University of Bologna
Dipartimento di Informatica – Scienza e Ingegneria (DISI)
Engineering Bologna Campus

Class of Infrastructures for Cloud Computing and Big Data M

C/S and Middleware, Multicast, and MOMs

Antonio Corradi

Academic year 2017/2018

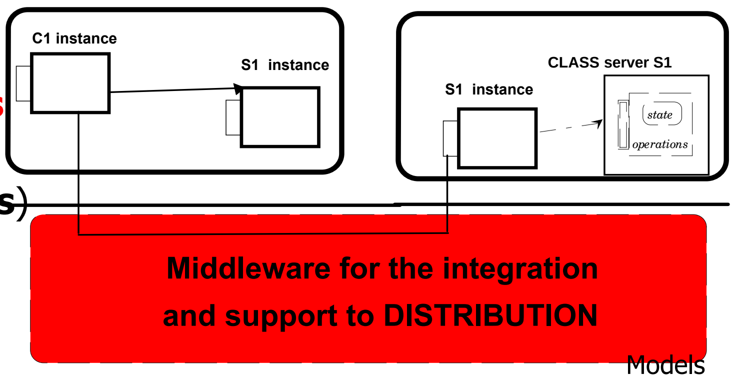
REMOTE REFERENCES

In **many local environments** (in **object-oriented** system), we need the capacity of **referring to some external resources**, in order to coordinate different machines (virtual or physical)

A C1 on one node must refer to a remote instance, the same as if they were local instances on the same node

To refer to a remote instance we need some **intermediary support that extends the visibility to remote nodes**

In some cases,
local and remote references are uniformed via
local intermediaries (proxies)
that play the enabling role and typically mask support transparently



RMI REMOTE REFERENCES

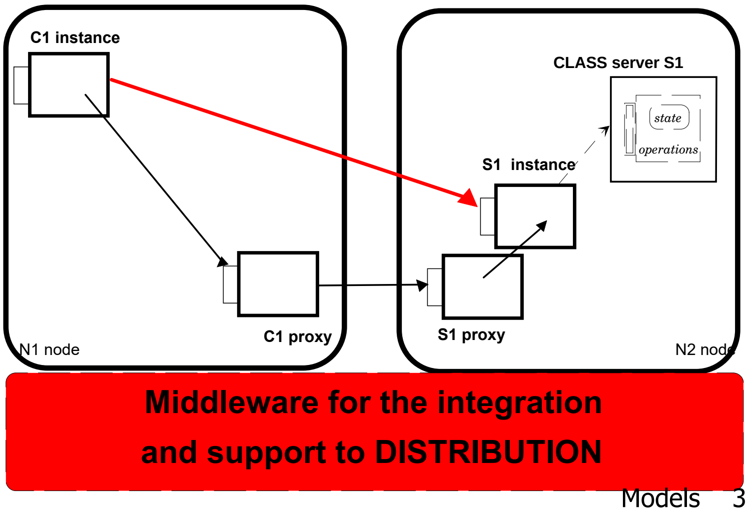
Between two JAVA JVM systems, we can use Java Remote Method Invocation (RMI) that build two proxies

-one from the customer (stub)

-one on the side of the servant (skeleton)

Such proxies are often **generated automatically** and make the user part reasoning regardless of the specific deployments

Similarly other environments (**CORBA**, **DCOM**, etc.) define their specific support for OO cases



REMOTE REFERENCES via PROXY

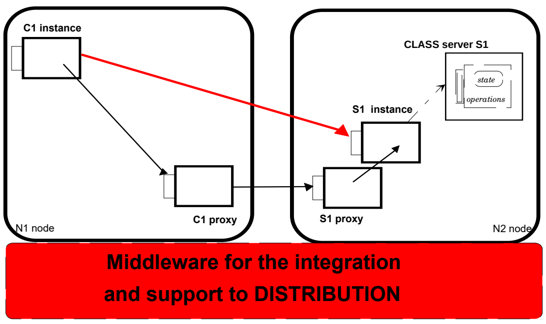
Two Java virtual machines can use **PROXIES to get remote visibility of object references**

RMI support many solutions but proposes problems:

- How do you get the reference to the server? (name system)
- Where are the ancillary classes?
- How to obtain them (while running)?
- And if there are any inconsistencies?
- And if the server is not active?
- And if you don't keep the status?

About **remote references**:

- two references to the same object?
- two references for the same service?



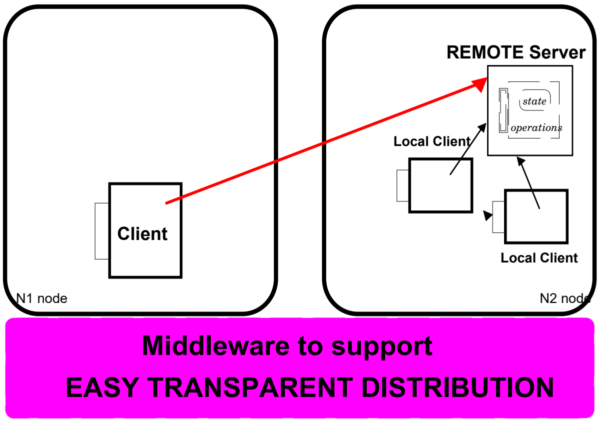
REMOTE REFERENCES & MIDDLEWARE

A central point in all middlewares that **abstract away and hide details from users for remote access** is how to enable and manage a **remote reference** in all its aspects

A remote reference allows access to non-local entity must surely be transparently

But costs must be considered and evaluated for each aspects of the support mechanism

- How does the remote reference cost?
- How is the cost of middleware to support organization?
- How to obtain remote references?
- Are inconsistencies possible?
- What are the responsibilities of the middleware? ...
- ...



INTERMEDIARIES & PROXIES

PROXY

In a communication we may have intermediaries placed and deployed either side, the client and the service provider

PROXY

from client or from server

proxy

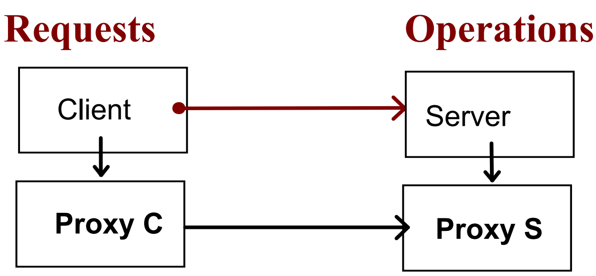
C/S stub & skeleton

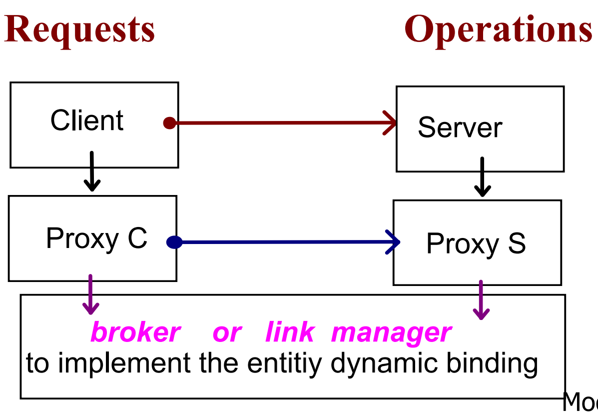
interceptor

to add functions

broker

something similar to a container





MIDDLEWARE: CORBA as a C/S MW

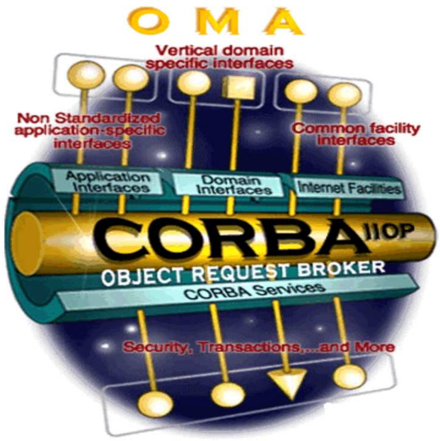
OMG- Object Management Group

CORBA started in 1989 with **440 company** Microsoft, Digital, HP, NCR, SUN, OSF, *etc.* with main objective to create a **use** and **management system** of a **distributed architecture**

Common **O**bject **R**equest **B**roker **A**rchitecture
CORBA standard v1 ⇨ 1991, v1.2 ⇨ 1992
v2 ⇨ 1996, v3 ⇨ 2000

Orbix *SunOS Solaris, Iris, Windows NT, HP/UX, AIX, OSF/1, UnixWare*
DSOM *IBM*

General specification of an Object (component) Middleware to use in heterogeneous distribute systems not tied to a specific language



MIDDLEWARE: CORBA

STANDARD OPEN SYSTEM based on OBJECT

models with heterogeneous components to implement **mutual** and **complete interaction** and **integration** between **such components,** inside **distributed environments also objects oriented (C/S model)**

CORBA requires:

- definition of a **language as service interface**
- definition and support to **objects interaction**
- **integration bus** for **different environments** objects (**ORB**)
- **interaction** between systems with different managers
- **different deployment languages** (**language mapping**)

The objective is to allow **services support** without posing **limits** on user application **lifecycle**

CORBA ARCHITECTURE

Common **O**bject **R**equest **B**roker **A**rchitecture **CORBA**, as a **common environment**, **O**bject **M**anagement **A**rchitecture, for multi-architecture and multi-language scenarios, with an optimal integration with legacy systems and best support for differentiated projects for server and clients

Object Request Broker (**ORB**) is the **heart** of the **architecture** and acts as a **broker of communication**, to allow both **static** and **dynamic** links (!?) between entities

ORB behave as an always available enabler and allows:

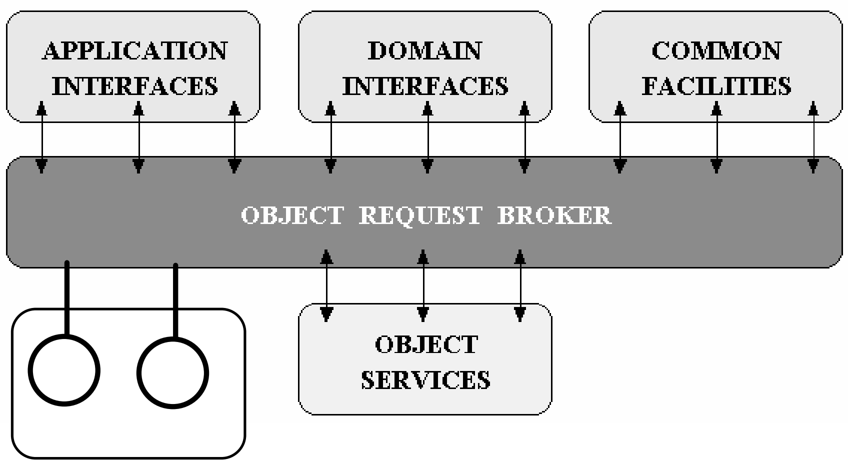
- control of **allocation** and **visibility** of objects
- control of **methods** and of **communication**
- control of **accessory services** always available inside OMA for every language mapping
- **simplified management** of every possible services

CORBA is middleware to support an infinite lifetime

CORBA as a BUS

ORB is the **center** of **O**bject **M**anagement **A**rchitecture
ORB as a **bus center of an architecture** that aims at the integration among **every resources of an organization**

Every managed application objects can belong to **different environments** and must be able to **mutually communicate** without any need of **redesign**



Applications Object

Object Management Architecture

Other additional environment components

Common Facilities CF (horizontal)

Set of specific features

User Interface (client-site),

System Management, Information, Task (server-site)

Domain Interfaces (vertical)

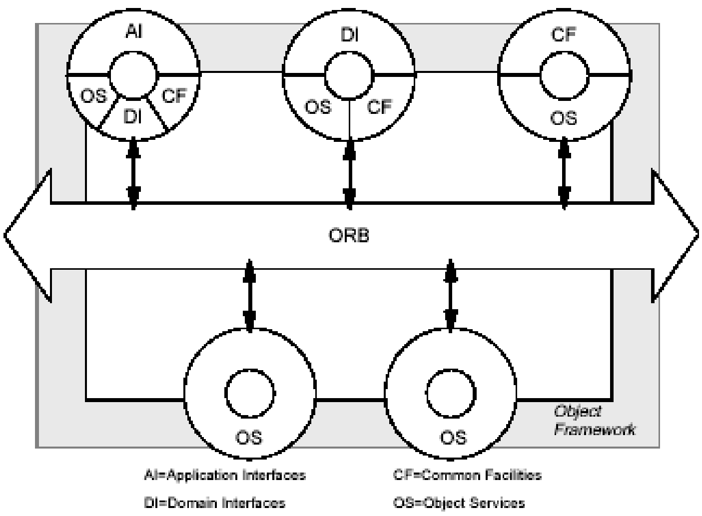
Features dedicated to application areas, for ex. manufacturing, telecommunications, electronic commerce, transportation, business objects, healthcare, finance, life science, …

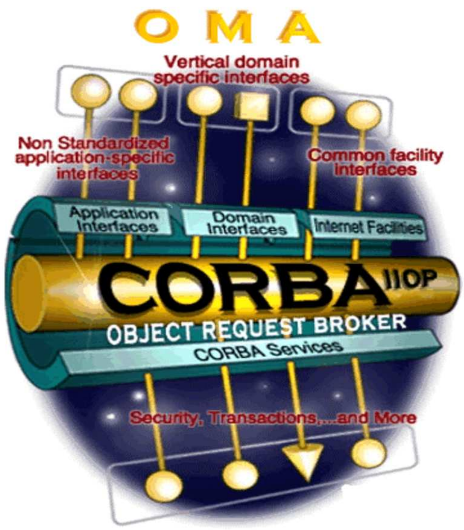
Application Interfaces

Non standard in any way and application-dependent

Object Management Architecture - OMA

Ambiente Object Framework





Object Management Architecture

Every component can connect to every other one, preparing link either before or during execution (if unknown before), using the service of **one or more ORB (known dynamically)**

Set of **additional environment components**

Object Services or CORBA Services (*Common Mw Services*)

Some operations are basic for object

- ***naming*** and ***trading*** service (compatible with OO)
- ***event*** and ***notification*** service (less Object-Oriented)

In addition to further operations (or services)

For lifecycle management, relational, transactional, concurrency control, security, …

CORBA COMPONENTS

The essential components of OMA architecture, i.e., CORBA, associated to an ORB:

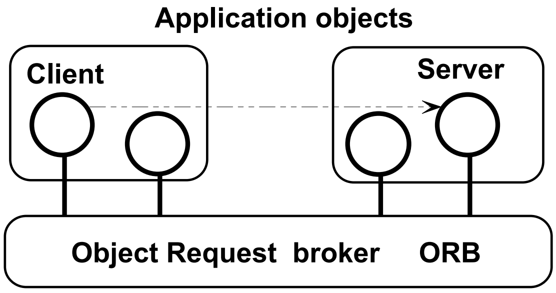
- **Object Request Broker** (ORB)
- **Interface Definition Language** (IDL)
- **Basic Object Adapter (e POA …)** (BOA e POA)
- **Static Invocation Interface** (SII)
- **Dynamic Invocation Interface** (DII)
- **Interface e Impl. Repository** (IR e IMR)
- **Integration Protocols** (GIOP)

Those components are at very different level

ORB CONTINUOUS SUPPORT

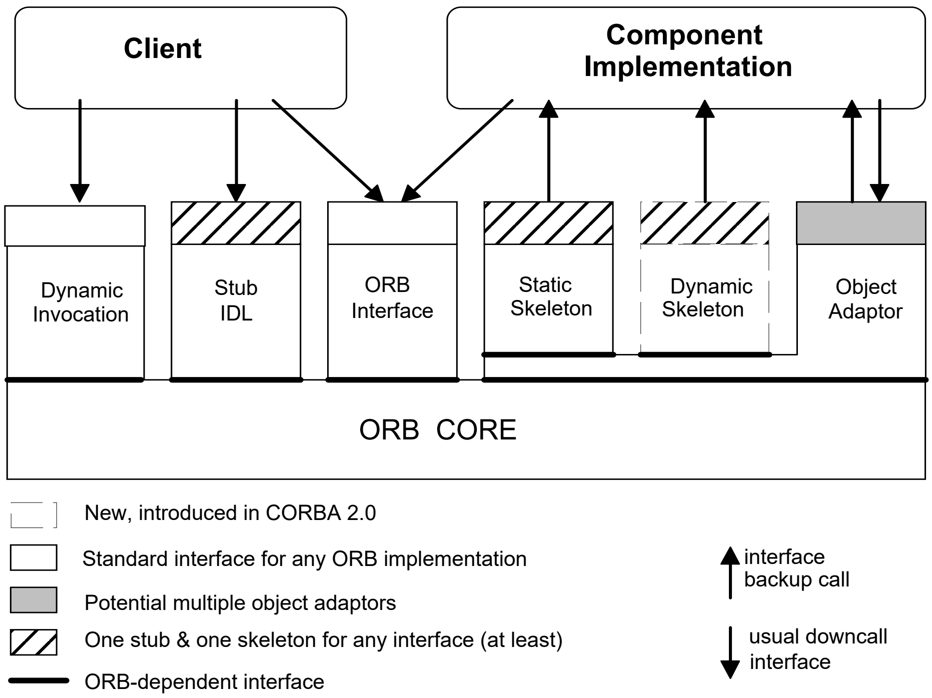
Object Request Broker (ORB) must **coordinate invocation of local and remote services (dynamically)**

- Identify **implementation of an abject** as a servant to requests (object location)
- prepare the **servant** to receive the request - via *adapter* (object creation, activation & management)
- transfer **the request** from the client to the servant
- return **reply** to client



CORBA: DYNAMIC VISION

Elements in action: overall user view



view of CORBA 1.x

not changed until CORBA 3

COMMON LANGUAGE in CORBA

Interface **D**efinition **L**anguage (**CORBA IDL**) must **identify** and **coordinate requested and offered services, local** and **remote (for** either **static** or **dynamic interactions)**

- Both **servants** and **clients** can **identify themselves** to **make themselves** mutually **known**
- Both **operations request** and **service** offers can be **optimally associated**
- CORBA reuse the experience from already developed and available **IDLs** for defining a general multi-language IDL

Unfortunately IDL prescribe predetermined identification and link and statically recognized (CORBA static binding)
And if we want bindings unknown at development time?

CORBA IDL for MULTILANGUAGE

Interface **D**efinition **L**anguage (CORBA IDL) **coordinates requested and offered services identification**, **with different languages**

```
interface Factory  //OMG IDL
  {         Object create(); // CORBA object or reference
  };
```

This interface permits to refer an object of type Factory (IDL) and to request the **create** operation (without **in** or **out** parameters) that returns a generic CORBA object (`type Object`, that is a reference to the object of interface `Object`)

IDL makes possible to define **new interfaces and new** general **types** and **abstract**, by need, to make them available and registered, and eventually concretely usable inside different language environments

CORBA does **not** provide any **object creation** (neither Factory): the creation is inside language environments and predefined there, outside CORBA scopes (the same as C does not provide any I/O)

CORBA IDL → STUB E SKELETON

The **Interface Definition Language** (CORBA IDL) allows to generate **support component** (**stub** and **skeleton**), for communication and data, inside **different languages**

The **stub** enable working on the *message from the client perspective* (marshalling) and acting as client proxy

The **skeleton** collaborate with the ORB *accepting service request and adapting it to the server* (unmarshalling), by managing requests and responses

DEPLOYMENT

Typically, there is a **static link** between ***interface - client - servant*** *(not between* **client** and **servant***, but between* ***client - service*** *and* ***service - servant****)*

*The **objects inside their different language environments** are bound to the stub and skeleton before execution*

(stub and skeleton are objects? no)

CORBA ADAPTER

Adapter (**Object Adapter**) **system component** to overcome **inhomogeneity** and **differences** among implementation of different **service environments** of different servants

(the Adapter does not connect with data presentation)

The Adapter is on the **server** side, with typical tasks of:

- object ***registration*** functions
- object ***external reference*** generation
- object and **internal process *activation*** *even on demand*
- *requests* ***demultiplexing*** to uncouple them
- ***send requests*** (upcall) to registered objects

Firsts adapters were Basic (**BOA**), then Portable (**POA**)

(OA are also CORBA objects? no, as OA are pseudo-objects)

INTERFACE REPOSITORY in CORBA

Interface Repository allows to know details about every **IDL data type** and to explore **interfaces**, exported from existent objects and available during execution

The interfaces are translated to different programming languages (**static binding**) where components are defined and compiled (**language mapping**)

IR allows to know and manage available interfaces **dynamically** and to **decide at runtime (dynamic binding)** what is available and convenient

Allows overcoming static approach: for example for a ***gateway*** *that allows access to CORBA interfaces of an environment and cannot be recompiled for every new interface*

IR service description system (it is not a naming system)

(IR is an object? yes)

ORB and IR in CORBA

In CORBA, **ORB is the middle enabler of any (remote) execution and operation request between different entities**

Every request **is always delivered** via the ORB and then server-side mediated BY the adapter

The ORB do not know about any **type information,** that are outside his scope and contained inside stub, skeleton and **language environment**

Interface Repository works as a **dynamic catalogue** of **interfaces** (not necessarily for **static** stub and skeleton),

And it is present for **dynamic explorations** at runtime**,** if it is necessary to retrieve information on dynamic interfaces

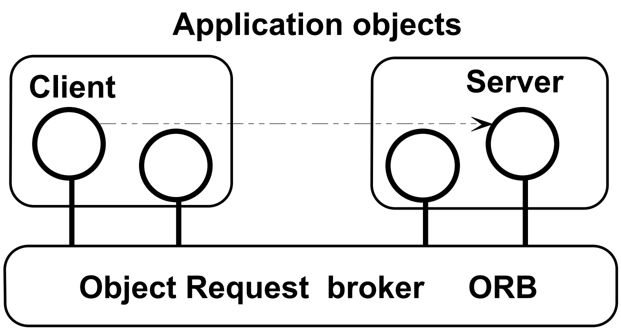
The interfaces must be always registered within the IR at their time of use and before consultation

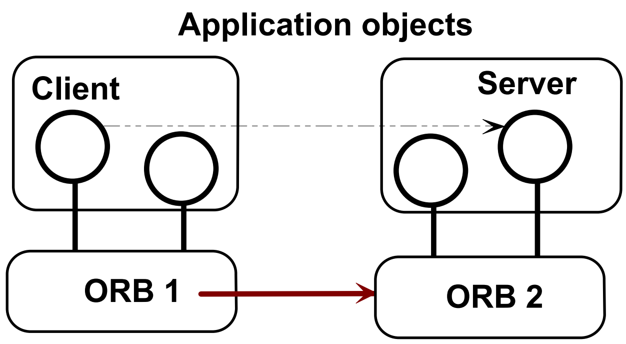
In the **static case**, the IR is generally not needed (its function is plaid by proxies)

DIFFERENT ORB SYSTEMS

ORB for communication of objects (intra-ORB) and also for communication between objects in different ORBs (inter-ORB)

In one CORBA system or in more CORBA systems managing different brokers





DIFFERENT CORBA SYSTEMS

Definition of Inter-ORB standards to establish how to integrate different CORBA systems without problems

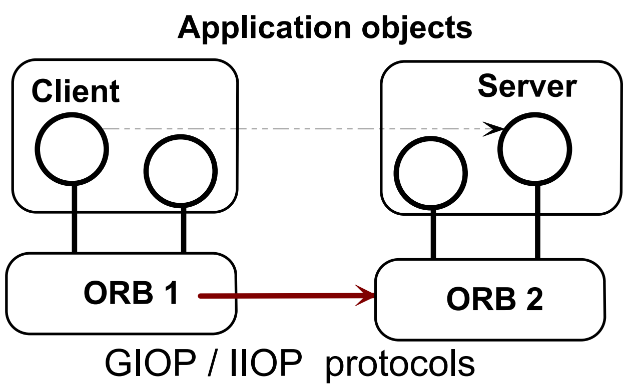
Necessity of standard protocols **ORB-to-ORB interoperability**

General **I**nter-**O**RB **P**rotocol (**GIOP**) that prescribe a standard message format

CORBA specifies a protocol between different ORBs in terms of architecture and data exchange

Binary Communication protocol: data are optimized and non user-readable (no source)

Common Data Representation (**CDR**) standard



INTER-ORB PROTOCOL: GIOP e IIOP

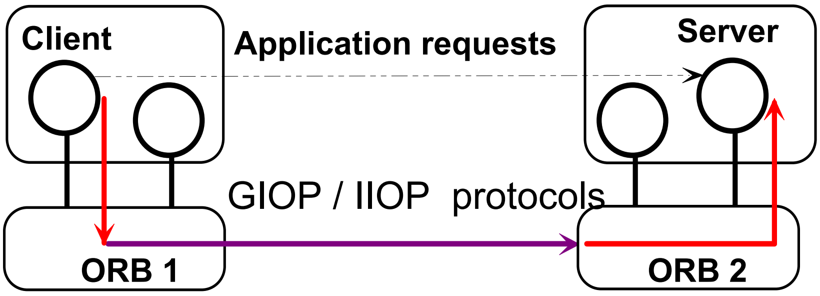
Definition (since version 2) of Inter-ORB Protocols to precisely the interaction between different CORBA systems

ORB interoperability protocol

General **I**nter-**O**RB **P**rotocol (**GIOP**) - **Binary protocol**

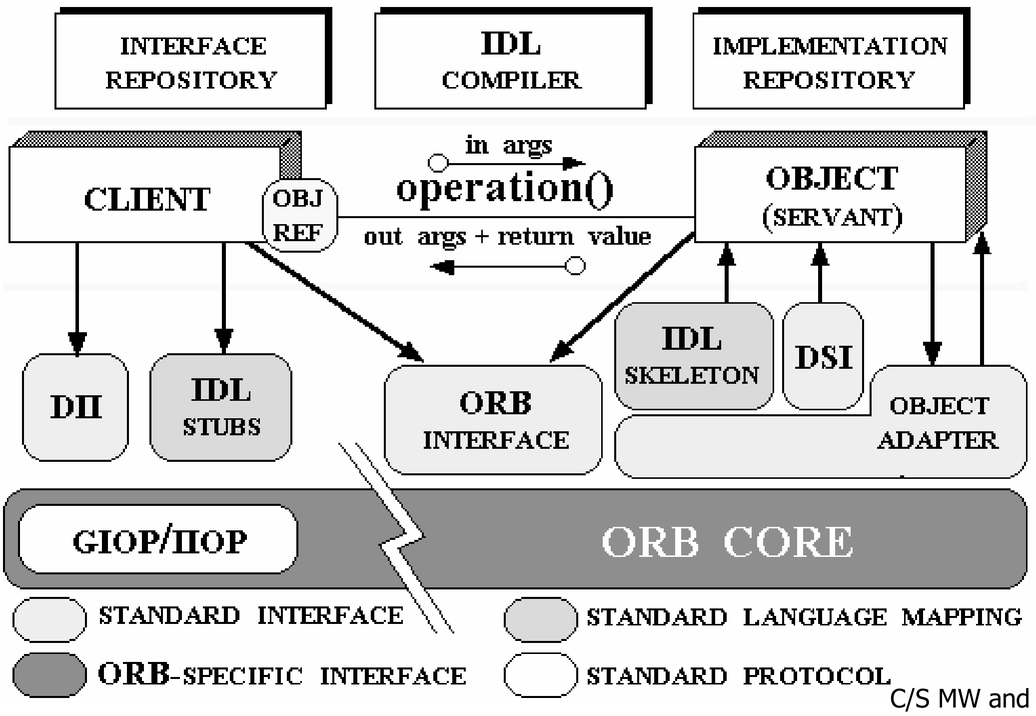
Common specification of data representation, data format, interaction with transport messages (semantic assumptions: reliable, connection, …)

for Internet using TCP/IP - **I**nternet **I**nter-**O**RB **P**rotocol (**IIOP**)



CORBA ARCHITECTURE

Overall picture of a communication between ORBs



CORBA: PSEUDO-OBJECTS

Support components and pseudo-objects

Stub generated from IDL interface for a specific language
Skeleton generated from IDL interface for a specific language

These components realize the **Static Invocation Interface SII**

The SII consists also of other architecture component, such as **IDL interfaces** (to generate stub and skeleton), (interface and implementation) **repositories** to find component specifications and implementation, and **object references**

The dynamic part is implemented in other **pseudo-objects**

DII, **D**ynamic **I**nvocation **I**nterface, or *Request* object
introduced for client dynamic invocation

DSI, **D**ynamic **S**keleton **I**nterface, or *ServerRequest* object
introduced for server dynamic invocation

ORB base functions

ORB acts as a coordinator, as an enabler, and as a manager of services available on the system

CORBA applications produces **objects** that become part of the system beyond **application lifetime**

The **applications** and the **objects** are developed using **different environments** to represent **stable resources** that can act to request **methods** and **execute operations**

ORB intermediates any interaction and

- **coordinates requests from client objects**, **transparently** from the position and the implementation of remote objects
- **facilitates** and **manages communication through the** use of **references** to existing **servant objects**
- **supports** and **controls** the whole **interaction**

ORB functions

ORB is a fully object interaction enabler, by suggesting a default blocking synchronous interaction

ORB limits its interaction responsibility by delegating individual language environments for final execution

CORBA is **not responsible for object creation and moving**

CORBA employs **external remote references** that are **externally created** by language implementation environments that must define their service objects (**servant**)

CORBA obtains **remote references** via:

- conversion of **string references** and vice versa (objects referred and translated into strings - stringification, and vice versa)
- use of **objects directory,** by using name services (Trading e Naming service)
- **Passing of reference parameters** to **servants**

CORBA IDL

INTERFACE DEFINITION LANGUAGE (**OMG IDL**) has been introduced to grant flexibility over heterogeneous platforms

IDL are **declarative languages** to **specify interfaces** and **involved data** (for API parameters)

Many common IDL are **procedural**

* OSI **ASN.1** / GMDO

* ONC **XDR** (SUN RPC)

* Microsoft **IDL**

CORBA IDL is an **object-oriented** language (***derived from C++***)

Obviously, different IDLs are **not compatible** with each other, even if often are different only for **syntax** and **identification systems** and **entity names**

CORBA IDL

CORBA IDL is a purely **description language for data** and **method interfaces**

- description of **interfaces definition**
- **interfaces** as set of method and attributes
- **multiple inheritance** of interfaces
- **exception** definition
- automatic management of **attributes**
- **mapping** for **different languages** and environments

The compiler can obtain automatically stubs for clients/servants even using different languages

We must consider different **language mapping** **for references to servant objects** (in different languages)

CORBA IDL EXAMPLE

```
module Stock
{exception Invalid_Stock {}; exception Invalid_Index {};
 const  length = 100;

 interface Quoter {
   attribute float quote; readonly attribute float quotation;
  long get_quote(in string stock_name) raises (Invalid_Stock);
 };
  interface SpecialQuoter: Quoter {
   attribute float quotehistory [length];
   readonly int index [length];
   long get_next (in string stock_name) raises (Invalid_Index);
   long get_first(in string stock_name) raises (Invalid_Index);
 };
 interface CancelQuoter: SpecialQuoter {
   long cancelhistory (out float cancelledquote [length])
 };
}
```

CORBA IDL SUPPORT

For any attribute, an automatic access function is provided suited for permitted operations
(_get for readings and _set for writings)

```
attribute float quote;
float    _get_quote ();
void     _set_quote (in float q);
readonly attribute ind index;
float    _get_index ();
```

For any exception, the state (completion_status) provides information on behavior semantics

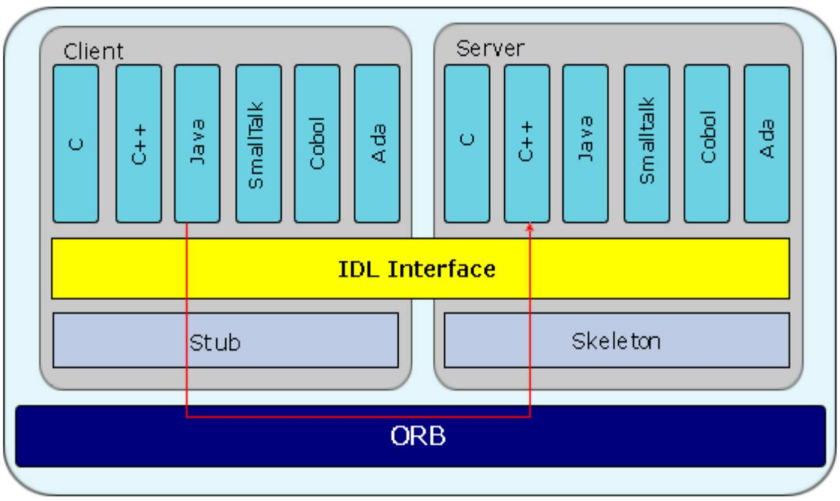
```
COMPLETED_YES,
COMPLETED_NO,
COMPLETED_MAYBE
```


CORBA IDL

Language to define CORBA interfaces, independently of a **specific programming language**

Naturally it is necessary **pass** from the abstract **CORBA level** to concrete **specific languages** (**language mapping**)

CORBA specifies the need of **mapping environments**
Servant creation is a responsibility of each language mapping



CORBA IDL ENVIRONMENT

CORBA is an **environment** where **we use remote references and do not move objects (static objects)** because of **the heterogeneity of single deployment environment**

Remote references allow to request operations to other components with known CORBA interface

Every object has **an interface** (**coarse granularity**)

Interfaces define: **attributes**, **methods**, **exceptions**

(*attributes accessed through* ***get*** *and* ***set*** *operations*)

(*operations with* ***in*** *or/and* ***out*** *arguments*)

The interfaces use **multiple inheritance**

The **interfaces** can be grouped also within **modules**
(for logical aggregations)

OTHER CORBA IDL EXAMPLE

```
module BankAccount {
struct transaction { string data; float amount;};
exception RedException {string message;};
typedef sequence <actions> list_ops;
interface Account {
  float balance(in string cc);
  list_ops bankStatement (in string cc);
  void withdrawal (in string cc, in float amount,
     out float balance) raises RossoException;
  Account accountTwin(); // returns an object };
};
```

Parameters passed by value (**CORBA** objects **by references**)

Problem of parameter handling in `out` and `in out`

DATA in CORBA IDL

Types in CORBA

Object Reference (references to **objects or interfaces**)
vs. even with inheritance between CORBA objects

Value (values copy) and **Exceptions**

Basic values short, long, ushort, ulong, float, double, char, string, boolean, octet, enum, Any

Constructed values Struct, Sequence, Union, Array

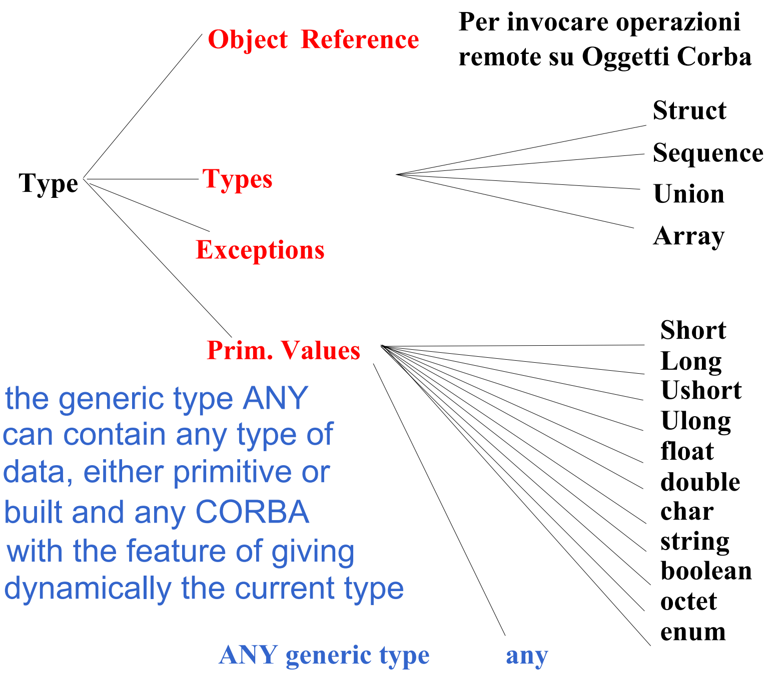
Any as general type that contains any type, primitive or from CORBA interface (analyzable during execution)

Object by value *(CORBA 3)*

*Objects that **cannot** be accessed remotely but only passed **by copy** from an environment to another one overcoming heterogeneity of different environments (no remote reference to them)*

TYPES in CORBA IDL

Types in CORBA IDL



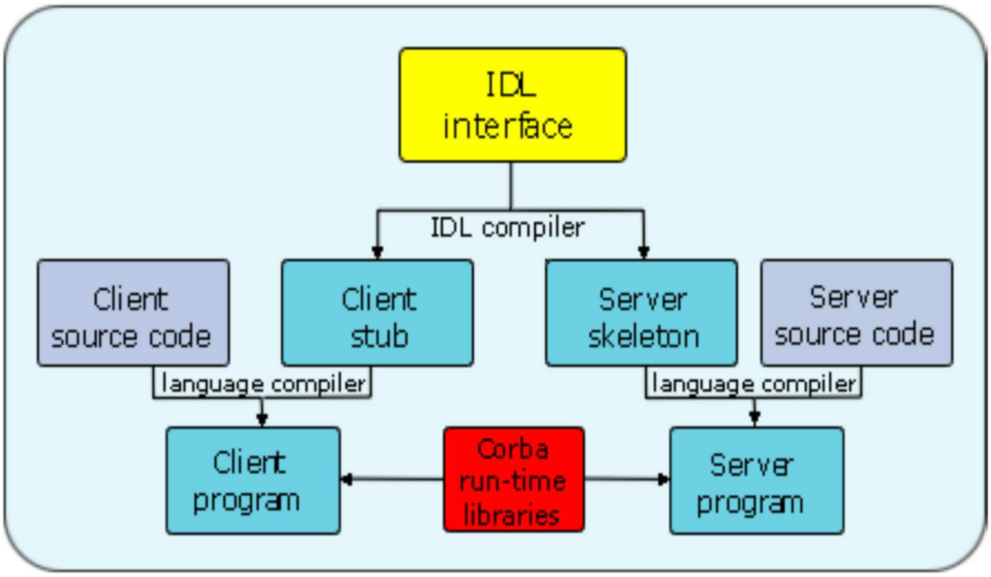
Types of CORBA IDL are than translated into types of different programming languages obtained for different language mapping

Type Object (IDL) represents any type of CORBA object without any information of the specific type

From CORBA IDL to Languages

Tools allows to build from CORBA IDL different components, essential to the project and to execution in **different language mapping**

stub and **skeleton**
+
file helper
and **of other help** (**holder**)
+ other operations



CORBA Language mapping

CORBA defines

interfaces (with inheritance), **exceptions**, **methods** with **objects as parameters** of different types and with different **modes** (`in`, `out`, `in out`)

Different languages must add **tools,** to harmonize their **structures** to obtain **interface conformance** and guarantee **run-time operations (OO languages must integrate inheritance)**

Strategy for consistency of concrete language types and possibility of integrating with the CORBA model

various transformation functions provided automatically management of types, to put together structures in simple way,

Apart from many other support functions (naming, trading, and suggested development methodologies) usable by user

CORBA vs LANGUAGES: HOLDER

Use of **holders** in JAVA as language where are output parameters for example

public final Class **BalanceHolder …**

{public float value;

public **BalanceHolder**() {}

float **_read**() {return value;}

void **_write**(float value) {this.value = value;}

};

for **`out`** and **`in out`** parameters (also other helps: helper)

In general, every **language** must create anything that is necessary to foster development inside its environment

CORBA HELPER

Helper use for Language mapping: in Java functions to

- **harmonize and treat language types and CORBA types**

*In Java the **CORBA Object** type is mapped in org.omg.CORBA.Object*

functions of **narrow**-ing that transform from the CORBA Object type to the one defined inside the interface

functions used for managing transformations from abstract CORBA type for the specific concrete type of interest

- **implement various utility functions**

functions for **reading** and **writing** a type on an object stream (associated to CORBA interface), to **treat type dynamically** during execution, …

Every **language** must guarantee interoperability with CORBA

CORBA ENVIRONMENTS AVAILABILITY

Widely used and still rising

`Object Broker`	`DEC`
`ORB`	`HP`
`DSOM`	`IBM`
`Orbix`	`IONA`
`Visibroker`	`Borland`
`(DOM Facility)DOE`	`Sun Studio Sun`
`PowerBroker`	`ExperSoft`
`JacORB, ...`	**`Open source tools`**

Even if the learning curve is high and there is overhead in performances

NOT ONLY C/S: ADVANCED C/S MODELS

Many variant of the Client/Server model

Novel variants

pull (synchronous non blocking)
(the client get afterwards the result, without waiting for it)

push (synchronous non blocking)
(the server gives the result afterwards to the client that do not wait for it)

delegation waiting for the result **(synchronous non blocking)**
(the delegate *waits for the client* and gives it the result)

notification for the result
(the delegate notifies the client that a result is arrived)

events (typically asynchronous, so non blocking)
(an event is generated from producer and advertised to consumers)

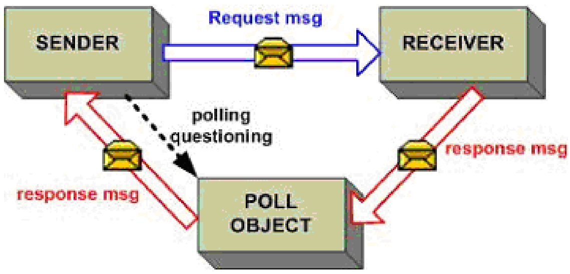
provisioning
(other parties can be interested in the call chain, apart from C/S)

DELEGATION – GET THE RESULT…

In a synchronous non blocking model, we may have a delegated entity for handling the result

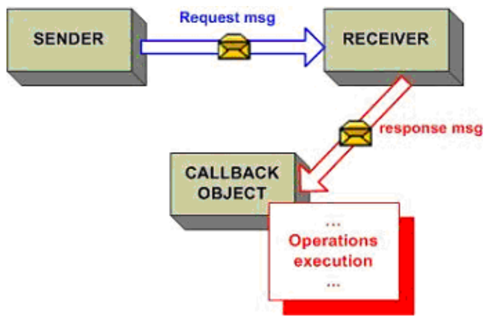
We add a new objects, typically called **Poll and Call-Back objects as intermediate entities**

Poll Object



Used for short operations response time

Call-Back Object



Even long operations and limited and independent from the client life cycle

We should define specifically the organization in any case

MESSAGE EXCHANGE

Model of MESSAGE exchange

very flexible but primitive, not user friendly

Sometimes the **message are only for the synchronization (signals) without any real data communication (carrying no information)**

Information exchange: **properties**

- **a/ synchronous** (no / result)
- **a/ symmetric** (the same knowledge of partner)
- **in/ direct** (intermediate entity or not)

Implementation

- **non/ blocking** (un/blocking of the sender)
- **un/ buffered** (non / message queuing)
- **un/ reliable** (with/without message loss)

Models with **multiple receivers** or **group messages**

multicast (MX) and broadcast (BX)

MODES of MESSAGE EXCHANGE

MESSAGE EXCHANGE varies a lot in different systems

Rendez-vous

One to one message exchange that is synchronous, blocking, symmetric, unbuffered, coupled (more than C/S)

With an intermediate entity (channel, …)

Message exchange typically asynchronous, non blocking, asymmetric, **decoupled** (less strict than C/S)

With intermediate entity & receivers group (events, …)

Message exchange typically asynchronous, non blocking, asymmetric, **decoupled and many to many**

C/S vs MESSAGE EXCHANGE

Client/Server

Model with strong coupling
implies **co-presence of interacting parties**

Mechanism suitable for high-level and simple communication

Very **high level** (very suitable for application usage)
but **not so flexible** for differentiated situations,
no Multicast (MX) and Broadcast (BX)

Sender/Receiver message exchange

Model with loose (minimal) coupling
imposes no **co-presence of interacting parties**

Very flexible, primitive, and expressive mechanism, maybe not so easy to use

Very **low level** (and suitable for any system potential usage):
many **differentiated modes of usage**, even easy support to any kind of needed communication, e.g., any form of **MX** and **BX**

DE / COUPLING

Communication tools can impose some ***constraints on the interacting entities*** ***(also no imposition)***

These constraints can even induce severe limitations on the interaction and force knowledge needs sometimes not required

Different ways of coupling

- space

The interacting entities must know each other and be co-located

- time

The interacting entities must be present at the same time (they should share some intervals of time)

- synchronization

The interacting entities must wait for each other and are subjected to reciprocal limitations and blocks

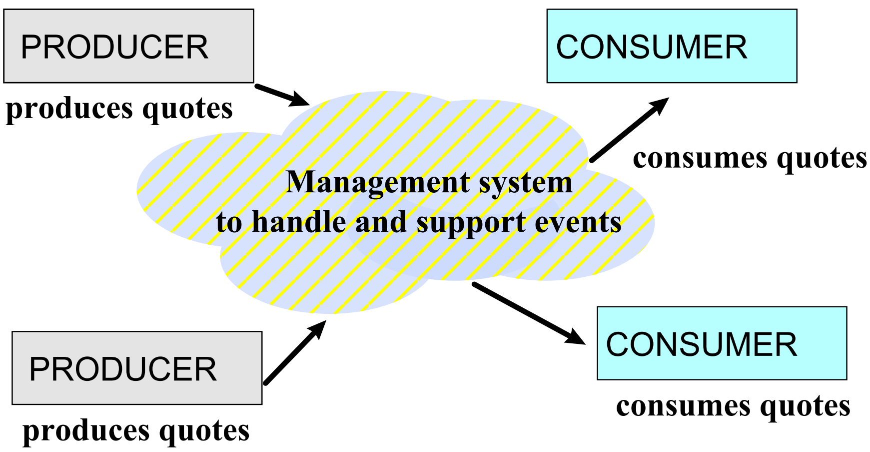
Decoupling becomes a factor to enable greater flexibility and to leverage the potential distribution of the load in a system

EVENT and PUBLISH-SUBSCRIBE

Decoupling between interacting entities

Events are generated by **producers,** free of doing it when they intend to generate events (**publish or PUB**) ***without worrying about delivery***

Consumers register their interest in specific events, topics, … (they have **subscribed SUB**) and the **event support** is in charge of the delivery



Producers and **consumers** are not required to be **present at the same time**

FROM LOCAL EVENTS

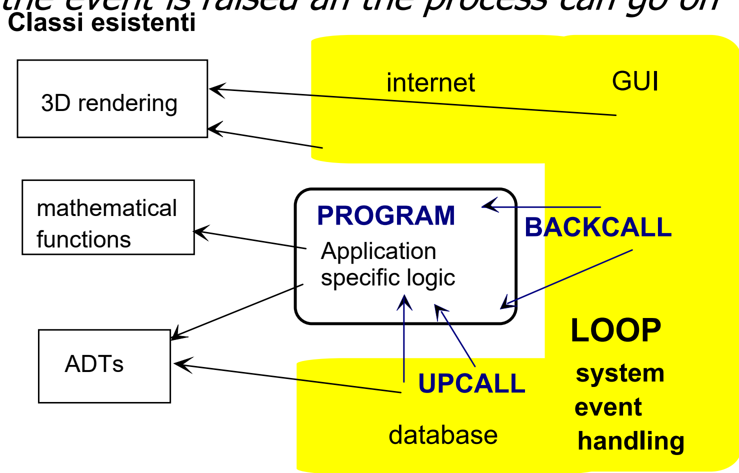
Different model than a synchronous requests of C/S t
The Framework tends to reverse the control for low level events

The user process does not wait for result but register with a handling action

Example: Windows asks all processes to provide a waiting loop to serve with the it is going to raise to them (and send to them)

When the result is produced the event is raised an the process can go on

Responses from the framework to the user are called **backcall** or **upcall**
They are similar to an asynchronous event generated by the framework and that application must manage through a **handler function** specified by the user



Available Services and Functions

EVENT SYSTEMS (DISTRIBUTED)

Event systems have been modeled and designed **without any locality constraints (no coupling)**

The model has its strength in the non-locality of interacting entities only local implementations

Local implementations are not interesting (*such as using the sharing on the same node, between producer and consumer*), ***arbitrary, and not meaningful downsizing*** *of the model*

Develop a system for events not taking into account the potential decoupling, ...

means to use badly the model properties, one of the worst things we can do to a technology

If you constrain the events to the co-residence and co-presence of interacting entities, you produce a deployment that contrasts with the basic event model

EVENT SYSTEMS: INDICATORS

Event systems have been defined to model **large systems** and **scalable ones**

Some **indicators are core ones**

- **Cost in distributing events (to limit)**
- **Performance (to optimize)**
- **Scalability (to keep high)**
- **Latency (da limit in time)**
- **Pervasivity of provided services (to keep high)**
- **Independent develop and execution (high)**
- **Fault tolerance (maximal possible)**

When you implement event systems you start from **viability**, to mean that you grant that the **indicators are scalable**, in other words for all distributed implementation indicators keep **acceptable values**, possibly **'costant'**… at least **tested**

EVOLUTION of EVENTS

Primitive events

some **events** are **on/off signals without any content information**

interrupt events and signals triggered by low-level handling functions

Events that carry contents

some contents carry information and one can also **filters** events based **on interest** about **specific information**

RSS as an example, where there is interest only to specified topic and users can register to specific interests

Events with quality - Quality of Service

These **events** can provide **differentiated service for different users:** they can persist and be maintained for all or some users, the delivery can be different depending on receivers, …

Persistent events*: users not online do not lose any event, kept to be delivered a.s.a.p. when they are on*

Event priority*, e.g., depending on the number of resources devoted to users*

PUBLISH-SUBSCRIBE SYSTEMS

PUB-SUB systems are **advanced distributed systems** based on the **event model** and **message exchange** to take the best advantage of the flexibility and the decoupling of interaction to increase **scalability and distribution**

The PUB-SUB model has also many other flexible aspects…

Message filtering based on

topic-based: based on a predefined topic (a specific interest between different channels: such as a specific RSS)

content-based: based on message contents (some keywords or also some more complex relationships)

type-based: based on message type (in case of different message types and a selection done on them)

Quality of Servizio (QoS) over messages

Persistency, Priority, Guarantee of maintenance and duration, …

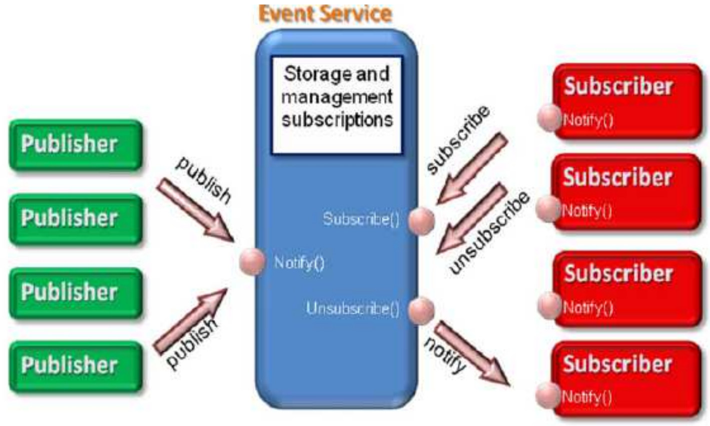
PUBLISH-SUBSCRIBE SYSTEMS

Real PUB-SUB systems support **operations for consumer subscription**

producers called also **publishers** provide events (they might ask which are current subscribers)

consumers or **subscriber** that have subscribed must receive events, via a notification

an **infrastructure** must ensure and grant the operations



DECOUPLED MODELS - TUPLE

TUPLE MODEL for loose and scalable interaction

A general model for **communication** and **synchronization**

designed as a *shared memory abstraction* + *communication*

A **tuple space** is a set of **structured relationships**, organized as a container for *attributes* and *values* for PUB-SUB

On a tuple space tuples can be deposited / extracted **high-level information** without **causing any interference** or incorrectness

A possible relationship: **message** (**from**, **to**, **body**)

The space is a container of **tuple values** according to the defined attributes

(the *attribute types,* here ASCII string)

Tuple values *message*: {**Antonio**, **Giovanni**, **msg1**}

{**Giovanni**, **Antonio**, **msg1**} {**Antonio**, **Giovanni**, **msg2**} …

There are no constraints on tuples that can be deposited and stay in the space forever (almost, it is a model) so **without time** or **space limits**

TUPLE - Linda (Gelernter)

Operations of **In** e **Out on the tuple space**

Tuple spaces offer operation always possible and correct for **readers (In consumer)** and writers (**Out producers**) competitors with access based on attribute contents

Out inserts one tuple in the space and **In** extracts one tuple from the space

The **Out** operation **emits a tuple** on the space available for a match with an In request and the tuple stays there until it is consumed by one corresponding In only

The **In** operation **extracts one matching tuple** from the space, if exists
If it dose not exists, the In waits until one is received for the **match that is based on pattern** on the attribute values

In case of match with **multiple tuples**, only one is **non-deterministically extracted**

Out: message (P, Q, text1)

In: message (?from, Q, ?body)

The **In** may have name of attributes for larger matches

The **In** waits for one tuple with the second attribute the string Q, and give to the consumer the values **from**(=P) e **body**(=text1) of the matching tuple

DECOUPLING TUPLE

Tuple spaces

The communication is rather **decoupled** and **asynchronous**

In time

A producer can deposit tuples and go away, and only after a **long time**, the consumer can arrive and get the tuples

In (reciprocal) knowledge (space & synchronization)

The consumers do not know the producers in any way, but only the tuple contents they cannot interfere in any way with production (*one* ***in operation*** *extract one tuple, other* ***in-s*** *are queued and wait for their matching tuples and* ***outs operations***)

In quality - QoS

Tuple spaces are **persistent** and their requirement is to **maintain deposited tuples without limit (in memory and time)** without any preference for a specific requesting process

Tuple spaces (local implementation) are available to favor local communication **well formed** and **with high level operations**

Javaspaces, …

GROUP COMMUNICATION

Communication within a set of processes

Broadcast e Multicast

How to send general messages either to all currently present processes in the system or to a subset of processes (a group) in the system?

In a **single location** you can easily achieve it (in the same LAN)

On different **networks and locations**, you cannot easily achieve it

expressive incapacity, excess overhead, lack of QoS, ...

There are some semantic problems to solve in multicast and broadcast

How to cope **with the answers** (if any)?

- **no wait – asynchronous** operations
- wait for **one answer only**
- wait for **some answers only** (how many?, how long?)
- wait for **all answers** (how many? how long? **When to stop**?)

GROUP COMMUNICATION

IP Broadcast

Broadcast limited and directed (inside local network)

IP Multicast **heavier duty and protocol**

Multicast for class D addresses

Local Multicast support **and …**

Internet uses **Internet Group Management IGMP** protocol since long ago (**RFC 1112** e **2236**) to **implement local multicast**

Often the **protocol could operate only on local subnetworks**, and it is implemented in different and not compatible forms

Multicast (more) global support

A multicast is realized by **flooding between networks**

a packet can traverse a node only once (node with state) and **is sent via any output queue apart from the one where it came in (how long to keep the state?)**

Traditional way of routing with simple and low cost (!) policies

IP GROUP COMMUNICATION

One can adopt some basic strategies with mechanisms

For example, we can use an a-priori dimensioning of time-to-live (TTL) of datagrams (to specify penetration and cost)

TTL=0 local send	TTL=1 local to connection
TTL<=32 local to area	TTL<=64 local to region
TTL<=128 local to continent	TTL>128 global

IP Multicast and the QoS?

How can we be sure that the message has been delivered (beyond best-effort semantics)?

There is a limited guarantee on IGMP implementations

that is

We do not know if messages were all delivered to all recipients and in which order

GROUP COMMUNICATION

IGMP as an example of local support to Multicast (RFC 1112 e 2236)

IP multicast allows to send a unique packet to multiple receiver in the same locality, by using class D names to identify a group, not necessarily a local one but spanning a few local networks

The IGMP needs a **support from management router**

Every local network must hosts at least an IGMP router capable of managing local incoming and outgoing traffic and it controls the group with IGMP messages. It is possible to provide more multicast routers

IGMP v1 considers only **two simple messages** with C/S approach

IGMPQUERY a **router** periodically verifies the existence of hosts that answer to a specific IP D address

IGMPREPORT a **node** signals a state change to the router related to the group (only **join the group** and no **leave**)

IGMP VERSIONING

IGMP v1

Routers are in charge of group management

There is only a **join** message**,** but no **leave** message from the group in v1
Any router has always an **active** role that require to regularly emit queries: nodes reply to the query to signal their presence or do not reply (problem with nodes that **answer late to the first join query**)

this version requires group operations (*only one single report from a node for a single local network*)

IGMP v2 (support for join / leave)

The second version consider the capability of nodes to send a message of **explicit leave** (i.e., leave the address group)

Nodes that leave the group must notify the manager

More routers can be in charge of the management

Interference between router is settled with IP numbers order

ROUTING MULTICAST PRINCIPLES

Multicast must employ the least resources as possible during data transmission to receivers

Some assumptions tend to obtain an optimal use of resources and to avoid an excess of bandwidth

- **single sender** support
- **variable number of receivers** support (**up to n**), that can be added or removed dynamically

The main idea is to maximize **sharing,** so to **send only one copy**, instead on N ones, of the same **multicast** message (1 message cost) **instead of different unicast** (**N** message cost)

Derived from assumptions, protocols identify a central **tree** starting from the sender with **optimal shared paths from sender to current receivers**

The goal is to employ most shared hops as possible from root to leaves

the continuously changing tree must consider only currently active receivers and disregard the ones where there are no currently active receivers

ROUTING MULTICAST (STABLE)

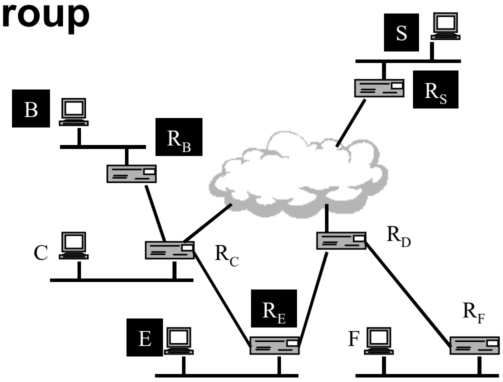
Multicast requires the **identification** of a (**dynamic**) **tree** from **sender to receivers for repeated forwarding**

the sender is the **root of the tree**, the intermediate routers are the **intermediate** nodes and identify subtrees**,** the receivers are the **leaf nodes** in the tree

- an open group of nodes with a **single sender**
- the group membership is **dynamic**
- leaves are responsible for **joining the group**
- **shared paths optimize bandwidth**

The tree is extremely dynamic

Consider the case where
an **host S** transmits and **B** and **E**
are in the receiver group



MULTICAST: SPANNING TREE

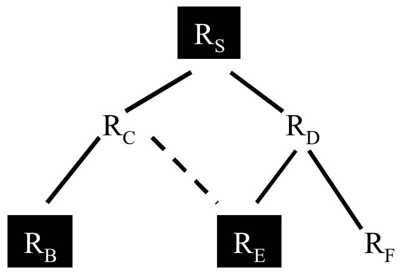
We consider only routers as participants (no nodes) and **we want to build a tree from the interconnection graph**

First step (request for leaf identification: root to leaves)

We want to build a tree (a **spanning tree**) that connects root to known leaf nodes, typically by using unicast routing protocol information and **organizing and aggregating paths**

- We start sending a **flooding** message towards **every** possible recipient with the main objective of creating a **bone multicast**
- The root identifies shortest paths by building it from replies from receivers

some receivers nodes are reached through multiple paths



MULTICAST: MULTIPLE PATHS

Second step (go back from leaves to root)

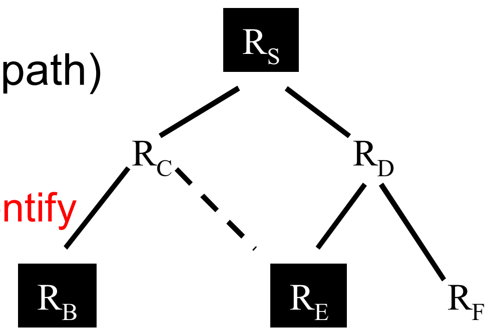
Every leaf signals direct **paths** (backwards) and can also **identify new paths** (even not shortest) for going from root to leaves

- *minimal path messages* are sent backward from leaves to root
- only some paths are selected, other are discarded
- some *shortest path messages* from the source are sent back in a **larger scope**: they are forwarded from leaves on all exit links, except the one where it was coming (to identify other better paths not traversed from root to leaves)

Reverse path forwarding (backward path)

For every router reached from several path, the root can so select the best

Re is reached from Rd but it can try to identify other routes in order to determine new shared parts



MULTICAST in ACTION

Normal routing: normal routing operation must work continuously while **tree identification is ongoing…**

Distance Vector

Next hop information must be used (or use poisoned reverse) in order to block too long paths

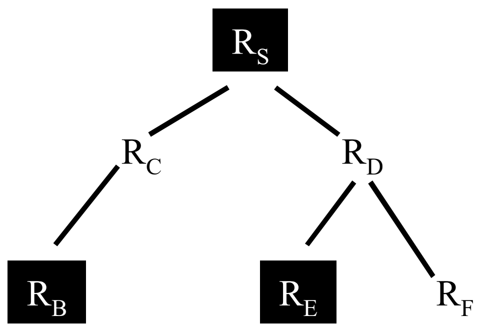
Link State

All shortest path trees must be built for every node and use “tie break” rules to settle conflicts

Reverse Path Broadcast (2 step) for **deleting Multiple Paths**

Leaves send a broadcast **towards the root** during normal routing operations

The root receive new paths and can reorganize the tree trying to aggregate several sub-paths and produce an optimal tree



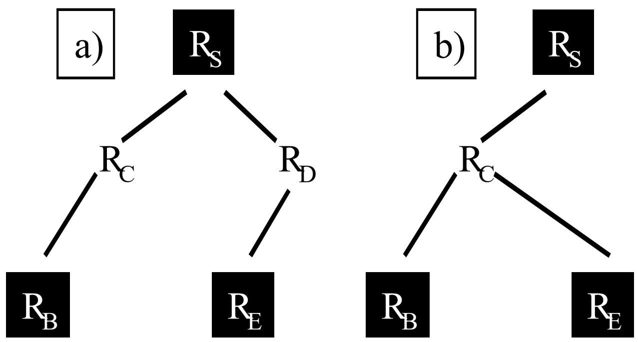
REVERSE PATH BROADCAST

Reverse Path Broadcast allows to choose between different paths to organize the optimal tree, while minimizing the number of sent messages and used bandwidth

With a broadcast from leaves (the **Reverse Path Multicast**) it is possible to **find paths, connecting leaves with the root, that have not been previously explored**

It is up to the root to choose the best tree organization

Reverse Path Multicasting (RPM) to reorganize the tree (even with a high cost)



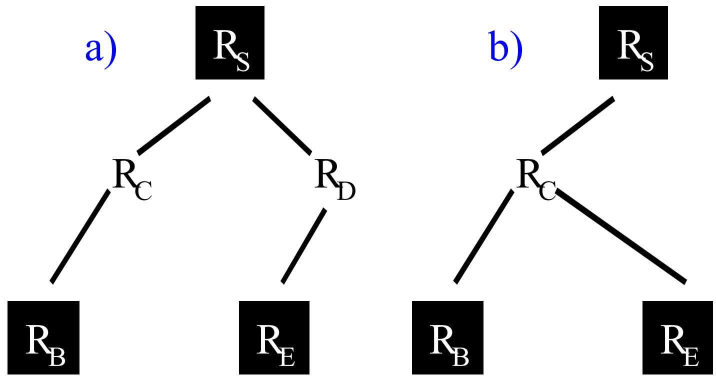
MULTICAST: PRUNING and GRAFTING

PRUNING and **GRAFT**

routers that have no receivers connected are excluded with '**cut**' messages that flows throughout the tree

The tree must be rebuilt in case of any modification

Reverse Path Multicasting (**RPM**) autonomously done by the leaves to consent **PRUNING** - from a) to b) and reinserting parts of the tree **GRAFT** - from b) to a)



REVERSE PATH MULTICAST

Reverse Path Multicasting from leaves to root (not a broadcast)

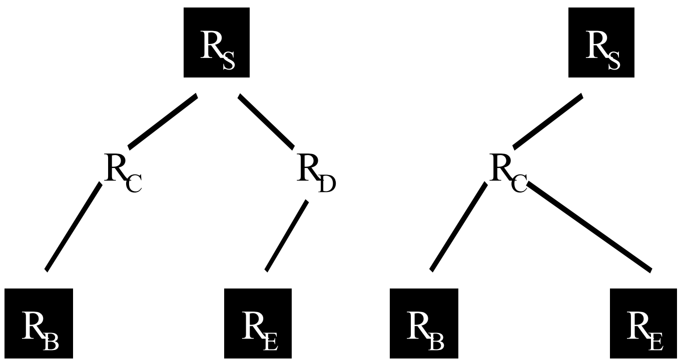
- used in a lot of multicast protocols
- keeps the state for communication **per-sender**, **per-group**

Networks with no members are **pruned out** from the tree and new ones can reenter the group (**explicit graft** from the bottom) without reorganizing the tree from scratch

The state (software) is kept for a limited and predetermined time

SOFT-STATE

The definition of the **RPM time interval** is **critical**



DIFFERENT MULTICAST PROTOCOLS

There are many different **routing multicast** protocols, **incompatible** with each other, even in competition between themselves and supported by different communities

DVMRP (**RFC 1075**) **D**istance **V**ector **M**ulticast **R**outing **P**rotocol

Employs RPM, based on a modified version of RIP and very used in MBONE (multicast backbone)

- Update messages are sent using special paths (tunnel) and using only some nodes

MOSPF (RFC 1584) **M**ulticast **O**pen **S**hortest **P**ath **F**irst Protocol

Extends link-state, suitable for big networks, based on RPM and soft-state

- It starts from networks map and uses them to calculate shortest path to every single destination
- It optimizes the trees and removes not used paths

MANY STANDARD MULTICAST

PIM (**RFC 2117**) **P**rotocol **I**ndependent **M**ulticast Protocol

Uses any unicast protocol in different ways so to suit different systems

Scattered intended when there is a low probability of multiple nodes on the same LAN and **Dense** where there are many neighbors routers

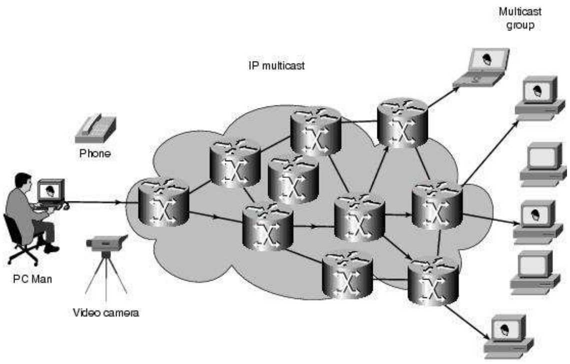
- **Scattered**: removing the most number of intermediate router to simplify the tree structure
- **Dense**: use of flooding and prune, simplified with regard to DVMRP

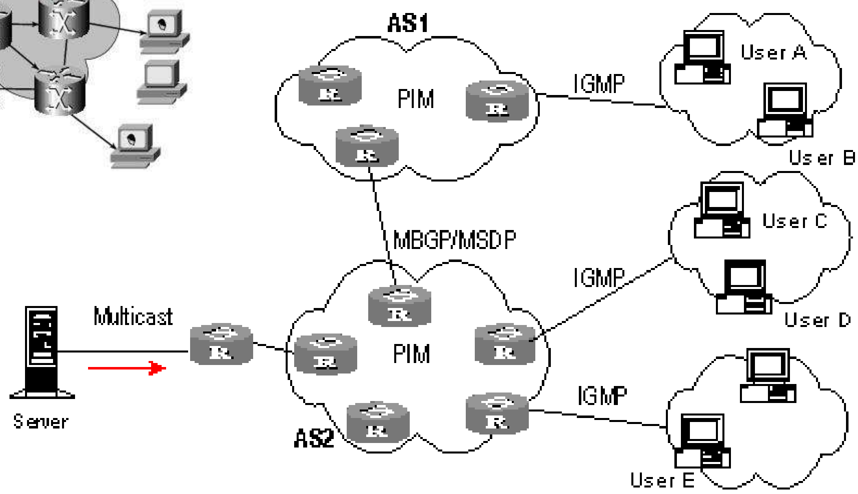
CBT (**RFC 2201**) **C**ore **B**ased **T**rees

suitable for an organization based on core routers to choose

- Some **nodes are fixed** (core) and **trees are unified** without defining a per-sender or per-group state
- It is possible to use sub-optimal tree organizations to avoid reorganizing connection for every multicast reconfiguration

MULTICAST PROTOCOLS





MOM MIDDLEWARE

Message Oriented Middleware (MOM)

Data and code distribution via **message exchange** between **logically separated entities**

Typed & **un-typed message exchange** with **ad-hoc tools** both **synchronous** and **asynchronous**

- **wide autonomy** between components
- **asynchronous** and **persistency** actions
- **handler (broker)** with different strategies and QoS
- easy in **multicast, broadcast, publish / subscribe**

Example: Middleware based on messages and queues
MQSeries IBM, **MSMQ Microsoft**, **JMS** SUN, **DDS**, **MQTT**, **RabbitMQ**, **Active MQ**, …

MOM DEPLOYMENT

The **specific deployment** and the **interconnection graph (OR)** is **always static** (**without the need of a name system**)

Network overlay model between different applications with specific support in distributed environment

- **Necessity of high-level Routing (as in ONs, but static)**
- **Data treatment** while communicating between **different environments**
- **Predefined and static** participating **entities**

Centralized model

MOM with a central node as hub-and-spoke that is responsible of support and pass messages between **different clients**

Distributed model

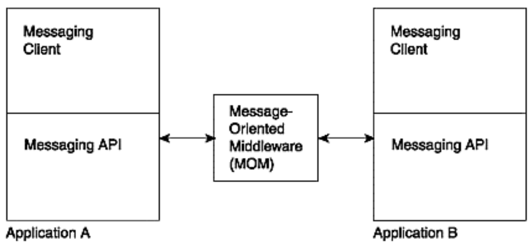
MOM is located on any client node to form a static ON network, that operate through P2P communication messages between nodes in need of communication

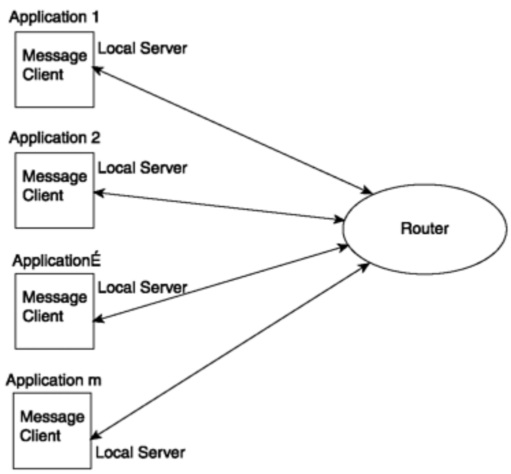
MIDDLEWARE MOM

MOMs provide simple and efficient services

Communication operations available via local ad-hoc API

MOMs put together different nodes and provide services on different **fruition nodes** arranging **queues for the support of every communication**





MOMs as **integrators use of routers, their interconnection and format conversion**

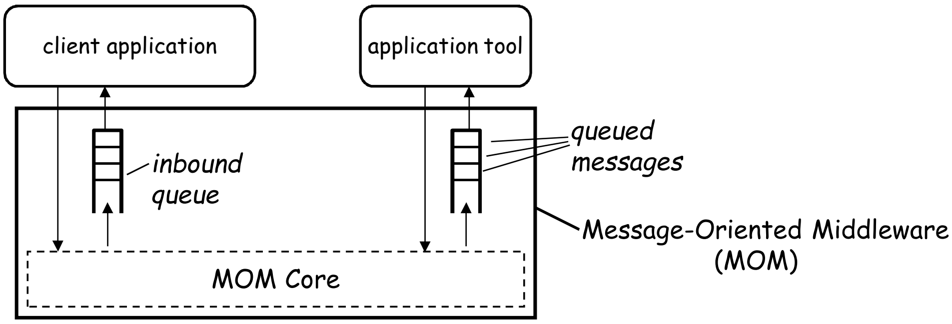
MESSAGE-ORIENTED MIDDLEWARE

MOMs use **queues local to interested nodes**

Inbound and outbound queues on interested **different machines** (connected in an univocal way)

Queue managers guarantee the expected operation level and message forwarding

Routing system to connect different queues (as an **overlay network** for application level routing)



MIDDLEWARE MOM or GLUE

By following a 'Glue' model

MOMs keep together **different autonomous systems** and organize their specific **interconnection**

Relay are **intermediate** entities that allow the implementation to scale and to organize high level **routing**

Message Broker are entities able to support message content transfer between **environments** with **different representations**

The **MOM** operations use **not** only **asynchronous point-to-point** messages, but also **many-to-many communications**

The **realization cost** must be **limited** and **reduced**: the main objective is to fast integrate **existent legacy systems**

MOM: MQSeries IBM

MOM proposal very popular and supported

Typically, the **interconnection graph** (**routing**) is controlled by an **always static** and **inflexible** system management (**no name servers and no dynamicity**)

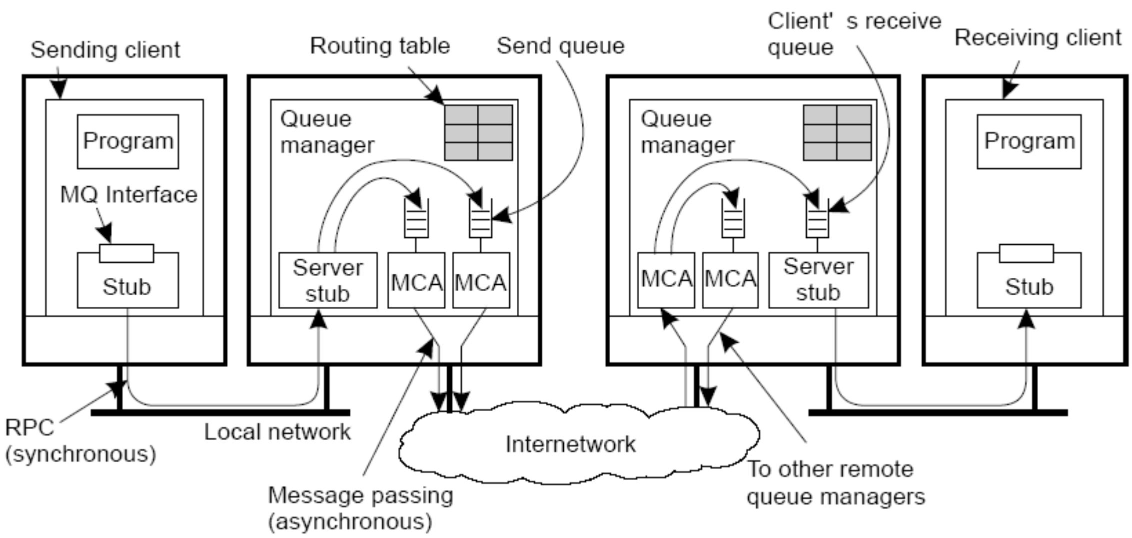
Application level messages are managed by a **queue manager**
Processes interact through *API RPC* to put/extract messages from local queues

Transfers are enabled by unidirectional **channels** managed by **Message Channel Agents** that deal with all details (different delivery politics, message type, etc.)

MCA coordination is offered via **primitives** that should enable flexible coordination (different activation policies, duration, maximum allowed cost, state persistence, etc.)

MQSeries IBM – Websphere part

For the deployment, **the system administrator** defines the **appropriate interconnections** by using **routing tables**, at the **configuration time**



MQSeries IBM: Broker

To achieve the best integration, an **MQ Broker** can operate on the messages by:

- modifying **formats**
- organizing **routing** based on **contained information**;
- working on **application information**, to **specify action sequence**