Middleware
**Middleware**

- Realisation of distributed accesses by suitable software infrastructure
- Hiding the complexity of the distributed system from the programmer
- Supporting the interaction between application components which could run on heterogeneous systems
- Separation of interface and implementation
- Is added to the operating system or build upon it to take over some of its tasks

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

The applications can be programmed in different languages.

... and the middleware realise the cooperation between the software components.
Middleware supports...

- The communication between application components (usually RPC- or RMI-based)
- Constructing modular applications
- Dynamic binding
- Supporting services, e.g.
  - Naming
  - Synchronization
  - Replication
Examples for Middleware

- **DCOM** as Microsoft's solution on distributed objects
- **GLOBE** as a research middleware for distributed objects (meant here: not a system composed of several objects forming a larger system, but single objects which by themselves are distributed, like in a strange form of replication)
- **The Common Object Request Broker Architecture (CORBA)** as a general (industry defined) way to support distribution in heterogeneous systems
- **Web Services** as an approach integrated with common Internet protocols forming a “lightweight version” of CORBA
CORBA

• Distributed object-based system
• **Common Object Request Broker Architecture (CORBA)**
• CORBA is a *specification* of a distributed system. There are no suggestions made about the implementation of the middleware itself
• These specifications have been drawn up by the Object Management Group (OMG), a non-profit organisation with more than 800 members
• Goals:
  - Make better use of distributed systems
  - Use object-oriented programming
  - Define a distributed system that could overcome many of the interoperability problems with integrating networked applications
• Core component: the *object request broker* (ORB), a 'software bus' for enabling and managing communication
CORBA

• No implementation guidelines were made, but many organisations thought that CORBA would be a good product:
  ➢ Large set of options to define every aspect of a distributed system
  ➢ Competition: fast implementations to be first-on-the-market
  ➢ Poor performance due to fast implementation and large set of specified options
  ➢ In the first versions: incompatibilities between implementations of different vendors
  ➢ Problems with the specification lead to new specifications, forcing vendors to re-build their implementations
  ➢ Re-building implementations lead to re-design of the developers software

• Today, lot of different implementations exist, e.g.
  ➢ Commercial: Orbix, VisiBroker, ...
  ➢ Freeware: MICO, OmniORB, ...
  ➢ Integrated: e.g. Java-ORB
Components of the CORBA Framework

CORBA specifies...

- A (raw) abstract object model
- An interface definition language (IDL)
- Mapping from IDL to different programming languages: Ada, C, C++, COBOL, Java, Lisp, Python, SmallTalk, ...
- A repository for managing IDL specifications
- The ORB as communication infrastructure
- A communication protocol (GIOP/IIOP), together with a transfer syntax and message formats
- A client interface to the ORB (proxy)
- A server interface to the ORB (skeleton)
- Interoperability with other standards, e.g. DCOM
- Services supporting management and use of distributed applications: Naming, Transactions, ...
- ...
CORBA Architecture

- Reference model consists of four components
  - The Object Request Broker (ORB) as communication infrastructure
  - CORBA services as useful services for all distributed applications
  - CORBA facilities as additional services used by some applications
  - Application Objects as the distributed applications implemented by a programmer for its special purposes
CORBA Components

Object Request Broker

- Core of any CORBA system
- Responsible for enabling basic (synchronous!) communication between objects and their clients while hiding issues related to distribution and heterogeneity

CORBA services

- Set of basic services (used by many applications) supporting a programmer in constructing a distributed application and supporting the application at runtime
- Examples: Naming, LifeCycle, Event, Security

CORBA facilities

- General purpose high level services with useful functions needed by several applications, but which are not necessary for basic functioning
- Horizontal facilities (CORBA facilities): services that are independent of an application domain such as user interfaces or system management
- Vertical facilities (CORBA domains): services that are targeted to a specific application domain such as electronic commerce, banking, medicine, telecommunications, ...

Application Objects

- Not standardised, these are all objects programmed by a user for a special purpose
Object Model

- CORBA bases on the remote object model: objects can be placed in the same process as the client, in another process on the same machine, or in a process on a remote machine
- RMI as communication principle between objects: used is multi-language RMI
- Basic concept:
  - Distinction between interface and implementation
  - The developer first specifies an object by its interface in an Interface Definition Language (CORBA IDL)
  - An object simply is an implementation of an interface
- IDL provides precise syntax for expressing methods and their parameters, but not the semantics. By using IDL, the whole interface of an object is specified
- The CORBA implementation provides compilers to translate the IDL specification into an object with proxies resp. stubs (client) and skeletons (server) for handling the communication. The programmer only has to specify the application-specific code, not the communication mechanism
CORBA Interfaces

- CORBA generates *language-dependent* stubs
- It is necessary to provide exact rules concerning the mapping of IDL specifications to existing programming languages (C, C++, Java, Smalltalk, Ada, Cobol)
- Each object is 'named' by an unique *object reference*
- The term *CORBA object* is used to refer to remote objects. A *CORBA object* implements an IDL interface, has a remote object reference and its methods can be invoked remotely

**IDL specification**

- A new specification for each mapping to a programming language
- Separate translation of the interface for each language
CORBA IDL

- Abstract interface definition
- IDL has the same lexical rules as C++ but has additional keywords to support distribution
- The grammar of IDL is a subset of ANSI C++ with additional constructs to support method signatures
- IDL provides facilities for defining modules, interfaces, types, attributes and method signatures
- It allows standard C++ pre-processing facilities, e.g. `#include` for including other interface definition files in the new file (inheritance)
- Through an interface, only CORBA data types can be passed for an invocation. CORBA provides a lot of data types, but own data types which have to be passed are to be defined together with the interface specification
- **Notice**: pointers are not supported!
### IDL Basic Types

Support of basic keywords and data types:

<table>
<thead>
<tr>
<th>any</th>
<th>default</th>
<th>inout</th>
<th>out</th>
<th>switch</th>
</tr>
</thead>
<tbody>
<tr>
<td>attribute</td>
<td>double</td>
<td>interface</td>
<td>raises</td>
<td>TRUE</td>
</tr>
<tr>
<td>boolean</td>
<td>enum</td>
<td>long</td>
<td>readonly</td>
<td>typedef</td>
</tr>
<tr>
<td>case</td>
<td>exception</td>
<td>module</td>
<td>sequence</td>
<td>unsigned</td>
</tr>
<tr>
<td>char</td>
<td>FALSE</td>
<td>Object</td>
<td>short</td>
<td>union</td>
</tr>
<tr>
<td>const</td>
<td>float</td>
<td>octet</td>
<td>string</td>
<td>void</td>
</tr>
<tr>
<td>context</td>
<td>in</td>
<td>oneway</td>
<td>struct</td>
<td></td>
</tr>
</tbody>
</table>

---

Chapter 8: Middleware  
13
IDL Definitions

IDL definition: definition of

- **Modules** to group interface definitions into name spaces
  
  ```
  module <ModulName> { ... ];
  ```

- **Interfaces** as access points to objects. Interfaces can inherit from several other interfaces
  
  ```
  interface <InterfaceName> { ... }; 
  ```

- **Constants**
  
  ```
  const <Typ> <ConstantName> = <Wert>
  ```

- **Types** to group basic data types
  
  ```
  typedef <TypeName> ... 
  enum <EnumName> { ... }; 
  struct <StructName> { ... }; 
  ```

- **Exceptions** for receiving ORB-specific or other (self-defined) distribution-specific error messages
  
  ```
  exception <ExceptionName> ... 
  ```
IDL Attributes

- Components of interfaces
- Equivalent to two operations: read and write an attribute value
- Definition:
  ```
  attribute <TypIdentifier> <AttributeName>;
  Example: attribute short x;
  ```
- Special attributes for read-only:
  ```
  readonly attribute short x;
  Meaning: a client can't modify the attribute value, but the object itself possibly can
  ```
- The IDL compiler generates a write operation and a read operation for such a definition
IDL Methods

• Similar to function declarations in C++
• Consist of an identifier for the resulting type, one for the function name, and a list of identifiers for arguments
  
  short operation(in short arg1, in short arg2);

• Optional: oneway (the client gets no response)
  oneway void operation(in char arg);

• Optional: Exceptions (for catching error messages)
  void operation(out short arg2) raises (Except);

• Arguments: differentiation between three types:
  ➢ in for parameters only used to pass information to the object
  ➢ out for parameters only used to give back a result to the client
  ➢ inout for parameters used in both directions

• General method signature:

  [oneway] <return_type> <method name> (parameter1,...,parameterL)
  [raises (except1,..., exceptN)] [context (name1,..., nameM)]
Data Types

• Basic types
  - **Any** char float Object short void unsigned short
  - **boolean**: defined values TRUE, FALSE
  - **char**: 1 Byte.
  - **octet**: similar to char, but with a constant value
  - **any**: can contain each combination of other IDL data types
  - **Object**: Interface type; all interfaces inherit from this type to offer special functions for binding or name resolving

• **Structs**
  - Combines several data type to a new one
  - **Example**:
    ```
    struct structure {
      char element1;
      short element2;
    }
    ```
## Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequence</td>
<td><code>typedef sequence &lt;Shape, 100&gt; All;</code>&lt;br&gt;<code>typedef sequence &lt;Shape&gt; All</code>&lt;br&gt;bounded and unbounded sequences of Shapes</td>
<td>Defines a type for a variable-length sequence of elements of a specified IDL type. An upper bound on the length may be specified.</td>
</tr>
<tr>
<td>string</td>
<td><code>String name;</code>&lt;br&gt;<code>typedef string&lt;8&gt; SmallString;</code>&lt;br&gt;unbounded and bounded sequences of characters</td>
<td>Defines a sequences of characters, terminated by the null character. An upper bound on the length may be specified.</td>
</tr>
<tr>
<td>array</td>
<td><code>typedef octet uniqueld[12];</code>&lt;br&gt;<code>typedef GraphicalObject GO[10][8]</code></td>
<td>Defines a type for a multi-dimensional fixed-length sequence of elements of a specified IDL type.</td>
</tr>
</tbody>
</table>
## Data Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Examples</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>record</td>
<td><code>struct GraphicalObject {</code> <code>string type;</code> <code>Rectangle enclosing;</code> <code>boolean isFilled;</code> <code>};</code></td>
<td>Defines a type for a record containing a group of related entities. <em>Structs</em> are passed by value in arguments and results.</td>
</tr>
<tr>
<td>enumerated</td>
<td><code>enum Rand (Exp, Number, Name);</code></td>
<td>The enumerated type in IDL maps a type name onto a small set of integer values.</td>
</tr>
<tr>
<td>union</td>
<td><code>union Exp switch (Rand) {</code> <code>case Exp: string vote;</code> <code>case Number: long n;</code> <code>case Name: string s;</code> <code>};</code></td>
<td>The IDL discriminated union allows one of a given set of types to be passed as an argument. The header is parameterised by an <em>enum</em> which specifies which member is in use.</td>
</tr>
</tbody>
</table>
IDL Interfaces and Modules

Interfaces group all methods and parameters:

```cpp
interface Grid
{
    readonly attribute short height;
    readonly attribute short width;
    void set(in short n, in short m, in long value);
    long get(in short n, in short m);
};
```

... and modules form a name space:

```cpp
module Whiteboard {
    struct Rectangle{...} ;
    struct GraphicalObject {...};
    interface Shape {...};
typedef sequence <Shape, 100> All;
    interface ShapeList {...};
};
```
IDL Interfaces Shape and ShapeList

```plaintext
struct Rectangle {
    long width;
    long height;
    long x;
    long y;
};

struct GraphicalObject {
    string type;
    Rectangle enclosing;
    boolean isFilled;
};

interface Shape {
    long getVersion();
    GraphicalObject getAllState(); // returns state of the GraphicalObject
};

typedef sequence <Shape, 100> All;

interface ShapeList {
    exception FullException{};
    Shape newShape(in GraphicalObject g) raises (FullException);
    All allShapes(); // returns sequence of remote object references
    long getVersion();
};
```
Parameters in CORBA IDL

• Passing CORBA objects:
  – Any parameter or return value whose type is specified by the name of a IDL interface, e.g. Shape, is a reference to a CORBA object (see newShape)
  – and the value of a remote object reference is passed

• Passing CORBA primitive and constructed types:
  – Arguments of primitive and constructed types are copied and passed by value. On arrival, a new value is created in the recipient’s process. E.g., the struct GraphicalObject (argument of newShape and result of getAllState)

• Note: the method allShapes returns an array of remote object references as follows:

  typedef sequence <Shape, 100> All;
  All allShapes();

• Type Object - is a supertype of all IDL interfaces (its values are object references)
IDL Compiler

The IDL compiler generates programming language constructs from the interface definitions:

Client

Server

IDL

IDL-Compiler

Client-Code

Server-Code

Client Stub

Header

Server Stub

Compiler & Linker

Compiler & Linker

Client Application (Object Code)

Server Application (Object Code)

Client

Server
IDL Language Mapping

- Mapping of IDL definitions to data structures. Here: C++ (generated code depends on CORBA implementation – here: only one example, Orbix)
- Generated: Stubs of client and server. Additionally: Management of dynamically allocated memory for IDL data types
- Determination of how to execute an operation call for an object reference

Simple mapping rules:

```c
module M { ... };  ➔ namespace M { ... };
interface I { ... };  ➔ class I { ... };
struct S { ... };  ➔ struct S { ... };
```

Very simple: interfaces can be mapped to classes; by this, objects to an interface can be instantiated easily.

Mapping of basic types: e.g.

```
short  ➔ CORBA::Short
unsigned short  ➔ CORBA::UShort
```
IDL Language Mapping

• Arguments

➢ The mapping of arguments depends on their usage:

  IDL: type0 op(in type1 arg1, inout type2 arg2, out type3 arg3);
  C++: type0 op(type1 arg1, type2& arg2, type3& arg3);

```
interface example {
  readonly attribute long ro_value;
  attribute long value;
  void op(in long in_par, out long out_par);
};
```

```
class example : public virtual CORBA::Object {
public:
  virtual CORBA::Long ro_value();
  virtual CORBA::Long value();
  virtual void value(CORBA::Long val);
  virtual void op(CORBA::Long in_par, CORBA::Long& out_par);
};
```

Note: the virtual class is generated by the IDL compiler and has to be implemented as a class by the programmer
IDL Language Mapping

**Strings**
- Are defined as `char*`
- ORB manages memory for character strings
- Two possible definitions of string variables:
  ```c
  char* string1;
  CORBA::String_var string2;
  ```
- ORB offers some functions for working with strings (recommended to use them):
  ```c
  char* CORBA::string_alloc(CORBA::Ulong len);
  char* CORBA::string_dup(const char*);
  char* CORBA::string_free(char*);
  ```

**Other types, e.g. sequence**
- `_var` is defined as above (regarding memory management)
- Mapped to a list, accessible by giving the index:
  ```c
  IDL:    typedef sequence<Content> Example
  C++:    Example_var ex_variable;
          ex_variable[2] = ... 
  ```
Writing Client and Server using the IDL-generated Stubs (Orbix)

Defining the interface

- Specify the IDL definition in file.idl
- Use IDL compiler to map it to C++. Generally, some stub files like fileS.cc for server stub and fileC.cc for client stub are generated.

Writing the object implementation

- Include fileS.cc (server stub)
- Implement the operations which were defined in IDL
- Register the server with the ORB

Writing the client

- Include fileC.cc (client stub)
- Implement client functionality, including operation requests
- Bind to the server
Simple Example – grid / C++

• Specifying the interface IDL

```
// In file grid.idl

interface Grid {
    readonly attribute short height;
    readonly attribute short width;
    void set(in short n, in short m, in long value);
    long get(in short n, in short m);
};
```

• Compiling the interface definition produces

  - `grid.hh` (general type information for types defined in file.idl)
  - `gridC.cc` (client stub)
  - `gridS.cc` (server stub)
Simple Example – grid / C++

The grid.hh file produced by the IDL compiler contains the following:

```cpp
// Automatically produced (in grid.hh).
class Grid : public virtual CORBA::Object {
public:

    virtual CORBA::Short height(CORBA::Environment& IT_env = CORBA::default_environment);

    virtual CORBA::Short width(CORBA::Environment& IT_env = CORBA::default_environment);

    virtual void set(CORBA::Short n, CORBA::Short m, CORBA::Long value,
                     CORBA::Environment& IT_env = CORBA::default_environment);

    virtual CORBA::Long get(CORBA::Short n, CORBA::Short m,
                            CORBA::Environment& IT_env = CORBA::default_environment);

};
```

Notice: each class inherits from the general CORBA object
Simple Example – grid / C++

(Partly) generated header file for your implementation - grid_i.h

```cpp
#include "grid.hh"

class Grid_i : public virtual GridBOAImpl {
    CORBA::Short m_height;
    CORBA::Short m_width;
    CORBA::Long** m_a;

public:
    Grid_i(CORBA::Short h, CORBA::Short w); // Constructor
    virtual ~Grid_i(); // Destructor

    virtual CORBA::Short width(CORBA::Environment&);
    virtual CORBA::Short height(CORBA::Environment&);

    virtual void set(CORBA::Short n, CORBA::Short m,
        CORBA::Long value, CORBA::Environment&);
    virtual CORBA::Long get(CORBA::Short n, CORBA::Short m,
        CORBA::Environment&);
};
```
Simple Example – grid / C++

Your server implementation - grid_i.h

```cpp
#include "grid_i.h"

Grid_i::Grid_i(CORBA::Short h, CORBA::Short w) : m_height(h), m_width(w) {
    m_a = new CORBA::Long*[h];
    CORBA::Short i;
    for (i = 0; i < h; i++) m_a[i] = new CORBA::Long[w];
}

Grid_i::~Grid_i() {
    CORBA::Short i;
    for (i = 0; i < m_height; i++) delete[] m_a[i];
    delete[] m_a;
}

CORBA::Short Grid_i::height(CORBA::Environment&) {
    return m_height;
}

... // the same for Grid_i::weight

void Grid_i::set(CORBA::Short n, CORBA::Short m, CORBA::Long value,
    CORBA::Environment&) {
    m_a[n][m] = value;
}

CORBA::Long Grid_i::get(CORBA::Short n, CORBA::Short m, CORBA::Environment&) {
    return m_a[n][m];
}
```
Simple Example – grid / C++

Providing the server - Srv_Main.C

```c++
#include "grid_i.h"
#include <iostream.h>

main() {
    // We could create any number of objects
    // here but we just create one.
    Grid_i myGrid(100,100);

    // Orbix objects can be explicitly named,
    // but this is not required in this simple
    // example.

    CORBA::Orbix.impl_is_ready("GridSrv");
    cout << "Server terminating" << endl;
}
```

Note: additionally it is necessary now to make an entry for GridSrv in the so-called Interface Repository which enables a mapping of the server name to an executable
Simple Example – grid / C++

**Writing a client - Client.C**

```cpp
#include "grid.hh"
#include <iostream.h>

main() {
    Grid_var gVar;

    gVar = Grid::_bind(":GridSrv");

    cout << "height is " << gVar->height() << endl;
    cout << "width is " << gVar->width() << endl;

    gVar->set(2,4,123);
    cout << "Grid[2,4] is " << gVar->get(2,4) << endl;

}
```

**Simple example – Orbix provides a comfortable bind function. In other implementations, you would have to work with object references – try on your own in exercise 5.**

Now: what makes all that work?
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.
**ORB services** *(from the perspective of a process):*

- The ORB is offering a few functions by itself:
  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

**Object adapter**
- Implements an activation policy for a group of objects (single thread, multithreading, ...)
- Responsible for providing a consistent image of what an object
- Adapts a program such that clients can see the program as an object

**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:
  ```
  ObjectId 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; mapped to a language dependent data type
- The returned *ObjectId* 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 send 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 facilities
- 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
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.
Object References

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)
• This IOR references directly to an object, thus this 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
Indirect Binding

- 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
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.
    The server is presented a normal (synchronous) invocation request
  - **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(in int I, in int j, out int k);
```

New methods:
```
void sendcb_add(in int I, in int j);
void replycb_add(in int ret_val, in int k);
```
• Again, the original method is replaced by two new methods
• The ORB has to provide the second method

Original method:
```c
int add(in int I, in int j, out int k);
```

New methods:
```c
void sendpoll_add(in int I, in int j);
void replypoll_add(out int ret_val, out int k);
```
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
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