Conception of Information Systems
Lecture 7: CORBA

26 April 2005
http://lsirwww.epfl.ch/courses/cis/2005ss/

Outline

- Distributed Object Computing
- Overview of CORBA
- Server-Side Objects in CORBA
- Object Transaction Monitors

Distributed Object Computing

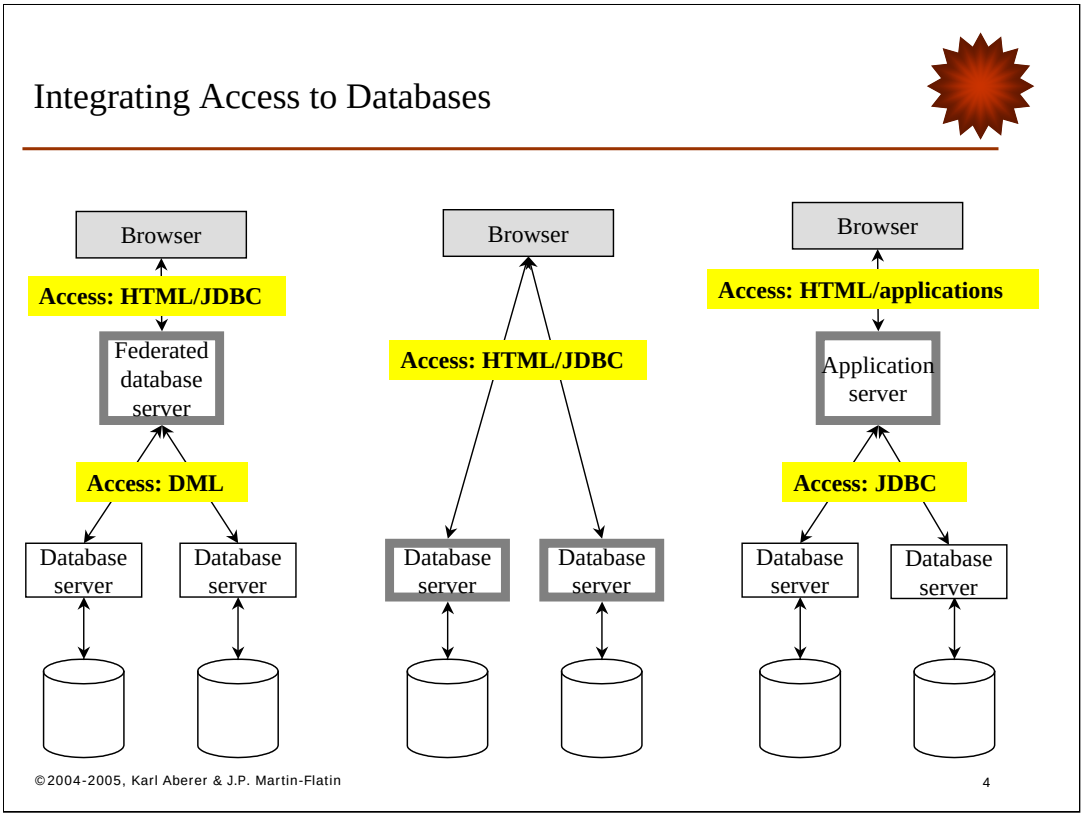
- *Goal:* It should be possible for object-oriented applications to be physically distributed over multiple machines
  - for scalability, robustness, etc.
- *Technique:* Enable an application (=client) to invoke methods of remote objects accessed via the network (=server objects)
- Main issues
  - global identification of an object
  - processing of distributed requests (communication, RPC)
  - implementation of server objects (possibly shared by many applications)
- Main approaches
  - DCE: OSF, open standard, platform independent (now passé)
  - CORBA: OMG, open standard, platform independent, language independent
  - RMI: Java, kind of open (platform independent but controlled by Sun)
  - DCOM/.NET: Microsoft, platform dependent

Distributed object computing is based on the idea that applications and objects should be enabled to invoke operations (methods) on any object that is accessible over the network as transparently as possible. It extends the mechanism of invoking remote functions through RPC, which is based on a procedural programming model, to the object programming model. When dealing with objects a first issue is object identification, which should be possible globally across the network. The processing of distributed requests on objects is analogous to RPC. As server objects on which methods are invoked typically are shared by many clients, the efficient implementation of server objects requires particular attention. The questions encountered to that extend are similar to those that are considered in the development of classical transaction monitors.

CORBA was the first standardization effort in order to enable interoperability among distributed objects and to support developers in developing distributed object applications. A lot of hope was put into CORBA to become a general basis distributed programming and distributed information management. In particular industry was investing substantially into large-scale CORBA projects (with mixed success). Nowadays CORBA is still a valid architectural alternative for distributed information systems development, but it appears that its main merits lie in the fact that it paved the way and introduced concepts for the distributed component architectures, such as EJBs (Enterprise Java Beans), which provide functionalities at higher level abstractions and in a more reusable form and Web services that are built around the Web infrastructure.

RMI is Java's distributed object programming model. Besides being bound to a specific programming language it is much less complex (and powerful) than CORBA.

DCOM is the Microsoft competitor to CORBA.

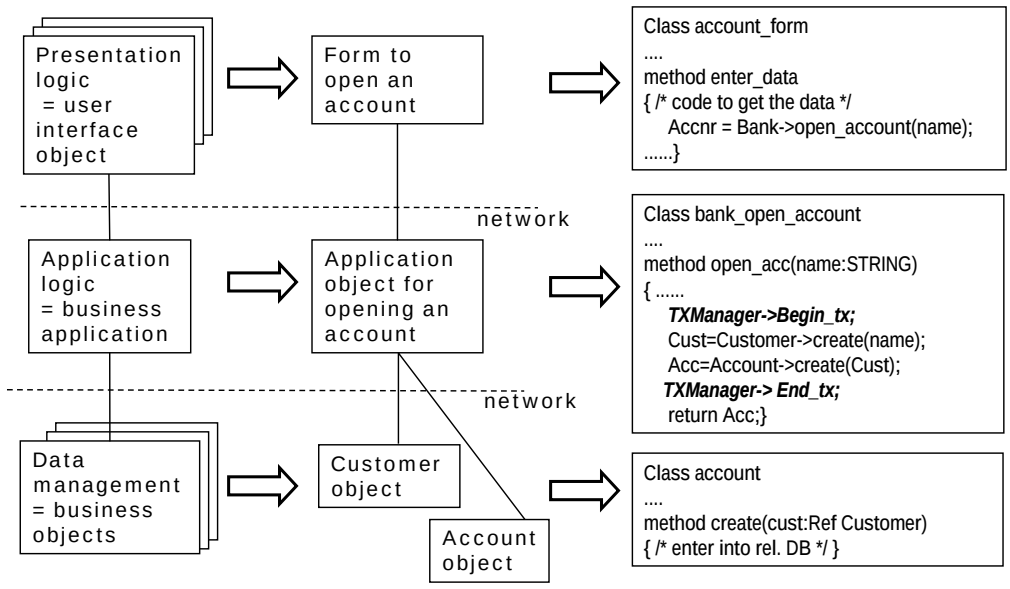


We have earlier studied the problem of accessing multiple databases at the same time. Two solutions have been discussed:

- Access through a federated DBMS. With respect to transactions the problem may occur that within the same transaction multiple databases are accessed at the same time, therefore we have what is called a "distributed transaction"
- Access through a user interface and application programs that are web-enabled. Here no distributed transactions occur since every database is accessed by a separate program.

However, we could easily imagine to use exactly the same mechanism that we used to Web-enable databases, i.e. Web-accessible Java applications, that access the DBMS via JDBC, for accessing multiple databases from within the same Java application. Thus we have found a different way to access multiple DBs, by integrating them at the application level. Now we can again have distributed transactions, and they need to be dealt with somewhere. The platform that will be doing this job is called an *application server*. This application server can also run (and in many cases has to run) on a computer (server) that is different from the database servers (remember: Java allows DB access over the network).

Example: Three-Tier Information System



Since 3-tier (and more generally n-tier) architectures are distributed by definition, it is natural to consider their implementation using distributed objects. It is important to keep in mind that the scope of distributed object technology goes beyond the support of distributed, heterogeneous information systems, but that they are one of the main applications of distributed object technology. For example, many real-time applications are developed based on CORBA. One can derive the importance of distributed objects for information systems from the fact that many of the services in CORBA, for example, are related to information management (e.g. transactions, persistence).

Thus one can say that distributed object standards are principally a *distributed application development platform* that in particular are used as *architecture for information systems integration*.

The example of a banking application shown demonstrates how a 3-tier architecture can be mapped onto an implementation that is based on distributed objects. For each of the three tiers, there exist distributed objects representing the corresponding functionality and accessing each other via (remote) method invocations. So, we have an object encapsulating the code for supporting the user interface of this application, that invokes the object that encapsulates the business logic. This object in turn invokes the (persistent) objects that represent the data stored in a database. Notice also, that the object that controls the business logic is invoking a transaction management service, which is implemented as another distributed object.

Overview of CORBA

- Object Management Architecture (OMA)
- Abstract CORBA model
- Functional components
- Inter-ORB protocols
- Object adaptors
- Interoperable object references
- Example of CORBA service
- Development process

Object Management Architecture (1/2)

- OMA = Object Management Architecture
  - Devised by Object Management Group (OMG)
  - Distributed object programming architecture (CORBA)
    - CORBA 2.0 specified in 1995 = the real start of CORBA business-wise
    - CORBA 3.0 = CORBA Component Model (CCM)
  - Standard services
- Deals with distribution
  - by supporting remote method invocation and global object identification
- Deals with heterogeneity
  - by providing a standard object-oriented Interface Definition Language (IDL)

OMA (object management architecture) is a standard architecture for distributed programming developed by the industry consortium OMG (Object Management Group). It is available on many system platforms.

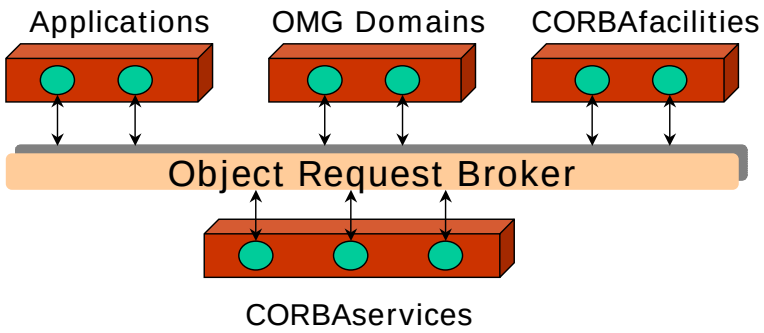
We can view the function of the OMA in two ways (according to our three-dimensional taxonomy on information systems integration)

-OMA hides distribution by supporting remote method invocation and global object identification. Thus it can be considered as an implementation platform for distributed information systems and in particular for multi-tier architectures (3-tier, n-tier).

-OMA provides a homogeneous application development layer. It allows to hide heterogeneity of underlying (legacy information) systems by providing a standard interface. This is done by encapsulating the legacy systems into distributed objects (comparable to wrappers) that provide their functionality through the standardized interfaces which are specified using the OMA interface definition language (IDL).

Object Management Architecture (2/2)

- OMA: two types of services
  - Standard low-level services for objects: CORBAservices
    - Life Cycle (create, move, copy, delete)
    - Naming, Concurrency, Persistency, Query, Security, Event, Notification, Time, Transaction, Trading, etc.
  - Standard high-level services for applications: CORBAfacilities
    - Horizontal CORBAfacilities (across business domains): Internationalization, Mobile Agent
    - Domain CORBAfacilities (OMG Domains): Air Traffic Control, Telecom, Genomic Maps, etc.



The main constituents of OMA are

-standard programming support through CORBA (common object request broker architecture): CORBA provides an object bus for making distributed objects interoperable over the network. The necessary functionality is provided by a so-called ORB (Object Request Broker). Since ORBs from different vendors are not using necessarily the same communication protocols there exists in addition a standardized protocol IIOP (Internet-Inter-ORB) for ORB to ORB communication over the Internet.

-standard service interfaces (horizontal): the most interesting for our purposes are the basic information system services (transactions, persistence)

-standard domain interfaces (vertical): these are interfaces for special application domains such as Finance, Healthcare, Manufacturing, Telecom, Electronic Commerce, and Transportation.

Common Object Request Broker Architecture CORBA

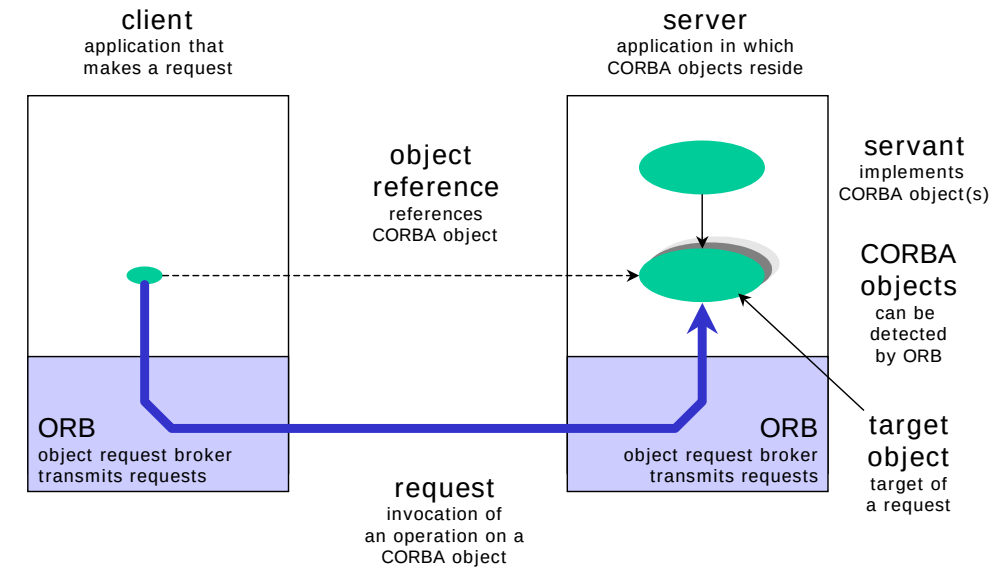
- CORBA is the core of the OMA architecture
- CORBA defines
  - Distributed object model and execution semantics
    - How are distributed objects identified?
    - How are they invoked over the network?
    - How are CORBA objects accessed upon invocation?
  - Architecture of the distributed object bus (Object Request Broker = ORB)
  - Interfaces among the different architectural components
  - Protocol for inter-ORB interoperability
  - Language for specifying object interfaces (IDL)
  - Language bindings
    - C, C++, Java, etc.

First we give an overview of the core of the OMA architecture, which is the Common Object Request Broker Architecture CORBA. This architecture consists of the following components, that are specified as part of the OMA standard document "Common Object Request Broker Architecture: Architecture and Specification" v 2.0, 1995.

- The distributed object model and its execution semantics: this part describes the model of how distributed objects are identified, of how they are invoked over the network, and of how their implementation is accessed upon invocation.
- The architecture of the distributed object bus: this part describes which are the functional components of an ORB and how they interact.
- The interfaces among the different architectural components of an ORB. These interfaces are in particular important for developers that use or provide services to certain of these components.
- The interface for ORB interoperability that is needed to enable the interoperability of ORBs from different vendors.
- The interface definition language that is a syntax for specifying object interfaces (both methods and attributes). It borrows from C++, however has mappings to various standard programming languages.
- The mappings to various programming languages that are called "language bindings".

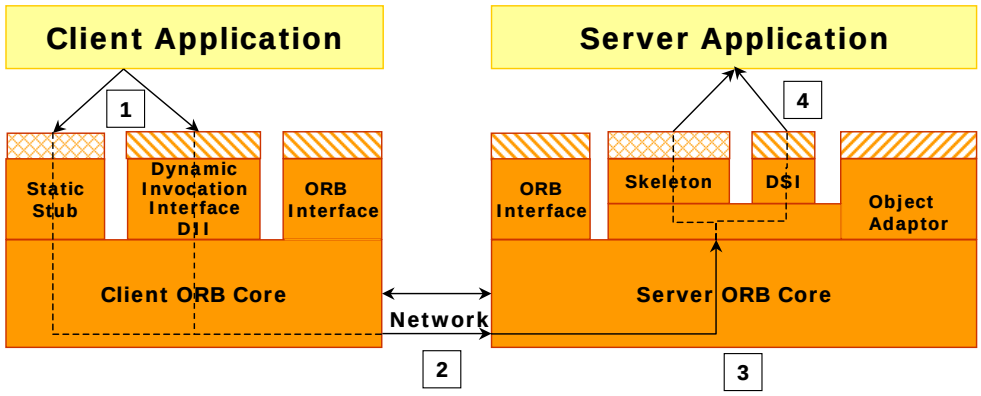
We will look now in more detail into each of these aspects of CORBA.

Abstract CORBA Model



The basic model of operation in CORBA is the following: a client that wants to invoke a service on an object through the network needs first of all the object's identifier, which is called the object reference. There exist different ways of how it can obtain such a reference, typically another CORBA service will be providing them. The invocation of a service (i.e. a method of the object) is transformed by the ORB into a request (marshalling), then transmitted by the ORB from the client node to the server node that has the object's implementation, and then transformed by the ORB (unmarshalling) into a method invocation on the server side representation of the object, which is a *CORBA object*. The ORB is a system layer (possibly distributed over the network) that performs the forwarding of the requests transparently for the client. The CORBA object has access to another object the *servant object*, which provides the actual implementation of the service. Thus the CORBA objects works like a proxy for the service provided by the servant object. The interface of the CORBA object is specified by means of IDL.

Common Object Request Broker Architecture:
Functional Components



This figure depicts the functional components that an ORB needs in order to forward a request from the client to the server. The client has access to three interfaces of the ORB:

- The *ORB interface*, which makes it possible to establish a connection to the ORB
- The *static stubs*, i.e. invocations to CORBA objects, which are statically compiled into the application code
- The *Dynamic Invocation Interface* (DII), which is used to dynamically invoke objects (i.e. decide at runtime which object is invoked)

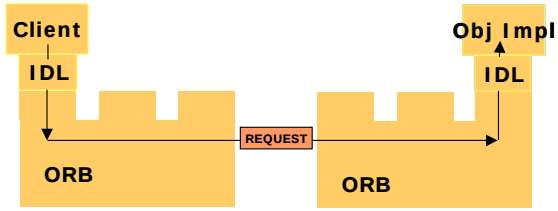
On the other side, the server application has four interfaces at its disposal:

- the *ORB interface*, again for establishing a connection to an ORB
- The *object adaptor*, which allows the application to register the implementation of a CORBA object at the ORB and provides to the ORB access to the object. We will discuss this interface later in much more detail.
- The *skeletons*, which are statically compiled (CORBA) objects providing access to the object implementation (servant)
- The *Dynamic Skeleton Interface*, which allows the server to serve untyped requests to objects (i.e. decide in runtime of which type the request is and how to access its implementation)

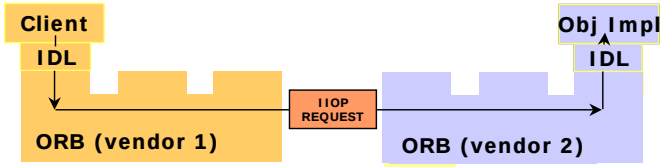
For processing a request, the client in step (1) either invokes a static stub method or uses the dynamic invocation interface to dynamically construct a request from an object interface definition. The Client ORB core forwards the request in step (2) to the server ORB core. There, depending on the object implementation (not on the way the request was constructed at the client) either the request is forwarded to the skeleton or it is dynamically associated with an object implementation using the DSI. Then in step (4) the request is sent to the server application (i.e. the servant) that implements the service.

Inter-ORB Protocols

- Different ORB processes communicate over proprietary protocols
  - Vendor specific



- GIOP = General Inter-ORB Protocol
  - defines transfer syntax
  - standard set of messages (request, reply)
- IIOP = Internet Inter-ORB Protocol
  - implementation of GIOP over TCP/IP

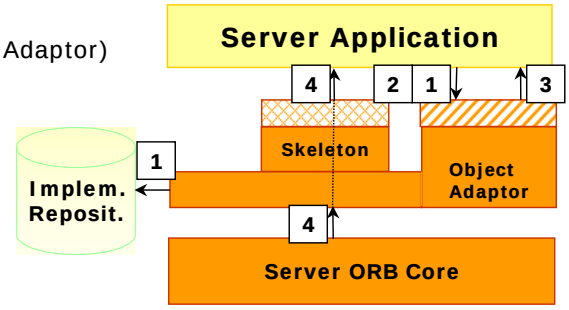


The fact that vendors are using proprietary protocols for having their ORBs to communicate turned out to be a major paradox when using CORBA for implementing distributed applications, presumably following a "standard". Since ORBs of different vendors could not communicate with each other, CORBA-based software was generally not interoperable.

To address this serious issue, a protocol was specified that standardizes also the interfaces for request interchange between ORBs, and thus allows different ORB implementations to communicate. More specifically, it standardizes the transfer syntax for requests and the messages that can be used. This is the GIOP, the general Inter-ORB protocol. A number of assumptions are made by the GIOP, such as connection-oriented transport (i.e. synchronous communication) and that connections are bidirectional, symmetric, reliable, and support byte-streams. The IIOP is the Internet-compliant implementation of GIOP and is the protocol that is frequently used in practice.

Object Adaptors: BOA → POA

- Interface between the object implementation (servant) and the ORB
- Functionality
  - Register and manage object implementations
  - Manage object references that allow clients to refer to objects
  - Ensure that target objects are instantiated by the server application
  - Dispatch requests from server-side ORBs to the servants of target objects
- Original design: BOA (Basic Object Adaptor)
  - C oriented (not OO)
  - Incomplete functionality
  - Required vendor-specific extensions
- Replaced by POA (Portable Object Adaptor)
  - Part of CORBA 2.3
  - Specified in 1999 (very late)



The object adaptors provide the interface between the object implementation (a.k.a. CORBA object, a.k.a. servant) and the ORB. One might ask why this functionality is not integral part of the server ORB core. The reason is that different types of object adaptors can be provided for different styles of programming and support different behavior of the implementation objects (e.g. related to performance). Putting all these options into the ORB kernel would overload it or unnecessarily limit the flexibility in interfacing to object implementations.

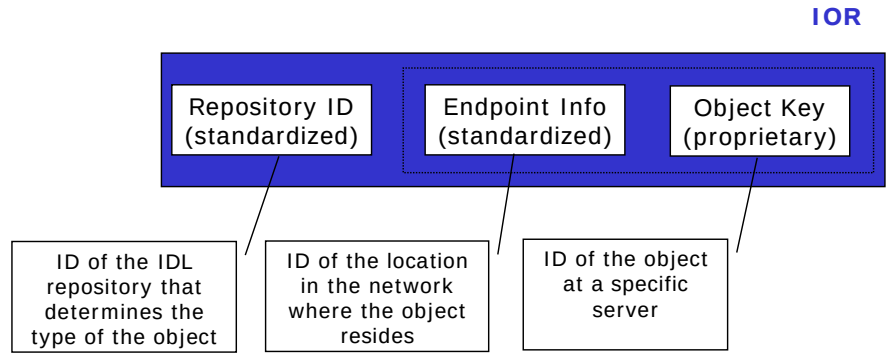
What are the functions of the object adaptor ?

- Applications need to be able to register implementations of CORBA objects (i.e. servants) at the ORB. To that end the object adaptors provide interfaces for registration and they access an implementation repository, where the information about the registered implementations is stored.
- The object adaptor generates object references for CORBA objects and thus make the objects accessible through an ORB. To that end it provides interface functions to the server application for providing the object key.
- When an object is requested by a client, the object adaptor needs to make sure that the CORBA object exists and an actual implementation object (servant) is instantiated at the server application. The server application has for that reason to implement callback functions in order to support the object adaptor interface in activating (resp. deactivating) the object implementations.
- Finally the central function of the object adaptor is to dispatch requests that arrive at the server-side ORB through the skeleton (in the static case) to the servant object that is provided by the server application.

The simplest form of object adaptor is the so-called BOA (basic object adaptor). It is generally provided with every ORB implementation but it is very limited in its functionality and it does not support an object-oriented style of programming. A more advanced type of object adaptors are the so-called portable object adaptors (POA) which provide in particular for the implementation of scalable applications, such as necessary information management, substantial flexibility to tune and optimize the implementation. They constitute also a predecessor to the component concept, that we will introduce later.

Interoperable Object Reference (IOR)

- Properties
  - Every object reference identifies exactly one object
  - Different IORs can identify the same object
  - IORs can be NIL and dangling
  - IORs are opaque (client does not see the content)
  - IORs are strongly typed, support late binding (subtypes) and can be persistent



Each CORBA object has at least one identifier, called *object reference*, that is used by the client in order to invoke methods on the object. A reference is uniquely associated with a CORBA object, but the same CORBA object may have multiple references. References may be NIL or they may point to CORBA objects that do not exist. The clients cannot see the content of the reference (opaque references) because it is the task of the ORB to use the object references in order to locate the CORBA objects and to transmit the requests. References are strongly typed, that is they are associated with a specific IDL object type. However, they can be bound to objects that have a subtype of the specified IDL type, and the binding to the subtype can be performed in runtime (late binding). References are also not necessarily bound to the lifetime of an application, i.e. they can be made persistent (together with the objects they identify).

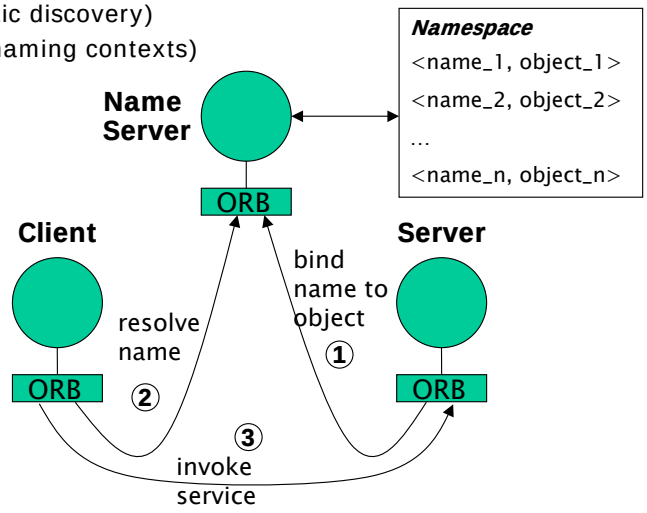
An object reference consists of three main parts

1. An identifier of the IDL which determines the type of the object (an IDL repository). For a specific ORB implementation the representation of this part of the identifier needs to be standardized, such that the ORB can determine the type of the object.
2. An identifier of the location in the network where the object resides. For the same reason as before this part of the object reference needs to be standardized for a specific ORB implementation.
3. The object key, identifying the object at a specific server. Part of this object key may depend on the implementation of the server object, and thus it is not standardized for an ORB implementation.

The endpoint info (network location) together with the object key determine uniquely the object, whereas the type information is needed to determine whether an invocation (static or dynamic) on an object is admissible.

Naming Service

- Ability to provide meaningful names to locate objects in distributed systems
- Server registers names, client must know them (no automated semantic discovery)
- Supports directories (naming contexts)



The Naming Service is a specific CORBA service providing to applications bindings among interpretable names and object identifiers. The name server maintains these bindings and may organize the names into different contexts (called directories). A server that is providing an object under a specific name has to register the name at the naming service (1). A client that wants to access the object has to know its name. Using the name it can access the name server and resolve the name to an object identifier. Using the object identifier it can then invoke the service on the CORBA object.

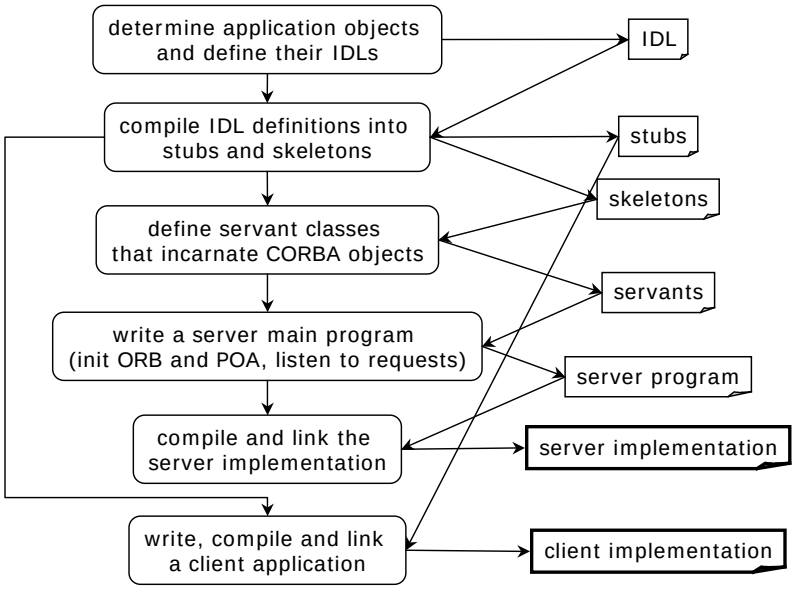
The Naming Service is, as any other CORBA service, defined through the interfaces of CORBA objects,

Naming Service IDL Specification

```
// File: CosNaming.idl, excerpt
module CosNaming
   {
   struct NameComponent {Istring id;Istring kind;};
   typedef sequence<NameComponent> Name;
   …
   struct Binding {Name binding_name; BindingType binding_type;};
   …
   interface NamingContext {
        void bind(in Name n, in Object obj)
                raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void bind_context(in Name n, in NamingContext nc)
                raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        Object resolve (in Name n)
                raises(NotFound, CannotProceed, InvalidName);
   NamingContext new_context();
   NamingContext bind_new_context(in Name n)
        raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
   void destroy() raises(NotEmpty);
   };
```


This is an example of an IDL specification for a CORBA object, the naming service. One can see that IDL allows to specify both attributes and methods for the objects, supports strong typing of attributes and method parameters, and supports standard built-in types and type constructors. We will discuss IDL not in more detail as it is not substantially different from other OO interface definition languages.

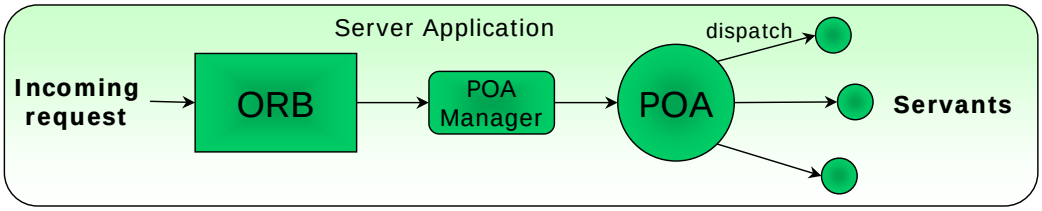
Development Process for CORBA Applications



This figure illustrates the basic development process that the implementation of each CORBA application has to follow. Notice that after defining the IDLs (i.e. the common interface definitions), the implementation of the clients and servers are performed independently. For the server side a substantial part of the development effort is dedicated to the binding of server-side implementations to the server-side CORBA objects. In addition, a server program needs to be defined that can listen to incoming requests and serve them.

Server-Side Objects in CORBA

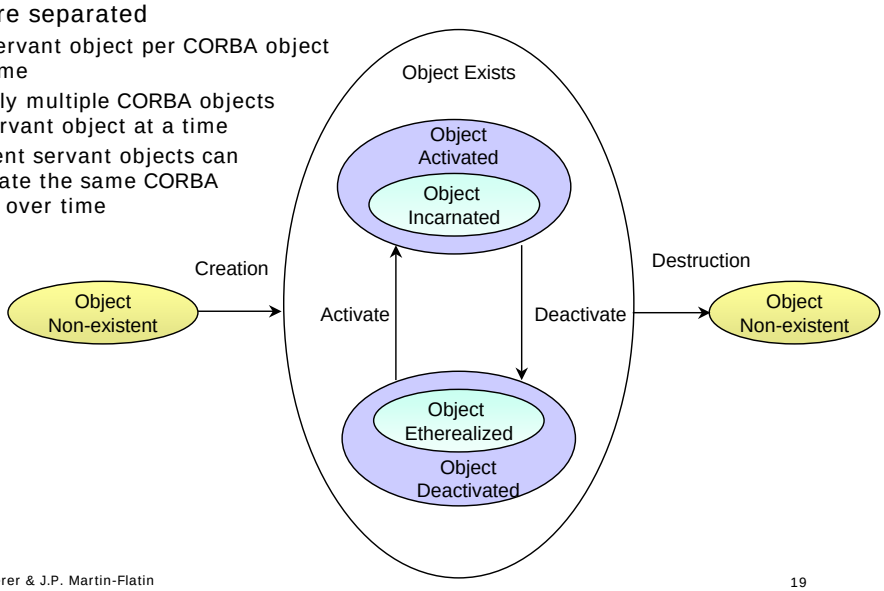
- Implementation of Servers using POA: Interface between ORB and programming environment
  - Creation of objects
  - Activation of objects
  - Dispatching of requests
- Multiple POAs can be used at the same time
  - Different characteristics
  - POA Manager dispatches the requests
- POA does not maintain persistent state for objects
  - Task of the application



The Portable Object Adaptor (POA) establishes the interface between the ORB and the programming environment: it manages the life-cycle of CORBA objects, consisting of creation of objects and their references, activation of objects, such that they are connected to a servant, dispatching of requests, and eventually their destruction. In order to obtain a larger degree of flexibility multiple portable object adaptors can be used at the same time, each of which can exhibit different characteristics related to the type of object management and the resulting performance desired. Since multiple POAs are used at the same time, there exists a POA manager that knows them and knows of how to dispatch the incoming requests to the right POAs such that the requests arrive at the right CORBA objects and servants. An important design decision, that has been made with respect to POAs, is whether they should manage a persistent state for the objects. This would be in principle reasonable since this aspect is closely related to the object lifecycle management. However, this is not the case in order not to hamper the POA's generality and is left as a task to the application implementing the object.

Life Cycles of CORBA Objects and Servant Objects

- Life cycles of CORBA and servant objects are separated
  - One servant object per CORBA object at a time
  - Possibly multiple CORBA objects per servant object at a time
  - Different servant objects can incarnate the same CORBA object over time



This diagram shows the life cycles of CORBA objects and servant objects as they are managed by the POA. One can see that the POA allows to decouple the life-cycles of the CORBA objects and the servant objects. One CORBA object can only be incarnated (=implemented) by one servant object at a time, but servant objects can be shared by multiple CORBA objects, even at the same time. CORBA objects can also be dynamically attached ore detached from their servant objects. This is called the *activation* or *deactivation* of the CORBA object. A CORBA object that has terminated the connection to its servant is called *etherealized*. The POA provides the necessary interfaces to the application in order to control the object life cycle as described.

Request-Related POA Policies

- Mapping objects to servants
  - Servants can incarnate single or multiple CORBA objects
- Mapping requests to servants
  - create servants upon activation or reuse servants
- Persistent and transient objects
  - Persistent objects exist independently of the actual running of any server process
  - Transient objects exist during the lifetime of the POA
- Retaining object-servant associations
  - Determines whether the associations are stored in the object association table or are supplied by the application with every call
  - Tradeoff: performance vs. memory consumption
- Allocation of requests to threads
  - Different policies for assigning processing threads to requests are possible: thread-per-request, thread-per-object, thread-per-POA, thread-pooling
  - The POA allows the ORB to implement a threading policy

The POA provides the possibility to develop scalable, high-performance servers by allowing a fine-tuned resource control for server objects. The control strategies are provided by so-called *policies*. Each policy addresses one aspect of controlling the mapping of CORBA objects to servants. Each POA can use a different policy. The ORB uses the object reference (which contains type information) to determine the right POA for processing a request to a CORBA object. By default there exists a root POA that implements a default policy. Policies themselves are represented as CORBA objects that are accessible through the local ORB. The choice of policies depends on application characteristics, such as space-time tradeoffs (number of objects to be supported, expected rates and durations of requests), the use of underlying persistent storage (need to support user-assigned object identifiers for persistent objects), available resources and services of underlying system, or access to applications that are wrapped as CORBA objects.

A basic decision is whether servants are shared by CORBA objects. Sharing of servants by multiple CORBA objects is analogous to sharing of processes for multiple client requests in transaction monitors. In transaction monitor terminology, a CORBA object corresponds to a service and a servant corresponds to the servers of a server class. Having separate servants for different objects is most appropriate in the case where only a few transient CORBA objects exist, whereas sharing of servants makes sense where CORBA objects are, for example, related to persistent data items stored in a database. Sharing of servants is in that case possible since the state of the object is not kept in the servant process but in the database. As a result not for every data object a separate servant process needs to be instantiated.

For many applications static object creation at startup is sufficient. Objects in information systems correspond frequently to database objects and creating for each of them a separate CORBA object would not be efficient. Therefore the POA allows to create CORBA objects upon actual requests to them only.

Persistent objects exist independently of the actual running of any server process. This is the original model of how objects are managed in CORBA. The transient objects were only introduced for efficiency purposes with the POAs. Transient objects exist only with the lifetime of the POA and are sufficient for many applications.

Retaining object-servant associations allows to optimize performance. If many such associations are kept the active object table holding these associations becomes more memory consuming. Not retaining them increases processing overhead, as the association has to be established for each request anew.

CORBA Transaction Service

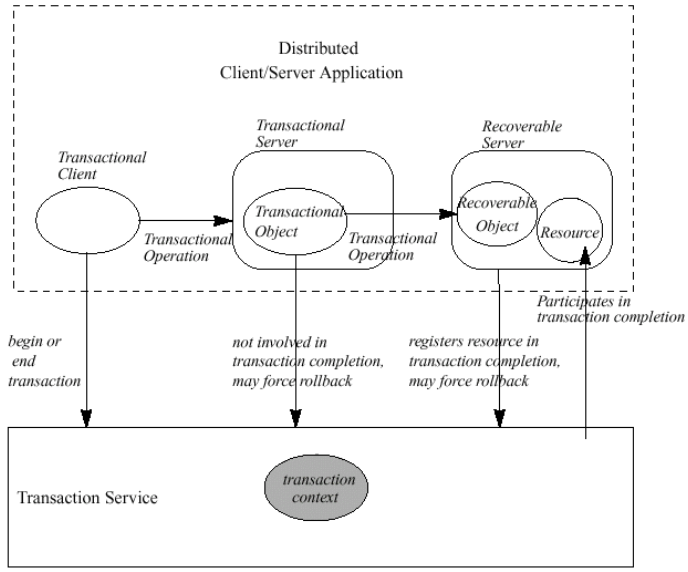
- CORBA counterpart to the X/Open Distributed Transaction Processing (DTP) architecture
- Defines interfaces that allow distributed objects to participate in an atomic transaction
- Supports standard transaction models
  - Flat transactions
  - Distributed transactions (2PC)
  - Nested transactions
- Designed for use in a transaction monitor

After having introduced the basic working of the CORBA architecture and in particular of the ORB and POAs we turn now our attention to another aspect of the OMA, the horizontal CORBA services. They cover in particular the functions that are found otherwise as components of application or database server architectures, such as transaction monitors or database management systems. Of particular interest for distributed information management is the transaction service, which constitutes the CORBA counterpart to the X/Open Distributed Transaction Processing (DTP) architecture. In the following we introduce the transaction service for two purposes:

- In order to give an example of how a typical horizontal CORBA service looks like.
- In order to understand the specific concepts for implementing distributed transactions that the CORBA transaction service introduces. Some of the technical aspects will be of importance for our subsequent discussion of component technologies (such as Enterprise Java Beans).

The specification of a CORBA service, and in particular of the transaction service, consists essentially of a set of interfaces that are defined for all the (distributed) objects participating in the service, i.e. in a transaction. Of course this specification of interfaces is based on an underlying processing model. In the case of the CORBA transaction service, the underlying model is compatible with the X/Open DTP Model. It supports a number of standard transaction concepts, including flat and distributed transactions, which we have introduced earlier in the part on transaction management. In addition, the transaction service supports the concept of nested transactions, which allow to structure transactions in an hierarchical manner, such that subtransactions can independently commit or rollback. The parent transaction is always committed when all its subtransactions are committed. This allows to do rollbacks of subtransactions only, and thus can offer advantages as compared to flat transactions, because in case of failure not the complete transaction needs to be aborted. The design of the transaction service has been guided by the intention to allow its implementation in a transaction monitor environment.

Basic Architecture



For understanding the transaction service specification one needs first to understand the relevant entities (objects) that participate in a distributed transactional application. To that end one distinguishes three types of objects: transactional clients, transactional servers, and recoverable servers. One can think of these three types of object as corresponding to the three layers of a three-tier architecture.

•Transactional clients are objects that can invoke calls to transactional objects and begin or start a transaction.

•Transactional servers contain (multiple) transactional objects. A transactional object can participate in a transaction and refer to some persistent data indirectly. For executing transactions the transactional context is transferred to them, as with transactional RPC.

•Recoverable servers contain (multiple) recoverable objects. A recoverable object registers a persistent resource (e.g. a database). Recoverable objects are transactional objects with additional properties. Since they represent persistent data objects, they must participate in the Transaction Service protocols (commit and rollback).

Both transactional objects and recoverable objects can participate in transactions and force the rollback (abort) of a transactions, but only recoverable objects need to participate in the commit processing. Transactional clients cannot distinguish transactional from recoverable objects.

The scope of a transaction is defined by a *transaction context:* it refers to a transaction (by means of a transaction identifier) and to transaction parents in nested transactions. The scope also contains a timeout value. The transaction context (object) is managed by the transaction service and passed with each invocation of a transactional operation.

Invoking Transactions in CORBA: Two Styles

- Indirect management
  - Transactions are started and ended explicitly
    - In C, via function calls
    - In Java, by invoking methods of a pseudo object `Current`
  - Advantage: easy to use
  - Disadvantage: coarse-grained control of transactions
- Direct management
  - The transaction objects implementing the transaction service are explicitly accessed
    - e.g., by creating a transaction control object
  - Advantage: fine-grained control of transactions
  - Disadvantage: more complex to use

CORBA offers different implementation styles for transactions: With respect to the invocation of transactions one distinguishes indirect from direct management. The indirect management corresponds to the way of how transactions are invoked in procedural programming languages, by explicitly starting and ending transactions through function calls, which are methods of a pseudo object Current. With direct management the transaction objects implementing the transaction service are explicitly accessed. For example, by creating a transaction control object a transaction is implicitly started. The advantage of this style of transaction management is, that it allows a finer-grained control of transactions, but it is more complex to use.

Propagating Transaction Context in CORBA: Two Styles

- Implicit propagation
  - The transaction context is propagated transparently by the ORB
    - Similar to TRPC, except that it is done by the ORB instead of a transaction monitor
  - Advantage: easy to use
  - Disadvantage: the ORB must support this functionality
- Explicit propagation
  - The transaction context is propagated explicitly by the application (i.e. developer)
  - Advantage: no changes required in the ORB
  - Disadvantage: complex to use
- Transaction invocation styles and context propagation styles are independent of each other
  - They can be combined arbitrarily

Similarly, for the transfer of the transaction call with a method invocation (corresponding to transactional RPC), the transaction context can be explicitly or implicitly propagated. The implicit propagation corresponds to the classical approach that we have already seen with transactional RPC. It requires that the ORB is able to support this functionality (similarly as a transaction monitor, which handles RPC requests, propagates "implicitly" the transaction context). This also shows that the transaction service is not a service that can be provided independently of the ORB, but requires extensions of the ORB implementation itself. This is a special feature of the CORBA transaction service. With explicit propagation the forwarding of transaction context is delegated from the ORB to the developer. The developer is responsible to extend the IDL interfaces of his transactional objects, such that they allow to include the transaction context as an explicit parameter into the method interface. Of course he has then also to provide the necessary values for these parameters in a method invocation. The trade-offs are similar as for direct/indirect management.

The two aspects of transaction management and context propagation are independent of each other. Both styles can be arbitrarily combined. The example gives an impression of the resulting different implementation styles:

With indirect management and implicit propagation the transaction is started by sending the begin() method to the Current object and invoking the method within the transaction boundaries. This corresponds exactly to the style of programming that is used also in the X/Open model and in procedural languages. With direct management and explicit propagation the application first creates the transaction object (Control) obtains from it the coordinator object and passes it then as a parameter to the makeDeposit method. The transaction is ended by invoking the commit() method that is provided by means of the Terminator object.

Transactions in CORBA: Code Examples

- Indirect and implicit transaction implementation

```
org.omg.CORBA.Object crtRef =
  orb_.resolve_initial_references("TransactionCurrent");
Current txn_crt = CurrentHelper.narrow(crtRef);
txn_crt.begin();
...
// the client issues requests, some of which involve transactional
objects;
BankAccount1->makeDeposit(deposit);
...
txn_crt.commit(false);
```

- Direct and explicit transaction implementation

```
CosTransactions::Control c = TFactory->create(0);
CosTransactions::Coordinator co= c->get_coordinator();
CosTransactions::Terminator t = c->get_terminator();
...
BankAccount2->makeDeposit(deposit, co);
...
t->commit(false);
```

Other CORBAservices

- Persistence Service
  - Make the state of CORBA objects persistent
  - Server-side service: invisible to clients
  - Support interfaces for efficient access to large numbers of objects
  - Reuse existing transaction service
  - This service influenced the persistence services offered by the component technologies that came afterward (e.g., EJBs)
- Query Service
  - Provides interfaces for query operations on object collections
  - Operations can return object collections
  - Visible to clients
  - Language independent, but support for SQL92 or OQL required
- Concurrency Service
  - Provides several locking types in order to ensure serializability in concurrent access of multiple clients to the same resource
  - Intentional read, read, upgrade, intentional write and write locks
  - Supports protocols for strict two-phase locking

We mention other horizontal services that are related to the access of persistent data.

The persistence service allows servants to persistently store their objects in a standardized way. Differently to the transaction service the persistence service is not visible at the client, but is just used for the implementation of servants and provides thus interfaces to the servants and POAs. The importance of the persistence service lies in the fact that its specification has influenced the development of persistence mechanisms for component technologies, such as EJB that we will introduced in the following.

The query service is on the other hand intended to be used by the clients and supports the access to large object collections. It is primarily intended to allow the use of declarative query languages, such as SQL or OQL, from within object-oriented applications.

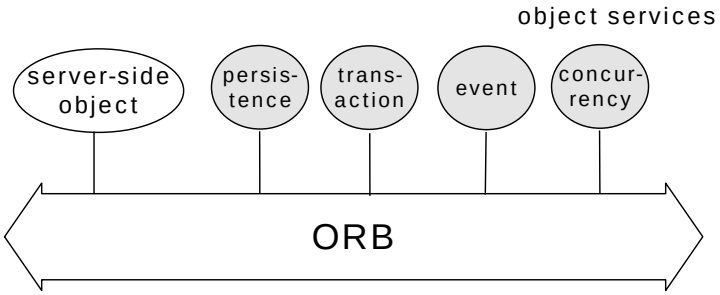
The concurrency service allows to implement locking mechanisms for resources. This is particularly useful for non-database resources (e.g. files). It supports different lock types and the strict 2PL protocol.

Summary (Server-Side Objects in CORBA)

- Distributed object architectures can be used
  - As distributed programming environment
  - Heterogeneous system integration platform
- Scalable CORBA servers require the flexibility of POA
  - Association of CORBA objects with servants
- CORBA Services provide all essential functions of information systems
  - Transactions, persistency
  - Support explicit and implicit programming styles

Object Transaction Monitors

- Implementing CORBA Server Objects
  - Provide calls to the horizontal services
  - Orchestrate those calls within the implementation
- The problem of server-side distributed object development
  - change of runtime behavior requires reprogramming
  - resource management needs to be implemented by developer
  - complex system-specific code, difficult maintenance
  - repeating implementation patterns



As we have seen in the previous part of the lecture, implementing a server-side CORBA object requires the developer to consider a number of aspects. He needs to establish policies for the object adaptor in order to control the object life-cycle and he needs to use horizontal CORBA services properly. Different parts of the implementation, such as client application, servants or persistent resources need to be combined in a way such that they are compatible with each other (e.g. when declaring an object to be persistent the developer needs to provide the necessary mappings to make objects persistent, has to implement the Resource interfaces etc.). This requires a thorough understanding on the side of the developer in order to coordinate correctly the interaction of the application objects and services.

All of this leads to a number of problems, in particular for developing complex applications:

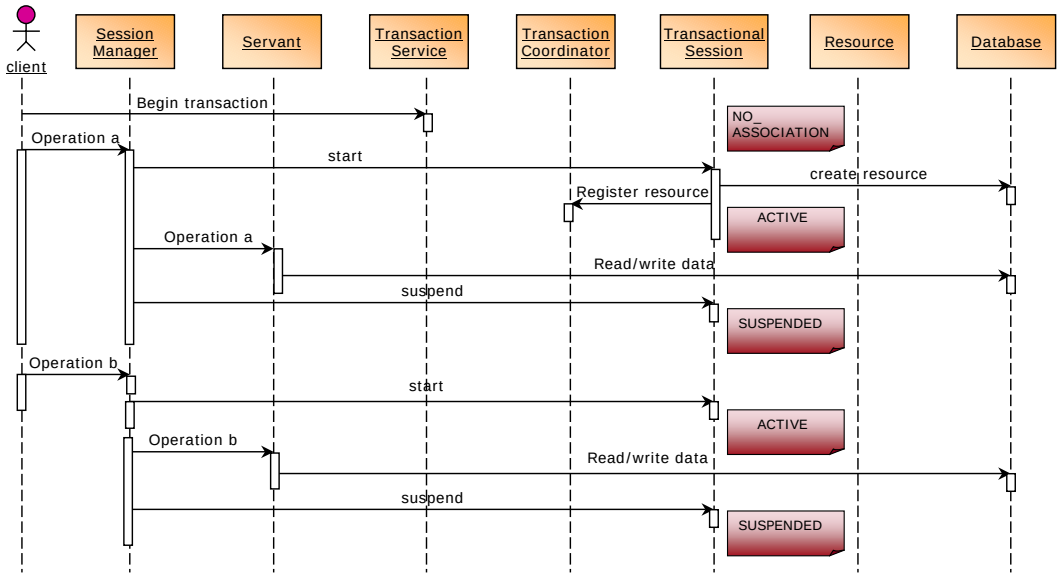
If a change in runtime behavior is desired, e.g. by changing POA policies or transactional boundaries, reprogramming and recompilation of the applications is required.

The developer has to be aware of the resource requirements, for all potential future uses of the application. Once an implementation is compiled, the runtime behavior can no more be changed.

The coding is, as elaborated before, a complex task and therefore error-prone. Changes can often not be performed in isolation, but require the adaptation of multiple parts of the implementation. Therefore maintenance becomes difficult. It requires substantial skills by the developers, which distracts their attention from the implementation of the business logic.

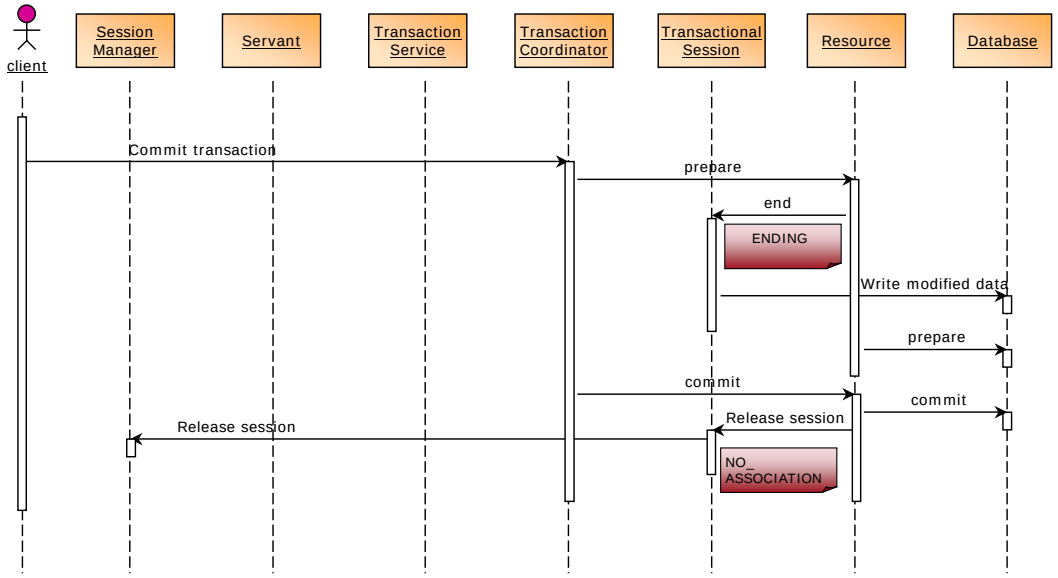
The implementation patterns for the system specific code are repeating, and it is not economical to repeatly perform the same system-specific implementation tasks for similar types of applications.

Example: Accessing Persistent Objects (1/2)



The difficulty of using the different horizontal CORBA services in a coordinated fashion, in particular those related to persistent data management, is illustrated by this figure. Without understanding all the details one can see that for accessing a persistent resource, such as a database, a number of CORBA objects provided by the different services need to be accessed. Some of the services and their corresponding objects, such as transaction service or resource we have introduced earlier. It is easy to imagine that following this complex interplay of multiple services poses to the developers substantial problems. This was one of the main reasons why the original intention to directly use these services for implementing scalable server applications encountered many difficulties and did not prevail in practice.

Example: Accessing Persistent Objects (2/2)



Server-Side Objects vs. Transaction Monitors

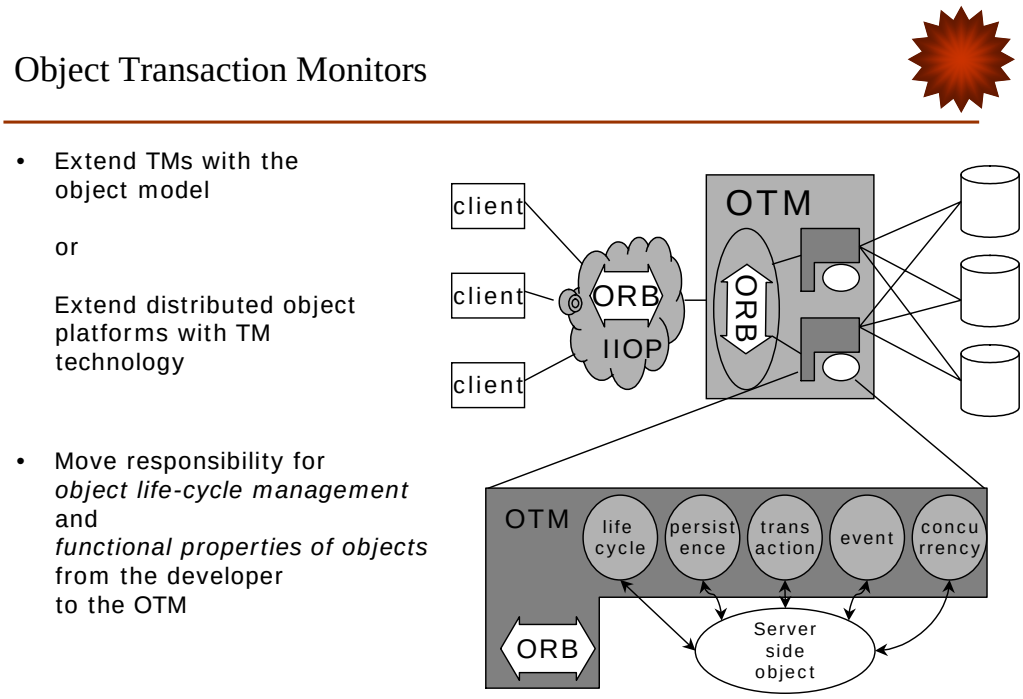
- Commonalities
  - 3-tier architecture
  - Transparency (heterogeneity and distribution)
  - Distributed transaction and communication management for applications with many clients and servers
  - Services: transaction, persistence, communication services, etc.
- Differences
  - Programming style: object-oriented vs. procedural
  - Explicit vs. implicit service representation
  - TM provide efficient process, reliable, scalable, performant, secure
  - Parametric resource configuration

Server-side objects play in an information system implemented on top of a distributed object platform the same role as applications running on a transaction monitor in a conventional 3-tier architecture. In both cases services, typically for managing access to persistent resources, are used by many clients in a multi-tier architecture. Both, the ORB and a transaction monitor, provide similar services, like distributed transaction and communication management and support the application developer in dealing with the problems of distribution and heterogeneity. In fact, the distributed transaction processing concept in CORBA is based upon the X/Open standard, that had been developed for transaction monitors.

Historically, there was from a practical perspective however a large gap among TM and CORBA platforms. At the first glance one recognizes a clear difference in the programming style, which is procedural for TMs and object-oriented for CORBA. This results also in the different ways of how services, such as transactions, are represented. In TMs these are provided as integral part of a system architecture and are typically accessed in an implicit manner (e.g. implicit transaction context propagation), whereas in CORBA these services are represented as objects in their own right and enable thus more explicit ways of dealing with them (e.g. explicit transaction context propagation).

The more important gap between the platforms was however much more related to the different functional features and performance. TMs are by construction high-performance transaction processing platforms, and thus exhibit all the desired functional features, such as efficient process management, high reliability and scalability, high performance and security, which the early CORBA implementations clearly didn't exhibit. Another very important difference is that TMs allow to adapt certain functional attributes, e.g. those related to resource usage, in runtime and therefore to tune the performance of an application in runtime (remember: the administration of a TM is a non-trivial task). This capability was lacking in early CORBA implementations, and any change in runtime required code changes as already pointed out.

Object Transaction Monitors



- Extend TMs with the object model

  or

  Extend distributed object platforms with TM technology

- Move responsibility for *object life-cycle management* and *functional properties of objects* from the developer to the OTM

Thus there was the quite obvious idea to combine the virtues of the two approaches of distributed object computing and transaction monitor technology by combining the two technologies. One can view this combination in two ways, and both ways reflect actual evolution paths that had been taken.

1. The extension of transaction monitors by the object model (based on CORBA): this approach has been taken by transaction monitor vendors, such as Tuxedo. They took their proven, robust technology and extended the architecture in way that they turned their products into full-fledged ORBs.

2. The extension of distributed object platforms by TM technology: this was the approach taken by CORBA vendors, such as IONA. They integrated into their ORBs increasingly features and technologies that are known from traditional transaction monitors.

In both cases the result were object transaction monitors. Object transaction monitors (OTMs) however introduce a new aspect: in order to combine the approach of having the important services, such as transactions, hard-wired into a system architecture such as an object transaction monitors, the responsibility of making use of those services needed to be moved from the developer to the OTM. Thus the original CORBA services are no longer made directly available to the developer, but they are internally used by the OTM. This is necessary, since the OTM provides implementations of these services that can take advantage of the specific characteristics of their architecture. The result is that an OTM turns into an environment (which is called *container*) where server-side objects, that implement the business logic, reside and make use of the OTM services via a high-level interface. In particular the OTM will be in charge of managing the object life-cycle. The functional characteristics of the object management, such as the properties that would be specified in POA policies, transaction demarcations or performance-related properties related to the specific OTM architecture, e.g. for process management, are supplied by the developer in a declarative manner, and can be changed in runtime.

In this way the OTM provides the following functions:

Integrated services: 2PC Transactions, Persistency, Resource access, Events, Security, Queued messaging, Object life cycle management

Communication: CORBA, DCOM (the Microsoft distributed object model), stubs, skeletons, object adaptors (as with classical ORBs)

Resource Management: Load Balancing, Message Routing, Multithreading (as with classical TM)

Developing Objects for the OTM

- The OTM provides more abstract interfaces for using the services
- The developer provides
  - Specification and implementation of business objects
    - IDL interfaces and object implementations
  - Support functions for the OTM for life-cycle management
    - *Object activation and deactivation* methods (e.g. managing persistence)
    - *Factory objects*: creation of object references
  - Specification of functional properties of the objects
    - *Deployment descriptor* (metadata)
- The OTM manages
  - object lifecycle (using POA and developer-supplied functions)
  - requests to and from objects
  - functional properties (according to the deployment descriptor using CORBA services)

For the developer an OTM presents itself as a quite different development environment as compared to a distributed object platform. The development process for OTMs differs substantially from developing server objects using directly the POA and CORBA services, since the OTMs provide more abstract interfaces for using these services.

The main focus of the developer is on the business logic of the objects as expressed through IDL interfaces and object implementations. In addition, he has only provide support functions that the OTM needs in order to deal with the application specific aspects of the object life-cycle. Most important this concerns the identification of objects, if it requires application-specific identification schemes. In addition, the developer has for example the possibility to provide application specific operations that should be performed upon activation or deactivation of objects. All the functional properties of the object, that should be guaranteed by the OTM, are provided by a so-called *deployment descriptors*, which are a declarative specification of the functional properties and can be changed in runtime.

Based on these specifications the OTM takes care of all the rest.

• It performs the object management: creates/destroys objects, automatically invokes activate and deactivate methods, allocates the required resources (all of this for example using a POA).

•It intercepts calls from/to other objects, forwards them.

•And it guarantees the functional properties. For example, it automatically invokes transaction services when accessing persistent objects within transactions.

Example: Deployment Descriptor

```
<?xml version="1.0"?>
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">
<M3-SERVER server-implementation="com.beasys.samples.BankAppServerImpl"
server-descriptor-name="BankApp.ser">

<MODULE name="com.beasys.samples">
<IMPLEMENTATION
      name="TellerFactoryImpl"
      activation="process"
      transaction="never"
/>
<IMPLEMENTATION
      name="TellerImpl"
      activation="method"
      transaction="optional"
/>
<IMPLEMENTATION
      name="DBAccessImpl"
      activation="method"
      transaction="optional"
/>
</MODULE>
. . .
</M3-SERVER>
```

implemented by POA

implemented by transaction service

For illustration, we give just one example of how in such an OTM environment the developer can specify, by using a deployment descriptor, the functional properties of an object in runtime, after he has specified its business logic. In this example, taken from a banking application, the developer has supplied implementations for certain business methods for accessing a teller machine. The TellerFactoryImpl is used to create a teller machine object that receives these methods, the TellerImpl is the implementation of the teller object machine interface, and the DBAccessImpl provides an application specific way of making the teller machine object persistent in a database.

There are two functional properties that are specified, which are related to the object activation policy and the transactional properties. With respect to the object activation policy for the TellerFactoryImpl object it is declared that it should stay activated, once it has been activated, till the process activating the object ends. The other two methods should be deactivated after the execution of a method on them (in order to free the resources for other objects). A third possible object activation policy would be to bind the activation time to a transaction (Note: activation of objects corresponds to the notion that we have introduced for the POA earlier).

The transactional properties relate to the different ways of how methods on the objects can be executed as part of transactions: The possibilities are

- Always (either the method is invoked within a transaction or a transaction is started)
- optional (only executed as part of a transaction if invoked within transaction)
- never (not invokable within transaction)
- ignore (not taking part in a transaction even if invoked within the transaction)

One can also see that the deployment descriptor is packaged as XML document.

Components

- Components
  - self-contained, reusable, portable, configurable SW pieces
  - equipped to live within a specific environment = *container*
  - standardized interface to the container
  - introspection, configurability, packaging
- Component model: Contract between components and container
  - a set of interfaces and classes that a component uses and supports
  - Difficulty: combining ease of use with flexibility
- Distinctions to be made from the software engineering viewpoint
  - Classes and Objects
  - Design Patterns
  - Frameworks
  - Components

From a software engineering perspective the server-side objects that are developed for an OTM and that are taking advantage of the environment of the OTM are what is called *components*.

Components are characterized by the fact that they are self-contained pieces of software that can be deployed (without further implementation effort) in a specific environment, which is the container. The idea is that such an approach will allow to assemble complex software constructs from smaller pieces - the components -, which have proven functionality, can be reused in many different application contexts, and therefore also can be easily adapted to different application environments. This is also a reason why the deployment descriptor plays such a central role. The components interact with the container through a standardized interface. They support introspection, i.e. they allow tools to discover their interfaces, and configurability, for customizing their properties for the applications. A packaging mechanism is provided, that allows to bind together all the necessary parts of a component into one file, that can then be *deployed* on a container.

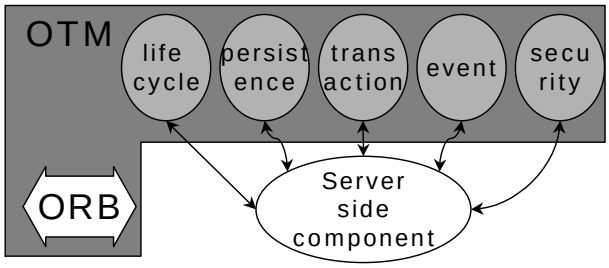
A *component model* describes the interface between components and the container. We may consider it as the language in which contracts between components and containers are established. The difficulty is to find for the component model a tradeoff between being sufficiently flexible and easy of use. For example, the generic CORBA model was arbitrarily flexible but very difficult to use.

Components are different from other paradigms for increasing reusability from software engineering.

Objects are encapsulated pieces of software that make up part of components, but are not self-contained. *Patterns* are proven solution methods that are described *textually* according to a specific method. *Frameworks* are partially implemented solutions of which the code must be completed.

Server-Side Components

- Client-side (e.g. Java beans) and server-side components
- Server-Side Components
  - serving multiple clients on a middle-tier server
  - transactional, persistent, secure, high-performing
  - Components once developed can be deployed on every application server (portability)
- Object Transaction Monitors
  - Are containers for server side components
  - Orchestrate the component interactions with services
  - Support declarative specification of functional properties



On the client side the component concept has been very successful for user interface development e.g. with the introduction of JavaBeans. The server-side objects running on OTMs are, following the characterization given for components before, clearly also components. Thus we will in the following no longer call them server-side objects, but server-side components. Their main properties are summarized on this slide.

Summary (Object Transaction Monitors)

- Object Transaction Monitors integrate the following three concepts

  1. Distributed Object Computing (CORBA)
  2. Transaction Monitor Technology
  3. Software Components

  in a non-standard fashion !

Summary (CORBA)

- Object Transaction Monitors
  - Combine the worlds of distributed object computing and transaction monitors
  - Provide runtime environments (containers) for components
- Components
  - Ready-to-run software packages that support an object-oriented client interface and support a component interface (contract) with the container
  - EJBs are the Java-based component model
- Web Application Servers
  - Are forming the backbone of the Internet computing infrastructure

References

- Books
  - R. Zahavi, *Enterprise Application Integration with CORBA*, Wiley, 2000.
  - M. Henning and S. Vinoski, *Advanced CORBA Programming with C++,* Addison Wesley, 1999.
  - K. Boucher and F. Katz, *Essential Guide to Object Monitors*, Wiley, 1999.
- Websites
  - http://www.omg.org/gettingstarted/corbafaq.htm
  - CORBA 2.0
    http://www.omg.org/cgi-bin/doc?ptc/96-03-04
  - CORBAservices
    http://www.omg.org/technology/documents/corbaservices_spec_catalog.htm
  - CORBAfacilities
    http://www.omg.org/technology/documents/corbafacilities_spec_catalog.htm
  - OMG standard documents on services
    - Transaction: http://www.omg.org/cgi-bin/doc?formal/2000-06-28
    - POS: http://www.omg.org/cgi-bin/doc?orbos/99-07-07
    - Concurrency: http://www.omg.org/cgi-bin/doc?formal/2000-06-14
    - Query: http://www.omg.org/cgi-bin/doc?formal/2000-06-23
  - WebLogic CORBA-based OTM
    http://edocs.bea.com/wle/wle42/index.htm