The ORB is not a single component, rather it is a service executed on each computer (maybe as a daemon). The skeleton is the generated stub for the server side, the generated proxy is the stub for the client side (implements the same interface as the object the client is using). Both stubs are specific for the current application and are the only parts the client resp. the server see; the ORB is invisible for them.

**CORBA Architecture**

**ORB services (from the perspective of a process):**

1. Manipulating object references (marshal and unmarshal object references to exchange them between processes, comparing object references)
2. Finding the services that are available to a process (an initial reference to an object implementing a specific CORBA service, in general a naming service)

**Object Adapter**

- Bridges the gap between CORBA objects with IDL interfaces and the programming language interfaces of the corresponding server
- Creates object references for CORBA objects
- Each active CORBA object is registered with its object adapter, which keeps a remote object table to map names of CORBA objects to servers
- Responsible for dispatching incoming requests via a skeleton to the addressed object
- Possibly has to activate an object when a request comes in

**Portable Object Adapter (POA)**

- Server-side code can be written independently of specific ORB - it is portable in means of ORBs from different vendors
- Assumes: object implementations are partly provided by servants - the part of an object that implements the methods that clients can invoke. Servants are programming-language dependent
- Each POA offers the following operation:
  
  ```
  Objectld activate_object(in Servant p_servant);
  ```

  This operation takes a pointer to a servant as input parameter and returns a CORBA object identifier as a result
- No universal definition of type servant; it is mapped to a language dependent data type
- The returned Objectld is an index into the POA's Active Object Map

**Dynamic Invocations**

**Dynamic Invocation Interface (DII)/ Dynamic Skeleton Interface (DSI)**

- There are occasions in which statically defined interfaces are simply not available to a client, instead it has to find out what it needs during runtime
- DSI and DII are providing generic interfaces for sending/receiving each request, independent of specific proxies and skeletons
- E.g. used for implementing gateways to achieve interoperability between different platforms, e.g. CORBA and DCOM
- Required knowledge: what does the interface to a specific object looks like?
- Subsequently compose an invocation request for that object
Dynamic Interfaces

What does DII do?
- CORBA offers DII to clients, which allows them to construct an invocation request at runtime.
- It provides a generic invoke operation, which takes:
  1. an object reference,
  2. a method identifier, and
  3. a list of input values as parameters.
- These information are marshalled into a request and sent to an object.
- Later it returns the result in a list of output variables provided by the caller.

What does DSI do?
- In the same way, a server object has a DSI, which is able to receive any request and decompose it into object reference, method identifier and parameters.

Interface Repository

How to find information about interfaces to construct a dynamic request? For this, the Interface Repository is needed.
- Stores all interface definitions.
- Often implemented by means of a separate process offering a standard interface to store and retrieve interface definitions.
- Can be viewed as the part of CORBA that assists in runtime type checking.
- Whenever an interface definition is compiled, the IDL compiler assigns a repository identifier to that interface.
- The repository ID can be used to retrieve an interface definition from the repository.
- By default, it is derived from the name of the interface and its methods (no guarantees are given with respect to its uniqueness).
- Because interface information are stored in IDL, it is possible to structure each interface repository in the same way. The interface repositories in CORBA all offer the same operations for navigating through interface definitions.

Implementation Repository

How to assign object references with real files?
- CORBA systems offer an Implementation repository.
- Contains all that is needed to implement, register, and activate objects, as well as locating running servers.
- Stores a mapping of the names of object adapters to the file containing the object implementation.
- Such functionality is related to the ORB (and its implementation) itself as well as to the underlying operating system. For this, it is difficult to provide a standard implementation for each CORBA system.
- Mainly used by an object adapter, which is responsible for dispatching a request to the right object. If this object isn’t running in the address space of a server, the object adapter could contact the implementation repository to find out what needs to be done: map the object reference to a binary file, start this file as a CORBA server in a specific way, ...

Communication in CORBA

Simple communication model: only synchronous communication.
(But: with the time, some invocation facilities were added to this model)

Object invocation model:
- When a client invokes an object, it sends a request to the corresponding object server and blocks until it receives a response. These semantics correspond exactly to a normal method invocation when the caller and callee reside in the same address space.
- In the presence of failures, a client will receive an exception indicating that the invocation did not fully complete.

One-way request:
- Problem with synchronous communication: if the client does not get back a result, it is blocked unnecessarily.
- Solution: a form of invocation, in which no result is expected. The client isn’t blocked, but it has no guarantees that the request is delivered.

Deferred synchronous communication:
- One-way requests are used by a client to make a request and by a server to pass back the result.
Interoperability

CORBA only specifies functionalities, not implementation issues.
- Each vendor of a CORBA implementation had his own way of enabling communication between clients and object servers as well as referencing objects → lack of interoperability
- This problem was solved by the

  **General Inter-ORB Protocol (GIOP)**
  - Standard protocol for communication in CORBA
  - Builds upon a reliable, connection oriented transport protocol
  - Specifies message types, a 'Common Data Representation' (CDR) as transfer syntax, interoperable object references (IORs), and more

  **Internet Inter-ORB Protocol (IIOP)**
  - The realisation of GIOP running on top of TCP
  - Not the only communication protocol implemented, but the widest used one

Message Types

GIOP (and thus IIOP or any other realisation of GIOP) knows eight different message types:

<table>
<thead>
<tr>
<th>Message type</th>
<th>Originator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td>Client</td>
<td>Contains an invocation request</td>
</tr>
<tr>
<td>Reply</td>
<td>Server</td>
<td>Contains the response to an invocation</td>
</tr>
<tr>
<td>LocateRequest</td>
<td>Client</td>
<td>Contains a request on the exact location of an object</td>
</tr>
<tr>
<td>LocateReply</td>
<td>Server</td>
<td>Contains location information on an object</td>
</tr>
<tr>
<td>CancelRequest</td>
<td>Client</td>
<td>Indicates client no longer expects a reply</td>
</tr>
<tr>
<td>CloseConnection</td>
<td>Both</td>
<td>Indication that connection will be closed</td>
</tr>
<tr>
<td>MessageError</td>
<td>Both</td>
<td>Contains information on an error</td>
</tr>
<tr>
<td>Fragment</td>
<td>Both</td>
<td>Part (fragment) of a larger message</td>
</tr>
</tbody>
</table>

Object References

- A client needs an object reference to invoke a method at an object
- A client resp. a server uses a language specific implementation of an object reference - in most cases this is a pointer to a local representation of the object
  - That reference cannot be passed from a process A to process B
  - Process A will first have to marshal the pointer into a process independent representation (done by the ORB)
  - The ORB uses an own, language-independent representation
  - Process B unmarshals it
- When a process refers to an object its underlying ORB is implicitly passed enough information to know which object is actually being referenced
- Common representation of an object reference: **Interoperable Object Reference (IOR)**
  - The IOR is used to pass references to other ORBs. Internally, ORBs can have their own representation.

Organization of an IOR with specific information for IIOP:
- **Tagged Profile**: complete information to invoke an object. If the object server supports several protocols, multiple tagged profiles are included
- **Object key**: server-specific information for demultiplexing incoming requests to the object
- **Components**: optionally contains additional information needed for invoking the referenced object (e.g. security information)
How does a client bind to an object to invoke a method?

- Use a name service to resolve a given name to an object reference (IOR).
- Because this IOR references directly to an object, the following is called **direct binding**.

- The client's ORB uses the repository ID to place a proxy at the client and pass a pointer to this proxy on to the client.
- The ORB uses a tagged profile, e.g. for IIOP and sets up a TCP connection with the object's server.
- A client's invocation is marshaled into an IIOP request message and sent over the TCP connection to the POA associated with the object key.
- The POA forwards the request to the proper servant where it is unmarshaled and transformed into an actual method call.

**Alternative:** **indirect binding**

- An implementation repository is involved.
- The implementation repository is identified in the IOR.
- It acts as a registry to locate and activate an object before transmitting invocations primarily used for persistent objects.

**Interceptors**

- Client implementations are simple: define IDL, generate proxy, done.
- But… if an object expects a client to enhance its functionality (e.g. caching), the client is not enabled to do so.
- Thus: some addition is needed enhancing the current software.

**Interceptors**

- Mechanism by which an invocation can be intercepted on its way from client to server and adapted as necessary before letting it continue.
- Piece of code modifying or analysing a request.
- General method to achieve extensibility.
- There may be various interceptors added to an ORB.
- Also possible for server-side.
- Are seen only by the ORB, the ORB has to invoke them.

**Indirect Binding**

1. First step: binding to the implementation repository.
   - The repository checks if the server is already running. If yes, it checks, where it is located. If no, the repository starts it.
   - When the client invokes the referenced object for the first time, the invocation request is sent to the implementation repository which responds by giving the details where the object's server can actually be reached.
   - So the invocation requests are forwarded to the proper server.

**Types of Interceptors**

1. **Request level interceptor**: is logically placed between a client's proxy and the ORB.
   - Comes in before an invocation request is passed to the ORB.
   - The interceptor may modify the request.
   - On server side, it is placed between the ORB and the object adapter.

2. **Message level interceptor**: placed between an ORB and the underlying network.
   - Knows nothing about the message content that is to be sent.
   - Only sees GIOP messages that it could modify.

- Add additional information, e.g. for instructing a server process or enhance a client by caching.
- Modify request, e.g. for fragmenting large GIOP messages or to transparently redirect a request by exchanging the IOR.
- Both types: add monitors to analyse the performance of communications.
CORBA services are general purpose and application independent services. CORBA services strongly resemble the types of services commonly provided by an operating system.

<table>
<thead>
<tr>
<th>Service</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collection</td>
<td>Facilities for grouping objects into lists, queue, sets, etc.</td>
</tr>
<tr>
<td>Query</td>
<td>Facilities for querying collections of objects in a declarative manner</td>
</tr>
<tr>
<td>Concurrency</td>
<td>Facilities to allow concurrent access to shared objects (locking)</td>
</tr>
<tr>
<td>Transaction</td>
<td>Flat and nested transactions on method calls over multiple objects</td>
</tr>
<tr>
<td>Event</td>
<td>Facilities for asynchronous communication through events</td>
</tr>
<tr>
<td>Notification</td>
<td>Advanced facilities for event-based asynchronous communication</td>
</tr>
<tr>
<td>Externalization</td>
<td>Facilities for marshaling and unmarshaling of objects</td>
</tr>
<tr>
<td>LifeCycle</td>
<td>Facilities for creation, deletion, copying, and moving of objects</td>
</tr>
<tr>
<td>Licensing</td>
<td>Facilities for attaching a license to an object</td>
</tr>
<tr>
<td>Naming</td>
<td>Facilities for system-wide name of objects</td>
</tr>
<tr>
<td>Property</td>
<td>Facilities for associating (attribute, value) pairs with objects</td>
</tr>
<tr>
<td>Trading</td>
<td>Facilities to publish and find the services on object has to offer</td>
</tr>
<tr>
<td>Persistence</td>
<td>Facilities for persistently storing objects</td>
</tr>
<tr>
<td>Relationship</td>
<td>Facilities for expressing relationships between objects</td>
</tr>
<tr>
<td>Security</td>
<td>Mechanisms for secure channels, authorisation, and auditing</td>
</tr>
<tr>
<td>Time</td>
<td>Provides the current time within specified error margins</td>
</tr>
</tbody>
</table>

Event Service

Asynchronous communication can be realised with one-way calls, but...

- One-way calls are best effort
- Overhead when simply a signalling information should be transmitted

→ Communication by events

CORBA’s Event Service

- Provide a service that could simply signal the occurrence of an event.
- In this model each event is associated with a single data item, generally represented by means of an object reference or otherwise an application-specific value.
- An event is produced by a supplier and received by a consumer
- The event service offers an event channel which is logically placed between suppliers and consumers and serves for delivering events

Push and Pull Model

Push model:
- Whenever an event occurs, the supplier produces the event and pushes it through the event channel
- Event channel passes the event on to its consumers
- In this model consumers passively wait for event propagation and expect to be interrupted when an event happens

Pull model:
- Consumers poll the event channel to check whether an event has happened
- The event channel in turn polls the various suppliers

Note: the models can be combined

How does it work?
- For both, consumers and suppliers, proxies (for push and pull operations) are implemented in the event channel.
- Consumer and producer use synchronous communication with these proxies for the time delivering or getting events

Event Channel

Notice: kind of multicast: all events are passed from a Supplier Proxy to all Consumer Proxies
Notification Service

Event services drawbacks:
- CORBA's event service does not support persistence of events. If a consumer connects to the event channel too late, events get lost.
- Consumers have no chance to filter events. Each event is passed to every consumer. If different event types need to be distinguished it is necessary to set on a separate event channel for each event type.
- Event propagation is unreliable. No guarantees need to be given concerning the delivery of events.

→ Notification service

Event typing
- Filtering capabilities have been added
- Offers facilities to prevent event propagation when no consumers are interested in a specific event.

Messaging

Communication in CORBA is transient.
- Many applications require persistent communication as offered by message queuing.
- CORBA supports this model as an additional messaging service.
- Messaging in CORBA is different because it is inherent object-based approach in communication.

- Two models for messaging:
  - Callback model: A client implements a callback method. The communication system calls this method to deliver a result from an asynchronous request. It is the client’s responsibility to transform the original synchronous invocation into an asynchronous one.
  - Polling model: The client is offered a collection of operations to poll its ORB for incoming results. It is the client’s responsibility to transform the original synchronous invocation into an asynchronous one.

Callback Model

Constructing an asynchronous invocation is done in two steps:
1. The original interface as implemented by the object is replaced by two new interfaces that are to be implemented by the client-side software:
   - Specification of methods that the client can call; none of these methods returns a value or has any output parameter
   - Callback interface
2. This step consists of simply compiling the generated interfaces

```
Original method:
int add(int i, int j, int k);

New methods:
void sendcb_add(int i, int j);
void replycb_add(int ret_val, int k);
```

Polling Model

- Again, the original method is replaced by two new methods
- The ORB has to provide the second method

```
Original method:
int add(int i, int j, int k);

New methods:
void sendpoll_add(int i, int j);
void repypoll_add(int ret_val, int k);
```
Naming

- CORBA supports different types of names
- Most basic types, object references and character-based names, are supported by the CORBA naming service
- There are a number of advance naming facilities whereby objects can found based on associated properties
- Independently from logical object names, CORBA objects can be addressed by URLs:
  - `iioploc://appserver.klick-and-bau.com:4711/object-24` references the CORBA object which can be connected to on host 'appserver.klick-and-bau.com' using port 4711 and object identifier 'object-24'
  - `iiopname://appserver.klick-and-bau.com/Buchhaltung/Stammdaten/ArtikelHome` references the CORBA object which the naming service on host 'appserver.klick-and-bau.com' knows by the name 'Buchhaltung/Stammdaten/ArtikelHome'
- In both cases, IIOP is used as communication protocol

CORBA Naming Service

- Used to look up object references using a character based name
- Names in CORBA are sequences of name components each taking the form of a \((id,kind)\)-pair, where \(id\) is used to name an object, \(kind\) is a simple indication of the name object (e.g. 'dir' for a directory object)
- The representation of a sequence is a language dependent
- There are no restrictions with respect to the structure of a naming graph.
- Each node in a naming graph is treated as an object
- Naming context: an object that stores a table mapping name components to object references (like directory node)
- Naming graph does not have a root context, however each ORB is required to provide an initial naming context which effectively operates as the root in a naming graph
- Names are always resolved with respect to a given naming context
- A client resolves a name by invoking the resolve method on a specific naming context
- If name resolution succeeds, it always returns either a reference to a naming context or a reference to a named object
- Name resolution proceeds as in DNS

Part of the CORBA Naming Service Interface in IDL

```
struct NameComponent { string id; string kind; }

typedef sequence <NameComponent> Name;

interface NamingContext {
    void bind (in Name n, in Object obj);
    binds the given name and remote object reference in my context
    void unbind (in Name n);
    removes an existing binding with the given name
    void bind_new_context(in Name n);
    creates a new naming context and binds it to a given name in my context
    Object resolve (in Name n);
    looks up the name in my context and returns its remote object reference
    void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
    returns the names in the bindings in my context
}
```

How to Find the First Naming Context?

- The CORBA name service allows you to search a name context and to navigate through several name contexts. This works as in other name services. But... how to find the root context, i.e. the naming service itself?
- The naming service is identified by an object reference...
- To solve the problem, the ORB itself is responsible to provide facilities to get initial object references. A reference to the root context can be got by calling the function `resolve_initial_reference("Name Service")` provided by the ORB:
  ```
  interface ORB {
    ...
    Object resolve_initial_references (in String name);
    ...
  }
  ```
- CORBA defines some more names the ORB has to resolve, e.g. "RootPOA" and "InterfaceRepository". Further names can be configured by the administrator.
Trading Service

Naming Service = White pages
But... sometimes, no concrete name but only a description of the needed service is known
→ Trading Service (= Yellow pages)

An object is not denoted by a logical name, but by a description of its capabilities:
• Service Type (description of object functionality, determines interface)
• Service Properties (non-functional description of a service)

Roles:
• Trader: stores and searches service descriptions
• Importer: a client searching for an object's service
• Exporter: an object offering a service

Example: searching a print service

Exporter: specifies its offer, i.e. type and properties, e.g.
• format={A3, A4, A5}
• cost_per_page=10
• pages_per_sec=5

Importer: specifies type and constraints, e.g.
• format=A4 AND cost_per_page<15
• minimize cost_per_page

Synchronisation and Transactions

Two important services that facilitate synchronisation in CORBA are the Concurrency control service and the Transaction service

• The two services collaborate to implement distributed and nested transactions using two phase locking
  ➢ A transaction is initiated by a client and consists of a series of invocations on objects. When an object is invoked for the first time it automatically becomes part of the transaction. This information is passed to the server when invoking the object
  ➢ Two types of objects can be part of a transaction:
    ➢ Recoverable object: is executed by an object server capable of participating in two phase commit protocol
    ➢ Transactional objects: executed by server that do not participate in a transaction's two phase commit protocol (typically read-only objects)

• CORBA transactions are similar to distributed transactions and their protocols as will be presented in the last part of the lecture
• The service is implemented using a central lock manager; it does not use distributed locking techniques
• The service distinguishes read from write locks and is also capable of supporting locks at different granularities (e.g. whole tables vs. single records)
Replication and Fault Tolerance

- CORBA offers no support for generic caching and replication.
- Application developers have to resort to an ad-hoc approach when replication is needed (such approaches are often based on using interceptors).
- Only the replication of objects for fault tolerance is included in the current CORBA version 3.

Fault Tolerance

- Dealing with failures: replicate objects into object groups, consisting of identical copies of an object referenced as if it would be a single object.
- A group offers the same interface as the objects it contains.
- Uses a special kind of an IOR, the Interoperable Object Group Reference (IOGR).

Fault Tolerance

- Supported strategies: passive replication, active replication, quorum-based replication.
- When a client passes an IOGR to the ORB, the ORB attempts to bind to one of the replicas. The Components file could refer to the primary or a copy of the object's replicas.
- If binding fails, the ORB can try another copy.

In the diagram:

- Repository identifier
- Profile ID
- Profile-1
- Profile-N
- Interoperable Object Group Reference (IOGR)
- IIOP version
- Host
- Port
- Object key
- Components
- TAG
- PRIMARY
- Other group-specific information
- TAG
- BACKUP
- Other group-specific information

- Replication manager: creating and managing object groups, replacing replicas in case of a failure.
- Interceptors are used to pass invocations to a separate replication component maintaining consistency and realising recoverability.