XII. Distributed Systems and Middleware

Outline

- **Distributed Systems Basics**
- Middleware generalities
- Middleware for Distributed Objects
- Distributed Computing Models

Distributed systems: a definition

"A distributed system is a collection of processors that do **not share memory or a clock**. Instead, each processor has its own local memory, and the **processors communicate** with each other through various **communication lines**. The processors in a distributed system **vary in size and function**. They may include small microprocessor, workstations, minicomputers, and large general purpose computer systems.

…

A distributed system must provide various mechanisms for **process synchronization and communication**, for dealing with the **deadlock problem**, and for dealing with a variety of **failures that are not encountered in a centralized system**."

(Silberschatz & Galvin, Operating System Concepts)

Highlights

- No shared memory and neither shared time
- Heterogeneity
- Typical problems of concurrency and necessity of suitable means to deal with them
- New and more points of failures
- Security and Integrity issues

CONCLUSION

Building a distributed system is really more difficult and expensive than building a centralized one. Hence the choice must be pondered. In general it is better not to build a distributed system if it can be avoided

Why building a distributed system?

- However some non-functional requirements cannot be achieved by a centralized system, and this lead to the construction of Distributed System, in particular:
  - **Scalability**
  - **Resource Sharing**
  - **Heterogeneity**
  - **Fault-Tolerance**
  - **Openness**

Non-functional requirements

- **Scalability**:
  - The system must be capable of accommodating growing load in the future
  - Distributed system can be "easily" scalable adding new computers in the original configuration
- **Resource Access and Sharing**:
  - Often it is necessary to share hardware, software and data
  - Resource manager and security issues

Non-functional requirements

- **Heterogeneity**:
  - Use of different technology for the implementation of services and legacy components
  - Differences in: programming languages, operating systems, hardware platforms, network protocols
- **Fault Tolerance**:
  - Operations that continue also in presence of faults (many components higher probability of having faults)
  - Generally obtained by means of redundant components

Non-functional requirements

- **Openness**:
  - System can be easily extended and modified with new functionalities
  - It is necessary to establish well-documented and well-defined interfaces

What about Performance?

- Distributed system can certainly improve performance (real parallelism available)
- However performance should not be the main motivation
- Communications really expensive
- Consider instead Multiprocessor and Massively Parallel System

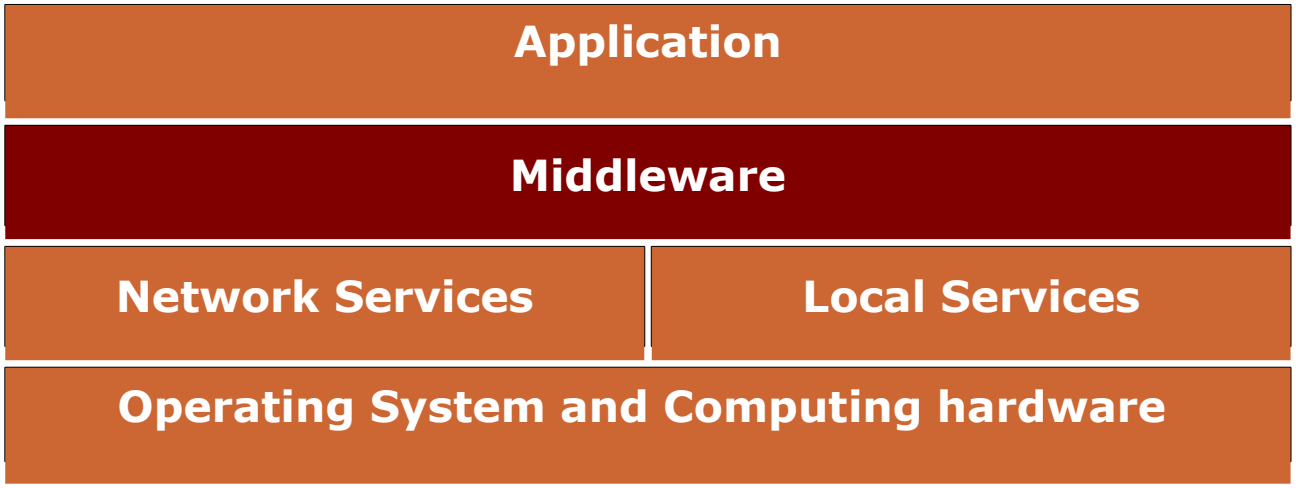
Outline

- Distributed Systems Basics
- **Middleware generalities**
- Middleware for Distributed Objects
- Distributed Computing Models

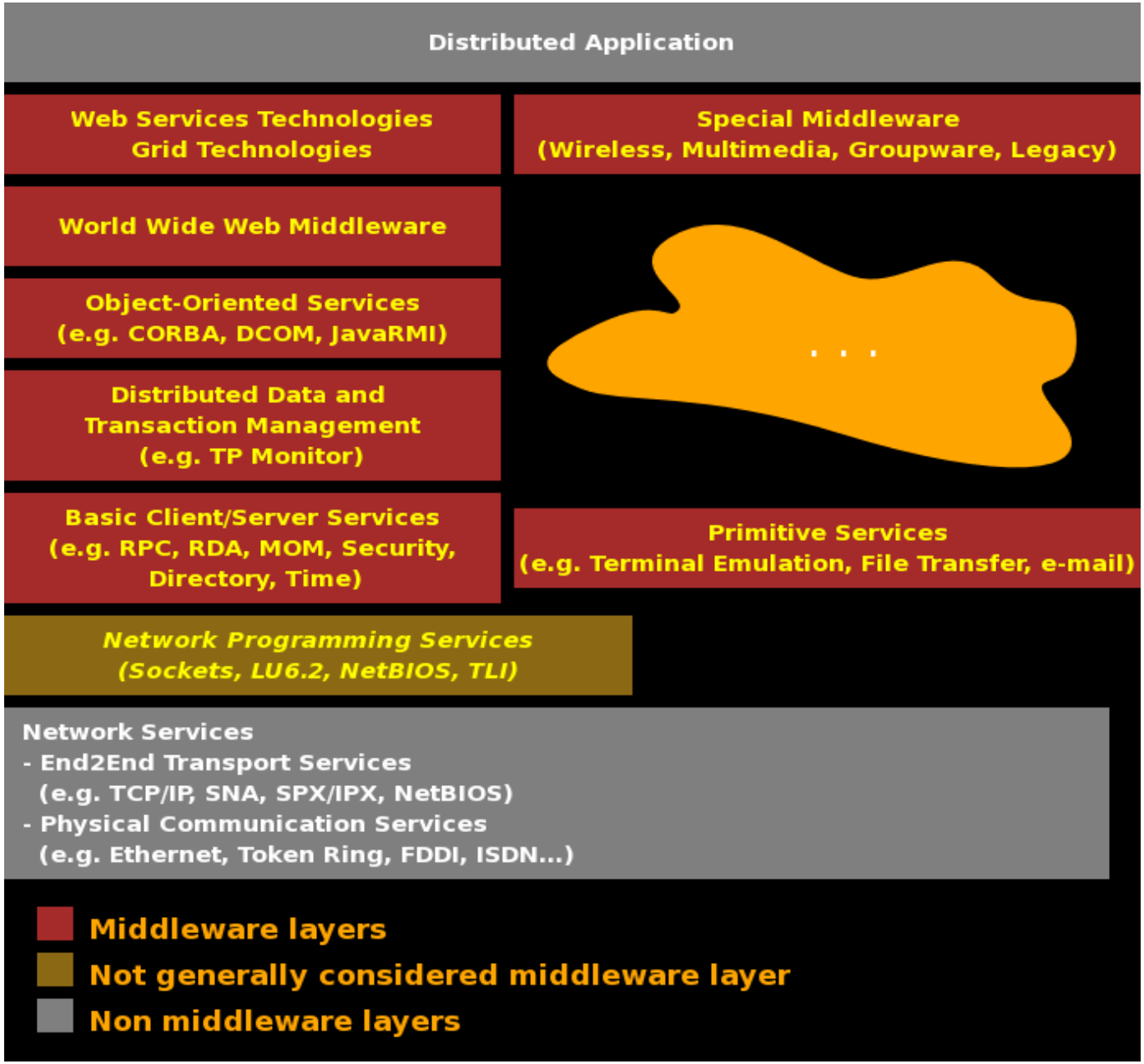
Middleware: a definition

"Middleware is a set of common **business-unaware** services that enable applications and end users to **interact** with each other **across a network**. In essence, middleware is the **software that resides above the network and below the business-aware application software**."

(Umar, *Object-Oriented Client/Server Internet environments*)

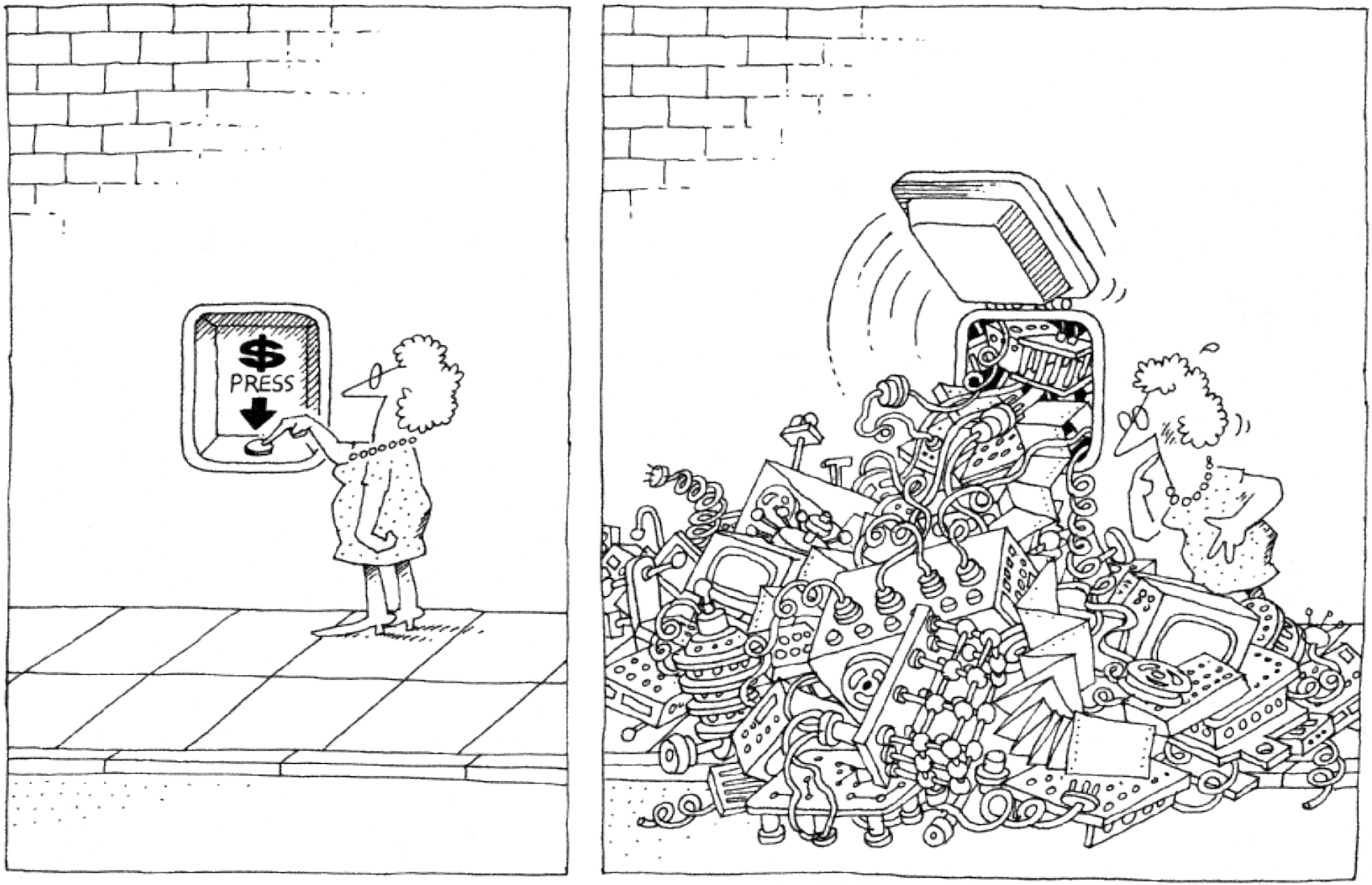


Middleware: a glance



Why do we need middleware?

- **Programming in heterogeneous distributed environment is made difficult by several factors:**
  - It is often necessary to exchange complex data
  - Different encoding of data types
  - References to other distributed components as return values
  - Distributed Components activation and deactivation
  - Synchronization and Real Parallelism
  - Need for atomic sequence operations
  - ...


$
PRESS

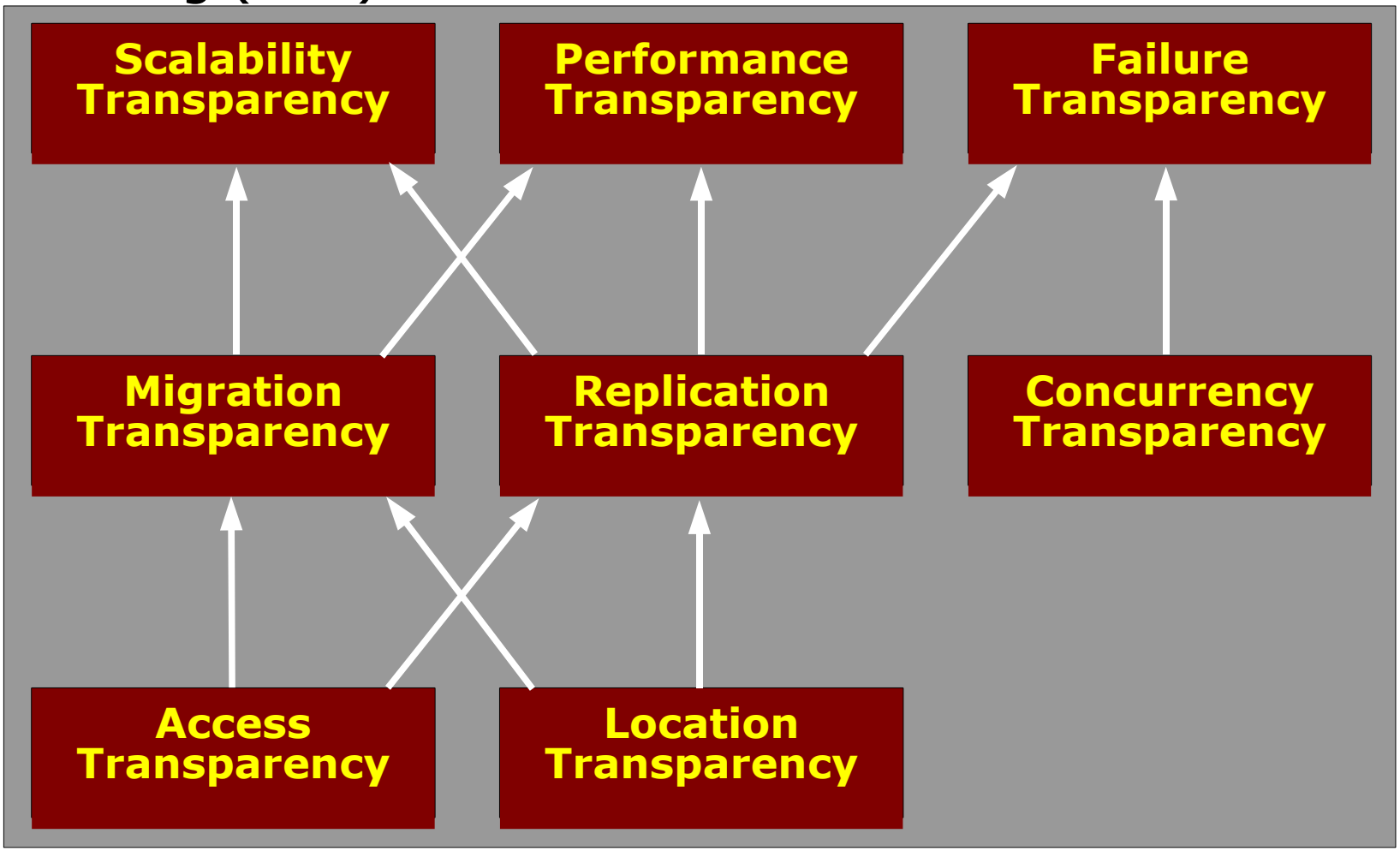
Trasparency

- **Trasparency – complexity introduced by distribution (mobility...) should disappear**
- The system should appear, to the final user, as an integrated computing facility
- Certainly useful to hide the distribution as much as possible to the application engineer

- Several dimension of transparency (not orthogonals) can be identified

Trasparency Relations

- Part of the International Standard on Open Distributed Processing (ODP)



(W.Emmerich – *Engineering Distributed Objects – John Wiley and Sons*)

Transparency Dimensions

- **Access Transparency**:
  - interface to a service do not depends from the location of the components that use it. Without it **is not an easy task to move the service** to a different host.
- **Location Transparency**:
  - a request for a service can be made without knowing the physical location of the components that provide the service. Without it moving components becomes almost impossible
- **Migration Transparency**:
  - Components can be migrated to different host without that the user are aware of that, and that the developer of clients components take a special consideration

Transparency Dimensions

- **Replication Transparency**:
  - The user of a service and the application programmer are not aware that a service they are using are provided by a replica
- **Concurrency Transparency**:
  - Several components may concurrently request services from a shared component while the shared component's integrity is preserved and neither users nor application engineers have to see how concurrency is controlled
- **Scalability Transparency**:
  - To the users and designers is transparent how the system scales to accommodate a growing load

Transparency Dimensions

- **Performance Transparency**:
  - The users and the application programmers are not aware of how the system performance is actually achieved (really difficult to obtain given the general unpredictability of load)
- **Failure Transparency**:
  - the user and the application programmer are unaware of how the system hides the failure. In some way they should believe that the service cannot fail

Outline

- Distributed Systems Basics
- Middleware generalities
- **Middleware for Distributed Objects**
- Distributed Computing Models

Types of middleware

- **Transaction Oriented Middleware**:
  - often used when the distributed components are database applications. It use the two-phase commit protocol to implement distributed transactions. (IBM CICS, BEA Tuxedo, Transarc Encina)
- **Message-Oriented Middleware**:
  - supports the communication by message exchange. Components use messages to require services and can wait for a response. It simplifies decoupling of clients and servers (then scalability) and supports multi-cast in a transparent way. (IBM MQSeries, Sun ToolTalk, NCR TopEnd)

Types of middleware

- **Remote Procedure Calls**: operation that can be invoked remotely across different hardware and operating systems platforms.
  - The intent was to provide a mean to invoke remote procedures as local procedures.
  - Interface Definition Language (IDL)
  - Client and Server stubs
  - RPC is not reflexive

Object-Oriented middleware

- Objective in the developing of OO Middleware is to **export the paradigm of OO into the distributed world**
- The development of a OO distributed application "must be" similar to the development of a normal OO application

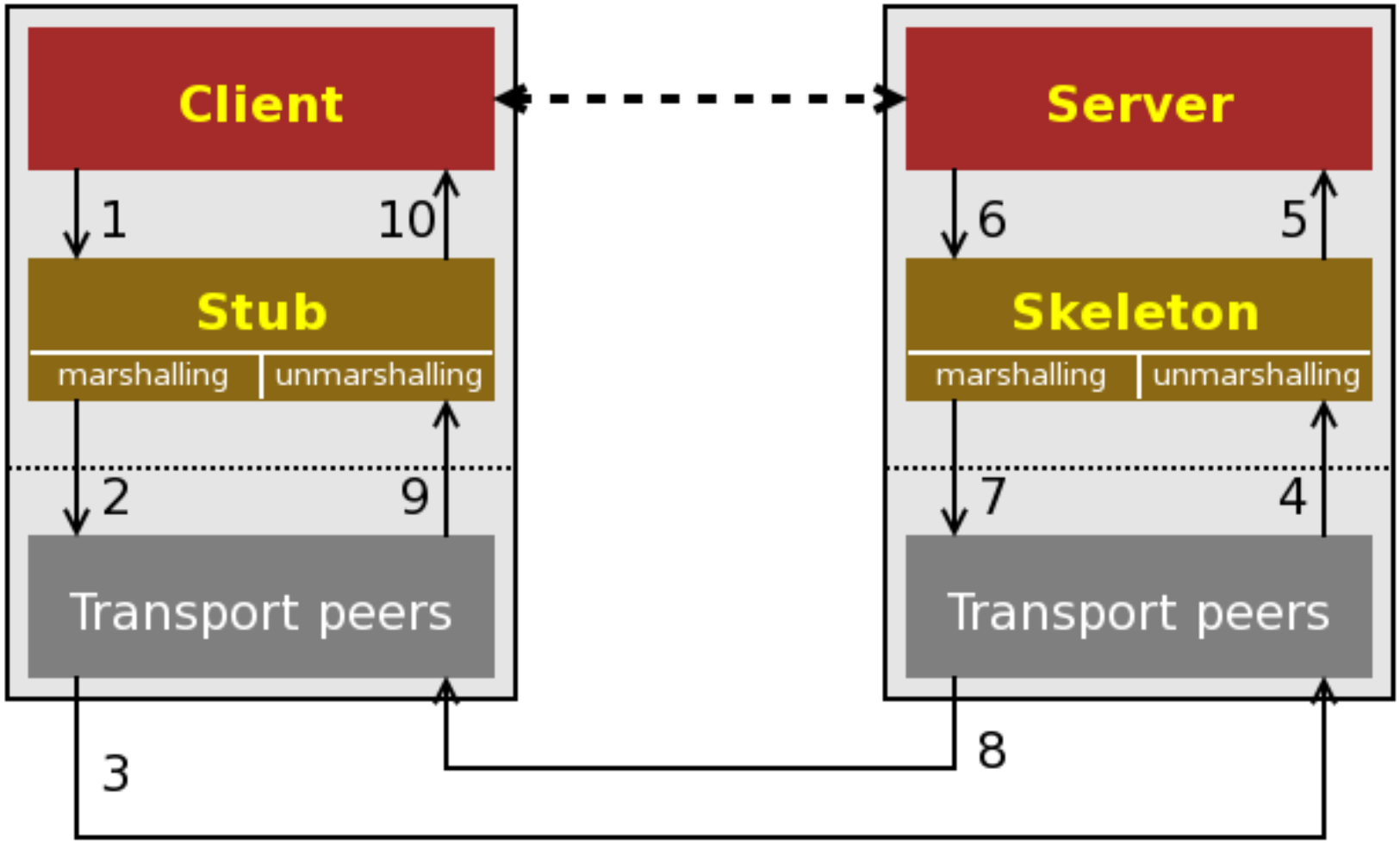
Distribution should not completely disappear!!!

OO Middleware

- Main elements of a OO middleware are:
  - **Object Request Broker**
  - **Interface Definition Language**
- Three main examples of OO middleware:
  - CORBA (OMG)
  - DCOM/.Net (Microsoft)
  - JavaRMI (Sun)

Just a glance on remote invocations

- Based on the same schema of RPC:



Just a glance on remote invocations

- A distributed AddressBook:

```
public interface AddressBookI {
  public String[] getAddress(String name) throws Trowable;
  public String[] getTown(String name) throws Trowable;
  public String getTelNumber(String name) throws Trowable;
}
```

Machine X

```
public class AddressBookServer implements AddressBookI {
   private AddressBookTable AddrBook;
   public String addEntry(String name, String street, String town, String tel) {
      AddressBook.addEntry(name,street,town,tel);
   }
   public String[] getStreet(String name) { return  AddrBook.getStreet(name); }
   public String[] getTown(String name) { return AddrBook.getTown(name); }
   public String[] getTelNumber(String name) { return AddrBook.getTel(name); }
}
```

Machine Y

```
public class AddressRemoteClient {
   public static void main(String[] args) {
      try {
         AddressBookI ab = new AddressBookServerStub();
         System.out.println("Via: "+ab.getStreet(args[1])+" Comune di: "+getTown(args[1])+
                  " Tel: "+ getTelNumber(args[1]);}
      } catch (Throwable t) {t.printStackTrace();}
   }
}
```

Class “Stub”

```
import java.io.ObjectOutputStream;
import java.io.ObjectInputStream;
import java.net.Socket;

public class AddressStub implements AddressBookI {
   Socket socket;

   public AddressStub throws Trowable {
      socket = new Socket(“localhost”,“9000”);
 }
   public String getStreet(name) throws  Throwable {
      ObjectOutputStream outStream =          new ObjectOutputStream(socket.getOutputStream());
      outStream.writeObject(“Street”);
      outStream.flush();
      outStream.writeObject(name);
      outStream.flush();
      ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
            return (String)inStream.readObject();
 }
   public String getTown() throws Throwable {
      ...
   }

}
```

Class “Skeleton”

```
import java.io.ObjectOutputStream; import java.io.ObjectInputStream;
import java.net.Socket; import java.net.ServerSocket;

public class AddressSkeleton extends Thread {
   AddressServer server;
   public AddressSkeleton (Address server) { this.server = server; }
   public void run() {
      try { ServerSocket serverSocket = new ServerSocket(9000);
         Socket socket = serverSocket.accept();
         while (socket != null) {
            ObjectInputStream inStream = new ObjectInputStream(socket.getInputStream());
            String method = (String)inStream.readObject();
      String name = (String)inStream.readObject();
            if (method.equals(“Street”)) {
               ObjectOutputStream OutStream = new ObjectOutputStream(socket.getOutputStream());
               outStream.writeObject(Server.getStreet(name)); outStream.flush();
            } else if (method.equals(“Town”)) { ... } } }
      } catch (Trowable t) {t.printStackTrace(); System.exit(0);}
   }
 public static void main(String[] args) {
      AddressServer server = new AddressServer(“Via Salaria”,“Roma”);
      AddressSkeleton addressSkel = new AddressSkeleton(server);
      addressSkel.start();
 }
}
```

Distribution can't disappear!!!

- Differences between local and distributed objects, and that must be considered by the application designer, concern:
  - **Life Cycle**
  - **Object References**
  - **Request Latency**
  - **Activation**
  - **Parallelism**
  - **Communications**
  - **Failures**
  - **Security**

Life cycle

- **Creation**: in traditional OO programming creation via constructor (compiler solve the problem). In distributed OO programming created object do not necessarily reside in the same process space, therefore is not possible to directly invoke constructors.
- **Migration**: after the creation an object could migrate to other hosts; when necessary the object should have been designed to permit migration
- **Deletion**:
  - difficulties in implementing garbage collections algorithms
  - **referential integrity is rather expensive**, it is difficult to know all the existing references to an object then it is necessary to deal with **server objects no more available**

Object references

- Parameter-passing
- Remote references are quite big data structure
- In complex middleware the necessity of space for a reference can be 100 times bigger than in "normal" OO programming

- Applications cannot maintain large numbers of object references
- Designing distributed object-based applications we have to minimize the number of objects

Request latency

- Local request call requires, in modern workstations, 250 nanoseconds (4ya)
- Remote request could require between 0.1 and 10 milliseconds (4ya)

- A remote request is about **400-4000 times** more expensive than a local one
- It is really important try to obtain locality. **Objects that communicate a lot between them, should reside on the same host!!**

Activation/Deactivation

- More new problems in distributed OO programming:
  - machines, hosting server objects, could be restarted
  - Resource required by all the server objects on a host may be greater than the resources available
  - Server object could be idle for long time between two invocations than could be opportune do not waste resources
- It is necessary to introduce other two operation to objects life cycle: **Activation** and **Deactivation**

Activation/Deactivation

- This operation can increase latency if the requested service is provided by a deactivated object
- Obviously activation and deactivation must be transparent to the client
- It is necessary to implement a "**persistence service**" to handle stateful objects

Parallelism

- Certainly we have to manage **real parallelism**, independently from the use of threads
- **Access to server objects must be controlled**

Communications

- The numerous cause of delay impose the possibility of using non-blocking calls invoking a service
- Often is also useful to use a form of multi-cast
- Request Synchronization (Synchronous, One way, extended rendez-vous, asynchronous)
- Request Multiplicity (Unicast request, Group request, Multiple request)

Failures

- Distributed objects have to deal with major probability of failures
- Partial failure (a new kind of failure)
- Different reliabilities available for distributed objects
  - Unicast request

    exactly-once, atomic, at-least-once, at-most-once, maybe
  - Group request and Multiple request

    k-reliability, totally ordered, best effort

Security

- **Distributed objects use the network for communications!!!**
- Centralized applications trust that the user will not make the session available to unauthorized users
- In distributed applications **each request** might be authenticated

Common problems

- Recurring problems not strictly related to the application logic
- Middleware can provide solutions known as “services”. We will see in brief four of them :
  - **Location**
  - **Life Cycle management**
  - **Object persistence**
  - **Transactions**

Location service

- **Object Naming**
  - a sequence of identifiers is bound to an object reference
  - Hierarchical structuring of name spaces (as DNS)
  - The object must be registered within a naming server that then can provide the reference to clients

    (CORBA Naming Service, COM Monikers, JavaRMI Registry)
- **Object Trading**
  - three main actors: **Trader**, **Exporter** and **Importer**
  - reference retrieved on the base of properties concerning provided functionalities and quality

Life cycle service

- CREATION:
  - The new operator is not available for remote construction
  - Client need reference to **Factory objects** (objects capable of create other objects)
  - Administrator influences distribution placing factories
- MIGRATION:
  - Moving or Copying (using a factory) an object from its current location
  - moved objects retain the same reference
  - more factories on heterogeneous platforms to solve heterogeneity machine code
- DELETION
- **Composite Objects Life Cycle Management**

Persistence service

- Three reason that can cause the necessity of storing object state information:
  - **deactivation**
  - **hosts have to be restarted**
  - **memory lacking**
- Java Serialization, CORBA Externalization, COM Structured Storage

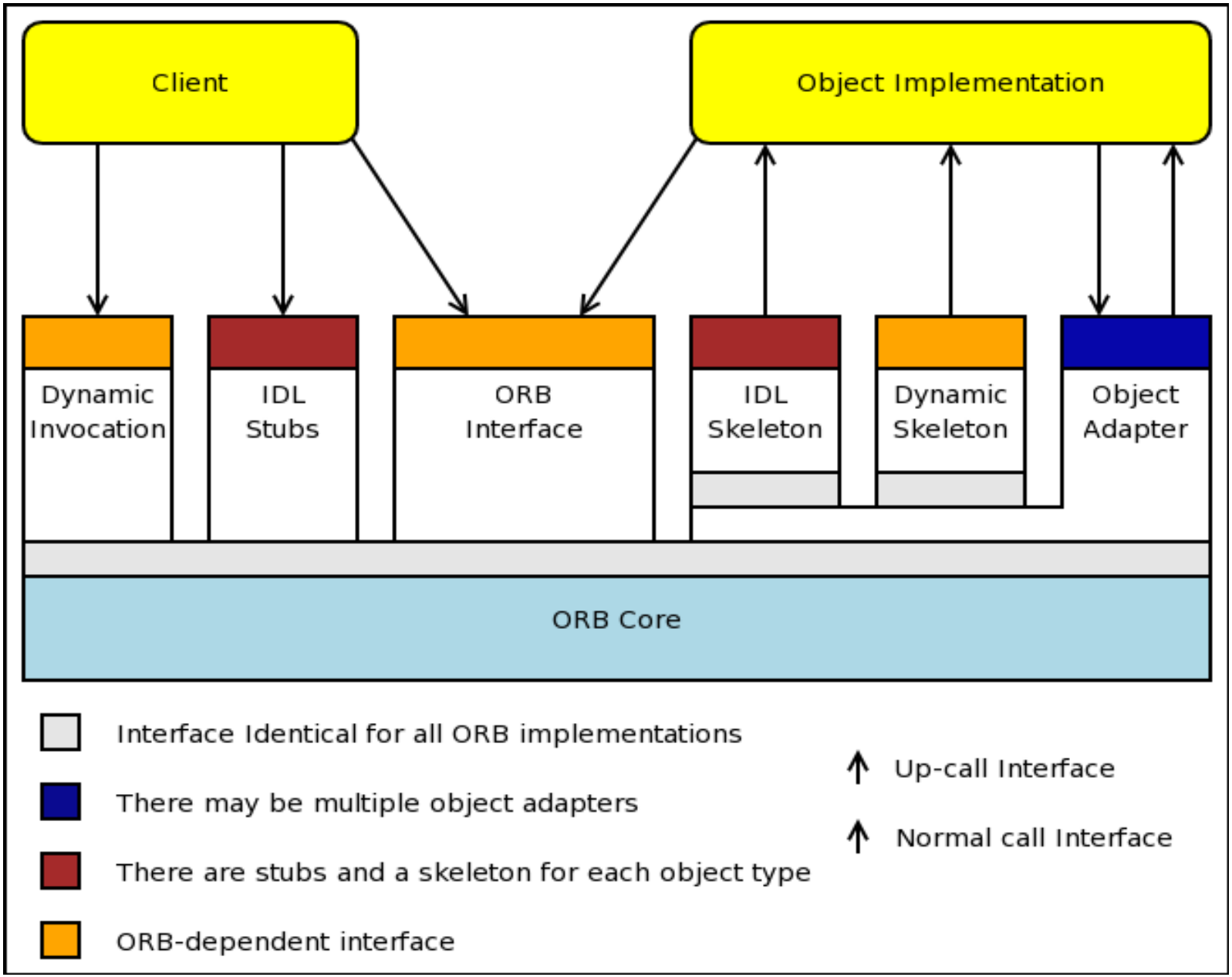
Transaction service

- Several object requests into a coarse grained one with ACID properties
- Middleware provide mechanisms for the implementation of the 2PC
- CORBA Transaction Service, Microsoft Transaction Server (MTS), Java Transaction API (JTA)

CORBA Short Overview

- **C**ommon **O**bject **R**equest **B**roker **A**rchitecture (CORBA)
- Open Standard defined by the Object Management Group (OMG) Last Formal Spec March 2004 (1152 Pages)
- Main Elements:
  - Object Model
  - ORB
  - CORBA IDL
  - CORBAServices, CORBAFacilities, CORBA Domain Interfaces
- http://www.corba.org/vc.htm (~220 companies committed)
  http://www.omg.org/technology/corba/corbadownloads.htm
  (~15 free CORBA implementation)

CORBA Architecture



CORBAservices

- Additional Structuring Mechanisms for the OTS
- Notification Service
- Collection Service
- Persistent State Service
- Concurrency Service
- Property Service
- Enhanced View of Time
- Query Service
- Event Service
- Lightweight Service
- Management of Event Domain
- Relationship Service
- Externalization Service
- Security Service
- Naming Service
- Time Service
- Licensing Service
- Trading Object Service
- Life Cycle Service
- Transaction Service
- Notification/JMS Interworking
- Telecoms Log Service

Outline

- Distributed Systems Basics
- Middleware generalities
- Middleware for Distributed Objects
- **Distributed Computing Models**

- **File Transfer**:
  - this was one of the first model trying to exploit the resources distribution. Following this paradigm the applications, that need data residing in another machine, make the log-on on it and then transfer the data. Therefore off-line the foreseen computations are performed
  - The distribution regards only the data
  - Applicable only with low load and low concurrency
  - e.g. e-mail

Distributed computing models

- **Client/Server**:
  - Client/server model is a concept for describing communications between computing processes that are classified as service consumers (clients) and service providers (servers)
  - Today the most used paradigm, therefore we will see major details in the following
  - **In this context C/S is a software architecture not hardware!!**

- **Peer-to-Peer (P2P)**:
  - following this paradigm more processes, located on different machines, cooperate to reach the solution of the problem. In different moments the process perform both server and client duties
  - Also consequence of the great underutilization of many resources linked to the net.

Client/Server model

- **Client**: process that requires services
- **Server**: process that provides services
- Three different logic levels of computation concerning:
  - **Graphical interface**
  - **Business logic**
  - **Data Management**

Client/Server model

- **Two-tier architecture**:
  - the business logic is divided between the two elements. We can have:
    - fat client and thin server
    - fat server and thin client
- **Three-tier and n-tier architecture**:
  - There is a tier for each of the logic levels
  - It is also possible to split the business logic in more than one tier (e.g. web applications)

- The J2EE example: