *Sending* a message N; j $\in$ G sends a message to all other group members.

    $z_{jj} := z_{jj} + 1$; the current state vector is appended to N and sent to all group members.

  – *Receiving* a message N sent by member i $\in$ G.

Message N contains state vector z_i. There are two conditions for delivery of N to the application process of j

(C 1): $z_{ji} = z_{ii} - 1$.

(C 2): $\forall\ k \neq i$: $z_{ik} \leq z_{jk}$.

Example of a fault-tolerant server

Function: A service manages the assignment of names to telephone numbers; for reasons of fault-tolerance, the service is realized by replicated servers. That means, the service consists of a group of servers. The servers support the operations update and query.

- **Server implementation**

```
# define UPDATE 1
# define QUERY 2
main ()
    isis_init (0);
    isis_entry (UPDATE, update, "update");
    isis_entry (QUERY, query, "query");
    pg_join ("Telephone", .., send_state, ..  );
    isis_mainloop (0);

update (mp)
    message *mp;
    char name [32]; int telephonenumber;
    msg_get (mp, "%s, %d", name, &telephonenumber);
    set_nummer (name, telephonenumber);

query (mp)
    message *mp;
    char name [32]; int telephonenumber;
    msg_get (mp, "%s", name);
    telephonenumber = get_number (name);
    reply (mp, "%d", telephonenumber);

send_state ()
    struct telephone_entry *entry;
    for (entry = head(telephone_db); entry =!
    tail(telephone_db); entry = entry ->next)
        xfer_out ("%s, %d", entry->name,
        entry->telephonenumber);

receive_state (mp)
    message *mp;
    update (mp);
```

- **Client implementation**

```
# define UPDATE 1
# define QUERY 2
main ()
    isis_init (0);
    server = pg_lookup ("Telephone");
    .......

update (name, telephonenumber)
    char *name; int telephonenumber;
    abcast (server, UPDATE, "%s, %d", name, telephonenumber,
    0); /* 0, because no result is expected */

query (name)
    char *name;
    int telephonenumber;
    abcast (server, QUERY, "%s", name, 1, "%d",
    &telephonenumber); /* telephonenumber is the expected
    result parameter */
    return (telephonenumber);
```

Chapter 7

Distributed file service

7.1 Issues

This section introduces schemes for replication and concurrency control that were originally developed in the context of distributed file systems but which could become prevalent in other application domains, as well, e.g. synchronous groupware.

- What are the general characteristics of a distributed file service?
- How to maintain consistency of replicated files?
- What are voting schemes?
- Presentation of the Coda file service.

7.2 Introduction

When a group of programmers has the task to build a distributed application, in addition to distributed code management there is also the need for distributed file services.

7.2.1 Definitions

- **Definition:** A **distributed file system** (e.g. $\rightarrow$ Sun Network File System (NFS) (see page 20)) is characterized by:

a logical collection of files on different computers into a common file system, and

computers storing files are connected through a network.

- **Definition:** A **distributed file service** is the set of services supported by a distributed file system. The services are provided by one or several file servers; a **file server** is the execution of file service software on a computer.
- **Definition: Allocation** is the placement of files of a distributed file system on different computers.
- **Definition: Relocation** changes file allocation within the distributed file system.
- **Definition: Replication**

  there exist multiple copies of the same file on several computers.

  Replication degree REP_d of a file d: total number of copies of d within the distributed file system.

  If $\rightarrow$ replication transparency (see page 29) is supported, the user is unaware of whether a file is replicated or not.

7.2.2 Motivation for replicated files

A distributed file system supporting replicated files has the following characteristics:

Less network traffic and better response times.

Higher availability and fault tolerance with respect to communication and server errors.

Parallel processing of several client requests.

The key concept of a distributed file system is transparency.

User's impression: interaction with a normal, central file system.

Goal to support the following $\rightarrow$ transparency (see page 28) types: location, access, name, replication and concurrency transparency.

7.2.3 Two consistency types

In the context of replicated files we can distinguish between two types of file consistency.

Internal Consistency

A single file copy is internally consistent, e.g. by applying a "2-phase commit" protocol.

Mutual Consistency

It is obvious that all copies of replicated information should be identical $\Rightarrow$ all file copies are mutually consistent, for example by applying the "multiple copy update" protocol.

- *Strict mutual consistency*: after executing an operation, all copies have the same state.
- *Loose mutual consistency*: all copies converge to the same consistent state of information.

7.2.4 Replica placement

A major issue of distributed data store is the decision when and where to place the file replicas.

- **Permanent replicas**

  The number and placement of replicas is decided in advance, e.g. mirroring of files at different sites.

- **Server-initiated replicas**

  They are intended to enhance the performance of the server.

  Dynamic replication to reduce the load on a server.

  file replicas migrate to a server placed in the proximity of clients that issue file requests.

- **Client-initiated replicas**

  Client-initiated replicas are more commonly known as caches.

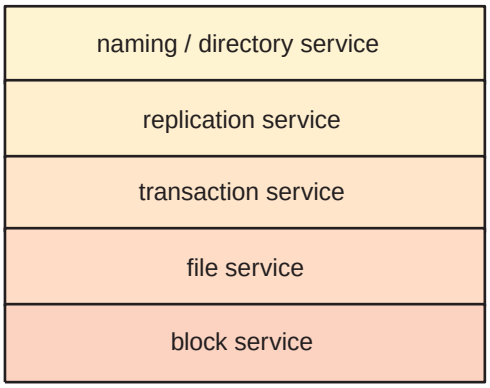
  Used only to improve access times to data.

  Client caches are normally placed on the same machine as its client.

  Replicas are only kept for a limited time.

7.3 Layers of a distributed file service

The functions of a distributed file service are usually arranged in a hierarchical way.



7.3.1 Layer semantics

Each layer of the distributed file service has a specific task.

- **Name/directory service**

  placement of files; file relocation for load balancing and performance improvement; localization of the server which manages the referenced file.

  mapping of textual file names to file references (server name and file identifier).

- **Replication service**

  file replication for shorter response times and increased availability.

  handles data consistency and the multiple copy update problem.

- **Transaction service**

  provides a mechanism for grouping of elementary operations so as to execute them atomically;

  mechanisms for concurrency control;

  Mechanisms for reboot after errors;

- **File service**

  relates file identifiers to particular files;

  performs read and write operations on the file content and file attributes.

- **Block service**

  accesses and allocates disk blocks for the file.

7.4 Update of replicated files

Basically, there are two types of approaches for multiple update control: the optimistic and the pessimistic approach.

7.4.1 Optimistic concurrency control

Data consistency is not guaranteed.

The concurrency control scheme does not constrain the activities of the user; it allows access to inconsistent data.

- Example: *Coda* file system of Carnegie-Mellon University.
- The Available Copy scheme

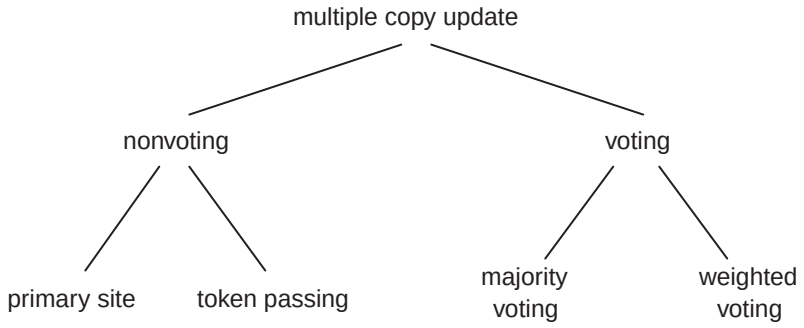
  read access to the local or to best-available file copy.

  in case of write access, all available file copies are updated.

7.4.2 Pessimistic concurrency control

Pessimistic concurrency control is particularly important for data-critical applications, e.g. banking applications. Access is always carried out to consistent data.

- **Classification of pessimistic concurrency control**



- **Primary site**

  A well-defined file copy, the primary site, serializes and synchronizes all (write) operations.

- **Token passing**

  Rather than making a specific site responsible for a file, this scheme introduces a token traveling the network between sites managing a replica. There is one token for each replicated file.

- **Voting schemes**

  The result of the negotiation between all file replicas determines whether a file access is granted or not.

  global consent is necessary, but control is decentralized.

  in case of consent, the relevant file block is locked.

  Examples: Majority consensus, weighted voting

7.4.3 Voting schemes

Voting schemes provide pessimistic concurrency control.

Introduction

Voting schemes are algorithms for maintaining mutual consistency of replicates even in situations of computer crashes and network partitionings.

- Let us assume, there exist REP replicas of file d.

- Let sg(r) be the weight of the vote of computer r; K be the set of all computers considered.

  Let the sum of all weights be SUM = $\sum_{r \in K}$ sg(r).

Definition

Definition: The **votum** for a desired access of a file is defined

as the sum of votes from the set of computers that have voted for the desired access.

Definition: The obtained votum is called successful if the sum of votes from the set of computers that have voted for the desired access is equal to or greater than a lower bound, the so-called **quorum**.

- File access is permitted (positive votum), if the following holds

  for read access: at least R positive votes (read quorum).

  for write access: at least W positive votes (write quorum).

Multiple-reader-single-writer strategy

The choice of the quorum must support the multiple-reader-single-writer strategy and must guarantee that among the computers that have allowed the access there is at least one computer with the most up-to-date physical replica of the file.

- Multiple-reader-single-writer strategy maintains file consistency:

  R + W $>$ SUM, i.e. a reader excludes a writer, and vice versa.

  W + W $>$ SUM, i.e. only one writer gets a positive vote.

Voting scheme variants

For further variants and details see the book Borghoff/Schlichter, Springer-Verlag, 2000.

- **Write-All-Read-Any**

  Write access to all copies; read access to any copy of the file.

  – The scheme can be considered an extreme case of a voting scheme

    write quorum: W = n, with n the number of computers.

    read quorum: R = 1

- **Majority consensus**

  In the majority consensus scheme, a votum is successful if at least a majority of computers with a right to vote have voted for the desired access.

  – the vote of each computer with a replicate has the same weight ("one person one vote"), i.e. $\forall$ r $\in$ K: sg(r) = 1.
  – A votum is successful if the majority of all relevant computers agree with respect to the desired access:

    W = R = REP/2 + 1, if REP is even.

    W = R = (REP+1)/2 + 1, if REP is odd.

- **Weighted voting**

  Each computer possessing a file copy receives a certain number of votes, i.e. $\forall$ r $\in$ K: sg(r) $\in$ {0, 1, 2, ...}.

  – We get: SUM = $\sum_{r \in K}$ sg(r)
  – Example: same read and write quorum

    W = R = SUM/2 + 1, if SUM even.

    W = R = (SUM+1)/2 + 1, if SUM odd.

- **Read algorithm for weighted voting**

```
function read (r: computer, k: index, B: block):
errorcode;
    /* computer r wants to read the block k; the read block is
    stored in the data structure B*/
    Let Q be the set of all reachable computers;
    if Σ_{i∈Q} sg(i) > R then
        Let s be the computer of Q with the current version of
        the block and which is easiest to reach;
        if version_r(k) < version_s(k) then
            request block k of s;
            update block in r;
            version_r(k) := version_s(k);
        read updated block;
        return (success);
    elsereturn (failure);
```

- **Write algorithm for weighted voting**

```
function write (r: computer, k: index, B: block):
errorcode;
    /* computer r desires to write block B to the file at
    index k; it is assumed there is first a read and the a
    write operation, i.e. the current version is known after
    reading.*/
    Let Q be the set of all reachable computers;
    if ∑_{i∈Q} sg(i) > W then
        version_r(k) := (max_{s∈Q} version_r(k) ) + 1;
        transmit block B with version_r(k) to all s ∈ Q;
        update block locally;
        return (success);
    elsereturn (failure);
```

7.5 Coda file system

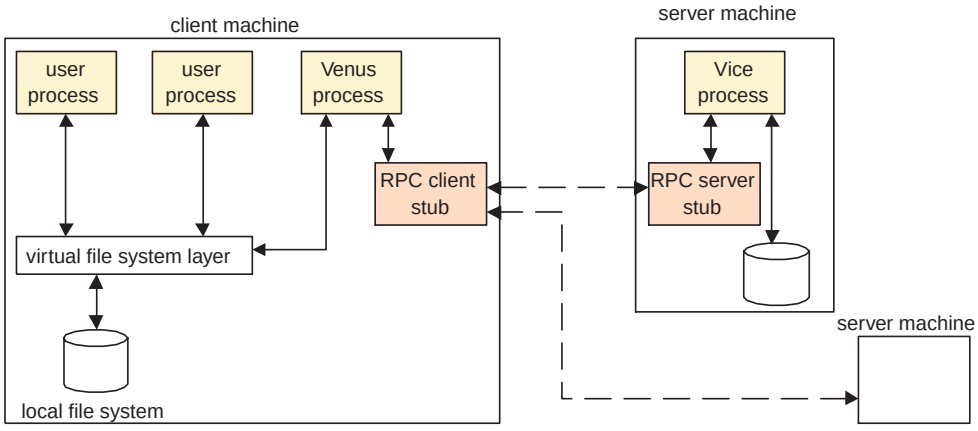
Coda was designed to be a scalable, secure, and highly available distributed file service.

supporting the mobile use of computers.

files are organized in volumes.

Coda relies on the replication of volumes.

7.5.1 Architecture



Venus processes provide access to files maintained by the Vice file servers.

role is similar to that of an NFS client.

responsible for allowing the client to continue operation even if access to the file servers is (temporarily) impossible.

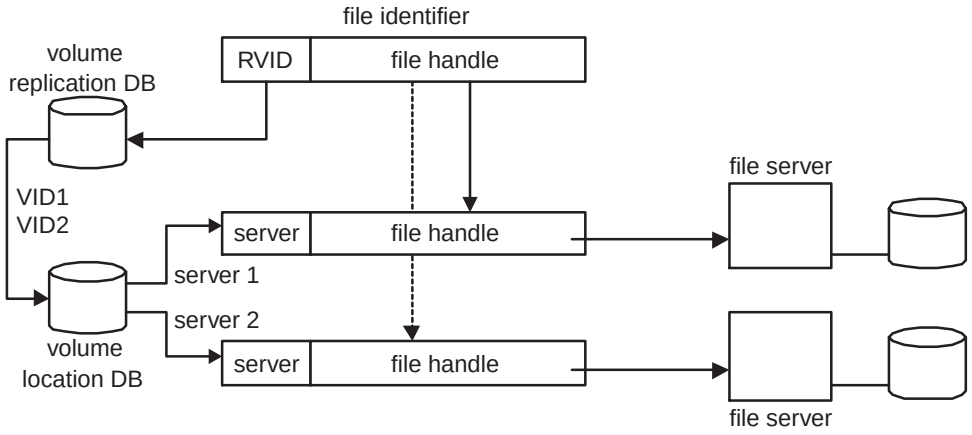
7.5.2 Naming

Each file is contained in exactly one volume. Distinction between

physical volumes.
logical volume (represents all replicas of a volume).

- RVID (Replicated Volume Identifier): identifier of a logical volume.
- VID (Volume Identifier): identifier of a physical volume.
- **File identifier**

  Coda assigns each file a 96-bit file identifier.



7.5.3 Replication strategy

Coda relies on replication to achieve high availability. It distinguishes between two types of replication.

Client caching

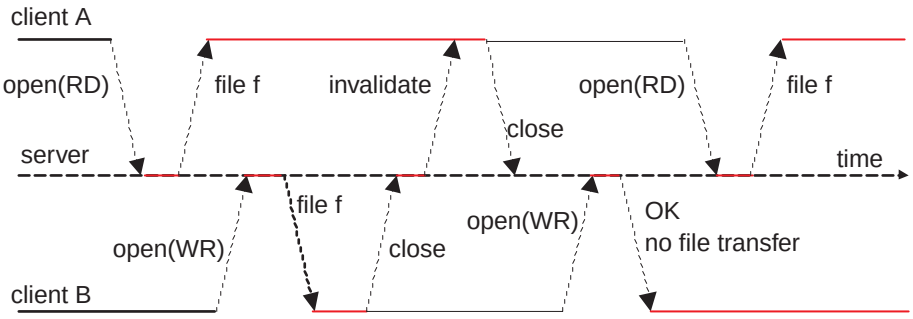
When a file is opened, an entire copy of the file is transferred to the client; caching of the file.

client becomes less dependent on the availability of the server.

- Cache coherence is maintained by means of callbacks.

  Server records a *callback promise* for a client.

  update of the file by a client ⇒ notification to the server ⇒ invalidation message to other clients.



Server replication

Coda allows file server to be replicated; the unit of replication is a volume.

Volume Storage Group (VSG): collection of servers that have a copy of a volume.

client's Accessible Volume Storage Group (AVSG): list of those servers in the volume's VSG that the client can contact.

AVSG = {}: client is disconnected.

Coda uses a variant of the "read-one, write-all" update protocol.

- **Coda version vector**

  Coda uses an optimistic strategy for file replication. For each file version there exists a Coda version vector (CVV).

  CVV is a → vector timestamp (see page 105) with one element for each server in the relevant VSG.

  CVV is initialized to [1, ..., 1].

  On file close the Venus process of the client broadcasts an update message to all servers in AVSG ⇒ all servers of AVSG update the relevant CVV entries.

– Let v1 and v2 are CVVs for two versions of a file f.

when neither v1 $\leq$ v2 nor v2 $\leq$ v1 $\Rightarrow$ there is a conflict between the two file versions.

7.5.4 Disconnected operation

In the disconnected situation a client will simply resort to its local copy in the cache.

AVSG = {} for the volume.

Venus supports a priority list of files which should be cached locally.

Venus supports *hoarding*.

- **Reintegration**

When disconnected operation ends, a process of reintegration starts.

for each cached file that has been modified, Venus sends update operations to all servers in AVSGs.

Chapter 8

Object-oriented distributed systems

Issues of the section

object migration in distributed systems.

what is the Linda tuple space?

what is Java RMI ("Remote Method Invocation")?

what are the basic concepts of Corba?

8.1 Introduction

Motivation:

handle increased software complexity,

reuse software modules.

8.1.1 Motivation

The next generation of client/server applications will increasingly be designed and implemented on the basis of distributed objects.

- Monolithic applications will be replaced by a set of autonomous components

  which interact with each other, and

  which migrate within the distributed system.

- Combination of objects to an *autonomous application component* which is created and maintained as a single piece of software; support of standardized interfaces to interact with other components.

$\Rightarrow$ *Componentware*

8.1.2 System approaches

- **DCOM**

  DCOM (Distributed Component Object Model) by Microsoft

  DCOM is an extension of COM; COM supports Compound documents.

  DCOM primarily offers access transparency.

  a DCOM component realizes a set of interfaces, each of which has a unique identifier (128 bit).

  DCOM objects are transient, i.e. as soon as an object has no more clients that reference it, the object is destroyed.

- RMI and JavaBeans by SUN

  Java classes which must satisfy certain conventions; internal data of components are accessed through the set/get methods; events have well-defined names.

- Distributed object management $\rightarrow$ Corba (see page 164)

- ObjectSpace's Voyager

  A collection of Java classes and interfaces providing a complete communication infrastructure; it supports the invocation of remote Java objects, object mobility, and the extension of objects to agents (see Voyager (URL: http://www.objectspace.com/voyager) Web-Site).

8.1.3 Objects as distribution entities

Basic object types

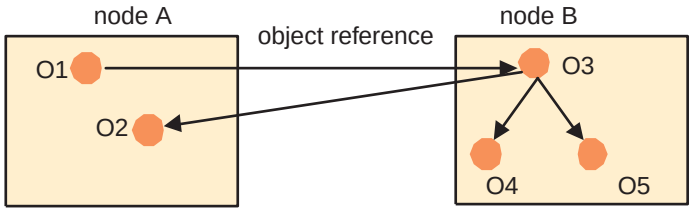
Distinction between

passive objects.

active objects.

Objects are elementary distribution entities

Objects reside on logical nodes of the distributed system (shared memory).



Distributed objects

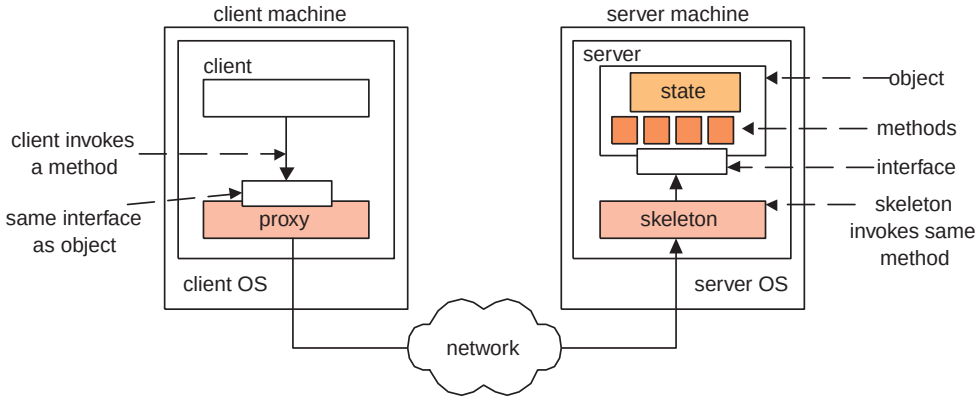
An application that allows for distributed objects has some interesting properties.

referencing and localization of remote objects, e.g. using an object directory.

the execution of methods is independent of the object distribution.

object migration between logical nodes, i.e. dynamic object migration.

- **Common organization**



Main advantage of the object-oriented approach: uniform communication among objects.

– local and remote method calls have the same semantics.
– allocation flexibility of objects; achieved through object migration.
– access of object data via method calls.
– **Example**
Emerald (Uni Washington): highly integrated object-oriented language concept which incorporates distribution aspects.

8.2 Object mobility

Object mobility refers to the migration of objects from one logical node of the distributed system to another.

8.2.1 Migration

Aspects of object migration

- In object migration, the suitable object class, too, must be available at the destination in order to execute the obejct methods.

  i.e. replication of object classes at all locations.

- time of migration

  static allocation: at initialization of the distributed application.

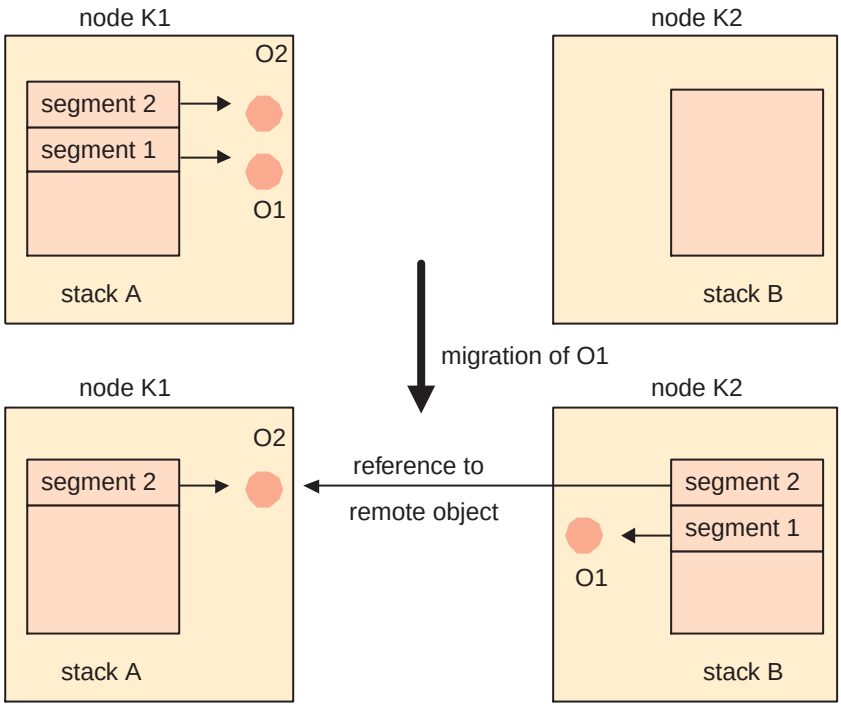
  dynamic allocation: during the execution phase of the distributed application.

- Execution state of the migrating object.

  passive object.

  active object.

8.2.2 Migrating an active object



8.2.3 Object localization

Approaches to localize an object in the distributed system.

1. using the object identifier.
2. querying an object server.
3. cache/broadcast of object locations.
4. forward addressing

   forwarding address : (host address, timestamp).
5. immediate update.

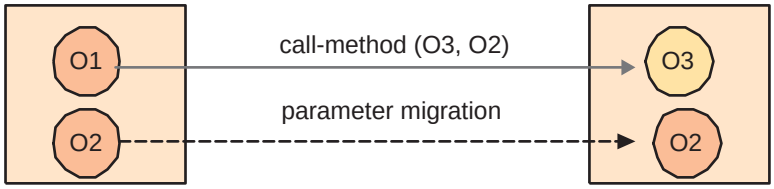
8.2.4 Object vs. process migration

	object	*process*
Layer	application layer	operating system layer
Node	logical node	physical node
Granularity	object and segments of active methods, i.e., fine-granular	process and data of the whole address space, i.e., coarse-granular
Motivation	joining together of communicating objects	load balancing

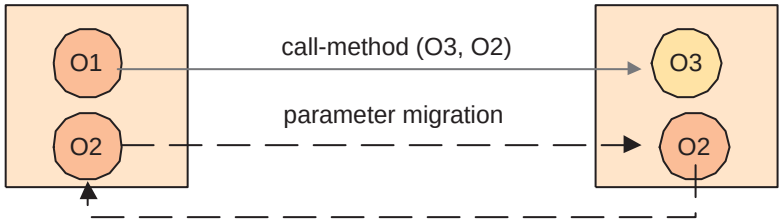
8.2.5 Object parameters in remote method call

Communicating objects may pass parameters not only as values but also as references to other objects.

- **call-by-move**

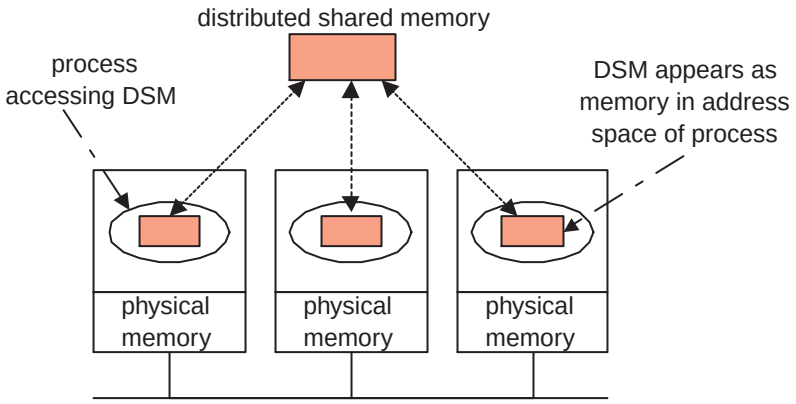


- **call-by-visit**



8.3 Distributed shared memory

Distributed shared memory (DSM) is an abstraction used for sharing data between computers that do not share physical memory.



8.3.1 Programming model

Message passing model

variables have to be marshalled from one process, transmitted and unmarshalled into other variables at the receiving process.

Distributed shared memory

the involved processes access the shared variables directly; no marshalling necessary.

processes may communicate via DSM even if they have non-overlapping lifetimes.

- **Implementation approaches**

  in hardware

  shared memory multiprocessor architectures, e.g. NUMA architecture.

  in middleware

  language support such as Linda tuple space or JavaSpaces.

8.3.2 Consistency model

The contents of the shared memory of DSM may be replicated by caching it at the separate computers;

data is read from the local replica.

updates have to be propagated to the other replicas of the shared memory.

Approaches to keep the replicas consistent

- **Write-update**

  updates are made locally and multicast to all replicas possessing a copy of the data item.

  the remote data items are modified immediately.

- **Write-invalidate**

  before an update takes place, a multicast message is sent to all copies to invalidate them;

  acknowledgement by the remote sites before the write can take place.

  other processes are prevented to access the blocked data item.

  the update is propagated to all copies, and the blocking is removed.

8.3.3 Tuple space

The so-called tuple space was invented by Gelernter (Yale University) as an object-oriented approach to managing distributed data. It was specially designed for Linda language.

Tuple space consists of a set of tuples that could be interpreted as lists of typed fields.

A tuple space has the following basic characteristics:

it is based on the shared-memory model.

tuples represent information, e.g. ("Linda", 3).

Atomic operations

Tuple space supports read and write operations on the shared memory.

1. Operations on a tuple t

   out(t): creates a new tuple t in the tuple space.

   in(t): reads and simultaneously removes a tuple from the tuple space.

   read(t): reads a tuple; t remains in the tuple space and subsequent operations can refer to it.

2. Read access is associative, e.g. *in*("order", ?i, ?j).

3. *in*, *read* are synchronous.
4. *inp*, *readp* are asynchronous.
5. Generation of new processes: *eval*(t).

Tuple space implementation

Implementation alternatives

1. central tuple space.
2. replicated tuple space,

   each computer maintains a complete copy of the tuple space.
3. distributed tuple space; division into subspaces

   each computer owns part of the tuple space; *out* operations are executed locally.

Example for client-server communication

The following simple example shows how the client-server style of communication could be programmed in the tuple space model.

- **/* Client */**

```
begin
    int client-id = unique identifier;
    listofunspecified parameterlist;
    listofunspecified resultlist;
    ..................
    collect arguments for server call;
    out ("request", client-id, parameterlist);
    in ("reply", client-id, ?resultlist);
    process results
end
```

- **/* Server */**

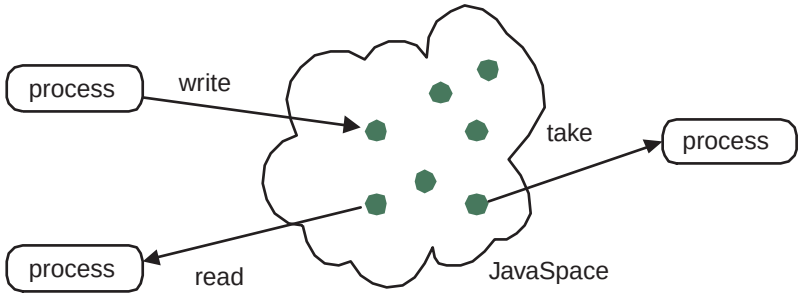
```
begin
    int client-id;
    listofunspecified parameterlist;
    listofunspecified resultlist;
    ..................
    while (true) do
        in ("request", ?client-id, ?parameterlist);
        compute result;
        out ("reply", client-id, resultlist);
end
```

8.4 JavaSpaces

JavaSpaces provide an environment to implement distributed applications based on the Linda tuple concept. Tuples are references to Java objects; see also JavaSpaces (URL: http://java.sun.com/products/javaspaces/) Web site.

8.4.1 Introduction

A *space* is a shared, network-accessible object repository. Processes use the repository as a persistent object storage and exchange mechanism.



JavaSpaces uses the Jini technology.

8.4.2 Features of JavaSpaces

The JavaSpaces programming interface is simple; a space provides the following key features.

- Objects in a space are passive.

processes do not manipulate objects directly in the space.

processes do not invoke methods of objects in the space.

- Spaces are shared: they represent a network-accessible memory that many remote processes can interact with concurrently.
- Spaces are persistent: objects are stored until a process explicitly removes them or until their *lease* time expires.
- Spaces are associative: objects are accessed via associative lookup, rather than by identifier or by memory address.
- Spaces are transaction oriented: access operations to the space are atomic.
- Spaces support the exchange of executable code.

8.4.3 Data structures

Entry interface

Objects in a space are realized via the `Entry` interface (net.jini.core.entry package).

- **Interface Definition**

```
public interface Entry extends java.io.Serializable {
    // this interface is empty
}
```

- Example of an object representing a shared variable in the distributed system

```
public class SharedVar implements Entry {
    public String name;
    public Integer value;
    public SharedVar() {
    }
    public SharedVar(String name, int value) {
        this.name = name;
        this.value = new Integer(value);
    }
}
```

- Instantiation of a shared variable within a process

```
SharedVar global_counter = new SharedVar("counter", 0)
```

SpaceAccessor

The shared space is identified via the method getSpace of the SpaceAccessor class.

```
JavaSpace space = SpaceAccessor.getSpace();
```

Access to the space identifier; there are two options

the space is registered as Jini service, i.e. Jini lookup services may be used.

the space is registered in the → RMI (see page 156) registry

8.4.4 Basic operations

Overview

For manipulation of space entries, there are the following basic operations available

read

take, i.e. read and remove

write

notify, i.e. inform the process when an entry matching the given pattern has arrived.

Write - operation

```
Lease write (Entry e, Transaction txn, long lease) throws
RemoteException, TransactionException
```

- Parameter semantics

  Entry e is entered into the space; e is transmitted, as well as stored, in a serialized form in the space.

  Transaction txn allows to group several operations to a transaction; the parameter value *null* represents a transaction with only one operation.

  long lease specifies how long the entry e is to be stored in the space before the space automatically removes the entry e.

  The result Lease specifies how long the space will store the entry e.

- **Example**

  ```
  space.write(global_counter, null, Lease.FOREVER);
  ```

Write can trigger the exceptions RemoteException (communication problems) and TransactionException (transaction txn not valid).

Read and take - operation

The methods read and take access an object in a space. read copies the object into the local process environment while take removes it from the space.

- For remote access, a process needs a template. A template is a kind of entry:

  containing some specified and some empty fields (i.e. the value null).

  matching associatively the relevant objects in the space.

- If several objects in the space match the template, then an object is selected at random.

- **Example**

```
SharedVar template = new SharedVar("counter");
SharedVar result = (SharedVar) space.take(template, null,
Long.MAX_VALUE)
```

- The take operation waits until there is a suitable entry in the space available.

Matching rules

An access template matches an object in the space if the following rules hold true

- the template class matches the object class, or else the template class is a super class of the entry's class.
- if a template field has a wildcard (null), then it matches the corresponding object field.
- if a template field is specified, then it matches the object's corresponding field if the two values are the same.

Atomicity

All basic operations are atomic.

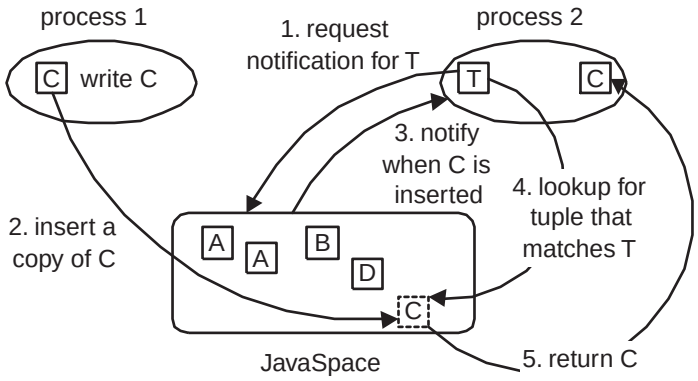
- The following code segment defines an atomic access of a shared global variable in the space

```
SharedVar template = new SharedVar("counter");
SharedVar result = (SharedVar) space.take(template, null,
Long.MAX_VALUE);
result.value = new Integer(result.value.intValue() + 5);
space.write(result, null, Lease.FOREVER);
```

- Thus, there are no race conditions between concurrent processes for the shared variable.

8.4.5 Events

A client can request to be notified whe a specific tuple instance is written to the JavaSpace.



8.5 Remote Method Invocation (RMI)

RMI supports communication among objects residing on different Java virtual machines (JVM). RMI (URL: http://java.sun.com/j2se/1.3/docs/guide/rmi/) is an → RPC (see page 59) in the object-oriented Java environment.

8.5.1 Definitions

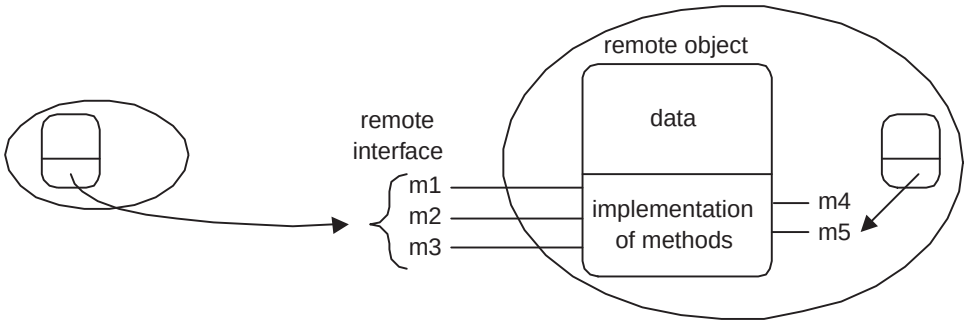
- **Definition: Remote object** is an object whose method can be called by an object residing on another Java Virtual Machine (JVM), even on another computer.

- **Definition: Remote interface**

  is a Java interface specifying the methods of a remote object.

- **Definition: Remote method invocation** (RMI) allows object-to-object communication between different Java Virtual Machines (JVM), i.e. it is the action of invoking a method of a remote interface on a remote object.

  The method calls for local and remote objects have the same syntax.

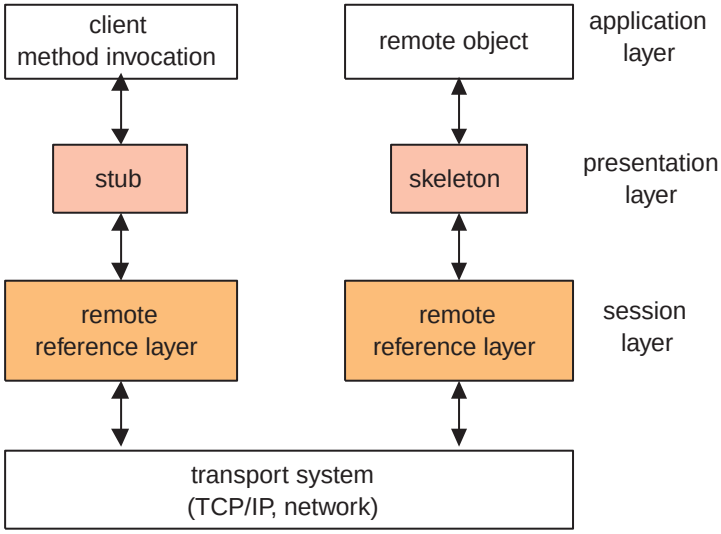


8.5.2 RMI characteristics

Note in RMI: client and server are objects.

- RMI supports location and access transparency.
- Localization of remote objects.
- Communication with remote objects (using method calls).
- Automated class loading for objects passed as parameters or results.
- Clients interact with remote interfaces, rather than with classes implementing these interfaces.

8.5.3 RMI architecture



Stub/Skeleton layer

Layer intercepts method calls by the client and redirects these calls to the remote object.

Object serialization/deserialization; hidden from the application.

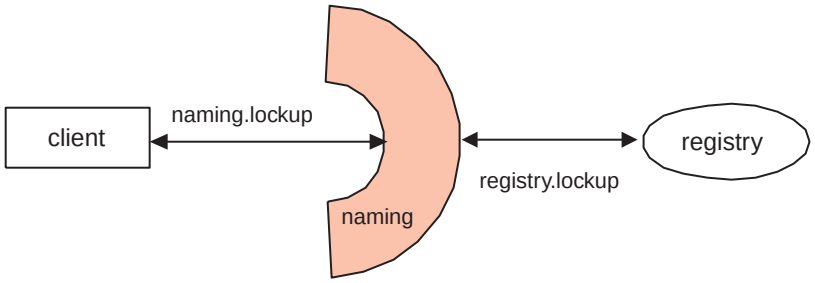
Remote Reference layer

Connects client and remote objects exported by the server environment by a 1-to-1 connection link.

- The layer provides JRMP (Java Remote Method Protocol) via TCP/IP.
- Mapping of stub/skeleton operations to the transport protocol of the host; it interfaces the application code with the network communication.
- The layer supports the method `invoke`.

```
Object invoke (Remote obj, java.lang.reflect.Method
method, Object [ ] params, long opnum) throws Exception
```

8.5.4 Locating remote objects



- RMI supports a special name service, the *RMI registry*

  mapping of names to remote objects.

  stand-alone Java application.

  the RMI registry runs on all those machines hosting remote objects.

  standard port for registry requests is 1099.

  the RMI registry is itself a remote object.

- access of the RMI registry via the java.rmi.Naming class.
- **Naming interface methods**
  - *publicstaticvoid* `bind (String name, Remote obj)`

    Throws AlreadyBoundException, java.net.MalformedURLException, RemoteException.

    associates the remote object obj with name (in URL format).

    example for name: rmi: //host[:service-port]/service-name

    if name is already bound to an object, then AlreadyBoundException is triggered.
  - *publicstaticvoid* `rebind (String name, Remote obj)`

    Throws java.net.MalformatURLException, RemoteException.

    associates always the remote object obj with name (in URL format).
  - *publicstatic* `Remote lookup (String name)`

    Throws NotBoundException, java.out.MalformedURLException, RemoteException.

    returns as a result a reference (a stub) to the remote object.

    if name is not bound to an object, then NotBoundException is triggered.

- *publicstaticvoid* `unbind (String name)`

  Throws NotBoundException, RemoteException.

- *publicstatic* `String [ ] list (string name)`

  Throws java.net.MalformedURLException, RemoteException.

  as a result, it returns all names entered in the registry.

  the name parameter specifies only the host and port information.

- **Registry-Lookup**

  The client invokes a lookup for a particular URL, the name of the service (rmi://host:port/service). The following describes the steps:

  1) a socket connection is opened with the host on the specified port.

  2) a stub to the remote registry is returned.

  3) the method Registry.lookup() on this stub is performed. The method returns a stub for the remote object.

  4) the client interacts with the remote object through its stub.

8.5.5 Developing RMI applications

Defining a remote interface

A remote interface is the set of methods that can be invoked remotely by a client.

- The remote interface must be declared public.
- The remote interface must extend the `java.rmi.Remote` interface.
- Each method must throw the `java.rmi.RemoteException` exception.
- If the remote methods have any remote objects as parameters or return types, they must be interfaces rather than implementation classes.
- **Example: remote inteface definition**

```
publicinterface HelloInterface extends java.rmi.Remote {
    /* this method is called by remote clients and it is
    implemented by the remote object */
    public String sayHello ( ) throws java.rmi.RemoteException
}
```

Implementing the remote interface

Definition of an implementation class that defines the methods of the remote interface;

the abstract class `java.rmi.server.RemoteServer` provides the basic semantics to support remote references.

`java.rmi.server.RemoteServer` has subclasses

`java.rmi.server.UnicastRemoteObject`: defines a non-replicated remote object whose references are valid only while the server process is alive.

`java.rmi.activation.Activatable`: defines a remote object which can be instantiated on demand (if it has not been started already).

- **Example: Remote interface implementation**

```
import java.io.*;
import java.rmi.* ;
import java.rmi.server.* ;
import java.util.Date.* ;
public class HelloServer extends UnicastRemoteObject
implements HelloInterface{
    public HelloServer( ) throws RemoteException {
        super( );
        /* call superclass constructor to export this object */
    }
    public String sayHello( ) throws RemoteException {
        return "Hello World, the current system time is " + new
        Date( );
    }
}
```

Client implementation

This step encompasses the writing of the client that uses remote objects.

The client must incorporate a registry lookup in order to obtain a reference to the remote object.

The client interacts with the remote interface, *never* with the object implementation.

- **Example**

```
import java.rmi.*;
public class HelloClient {
    public static void main (String args [ ]) {
        if (System.getSecurityManager( ) == null)
        System.setSecurityManager (new RMISecurityManager( ) );
        try { String name = "//" + args [0] + "/HelloServer";

            HelloInterface obj = (HelloInterface)
            Naming.lookup (name);
            String message = obj.sayHello( );
            System.out.println(message);


        } catch (Exception e) {
            System.out.println("HelloClient exception:  " + e);
        }
    }
}
```

Generating stubs and skeletons

The tool rmic generates stub and skeleton from the implemented class.

```
rmic HelloServer.
```

Remote object registration

Every remotely accessible object must be registered in a registry in order to make it available;

stubs are needed for registration.

the registry is started at the host of the remote object.

- **Example for object registration**

```
import java.rmi.* ;
public class RegisterIt {
```

```
    public static void main (String args [0])
        try { // Instantiate the object
            HelloServer obj = new HelloServer( );
            System.out.println ("Object instantiated:  " + obj);
            Naming.rebind("/HelloServer", obj);
            System.out.println("HelloServer bound in registry");
        } catch (Exception e) {
            System.out.println(e)
        }
    }
}
```

8.5.6 Parameter Passing in RMI

Parameters with primitive data types are passed with their values between JVMs; for object parameters, a distinction is made between local and remote:

1. local object parameter

   RMI passes the object itself, rather than the object reference.

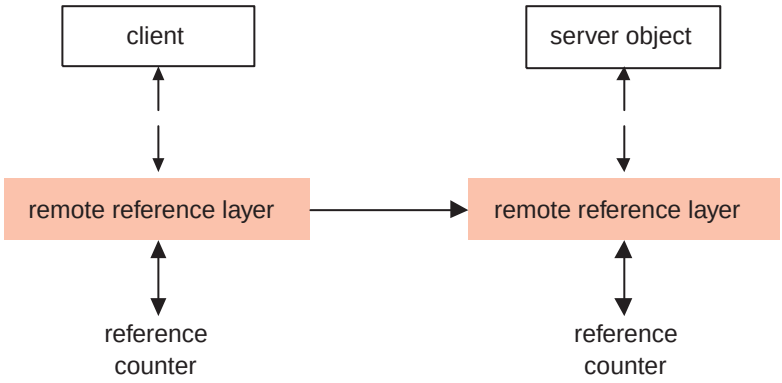
   The transmitted object must implement the interface `java.ia.Serializable` or `java.io.Externalizable`.

2. remote object parameter

   RMI transmits the stub of the remote object; the stub is a reference to the remote object.

8.5.7 Distributed garbage collection

Utilization of life references for each JVM; reference counter represents the number of live references.



- The first client access creates a referenced message sent to the server.
- If there is no valid client reference, then an unreferenced message is sent to the server.
- Time limit of references ("lease time", e.g. 10 minutes); the connection to the server must be renewed by the client, otherwise the reference becomes invalid.

8.6 Distributed object management - Corba

There are a number of object-oriented systems, some of which have been ported to a distributed environment, whereas others have been designed especially for distributed environments, e.g. Emerald, Argus and Linda. All these systems, however, have in common that they are usually targeted for homogeneous environments. The OMG (Object Management Group) was founded in 1989 by a number of companies to encourage the adoption of *distributed object systems* and to enable *interoperability* for heterogeneous environments (hardware, networks, operating systems and programming languages).

8.6.1 Introduction

OMG intends to enhance object-oriented technology in order to support cooperation between applications in heterogeneous, distributed environments.

1. interoperability among heterogeneous software components and reusability of portable software

integration of independently developed applications by the user and information exchange between these applications (using object interaction).

isolation of common functionalities, e.g. storing and managing of objects.

embedding of legacy applications into OO environments.

2. Technical goals of OMG

   transparency with respect to object distribution.

   good performance both for local and remote server object calls.

   dynamic object evolution.

   access through object names, object attributes, or relations between objects.

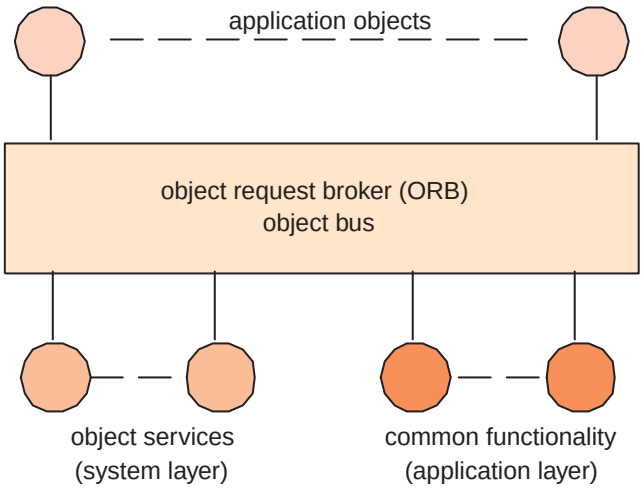
   concurrency control for objects; controls concurrent access of several client objects.

3. OMG defines a common architectural model for heterogeneous environments, namely OMA ("Object Management Architecture"); see also OMG (URL: http://www.omg.org/) Web Site.

8.6.2 Objekt Management Architecture - OMA

The architecture is also referred to as CORBA ("Common Object Request Broker Architecture").

OMA is the "middleware" for future object-oriented distributed applications.



- ORB supports the communication among the objects through a request/reply protocol.

  ORB includes object localization, message delivery, method binding, parameter marshalling, and synchronization of request and reply messages.

  ORB itself does not execute methods. Rather, it mediates between application objects, service objects, and shared functionalities of the application layer ("application frameworks").

8.6.3 Object Request Brokers ORB

The ORB connects distributed objects dynamically at runtime and supports the invocation of distributed service objects.

General features

ORB supports the following general characteristics

1. static and dynamic invocation of object methods

   static: method interface is determined at compilation time.

   dynamic: method interface is determined at runtime.

2. interfaces for higher programming language, e.g. C++, Smalltalk, Java.

3. a self-descriptive system.

4. location transparency.

5. security checks, e.g. object authentication.

6. polymorphic method invocation, i.e. the execution of the method depends on the specific object instance. Difference between RPC and ORB

   RPC calls a specific server function; data are separated.

   ORB calls the method of a specific object.

7. hierarchical object naming.

Interface definition language

The interfaces of methods are specified by Corba IDL; it is programming language independent.

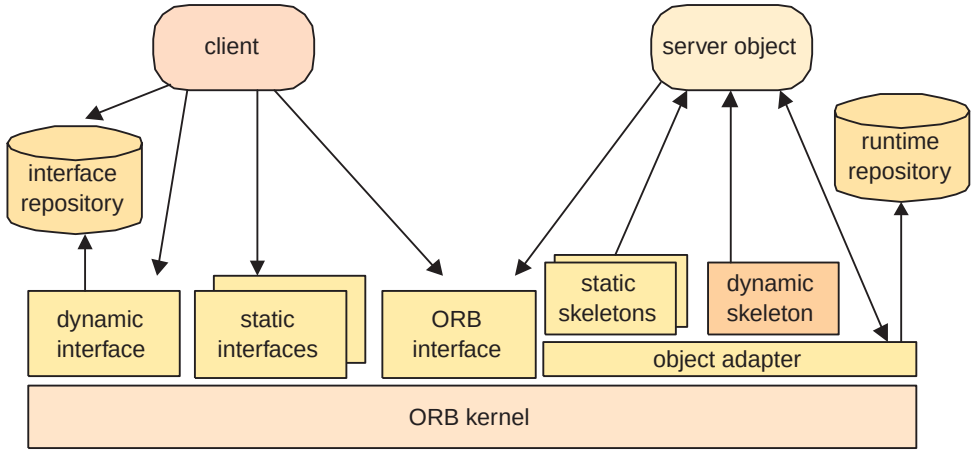
declarative specification.

similar to RPC interface descriptions; IDL supports an inheritance hierarchy; it is similar to ANSI C++.

- **Example for an interface description**

```
module <identifier> { /* defines a naming context */
    <type declarations>
    <constant declarations>
    <exception declarations>
    interface <identifier> [:<inheritance>] { /* defines a
    Corba class */
        <type declarations>
        <constant declarations>
        <exception declarations>
        [op_type] <identifier>
            (<parameters>) [raises exception][context] /*
            defines a method */
        /*context describes a client context by a set of
        attributes */
        ..........................
    }
    interface <identifier> [:<inheritance>]
    ..........................
}
```

Object Request Broker ORB



- **ORB components**
  - **ORB core (kernel)**
    mediates requests between client and server objects; handles the network communication within the distributed system.
    operations to convert between remote object references and strings.
    operations to provide argument lists for requests using dynamic invocation.
  - → Static invocation interface (see page 169)
    at compile time, operations and parameters are determined.
    an object class may have several different static interfaces.
  - → Dynamic invocation interface (see page 170)
    Procedures and parameters are determined at runtime; the interface is identical for all ORB implementations, i.e. there is *only one* dynamic invocation interface.
  - ORB interface
    supports ORB service calls, e.g. conversion of object references to strings and vice versa; the interface is determined by the ORB.
  - Interface repository
    stores at runtime the signatures of the available methods; the signatures are described by the IDL notation; in case of the dynamic invocation interface a lookup within the interface repository is performed.
  - **Object adapter**

bridges the gap between Corba objects with IDL interfaces and the programming language interfaces of the server class.

forwards client calls to the appropriate server object using the skeletons.

defines a runtime environment for initialization of server objects and assignment of object identifiers.

activates objects.

– Runtime repository

stores information about the object classes supported by the server, as well as the already instantiated objects and their identifiers.

– **Skeletons**

Skeleton classes are generated in the language of the server by the IDL compiler:

contains stubs for server object calls.

static: stubs for the static interfaces; they are generated along with the static stubs on the client side by the IDL compiler; there may be several static skeletons.

dynamic: provides a runtime binding mechanism for servers that need to handle incoming method calls for objects that do not have a static skeleton (generated by the IDL compiler).

Static invocation interface

The following steps are required for generating static invocation interfaces:

- Step 1: define the object class using IDL.
- Step 2: run the IDL file through the Corba compliant language precompiler to generate the skeletons for the implementation server classes.
- Step 3: server code implementation, i.e. implementation of the methods described by the skeletons; the generated skeleton is extended.
- Step 4: compilation of the entire code of step 3; compilation generates at least 4 output files:

static invocation interfaces for the client (client stub).

static invocation interfaces for the server (server skeletons).

the object code implementing the server object classes.

interface descriptions for the interface repository.

- Step 5: instantiate the server objects using the object adapter.
- Step 6: register the instantiated objects with the runtime repository.

Dynamic invocation interface

Steps during a dynamic method call on the client's side:

- Step 1: obtain the method description by extracting the respective information from the interface repository;

  CORBA supplies two functions: lookup_name(), describe().
- Step 2: create parameter list for the method call.
- Step 3: create method call (request) using the function:

  create_request(object-reference, method, argument-list)
- Step 4: invoke the request through

  RPC: invoke().

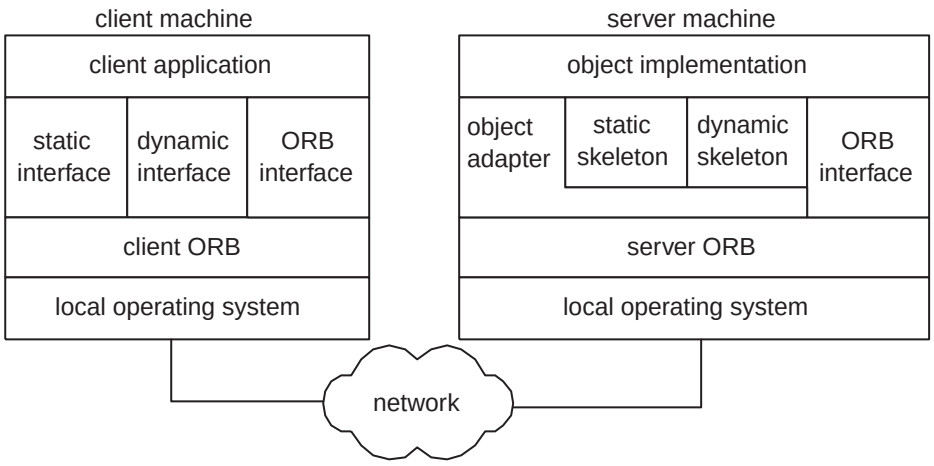
  send/receive: send() und get_response().

  Datagram (unidirectional): send().

Variants of the ORB implementation

Among others, the following choices for the ORB implementations are supported:

1. ORB is a resident component of the client and the object implementations.
2. ORB is a server.
3. ORB is a service of the (distributed) operating system.
4. ORB is a library function.



Commercially available ORBs

There are several commercially available ORBs, for example

- ORBIX by Iona (URL: http://www.iona.com/) Technologies

  available as a libraries: client and server library.

  based on TCP/IP transport mechanism; supports sockets and Sun RPC.

- DSOM (Distributed System Object Model) by IBM

  based on TCP/IP, NetBIOS transport mechanism; finely granular objects run within one process.

  serves as the basis of OpenDoc.

- DOE (Distributed Objects Everywhere) by Sun

  provides a toolkit based on TCP/IP.

8.6.4 Common object services

A collection of system level services which can be utilized by the application objects; they are extending the ORB functionality.

Some CORBA object services

Among others, the CORBA standard supports the following services

- *Life-cycle Service*: defines operations for object creation, copying, migration and deletion.
- *Persistence Service*: provides an interface for persistent object storage, e.g. in relational or object-oriented databases.
- *Name Service*: allows objects on the object bus to locate other objects by name; integrates existing network directory services, e.g. OSF's DCE, → LDAP (see page 52) or X.500.
- *Event Service*: register the interest in specific events; producer and consumer of events need not know each other.
- *Concurrency Control Service*: provides a lock manager.
- *Transaction Service*: supports 2-phase commit coordination for flat and nested transactions.
- *Relationship Service*: supports the dynamic creation of relations between objects that know nothing of each other; the service supports navigation along these links, as well as mechanisms for enforcing referential integrity constraints.
- *Query Service*: supports SQL operations for objects.

Example for a Query Service

The query service supports the search for objects whose attributes meet the search criteria. This service has been specified and approved by the important object-oriented database producers, as well as IBM, SunSoft and Sybase in 1995. Queries can be specified in the following languages: OQL (object query language) and SQL with object extensions. The queries provide no direct access to the object internals, i.e. encapsulation is maintained. Internal data structures remain hidden and only the externally visible data structures may be used for the search criteria.

- **Interfaces**

  Query Service consists of the following interfaces

- QueryEvaluator: performs the query, e.g search in a database;
  Method: evaluate
- Query: represents a new query;
  Methods: prepare, execute, get_status, get_result
- QueryManager: used to create query objects.
- Collection: object collection
  Methods: add_element, add_all_elements, insert_element_at, replace_element_at, retrieve_element_at, remove_element_at, create_iterator
- Iterator: supports navigation in a collection
  Methods: next, reset, more
- QueryableCollection: inherits the functions from the interfaces Collection and QueryEvaluator.

- **Query scenario**

  A simple query scenario:

- Step 1: Client $\rightarrow$ QueryManager (create); results in the creation of a query object.
- Step 2: Client $\rightarrow$ Query object (prepare); precompiles and stores the query for later execution.
- Step 3: Client $\rightarrow$ Query object (execute): creates a QueryableCollection object in which the query results are stored; the precompiled query is executed.
- Step 4: Client $\rightarrow$ Query object (get_result): returns the object reference to the QueryableCollection object.
- Step 5: Client $\rightarrow$ QueryableCollection object (create_iterator): creates an iterator object for QueryableCollection.
- Step 6: Client $\rightarrow$ QueryableCollection object (retrieve_element_at): retrieve the element pointed to by the iterator.
- Step 7: Client $\rightarrow$ Iterator (next): increment the counter.

8.6.5 Common facilities

Collection of objects that provide service of direct use to application objects; the objects are on a higher abstraction level than the system level object services. Distinction between

- **Horizontal category**

  There are 4 types of horizontal facilities:

  user interfaces for information editing, e.g. how they are supplied by OpenDoc or OLE.

  Information management services, e.g. compound documents.

  System management services, e.g. for installing, managing, configuring and operating of distributed application components.

  Task management services, e.g. agents, scripting, persistent transactions and workflow.

- **Vertical category**

  The facilities provided by objects depend on the application domain, e.g. for banking, marketing, health care, telecommunication etc.

8.6.6 Inter-ORB protocol

Communication between ORBs is based on GIOP ("General Inter-ORB Protocol").

GIOP Features

GIOP specifies

a set of message formats, e.g. request, reply, cancelrequest, and

common data representations (CDR)

for communication between ORBs. It also specifies a standard form for *remote object references*.

- GIOP works directly over a connection-oriented, reliable transport system

  IIOP (Internet Inter-ORB Protokoll) is GIOP based on TCP/IP.

External data representation

Distinction between primitive and complex data types, so-called typeCodes; assignment of integer values to identify data types

primitive: char, octet, short, long, float, double, boolean

complex: struct, union, sequence, symbol chains, fields

- The format of complex data types is described in the interface repository.
- **Example**

```
tk_struct (Typecode struct):
    string:  repository_ID
    string:  name
    ulong:  count
        {string:  member name
        TypeCode:  membertype }
```

Object reference

The object reference identifies the object that can be accessed through the Inter-ORB protocol. For IIOP, the object reference (IOR profiles) consists of

IP host address (e.g. host name).

TCP port number.

object key.

GIOP message

A GIOP message has three components

the GIOP message head.

a header which depends on the message type, e.g. request message, reply message.

the message content.

- **GIOP message head**

  The GIOP message head has the same format for all message types; it identifies the message type sent to another ORB.

  - **GIOP message head structure**

```
module GIOP
    struct Version {octet: major; octet: minor};
    enum MsgType {Request, Reply, CancelRequest,
    LocateRequest, LocateReply, CloseConnection, MessageError,
    Fragment}
    struct Message_Header {
        char magic[4]; this is the string "GIOP"
        Version GIOP_version;
        octet flag
        octet message_type
        usigned long message_size
    }
```

- the component `flags` determines the used byte ordering (big/little endian) and whether or not the entire message has been divided into several fragments.
- `message_type` is an element of the MsgType enumeration; it identifies the message type.

- **GIOP message types**

  GIOP supports the following message types:

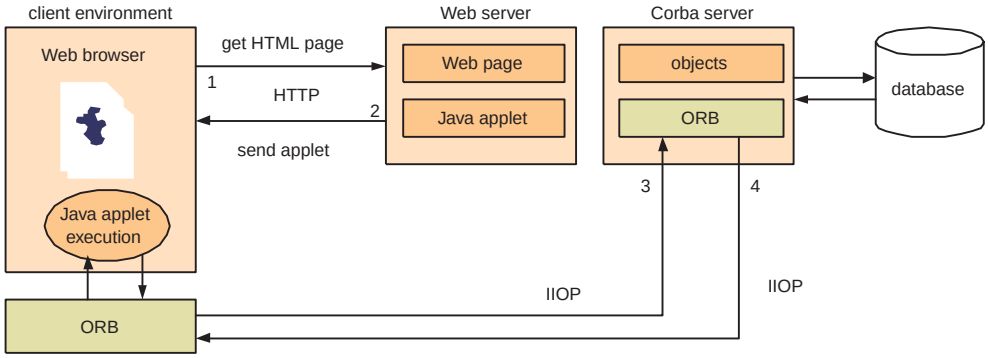
  1. *Request*: request the execution of an operation at the remote object, e.g. access of an attribute; the message contains the call parameters.
  2. *Reply*: answer to a request message.
  3. *CancelRequest*: termination of a request; the calling ORB does not expect an answer to the original request.
  4. *LocateRequest*: is used to determine

     whether the given object reference is valid, or

     whether the destination ORB processes the object reference, and if not, to which address requests for the object reference are to be sent.
  5. *LocateReply*: answer to LocateRequest
  6. *CloseConnection*: the destination ORB notifies the calling ORB that it closes the connection.
  7. *MessageError*: exchange of error information.
  8. *Fragment*: if, for instance, the request consists of several parts, then first a request message is sent, and then the remaining parts are sent using fragment messages.

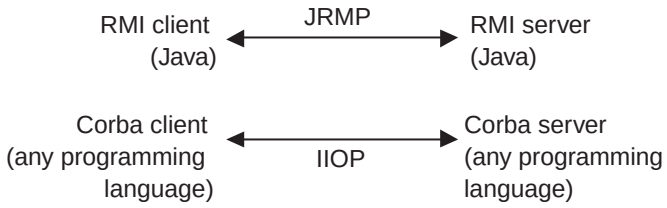
Example for IIOP use

Web access of a database using a Java applet and Corba.



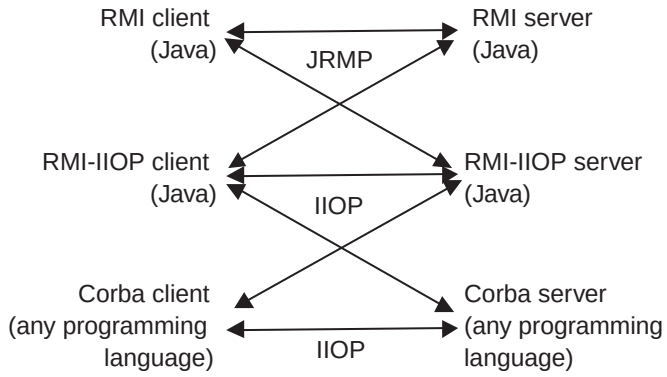
RMI over IIOP

RMI uses JRMP (Java Remote Method Protocol) for the communication between client and server objects, i.e. there is no interoperability with Corba.



- **Extension of RMI to RMI-IIOP**

  RMI-IIOP uses JNDI in order to register objects by their names.



Moreover, there is a Java IDL for Corba

- does not use JRMP for communication between remote objects.
- no interaction with RMI objects is supported.

Chapter 9

Secure communication in distributed systems

9.1 Motivation

The increased commercial usage of the Internet, in particular the Web, has made security a more and more pressing issue. Business and money transactions via networks are just as important domains as the *authentication of persons and software components*. The following section gives only a very brief introduction into the field of *cryptography* from the user's perspective.

9.1.1 Background

What is security in computer systems?

Prevent: By technical and organizational means

someone: differentiation between individual persons and groups

from doing some: limited by the human imagination

nasty things:

1) unauthorized reading of data (secrecy),
2) unauthorized writing of data (integrity),
3) pretending to be some else (authenticity),
4) unauthorized use of resources (availability).

Study by Coopers & Lybrand

Sources: Coopers & Lybrand Study [1988], ICL-Study (initiated by the EU)

Increased utilization and dependency on networks,

Increased number of security violations.

Reasons/factors	Frequency	Cost
human failure	70 %	25 %
unintentional break-downs	25 %	30 %
malicious attacks	5 %	45 %

- **Result of the study**

  High demand on security services (prioritized from top to bottom)

  User authentication,

  Message integrity and availability of network services,

  Keeping messages secret.

  Nevertheless, security aspects still do not have enough prominence in the design and management of hardware and software systems.

9.1.2 Issues

This section discusses the following issues

short introduction into cryptographic systems.

an authentication service for distributed systems.

secure RPC.

web security.

9.2 Introduction

9.2.1 Security threads

Main goal of security

restrict access to information and resources to those entities that are authorized to have access.

Three broad classes of security threads

leakage: the acquisition of information by unauthorized recipients.

tampering: the unauthorized alteration of information.

vandalism: interference with the proper operation of the system without gain to the perpetrator.

Factors threatening security

Examples for factors threatening security in distributed systems are

- tapping communication paths.
- intercepting and eavesdropping on messages.
- replay of messages which have been intercepted and copied.
- alterations of the message sequence.
- modifying the source or destination address of a message.
- host name and network addresses alone are not trustworthy.
- someone pretends to be a certain client, or a certain server.
- distribution of viruses.
- blocking servers by sending a huge number of requests (see yahoo, Microsoft)

Therefore, it is necessary to authenticate client and server.

9.2.2 Security requirements

There are various security requirements:

- **Authenticity**

  the message source, the sender identity and the data authenticity must be guaranteed, i.e. the alleged data source must be identical with the actual data source.

- **Confidentiality, secrecy**

  confidential data, even if transferred through an open network.

- **Non-repudiation**

  a seller, for instance must be able to prove to his customer that he initated an order. It prevents the sender from disavowing a legitimate message or the receiver from denying reception of a message.

- **Integrity**

  Data must be protected against non-authorized writing; writing includes insertions, modifications, deletions, etc.

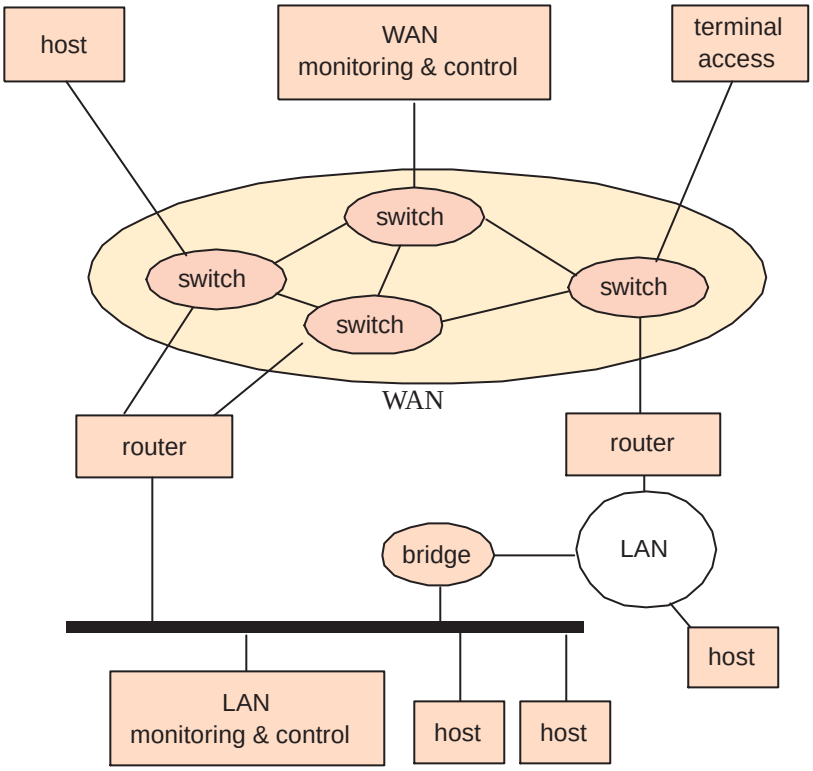
- **Availability**

  Protection of system resources against non-authorized use in order to guarantee availability for authorized users.

  System resources are: processors, memory, communication channels, programs, etc.

9.2.3 Attacks against secure communication

Points of attack: overview



Examples of points of attack:

network access points.

switches, routers, bridges, repeaters.

communication links, terrestrial links (glass fibre, copper), satellite channels, radio communication.

control centers.

hosts.

Attacks against the communication system

Attacks are mostly against the information transmitted through communication links.

cable tapping
especially against Ethernet networks (e.g. thick cable, 10Base T).
less realistic for glass fibre.

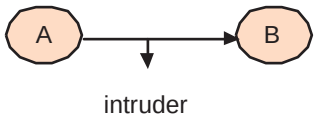
eavesdropping display screens.

Types of attacks

Distinction between passive and active attacks.

- **Passive attacks**

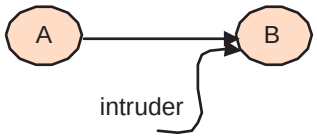
  intruder merely eavesdrops the communication.



  passwords are often transmitted unencrypted or lightly encrypted; data are readable by intruders.

- **Aktive attacks**

  intruder eavesdrops, sends and disturbs the communication.



  masquerading is possible; replay of past messages; messages can be delayed, manipulated, removed, etc.

Basic security structure

The following steps must be taken to analyze and design the basic security structure

- *Step 1*: identify subjects and objects

  subjects access objects

  subjects: users, processes, etc. ...

  objects: computers, programs, data, etc., ...

- *Step 2*: define mechanism for selective assignment of access rights for certain objects to certain subjects ⇒ access control.

- *Step 3*: make sure that these mechanisms cannot be circumvented

  secrecy, integrity.

  authentication.

9.3 Cryptography

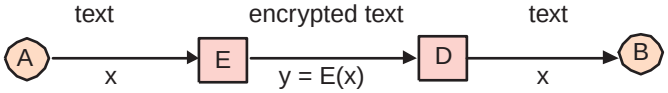
Cryptography is a practical technique for guarding against security violations in network-based distributed systems. In addition cryptography can be used for authentication.

9.3.1 Areas of cryptography

- **Cryptography**

  science of secret writing and encrypting; the process is occasionally refered to as a cipher; transfer of encrypted text instead of plain text.

  between E and D, encrypted information is transmitted.



- **Cryptoanalysis**

  science of breaking the encryption code; disclosure of the plain text or the encryption key.

9.3.2 Definitions

Definition: A **cryptosystem** is a tuple (M, C, K, E_k, D_k) with

a finite plaintext space M

a finite ciphertext space C

a finite key space K

a set of encryption transformations E_k: M $\rightarrow$ C with k $\in$ K

a set of decryption transformations D_k: C $\rightarrow$ M with k $\in$ K

Cryptosystems must satisfy the following requirements

- $\forall$ k $\in$ K, m $\in$ M : D_k (E_k (m)) = m.
- **Secrecy**

  it is impossible to determine the plain text from the encrypted text c (ciphertext).

  it is impossible to determine D_k systematically from the ciphertext c, even if m is known to be E_k(m) = c.
- **Authenticity**

  it is impossible to find a ciphertext c' for which D_k (c') $\in$ M is a valid plain text.

  it is impossible to determine E_k systematically from a ciphertext c, even if m is known to be E_k (m) = c.

9.3.3 Classes of cryptosystems

We can distinguish between 2 classes of cryptosystems:

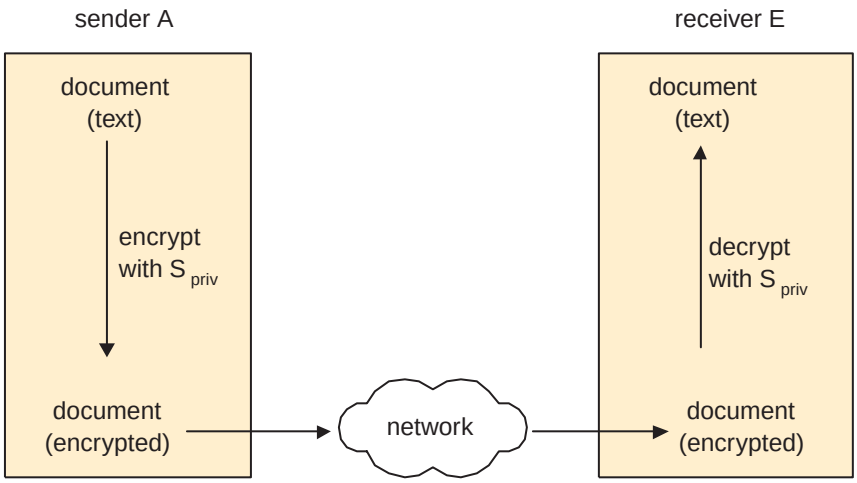
symmetric: sender and receiver have a single private key for encrypting and decrypting messages.

asymmetric: sender uses a public key to encrypt and receiver uses a private key to decrypt the message.

Symmetric cryptosystems (private-key)

Sender and receiver use a single private key to encrypt and decrypt the messages to be exchanged.

- **Process scenario**



  S_{priv} is a private key for communication between A and E.

- **Example: DES (Data Encription Standard)**

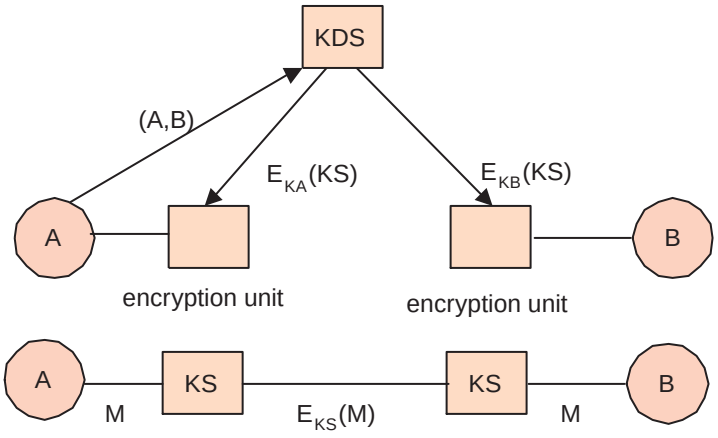
  Originally, DES presumed a fixed key length of 56 bit plus 8 bit parity; these days, keys have 112 bit and more.

  – Implemention of the encryption/decryption algorithm in hardware.

- **Key distribution**

  Problem: Secure distribution of the private keys, i.e. secure exchange of private key between sender A and recipient E $\Rightarrow$ a secure communication channel is required.

  – Possible solution: key distribution server (KDS)



– Characteristics

  users share a secret key with KDS (KA and KB for A and B, respectively).

  on request, KDS generates a private session key.

  KDS distributes encrypted session keys to the communication partners.

  partners communicate using a common private key.

– Another solution for key distribution is the use of an asymmetric cryptosystem.

Asymmetric cryptosystems (public-key)

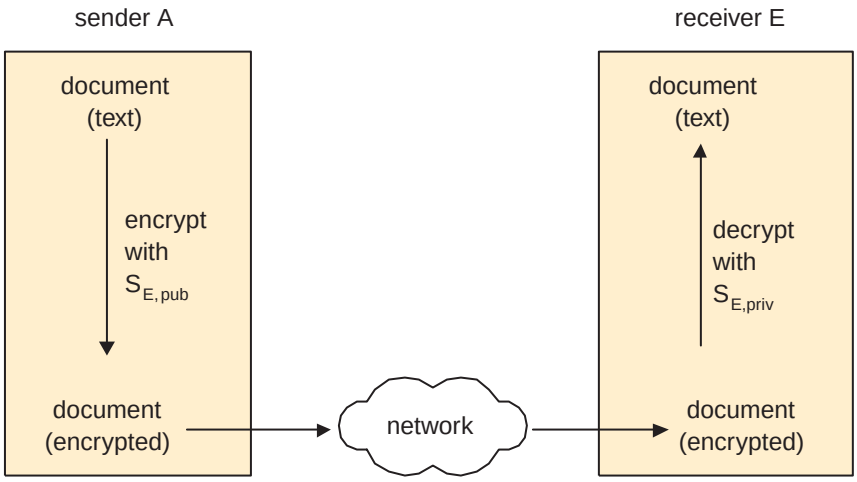
Each user has a public key which is published to everyone through a directory, and a secret key which is not revealed to anyone

  $\Rightarrow$ keys define a pair of transformations, each which is inverse of the other.

- **Process scenario**

  Support of a secure data exchange using a pair of keys (S_{priv}, S_{pub})

  S_{priv} is the private key, S_{pub} is the public key



- **Example: RSA**

  In general, RSA uses a key length $>$ 200 bits.

  The encryption process is slow; therefore, it is often combined with symmetric cryptosystems.

  – Distinction following security categories

    home: 512 bit

    standard: 1024 bit

    military: 2048 bit

- **Computation of private and public keys**

  – Computation of public key S_{pub}

1.	select 2 positive prime numbers p and q;	(p = 7, q =17)
2.	x = (p-1) * (q-1);	(x = 96)
3.	determine a number e with e not dividing x;	(e = 5)
4.	n = p * q;	(n = 119)
5.	S_{pub} = (n, e);	(S_{pub} = (119, 5))

  – Computation of private key S_{priv}

6.	determine d so that mod(d*e, x) = 1;	(d * 5/96 = 1 $\Rightarrow$ d = 77)
7.	S_{priv} = (n, d)	(S_{priv} = (119, 77))

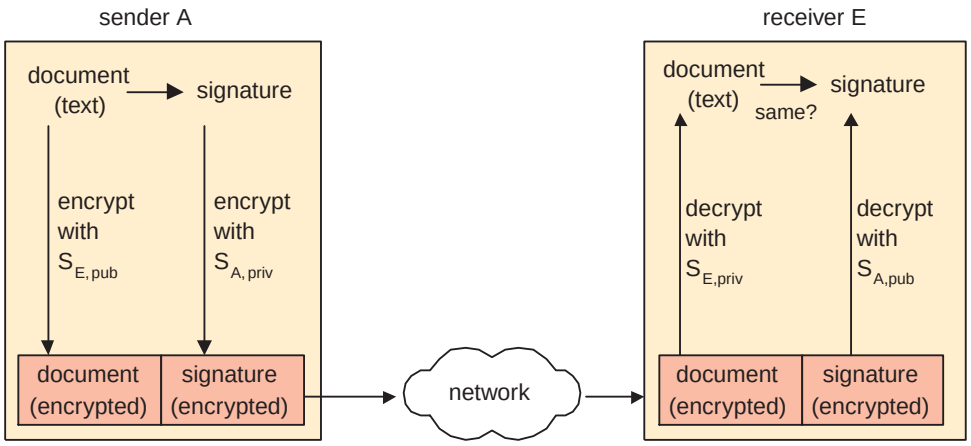
– computation of the cipher text c from the plain text m (m = 19), i.e. the plain text is interpreted as a numeric value; apply the key S_{pub} to encrypt m.

8.	$c = mod\ (m^e, n)$	$(c = mod(19^5, 119) = 66)$

– computation of the plain text m from the cipher text c; apply the key S_{priv}

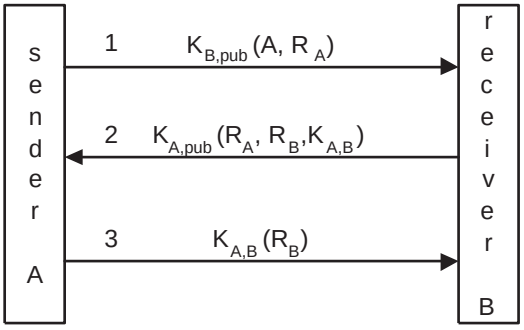
9.	$m = mod\ (c^d, n)$	$(p = mod(66^{77}, 119) = 19)$

- **Digital signatures**



Authentication using public key

A public key cryptosystem allows authentication without a key distribution center.



Comparison of cryptosystems

- Symmetric cryptosystems: e.g. DES

  (plus) low complexity, high efficiency (hardware implementation).

  (minus) complex key distribution.

  (minus) complex implementation of digital signatures.

- Asymmetric cryptosystem: e.g. RSA

  (plus) simple key distribution

  (plus) digital signatures easy to realize.

  (minus) high complexity, low efficiency (despite of hardware implementation).

- A combination of both approaches seems to be advisable

  at communication begin start with asymmetric cryptosystem (session key exchange).

  switch to symmetric cryptosystem for data exchange.

- Maintain data integrity during communication

  encryption of check sum (hash code) guarantees integrity.

  encryption of entire message $\Rightarrow$ integrity and secrecy.

Animation Encryption

siehe Online Version

9.4 Authentication

Definition: Authentication means verifying the identities of the communicating partners to one another in a secure manner.

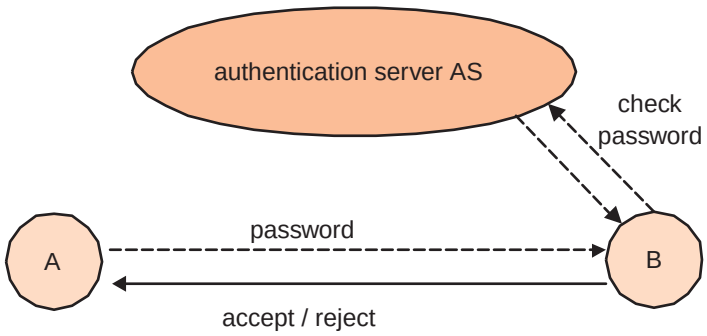
9.4.1 General authentication schemes

There are different levels and types of authentication:

simple: based on simple password schemes, and

strong: based on cryptographic techniques.

One-way authentication using passwords



Shared secret of A and AS: password

password is sent as plain text!

- **Risks**

  During typing: "spying over one's shoulder".

  Storing the password in the computer: password file is a high security risk.

  Storing the password with the user
  by heart if chosen by the user (often easy to guess).
  on paper or other means if chosen by the system (danger of theft).

On-way authentication: other schemes

Better than passwords

fingerprint

analysis of user spoken words

signature, etc.

These identifiers cannot be "stolen" by other users.

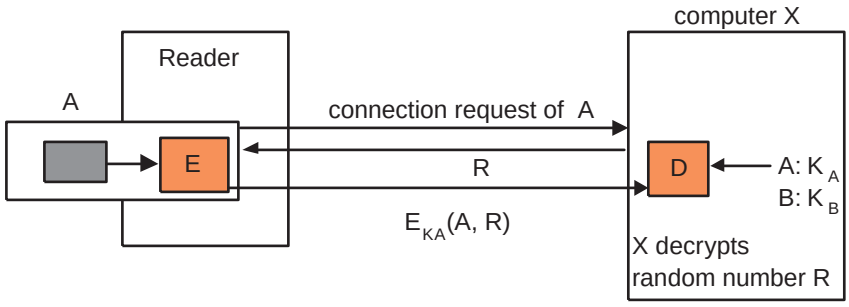
- But: during network transfer, these identifiers are represented as bit strings

  they can be copied and replayed.

  $\Rightarrow$ masquerade is possible.

SmartCard

SmartCard contains a coder (E) and a secret key K_A of user A.



- If decryption at the destination system results in the value (A, R), then A is authenticated.
- The card is password protected: protection against nonauthorized usage in case of loss.
- Characteristics:

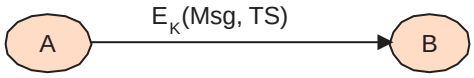
  no transfer of secrets.

  actuality is guaranteed by the random number (copy and replay is useless).
- But: destination system must know the user secret (security risk).

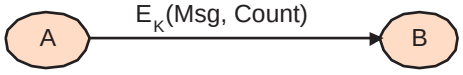
Discovery of replay

An intruder might monitor the exchange of messages, record them and replay them at later time (pretending to be the original sender).

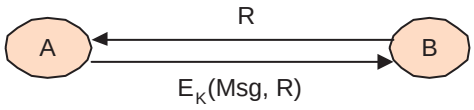
- Timestamps: clocks must be synchronized.



- Counter: recipient must record the current value.



- Random number (Nonce): additional message transfer to send the random number R.



9.4.2 Authentication service Kerberos

Kerberos has been developed at the MIT as part of the distributed framework Athena. It is also a part of the DCE framework. The Kerberos authentication protocol is based on the protocol by Needham and Schröder.

Introduction

This course provides only a short introduction to Kerberos (for further information, consult the Kerberos Web-Site (URL: http://web.mit.edu/kerberos/www/))

- **Motivation**

  Kerberos assumes the following components

  Client C,

  Server S,

  Key distribution center KDC, and

  Ticket granting service TGS.

  - **Goal of Kerberos**

    A client C requests the service of the server S. KDC and TGS are supposed to guarantee the secrecy and authenticity requirements.

    1. KDC manages the secret keys of the registered components.
    2. Within a session TGS provides the client C with tickets for authentication with servers of the distributed system.

- **Security objects of Kerberos**

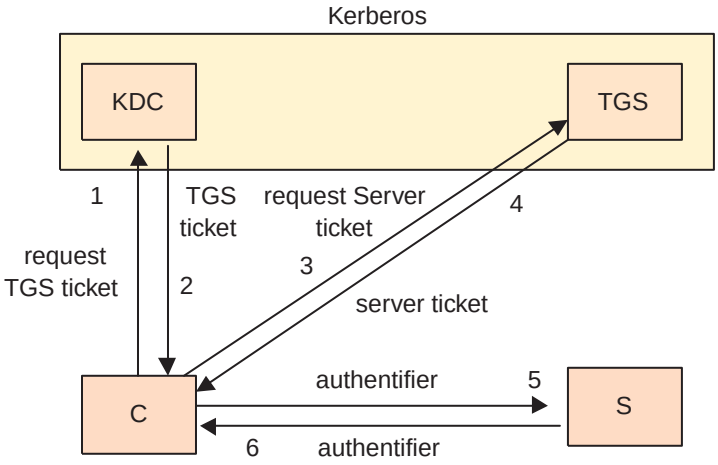
  Kerberos enables authentication through the following three security objects.

  1. *TGS ticket*: issued by KDC to the client C for presentation at TGS.

2. *Authentifier*: generated by client C; it identifies the client and guarantees the validity for the communication with server S.
3. *Session key*: generated by Kerberos for the communication between client C and server S.

Authentication process scenario

- **Graphical representation**



- **Description of exchanged messages**
  - **Message 1: C to KDC**

C → KDC with information	C, TGS

    * KDC determines by querying a database the secret key K_C for communciation between Kerberos und C; KDC generates a good random session key $K_{C, tgs}$.
  - **Message 2: KDC to C**

KDC → C with information	$(K_{C, tgs})_{K[C]}$
	$(C, TGS, T_{kdc}, L_{kdc}, K_{C, tgs})_{K[tgs]}$ = $ticket(C, TGS)_{K[tgs]}$

    In the following the terms K_C and K[C] are equivalent.
    * secret key K_C is derived from the user password.

- the second part of the message is not interpreted by C. Rather, it is forwarded to TGS as a whole (representing the *TGS ticket*). The ticket is encrypted with the secret key K_{tgs} of TGS.

  T_{kdc} timestamp for ticket creation time.
  L_{kdc} life-span of the ticket.

– **Message 3: C to TGS**

C → TGS with information	$(C, T_C)_{K[C,tgs]}$
	$ticket(C, TGS)_{K[tgs]}$
	S

- TGS determines a random session key $K_{c,s}$, if

  TGS ticket is still valid,
  T_C is current, and
  field C matches (of the first parameter and of the ticket).

– **Message 4: TGS to C**

TGS → C with information	$(K_{C,S})_{K[c,tgs]}$
	$(C, S, T_{tgs}, L_{tgs}, K_{c,s})_{K[S]} = ticket(C, S)_{K[S]}$

- The second part of the message serves as a ticket of C for server S.
- K_S is the secret key of server S known to Kerberos.

– **Message 5: C to S**

C → S with information	$(C, T_C)_{K[c,s]}$
	$ticket(C, S)_{K[S]}$

Messages 5 and 6 support the mutual authentication of C and S, respectively.

– **Message 6: S to C**

S → C with information	$(T_C)_{K[c,s]}$

- **Problems with Kerberos**

  Manipulation of local computer clocks to circumvent the validity time of tickets

  i.e. synchronization of clocks in distributed systems must be authorized and authenticated.

- **Example: user login with Kerberos**

1. login program of the workstation W sends user name N to KDC.
2. if the user is known, then KDC sends a session key K_N encrypted with the user password, as well as a TGS ticket.
3. login program requests the password from the user and decrypts the session key K_N using the password; if the password was correct, then the decrypted session key K_N and the session key K_N within the TGS ticket are identical.
4. the password can be removed from the main memory because for further communication, only K_N and the TGS ticket are used; both are used to authenticate the user at TGS if the user requests a server S.
5. establish a user login session on workstation W.

9.4.3 Authenticity in RPC´s

A variety of distributed applications based on the client/server paradigm apply the RPC mechanism for communication, e.g. the NFS file system applying the Sun RPC. However, in most implementations, the authenticity is not checked.

Background

Let us assume a client C requires a service provided by server S. The request message by client C to server S contains the following authentication information:

credentials.

verifier.

The reply of S to C only contains the verifier.

- Distinction between different authentication types:

  AUTH_NULL: no authentication.

  AUTH_UNIX: credentials contain host name, user name and group login; verifier is not used.

  AUTH_SHORT: is used as a reply after a AUTH_UNIX request; the server sends an alias which the client uses instead of the credential for the subsequent communication.

  AUTH_DES: message contains a verifier encrypted by DES.

DES authentication

If there is no verifier, the credentials are easy to fake. The server has no means of verifying the client's credentials.

⇒ DES authentication rectifies this problem.

- **Authentication: request message**

C → S sending the information	C
	$(K_{CS})_{KP[CS]}$
	$(L)_{K[CS]}$
	$(T, L\text{-}1)_{K[CS]}$

  - KP_{CS} (KP[CS]) is a key which can be computed locally by C and S; it is derived from their respective public keys.
  - Information provided in the request message

    1. Credentials C, $(K_{CS})_{KP[CS]}$, $(L)_{K[CS]}$
    2. Verifier $(T, L\text{-}1)_{K[CS]}$
    3. Validity window L
    4. Current timestamp T

- **Authentication: reply message**

S → C sending the information	$(T\text{-}1)_{K[CS]}$
	N_{CS}

  N_{CS} is a nickname for C in the further communication.

- **Communication after authentication**

  The following RPC calls of C sent to S utilize the values N_{CS} and $(T')_{K[CS]}$ for credentials and verifiers.

  - **Client call**

C → S sending the information	N_{CS}
	$(T')_{K[CS]}$

    Credentials N_{CS}

    Verifier $(T')_{K[CS]}$

– **Server reply**

S → C with information	(T' -1) $_{K[CS]}$

Verifier (T'-1) $_{K[CS]}$

9.5 Secure web transfer

The commercial use of the Internet, and the web in particular, has increased the importance of secure transfer mechanisms. Important application domains are electronic commerce, electronic payment (credit card number, but also CyberCash, etc.).

9.5.1 Basic authentication

The Web server maintains a protected document D which client C wants to access.

- C executes a normal HTTP request to D.
- Server S denies delivery of D with the error message "unauthorized to access the document" (Code 401).
- C requests login name and password from the user and encodes them using the base-64-scheme.
- Server S checks login name and password, in case of success, the document is supplied.

Disadvantage: both login name and password are practically transferred as plain text; no encrypting of information.

9.5.2 SSL (secure socket layer)

Introduction

This cryptographic scheme has been designed by Netscape. SSL, which supports secure data transfer, incorporates an additional layer between the application and socket layer. See information at Netscape (URL: http://home.netscape.com/eng/ssl3/index.html) Web site.

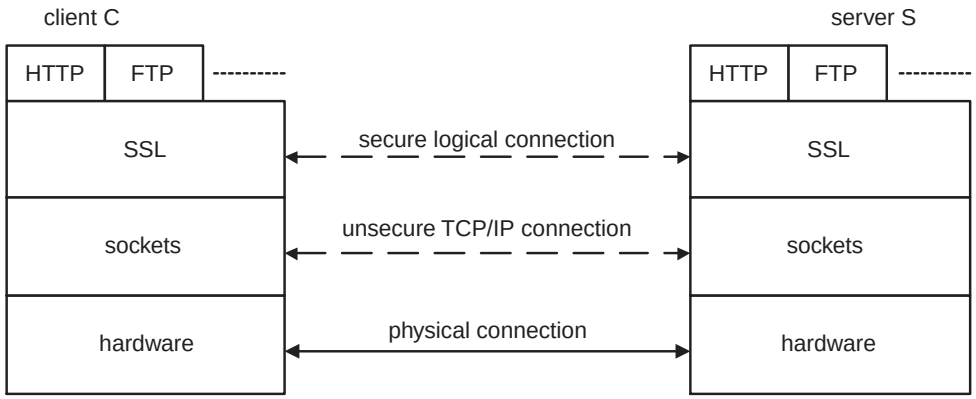
- SSL has the following goals:

  1. Authenticity, reliability and integrity should be supported independent of the used protocol.
  2. existing software should be used as before, without modifications.
  3. extensability with respect to new cryptographic schemes.

- SSL has the following basic features

  – secret data transfer through the communication channel.
  – partner authentication using asymmetric schemes or public key schemes.
  – the connection is reliable; integrity check by using MAC (message authentication code).

Embedding into the Internet reference model



Logical structure of SSL

SSL contains two components: encryption/transfer layer and protocols

- **Encryption and transfer layer**

  The encryption and transfer layer (SSL Record Layer) is based on a reliable transport protocol. The application data transferred via SSL are processed as follows:

1. the data are divided into blocks with a maximum size of 16K; data of different clients can be combined in one block (provided they have the same type).
2. adding status information (e.g. protocol version, block length, message type).
3. compression of the blocks.
4. encryption of the compressed blocks.
5. transfer of encrypted blocks to the underlying transport layer (e.g. TCP).

- **Protocols**

  The protocols are based on the SSL record layer. Support of different protocol types:

1. *handshake protocol:* client and server negotiate cryptographic parameters, e.g. encryption, key exchange, and MAC schemes.
2. *alert protocol*: for problems arising during handshake.
3. *ChangeCipherSpec protocol*: informs the partner that the cryptographic parameters agreed upon in the handshake protocol are now valid.
4. *Application Data protocol*: contains encrypted compressed application data.

SSL sessions

- **Session parameters**

  A session is defined by the following parameters:

  a session identifier.

  a certificate of the communication partner; this parameter is optional and is based on X.509.

  compression algorithm.

  encryption scheme: symmetric (e.g. DES) and MAC algorithm (e.g. MD5 or SHA).

  master secret: a master secret is a 48 byte sequence which client and server keep secret.

  flag, whether the session may establish a new connection.

- **Connection parameters**

  A connection is determined by the following parameters:

- Data structure containing random numbers; numbers are selected by client and server for each connection.
- MAC Keys for client and server, respectively.
- Keys for data encryption, both for client and server.
- Initialization vector for encryption schemes.
- Sequence number.

Handshake protocol

The protocol supports negotiations between client and server to determine the cryptographic parameters:

> the protocol version, the selection of the applied encryption schemes, the mutual authentication, usage of a public key scheme for generating shared secrets (so-called master secrets).

- **Process scenario**

  The Handshake protocol encompasses 10 steps before the application data are exchanged.

  - *step 1*: client sends ClientHello message to server.

    ClientHello message structure

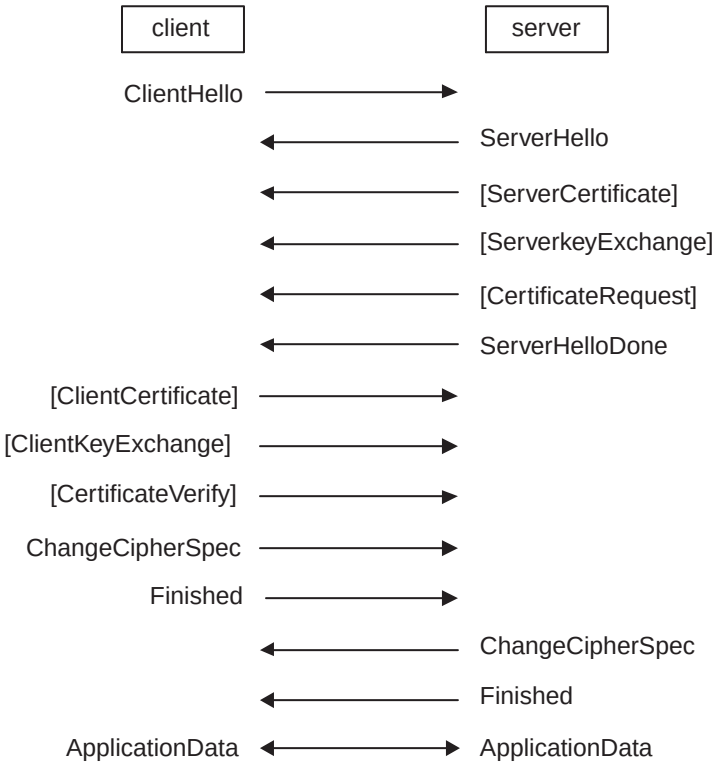
    > SSL version number of client.
    >
    > random data structure (ClientHello.random) containing client system time (32 bits) and a random number (28 bytes).
    >
    > session identifier.
    >
    > list of encryption and MAC schemes supported by the client.
    >
    > list of compression schemes supported by the client.

  - *step 2*: server sends ServerHello message back to client. ServerHello message structure:

    > SSL version number of the server.
    >
    > random data structure (ServerHello.random) containing server system time (32 bit) and a random numeral (28 bytes).
    >
    > session identifier.
    >
    > (symmetric) encryption and MAC schemes, selected from those available at the client site.
    >
    > compression schemes, selected from those available at the client site.

- *step 3*: determine how the session keys are to be exchanged; there are two alternatives:
  1. server authentication by ServerCertificate message; server sends X.509 certificate to client; certificate contains the public key of the server.
  2. server does not have an X.509 certificate; server sends an ServerKeyExchange message to client; this message is used to exchange the public key either by:
     Diffie-Hellmann, or
     an uncertified public RSA key, or
     Fortezza scheme.
     When Diffie-Hellmann or RSA are used, a hash code is included for integrity, e.g. MD5(ClientHello.random + ServerHello.random + key information).
- *step 4*: server sends CertificateRequest message to client [optional].
- *step 5*: server sends ServerHello message to client; control is handed over to the client.
- *step 6*: client sends ClientCertificate message [optional].
- *step 7*: client sends ClientKeyExchange message. The message contains a pre-master secret encrypted by public key scheme.
  the pre-master secret contains 48 bytes and is a preparatory step towards the actual session key.
- *step 8*: if the client has only a certificate for signatures, then it sends a CertificateVerify message.
- *step 9*: client sends a ChangeCipherSpec and then a finished message to announce the completion of the handshake protocol. The finished message contains two MACs
  md5_hash = MD5(master_secret + pad2 + MD5(handshake_messages + client + master_secret + pad1))
  sha_hash = SHA(master_secret + pad2 + SHA(handshake_messages + client + master_secret + pad1))
- *step 10*: server sends a ChangeCipherSpec and then a finished message to announce the completion of the handshake protocol.
- *step 11*: sending the encrypted application data.

- **Overview of handshake protocol**



Creation of a session key

The creation consists of two steps:

1. Master secret is generated from the pre-master secret and the random data structures ClientHello.Random and ServerHello.Random (+ for concatenation);

   Master-Secret contains 48 bytes

```
master_secret =
    MD5(pre_master_secret + SHA('A' + pre_master_secret +
    ClientHello.random + ServerHello.random)) +
    MD5(pre_master_secret + SHA('BB' + pre_master_secret +
    ClientHello.random + ServerHello.random)) +
    MD5(pre_master_secret + SHA('CCC' + pre_master_secret +
    ClientHello.random + ServerHello.random))
```

2. Master secret is converted into the key and MAC secrets.

```
key_block =
    MD5(master_secret + SHA('A' + master_secret +
    ClientHello.random + ServerHello.random)) +
    MD5(master_secret + SHA('BB' + master_secret +
    ClientHello.random + ServerHello.random)) +
    MD5(master_secret + SHA('CCC' + master_secret +
    ClientHello.random + ServerHello.random)) + ...
```

from the key_block, the respective keys can be extracted, e.g. client_write_MAC_secret = key_block[0 : 15].

Problems of SSL

SSL has a number of problems

no reliable authentication possible if X.509 certificates are missing.

no utilization of keys which were exchanged over other, secure channels.

generation of random numbers (quality depends on locally available random number generator).

proxy server might cache SSL messages.

Chapter 10

Summary

This lecture discussed basic concepts which are important for the design and implementation of distributed applications. Major issues presented were:

basic interaction models, such as client-server, remote procedure call, tuple space.

distributed transactions and group communication.

distributed file systems, replication, voting schemes.

design issues for distributed applications.

object-oriented distributed systems.

secure communication in distributed systems, especially authentication.