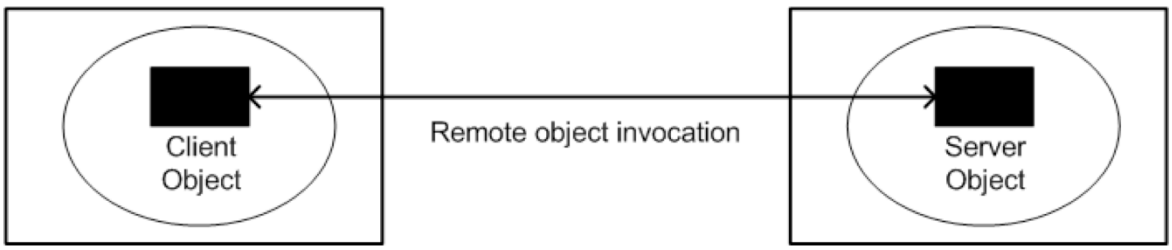
Chapter 2:
Middleware

1. Situating Middleware
2. Communication between distributed objects
3. Remote procedure call
4. Java RMI
5. CORBA RMI
6. Middleware Services

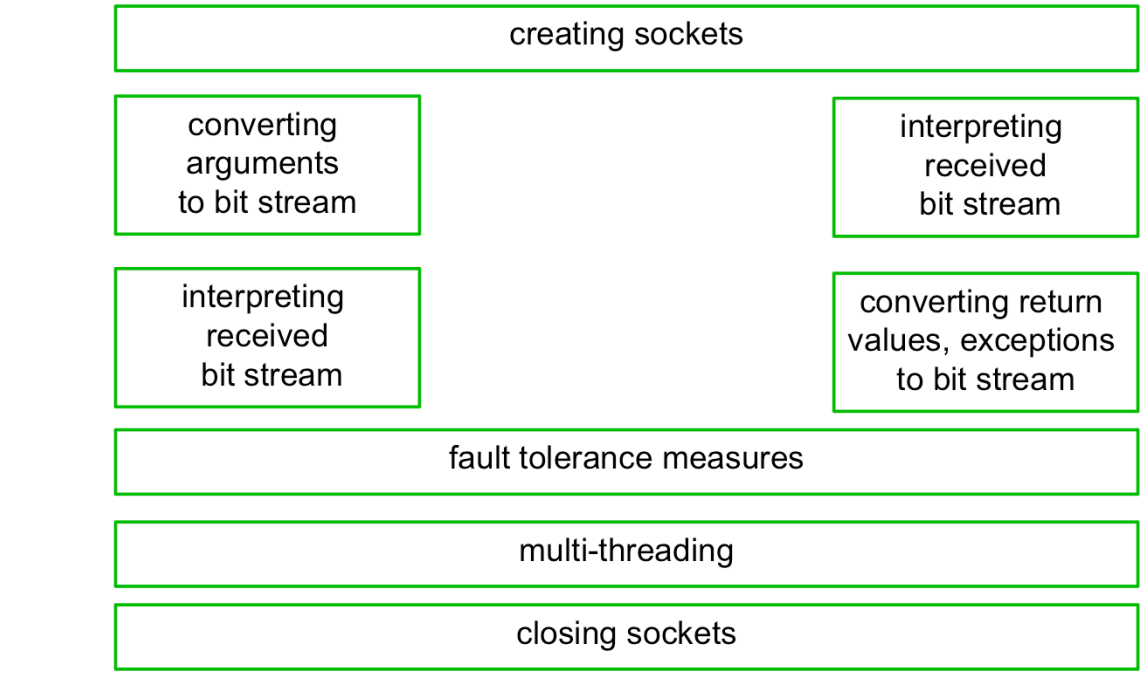
UNIVERSITEIT GENT

Remote invocation

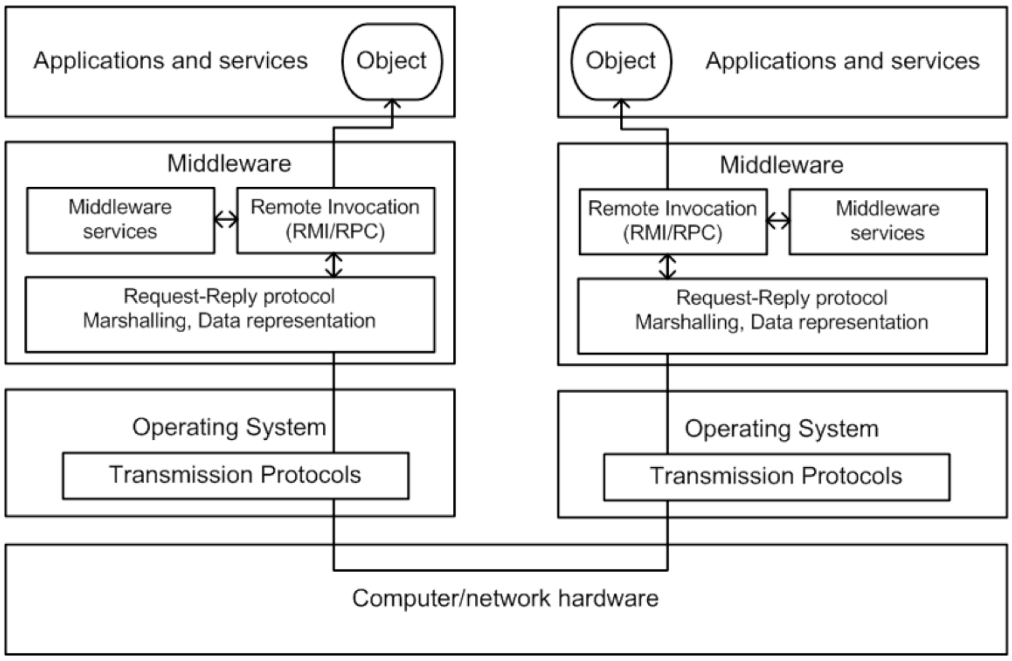


UNIVERSITEIT GENT

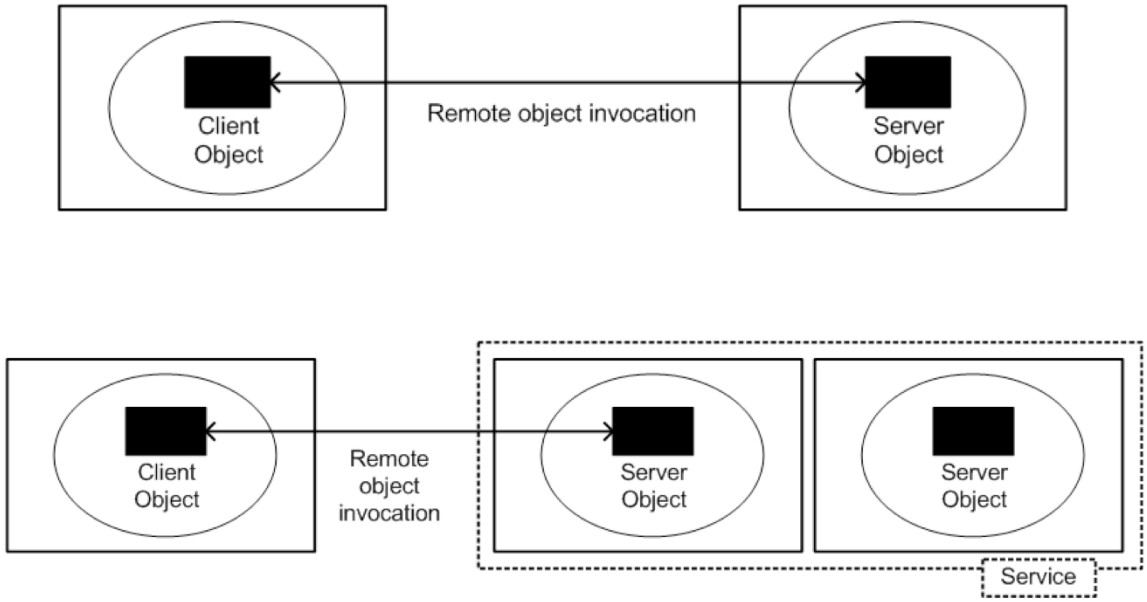
Remote invocation: requires …



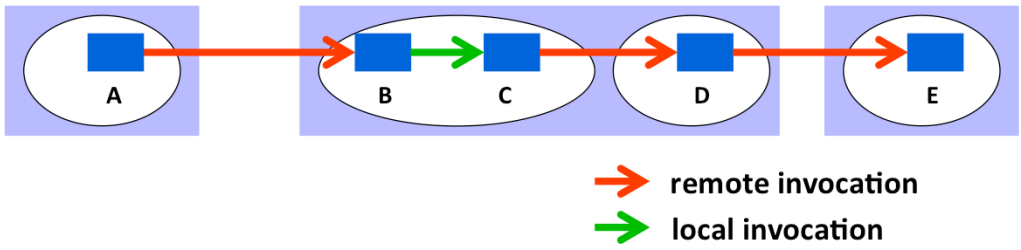
Middleware: positioning



Remote (service) invocation



Local and remote invocations



remote invocation

local invocation

Local invocation	Remote invocation
Use Object Reference Any public method	Use Remote Object Reference Limited access, remote interface

Invocation: syntax

The transparency argument

Make invocation syntax similar to local invocation, hiding:

- locate/contact remote object
- marshalling
- fault tolerance measures
- communication details (sockets)

But ...

Not everything should be hidden ...

programmer should know object is remote (network latency !) !

Typical approach

- same invocation syntax (but catch exceptions ...)
- object interface reflects remoteness

Remote interface : Java RMI example

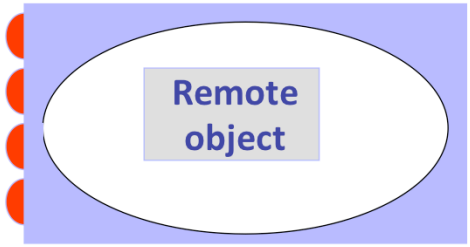
```
import java.rmi.*;

public interface UpperServerInterface extends Remote {
      String toUpper(String s) throws RemoteException;
      String getServerName() throws RemoteException;
}
```

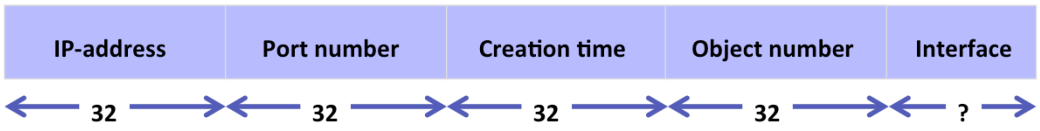
UpperServerInterface.java

Remote object references

- purpose : unique ID for objects in distributed system
  - uniqueness over time
  - uniqueness over space



Host : Internet Address
Process : Port number + process creation time
Object : object number
Object type : remote interface



Interface Definition Language (IDL)

Some facts ...

- **purpose : specify remote interfaces in a language neutral way**
- **standardized in 1990**
- **uses C++-like syntax (C++ has no interface construct ...)**
- **heavily used in CORBA (Common Object Request Broker Architecture)**

Syntax

keyword	usage
`interface`	-> specifies interface name
`attribute`	-> automatically make getters/setters
`in`	-> input parameter (read)
`out`	-> output parameter (out)
`inout`	-> input & output
`raises`	-> throwing exceptions
`Object`	-> supertype of all IDL types
`any`	-> anything can be passed (void*)
`readonly`	-> no setter, only getter for this attribute
`module`	-> groups interfaces and types in own name space (use with :: operator)

IDL type system

built-in types

```
short          unsigned short
long           unsigned long
float          double
char
boolean        octet
any
```

- use **`typedef`**-construct to extend IDL-types

- inheritance supported on interface level
  - C++ syntax
  - support for multiple inheritance


IDL example

```
#ifndef __ECHO_IDL__
#define __ECHO_IDL__

interface echo {
      string echoString(in string mesg);
      long   times_called();
};
#endif
```

echo.idl

More advanced IDL example

Whiteboard.idl

```
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() ;
};
```

differs from Java in that Java has classes but IDL does not.

CORBA must define anything that will be passed as argument or returned as result.

in Java the argument types are classes which can be defined in the language.

IDL module

- Modules allow interfaces and associated definitions to be grouped.
- A module defines a naming scope.

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

Chapter 2:
Middleware

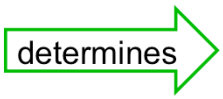
1. Middleware situation
2. **Communication between distributed objects**
    1. Invocation semantics
    2. RMI-architecture
    3. Request-Reply protocol
    4. Marshalling
3. Remote procedure call
4. Java RMI
5. CORBA RMI
6. Middleware Services

UNIVERSITEIT GENT

Fault tolerance

Techniques:

1. Retry-request message
2. Duplicate request filtering
3. Retransmission of results
    - 3.1 Re-execute call
    - 3.2 History table of results - Retransmit reply



Invocation semantics:

1. Maybe
2. At-least-once
3. At-most-once

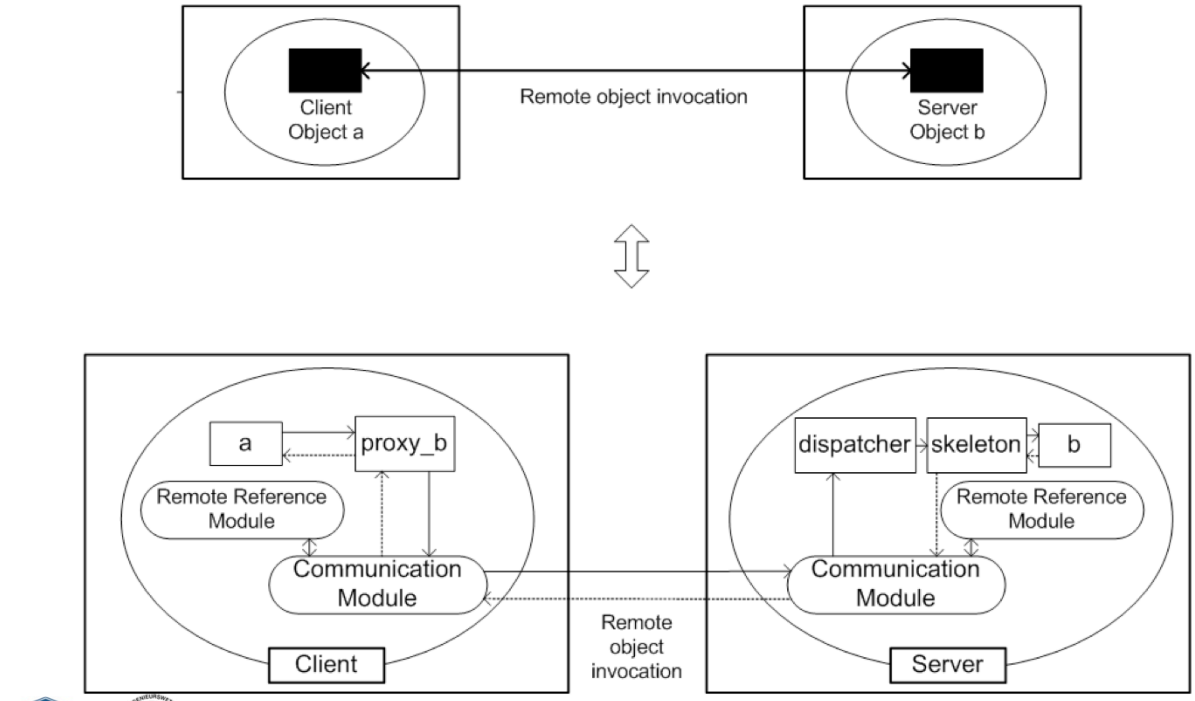
(Java RMI : at-most-once, CORBA : at-most-once or maybe Sun RPC : at-least-once)

Invocation semantics

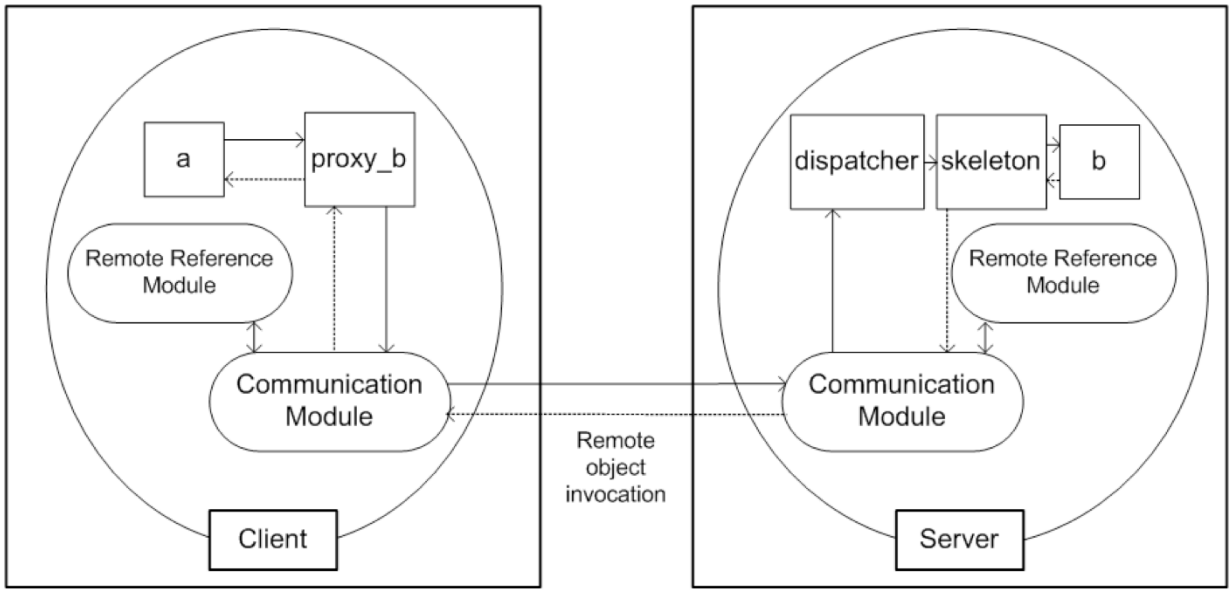
Retransmit request	Duplicate filtering	Re-execute call	Retransmit reply	Invocation semantics
NO	NA	NA	NA	*maybe*
YES	NO	YES	NO	*at-least-once*
YES	YES	NO	YES	*at-most-once*

(Java RMI : at-most-once,
CORBA : at-most-once or maybe,
Sun RPC : at-least-once)

RMI architecture



RMI architecture



Request-Reply protocol

Runs in the communication module

Purpose

- **send message to server object**
- **send back return value or exception info to requestor**
- **enforce appropriate invocation semantics**

Message format:

1.	2.	3.	4.	5.
messageType [int]	requestID [int]	objectReference [RemoteObjectRef]	methodID [int or Method]	arguments [byte [ ]]

1. request = 0, reply = 1
2. allows to identify matching request – reply pairs
3. marshalled remote object reference
4. method to invoke (e.g. numbering convention between communication modules)
5. marshalled method arguments

UNIVERSITEIT GENT

RR protocol: Duplicate filtering algorithm

if duplicate request arrives (check client + requestID)

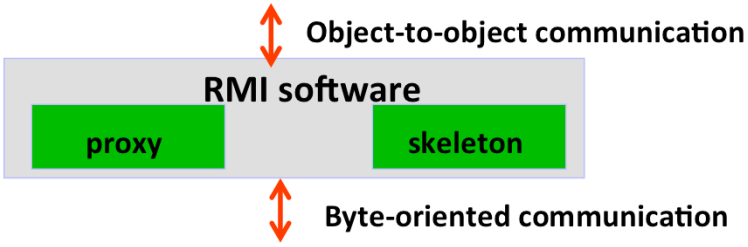
- discard if request handling still in process
- if request already executed
  - re-execute (idempotent operations)
  - retransmit reply (keep history)

scaling of history table ?

- timeout (FIFO)
- interpret new request from same client as acknowledgement


Marshalling

= transform object in memory to bitstream



Options

- **marshalling using external standard**
  - = External Data Representation
  - e.g. CORBA's common data representation (CDR)
  - Java serialization
- **marshalling using sender standard AND send this standard along**

Java standard serialization

- class implements `Serializable` interface
- writing
  - create `ObjectOutputStream (out)`
  - call `out.writeObject(<object>)`
- reading
  - create `ObjectInputStream (in)`
  - call `in.readObject()`
  - cast to specific class (result is of type `Object`)
  - BUT : class file must be reachable by virtual machine !

Standard serialization is very simple (the mechanism is quite powerful and well designed !). It suffices to implement the interface Serializable, creating appropriate input/output streams (ObjectInputStream and ObjectOutputStream respectively) and calling the read/write methods (readObject() and writeObject).

Implementing the Serializable interface literally involves nothing : the interface is empty and only serves as a marker for serializability.

When reading a serialized object, one must ensure that the “class”-file is reachable by the JVM (this is logical : only data is serialized – methods are found in the class file).

Java Serialization

Special care when:

1. `static` attributes
2. attribute contains reference to object of another class
3. attribute marked by `transient` keyword
4. update of serialized object and serialize again to same stream

RMI: Binding service

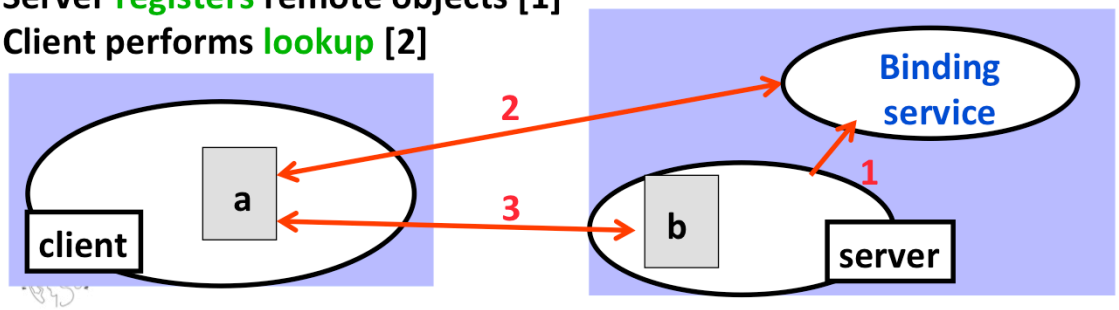
disadvantages of remote object references

- no logical meaning
- direct reference to physical location
  - server might crash -> automatic failover to other server ?
  - object might migrate ?
  - load balancing through replication not possible

binding service =

registry of remote objects reference
<-> textual name mappings

Server registers remote objects [1]
Client performs lookup [2]



Chapter 2:
Middleware

1. Situating middleware
2. Communication between distributed objects
3. **Remote procedure call**
    1. RPC architecture
    2. SUN RPC details
    3. Example : date service
4. Java RMI
5. CORBA RMI
6. Middleware Services

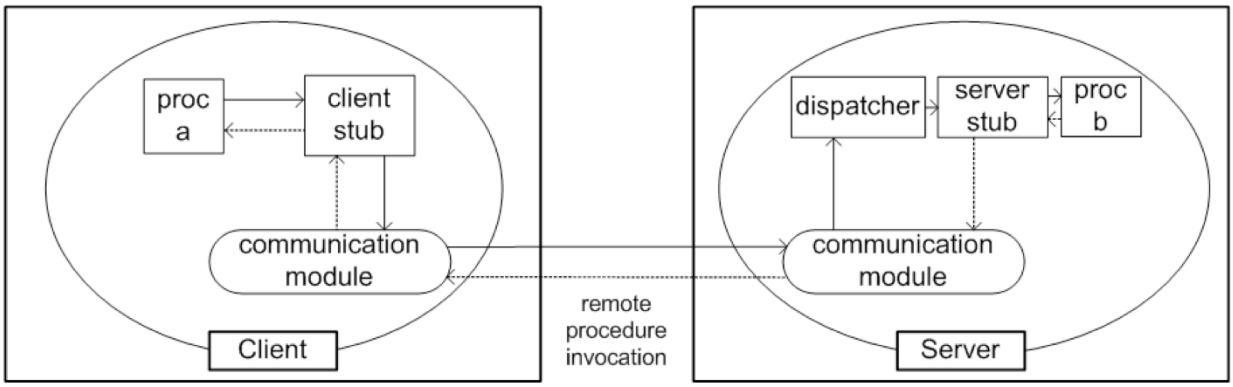
UNIVERSITEIT GENT

RPC History

= non OO version of RMI
"RMI for C programs"

- "invented" in 1976 (!), IETF RFC 707
- popular implementation "SUN RPC"
  (official name : ONC RPC (Open Network Computing)
- competing implementations not always compatible !
- programming language : mostly C
- used to implement Network File System (NFS)

SUN RPC Architecture



- no remote reference module
- dispatcher selects function stub at server side
- client procedure stub = RMI proxy
- server procedure stub = RMI skeleton
- stubs can be generated from service interface definition

Details

- marshalling : external data representation (XDR, RFC 1832)
- interface compiler : rpcgen
- binding service : portmapper

Steps for developing RPC-program

1. Interface definition (XDR)

   procedures have only 1 argument -> use C-structs

   additional keywords:

`program`	logical grouping of procedures
`version`	version control

2. Compile XDR-file with rpcgen

   -> client stub procedure

   -> server main procedure, dispatcher, server proc stubs

   -> XDR marshalling and unmarshalling procedures

3. Implement application logic (client and server proc's)
4. perform RPC in client code

   `clnt_create(hostname,program,version)` returns handle

   `clnt_destry(handle)` matching closing procedure

5. compile + run

UNIVERSITEIT GENT

Date example

(credits: Unix Networking, Stevens)

date.x

```
/*
 * date.x Specification of the remote date and time server
 */
/*
 * Define two procedures
 * bin_date_1() returns the binary date and time (no arguments)
 * str_date_1() takes a binary time and returns a string
 *
*/
program DATE_PROG {
        version DATE_VERS {
                long BIN_DATE(void) = 1; /* procedure number = 1 */
                string STR_DATE(long) = 2; /* procedure number = 2 */
        } = 1; /* version number = 1 */
} = 0x31234567; /* program number = 0x31234567 */
```

- Start numbering procedures at 1 (procedure 0 is always the ``null procedure''.)
- Program number is defined by the user. Use range 0x20000000 to 0x3fffffff.
- Provide a prototype for each function.

UNIVERSITEIT GENT

Date example : server side

dateproc.c

```
/*dateproc.c   remote procedures; called by server stub */
#include <rpc/rpc.h>          /* standard RPC include file */
#include "date.h"             /* this file is generated by rpcgen */

/*Return the binary date and time*/
 long *bin_date_1()
 {
    static long timeval;      /* must be static */

    timeval = time((long *) 0);
    return(&timeval);
 }

/*Convert a binary time and return a human readable string*/
char **str_date_1(long *bintime)
{
    static char *ptr;         /* must be static */
    ptr = ctime(bintime);     /* convert to local time */
    return(&ptr);
}
```

- No **main()** routine in server code.
- Extra level of indirection. **bin_date()** does not return a **long**, but **long***.
- return variable needs to be declared static

UNIVERSITEIT GENT

Date example : client side

rdate.c

```
/* rdate.c client program for remote date program */
#include <stdio.h>
#include <rpc/rpc.h>
/* standard RPC include file */
#include "date.h" /* this file is generated by rpcgen */
main(int argc, char *argv[]) {
   CLIENT *cl; /* RPC handle */
   char *server;
   long *lresult; /* return value from bin_date_1() */
   char **sresult; /* return value from str_date_1() */
   if (argc != 2) {
        fprintf(stderr, "usage: %s hostname\n", argv[0]); exit(1);
   }
   server = argv[1];
   /* Create client handle */
   if ((cl = clnt_create(server, DATE_PROG, DATE_VERS, "udp")) == NULL) {
    /*can't establish connection with server */
        clnt_pcreateerror(server);
        exit(2);
   }
   /*... */
}
```

Date example : client side

rdate.c

```
    /* ... */
    /* First call the remote procedure "bin_date". */
    if ( (lresult = bin_date_1(NULL, cl)) == NULL) {
         clnt_perror(cl, server); exit(3);
    }
    printf("time on host %s = %ld\n",server, *lresult);
    /* Now call the remote procedure str_date */
    if ( (sresult = str_date_1(lresult, cl)) == NULL) {
         clnt_perror(cl, server);
         exit(4);
    }
    printf("time on host %s = %s", server, *sresult);
    clnt_destroy(cl); /* done with the handle */
    exit(0);
}
```

- client calls client stubs (_1) suffix
- close handle after use !

Chapter 2:
Middleware

1. Situating middleware
2. Communication between distributed objects
3. Remote procedure call
4. **Java RMI**
    1. Philosophy and design choices
    2. Example
5. CORBA RMI
6. Middleware Services

Java RMI : philosophy

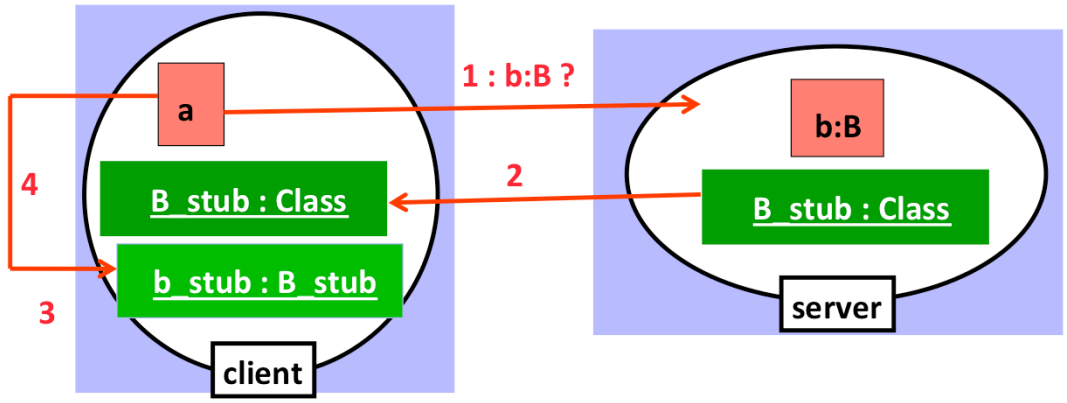
- **Java-only technology**
  alternatives : CORBA, SOAP

- **Remote invocation semantics identical to local ones BUT :**
  - client knows invocation is remote
    (MUST catch **`RemoteException`**)
  - server object MUST implement **`Remote`** interface

- **Terminology**
  - **client-side RMI-component (proxy) : stub**
  - **server-side RMI-component : skeleton**
  - **no dispatcher per class (single generic dispatcher)**
  - **binder : RMIregistry (1 instance on every server computer)**
  - **stubs and skeletons automatically generated by RMI-compiler rmic**

Java RMI : Code download

Automatic code download !



Easy to keep clients and servers synchronized (same version)

Security risk → Specialized security limitations for downloaded classes
Implemented in `RMISecurityManager()`

Java RMI : Parameter passing

- local invocation
  pass by reference
- remote invocation
  - remote objects : pass by reference
    (remote object reference)
  - non-remote objects : pass by value
    - make object COPY (MUST be serializable)
    - send copy to other side

Java RMI : Remote interface

- extends `java.rmi.Remote`
- all methods throw `java.rmi.RemoteException`
- has `public` visibility

```
import java.rmi.*;

public interface MyRemoteInterface extends Remote {
       int getData() throws RemoteException;
       MyRemoteInterface getNewObject() throws RemoteException;
       String getMessage(int n) throws RemoteException;
}
```

Java RMI : Remote objects

extend

- `java.rmi.server.UnicastRemoteObject`
- or `java.rmi.server.Activatable`

implement desired remote interface

```
import java.rmi.*;
Import java.rmi.server.UnicastRemoteObject;

public class MyServer extends UnicastRemoteObject
                implements MyRemoteInterface {
   // …
   public int getData() throws RemoteException {/* … */}
   public MyRemoteInterface getNewObject() throws RemoteException {/* … */}
   public String getMessage(int n) throws RemoteException {/* … */}
}
```

Java RMI : Registry

RMI registry must run on every server computer

- maps `String` <-> remote object reference
  - `String : //computerName:port/objectName`
- accessed through the class `java.rmi.Naming`
- server methods (registration of objects)
  - `public void bind(String name, Remote obj)`
  - `public void unbind(String name, Remote obj)`
  - `public void rebind(String name, Remote obj)`
- client methods (lookup of server objects)
  - `public Remote lookup(String name)`
  - `public String[ ] list()`
    (returns all names bound in registry)


Server Remote Interface

put String to upper case remotely

```
import java.rmi.*;

public interface UpperServerInterface extends Remote {
       String toUpper(String s) throws RemoteException;
       String getServerName() throws RemoteException;
}
```

Example : Server Object

```
import java.rmi.*;
import java.rmi.server.*;

public class UpperServer extends UnicastRemoteObject
                  implements UpperServerInterface {
      private String name;
      public UpperServer(String n) throws RemoteException {
            name=n;
      }
      public String toUpper(String s) {
            return "Server : "+name+" : "+s.toUpperCase();
      }
      public String getServerName() {
            return name;
      }
}
```

Example : Server program

```
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

public class UpperServerProgram  {
        public static void main(String[] args) {
                if(System.getSecurityManager()==null)
                        System.setSecurityManager(new RMISecurityManager());
                for(int i=0;i<args.length;i++)
                        registerServer(args[i]);
        }
        public static void registerServer(String s) {
                try {
                        UpperServer server=new UpperServer(s);
                        Naming.rebind(s,server);
                        System.out.println("Server <"+s+"> running ...");
                } catch(Exception e) {
                        System.err.println("Failing starting server : "+s+e);
                }
        }
}
```

Example : Client program

```
import java.rmi.*;
import java.rmi.server.*;


public class UpperClient  {
   public static void main(String[] args) {
        System.setSecurityManager(new RMISecurityManager());
        UpperServerInterface server=null;
        try {
                server = (UpperServerInterface)
                        Naming.lookup("rmi://localhost/"+args[1]);
                System.out.println("Reply from server : "+
                        server.toUpper(args[0]));
        } catch(RemoteException e) {System.err.println("Remote exception : "+e);
        } catch(Exception ee) {System.err.println(ee);
        }
   }
}
```

Java RMI : a Client recipe

1. Create and install RMISecurityManager
2. Lookup remote object
   (specify host, port number, textual representation of server object)
3. Perform remote invocations, catching remote exceptions

Java RMI : a Server recipe

Remote interface

1. Must be public
2. Must extend java.rmi.Remote
3. Each method must throw java.rmi.RemoteException
4. Remote objects type : use interface type

Remote Server object

1. Implement Remote interface
2. Extend java.rmi.server.UnicastRemoteObject
3. Constructor must throw RemoteException

Remote Server program

1. Create and install RMISecurityManager
2. Create server object
3. Register at least one server object

Chapter 2:
Middleware

1. Situating Middleware
2. Communication between distributed objects
3. Remote procedure call
4. Java RMI
5. **CORBA RMI**
    1. CORBA architecture
    2. Designing a CORBA application
6. Middleware Services

Situating CORBA

Socket Interface:

- no concept of methods, objects
- no services
- support for multiple programming languages

JAVA RMI/ C RPC

- concept of methods, objects
- services:
  - Binding service
  - Activation Service
- single programming language

In 1989, the need was generally recognized in industry for a middleware that allows applications to communicate irrespective of their programming language and that offers all required middleware services. Designing distributed application based on low level socket handling indeed also for support of multiple programming languages (client and server can be written in a totally different programming language). However, distributed application design through sockets is not very flexible as no middleware services are available for the developer. In the case of RPC or Java RMI, there is only support for a single programming language and the support for middleware services is limited (for instance, Java RMI offers a Binding service, Activation Service, and a “light” Persistent object storage service).

CORBA: Basic Idea

Need for middleware that allows applications to communicate irrespective of their programming language and that offers all required services

Recognized by OMG in 1989

- OMG = Object Management Group
- Industry Group of over 800 participants
- Their goal : design middleware that allows applications to communicate irrespective of (i) their programming language, (ii) their hardware and software platforms and (iii) the networks they communicate over.

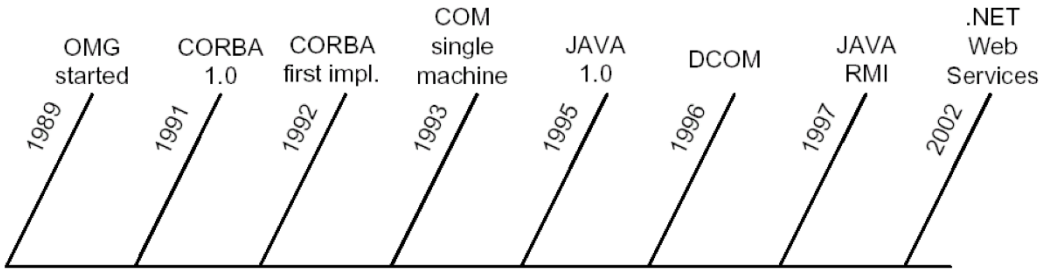
The OMG (Object Management Group) is an industry Group of over 800 participants, including the most important software vendors. In 1989, the recognized the need for a middleware that allows applications to communicate irrespective of (i) their programming language, (ii) their hardware and software platforms and (iii) the networks they communicate over. Their goal was to develop a common standard to be used by all participating companies.

CORBA ORB

OMG introduced ORB

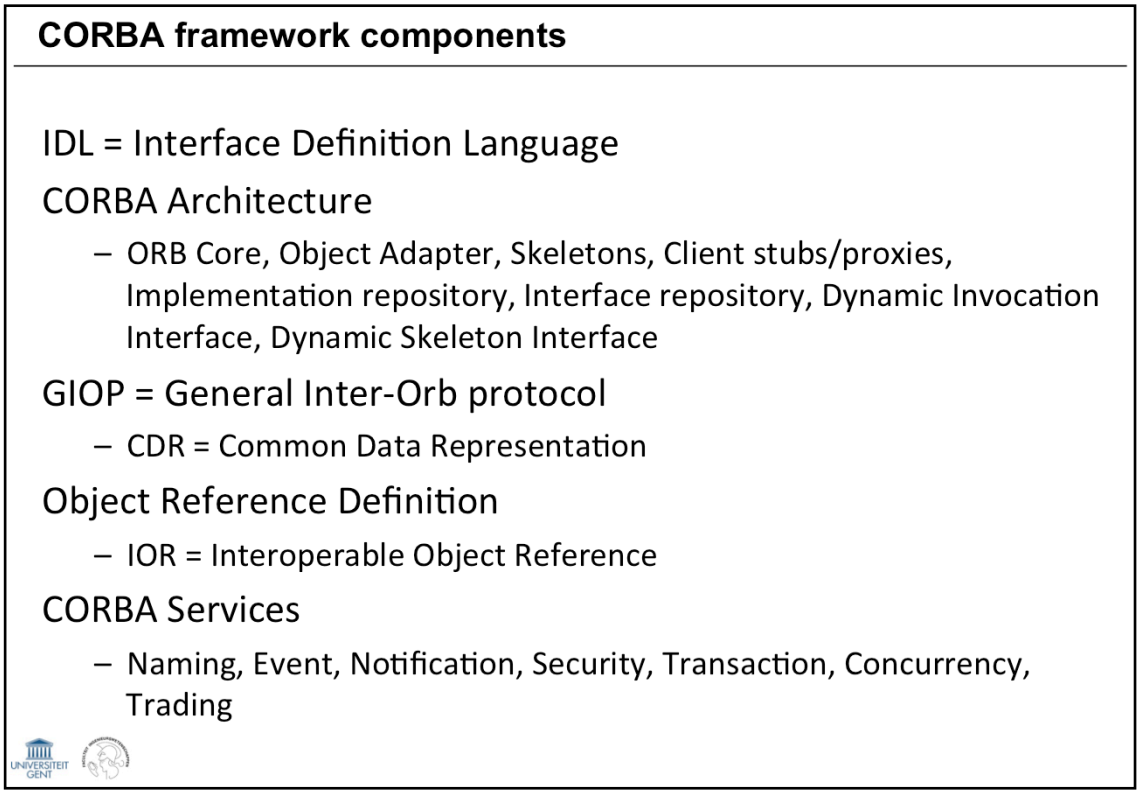
- ORB = Object Request Broker
- Software component which helps a client to invoke a method on an object

Timeline :



The OMG introduced the concept of an ORB (Object Request Broker), which is a software component that helps a client to invoke a method on a remote object. This ORB plays a central role in the CORBA architecture. A broker is an often used concept in distributed software (Dutch: makelaar): it offers a service to the interested clients by hiding the complexity of the underlying objects or components.

Due to the fact that CORBA was developed in the early 90ties and the fact that a lot of CORBA-based applications and tools were developed since then, CORBA is now generally considered as a mature software technology.



CORBA stands for Common Object Request Broker Architecture. There are five main parts in the CORBA framework:

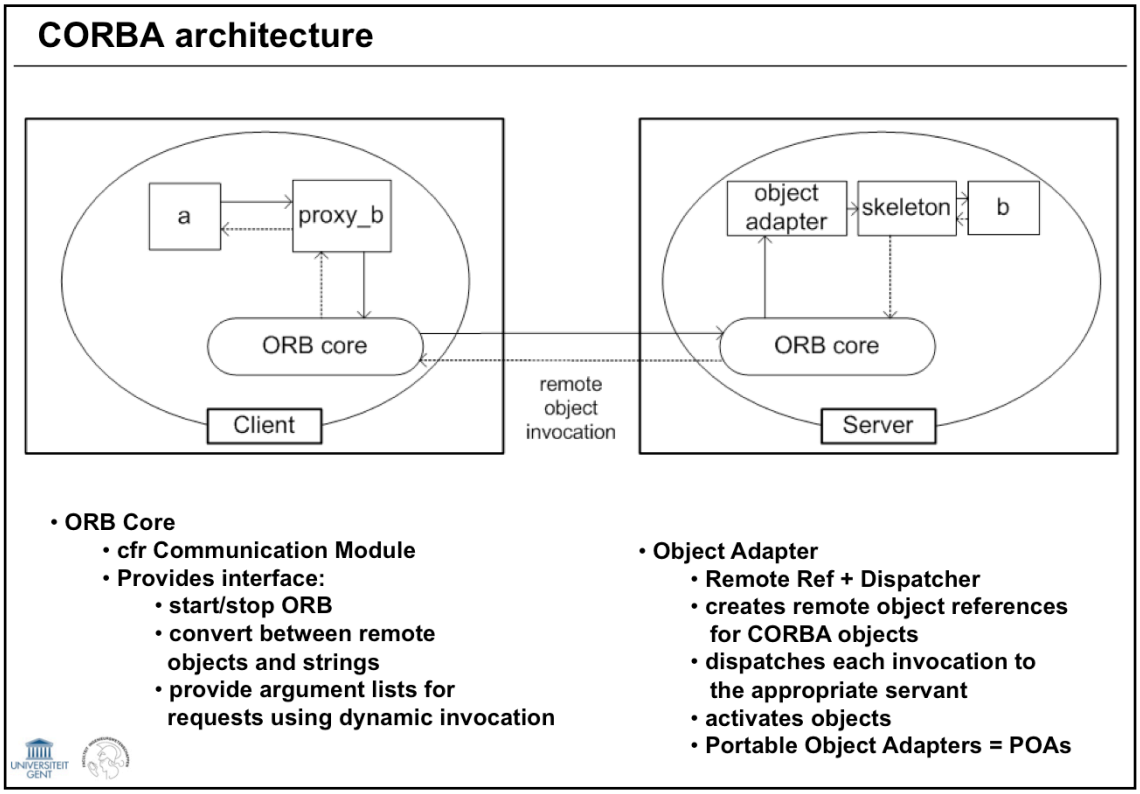
- IDL (Interface Definition Language): for defining the interfaces of the CORBA objects.

- CORBA Architecture: similar to the already detailed RMI architecture. The main components are: the ORB Core, Object Adapter, Skeletons, Client stubs/proxies, Implementation repository, Interface repository, Dynamic Invocation Interface, Dynamic Skeleton Interface.

- GIOP (General Inter-Orb protocol): the request-reply protocol for communication between the ORBs, CDR (Common Data Representation) is used for marshalling.

- Object Reference Definition: remote references of CORBA objects are referred to as IORs (Interoperable Object References), which are similar to the remote references, described in Section 1.

- CORBA Services: extra services for the applications developers, e.g Naming, Event, Notification, Security, Transaction, Concurrency, Trading.



The figure above shows the CORBA RMI architecture. In comparison to the general RMI architecture of lecture II.1, following new parts can be distinguished:

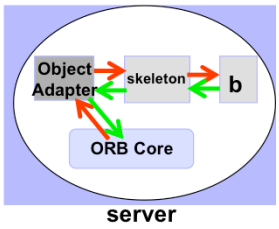
ORB Core: has comparable functionality to the Communication Module and provides an application interface to (i) start/stop the ORB, (ii) convert between remote objects and strings and (iii) provide argument lists for requests using dynamic invocation.

Object Adapter: provides the functionality of the Remote Reference and Dispatcher module, i.e. it creates remote object references for CORBA objects and dispatches each invocation to the appropriate servant.

Servant: the object implementing the business logic of the server object

(enumeration continued on next page)

Portable Object Adapter



object adapter : connects a request using an object reference with the proper code to service that request

Portable Object Adapter (POA):

Allow programmers to construct object implementations that are portable between different ORB products.

Allow a single servant to support multiple object identities simultaneously.

Overview of the advantages of the POA approach.

CORBA architecture (2)

Skeletons

- generated by IDL compiler
- remote method invocations are dispatched via appropriate skeleton to a particular servant
- unmarshals the arguments
- marshals exceptions and results

Client Stubs/proxies

- generated by IDL compiler
- marshals the arguments
- unmarhals exceptions and results

Skeletons: are generated by the IDL compiler and makes sure that remote method invocations are dispatched via appropriate skeleton to a particular servant. The skeleton unmarshals the arguments and marshals exceptions and results.

Client stubs/proxies: are generated by the IDL compiler, marshals the arguments and unmarshals exceptions and results

CORBA : pseudo objects

CORBA implementations provide some interfaces to the functionality of the ORB

- pseudo objects:
  - Cannot be passed as arguments
  - IDL interface
  - Implemented as libraries

Example: ORB interface

- represents the functionality that programmers need to access:
  - **`init`** method : to initialize the ORB
  - **`connect`** method : to register CORBA objects with the ORB
  - **`shutdown`** method : to stop CORBA objects
  - conversion between remote object references and strings

There is also the concept of pseudo objects, which have following properties:

- they cannot be passed as arguments

- implement an IDL interface

- are implemented as libraries

Depending on the CORBA implementation, there are some interfaces to the functionality of the ORB

An important example of a pseudo object is the ORB interface, which depending on the particular CORBA implementation can allow for following the functionality to the programmers:

- init method : to initialize the ORB

- connect method : to register CORBA objects with the ORB

- shutdown method : to stop CORBA objects

- conversion between remote object references and strings

The CORBA - IOR format is similar to the general RMI remote object reference.

CORBA IDL interface example

echo.idl :

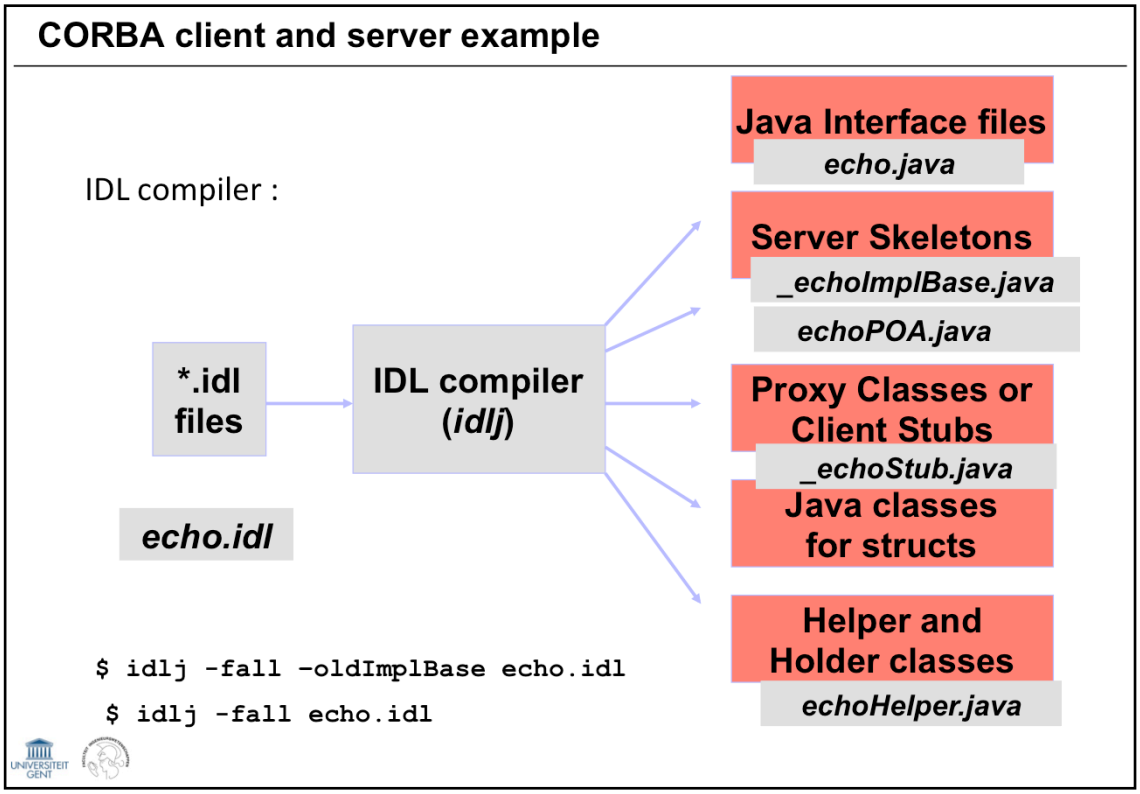
```
#ifndef __ECHO_IDL__
#define __ECHO_IDL__


  /**
   * The Echo interface which is basically an interface that contains an echo
operation and an accompanying operation.
   * @author AT&T omniORB, adapted by Filip De Turck
   * @version $Revision: 1.6 $, $Date: 2001/06/26 00:26:46 $
 **/
  interface echo {
    /**
     * Echoes the string received
     * @param mesg The received string which has to be echoed
     * @return The echoed string, which has to be <U>freed by the caller</U>
     **/
    string echoString(in string mesg);
    /**
    * @return The number of times the <code> echoString </code> operation is called
     **/
    long   times_called();
  };
#endif
```

CORBA : IDL

CORBA IDL

- IDL interface specifies a name and a set of methods that clients can request
- Parameters and results: in, out, inout keywords
- Passing CORBA objects, primitive and constructed types
- Type Object: common supertype of all IDL interface types
- Exceptions: defined in interfaces and thrown by their methods
- Invocation semantics: at-most-once



When writing a CORBA client server application, the following 6 steps need to be taken (detailed on this and the subsequent pages):

1. Specification of an IDL interface of the involved objects

2. Compilation of the IDL interfaces, which generates the stub code, the skeleton code and the header files (in case of C/C++ programming language) or interface files (in case of Java programming language)

Java interface file

```
/**
* echo.java .
* Generated by the IDL-to-Java compiler
* from echo.idl
*/

public interface echo extends org.omg.CORBA.Object
{
   public String echoString(String mesg);
   public int times_called();
} // interface Echo
```

Servant Class Implementation

```
import org.omg.CORBA.*;

class echoServant extends _echoImplBase {
   private ORB theORB;
   private int counter;

    public echoServant(ORB orb) {
       theOrb = orb;
       counter = 0;
    }

    public String echoString(String mesg) {
       counter++;
       String p = mesg;
       return p;
    }

    public int times_called() {
       return counter;
    }
}
```

3. Implementation of the Servant class: there are two options

   extending the corresponding skeleton class, often referred to as the BOA (Basic Object Adapter) approach: in this case the servant class implements the interface methods and uses exactly the method signatures defined in the equivalent Java interface, an ORB private attribute is used to connect new CORBA objects to the ORB (connect method) or shutdown objects (shutdown method)

   using a Portable Object Adapter (POA): allows programmers to construct object implementations that are portable between different ORB products and allows a single servant to support multiple object identities simultaneously.

The source code above details the BOA approach, more details on the POA approach are provided on the one but next slide.

Servant Classes

Extends the corresponding skeleton class

Implements the interface methods

Uses the method signatures defined in the equivalent Java interface

ORB private attribute: to connect new CORBA objects to the ORB (**`connect`** *method*) or shutdown objects (**`shutdown`** *method*)

Summary of the BOA approach, detailed on the previous page.

Server Class Implementation

```
import org.omg.CORBA.*;

public class echoServer
{
    public static void main(String args[]) throws
        org.omg.CORBA.UserException {
        try{
            java.util.Properties props = System.getProperties();
            ORB orb = ORB.init(args, props);
            echoServant echoRef = new echoServant(orb);
            orb.connect(echoRef);

            System.out.println("echoServer ready and waiting ...");
             orb.run();
        } catch (Exception e) {
             System.err.println("ERROR: " + e);
             e.printStackTrace(System.out);
         }
         System.out.println("Echo Server Exiting...");

    } //main
} // class
```

4. Implementation of the Server program: contains the main method (in case of C/C++/Java programs), depending on the servant implementation choice, two options for the code in the main method can be distinguished:

BOA approach: the ORB needs to be created and initialized, an instance of the Servant class is created and registered to the ORB (through the connect method), and subsequently waits for incoming client requests

POA approach: the rootPOA object is created first, next a POA object is created, the POAManager is activated, the servants are activated and the object references are created from the POA

CORBA server

Java class with *Main* method

Creates and initializes the ORB

Creates an instance of Servant class

Registers it to the ORB (through the *connect* method)

Waits for incoming client requests

Summary of the implementation of the Server program in the BOA approach, as detailed on the previous page.

Client Implementation

```
import org.omg.CORBA.*;

public class echoClient
{
    public static void main(String args[])

      java.util.Properties props = System.getProperties();
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);

      if (args.length == 0) {
           System.err.println("usage: java echoClient <object reference>\n");
           System.exit(1);}

      try {
        org.omg.CORBA.Object obj = orb.string_to_object(args[0]);
        if (obj==null) {
          System.err.println("Got a nil reference.\n"); System.exit(1);}
        Echo echoRef = echoHelper.narrow(obj);
        String src = "Hello!";
        String answer = echoRef.echoString(src);

        System.out.println("I said:\""+src+"\"."+
                     " The Object said:\""+answer+"\".");

        System.out.println("The function echoString on the object is called already "+
                           echoRef.times_called()+" time(s)");

      } catch(Exception ex) {
        System.err.println("Caught an exception!!.\n");
      }

      orb.destroy();
      System.exit(0);
   } //main
} // class
```

5. Client program implementation: contains a main method, in which the following takes place: creation and initialization of the ORB, invocation of the narrow method to cast an Object to particular required type, invocation of the remote methods, catching of CORBA System exceptions

Client Program

Creates and initializes the ORB

Narrow method to cast Object to particular required type

Invokes the methods

Catch CORBA System exceptions

Objects cannot be passed by value in CORBA

Summary of the implementation of the client program, as detailed on the previous page.

Running the server and client

Start orbd :

- UNIX command shell : $ orbd
- MS-DOS system prompt : start orbd

Start the Echo server :

- UNIX command shell :
  $ java echoServer
- MS-DOS system prompt :
  start java echoServer
- Result: echoServer ready and waiting ...

Run the client application :

- $ java echoClient *ior_server*
- Result: I said: “Hello!“. The Object said: “Hello!“.

6. Running the server and client: the orb daemon needs to be started on the involved machines (typical name: orbd, can differ from one vendor to another), next the CORBA server can be started on a certain machine. When it is started, the unique IOR of the server can be determined (and for instance printed to file). Next the client application can be started: in order to identify the remote server object, the IOR is required, and can be provided for instance as a command line argument. An alternative for using the IORs to identify the server objects, is by using the CORBA Naming Service (detailed further in this section).

CORBA implementations

Name	Language Binding	Services	Freeware
Jacorb	Java	Concurrency, Event, Naming, Notification, Trading, Transaction	x
omniORB	C++, Python	Event, Naming, Notification	x
TAO	C++	Event, Naming, Notification, Security, Time, Trading, Real-time Event, Scheduling, Load Balancing	x
Orbacus	C++, Java	Concurrency, Event, Naming, Notification	x
Orbix	C++, Java	Concurrency, Event, Naming, Notification, Transaction, Load-balancing	
VisiBroker	C++, Java	Concurrency, Event, Naming, Notification, Transaction, Load-balancing	

Some interesting URLs with links to:

- Commercial CORBA products
- Open Source freely available CORBA products
- SUN Java implementation of CORBA

Chapter 2:
Middleware

1. Situating middleware
2. Communication between distributed objects
3. Remote procedure call
4. Java RMI
5. CORBA RMI
6. **Middleware Services**
    1. Trading service
    2. Event Service
    3. Notification Service
    4. Activation Service
    5. Dynamic Invocation Service

Overview Of Middleware Services

Naming Service

Event and Notification Service

Messaging Service

Persistence Service

Transaction Service

Activation Service

Loadbalancing Service

Dynamic Invocation Service

Security Service

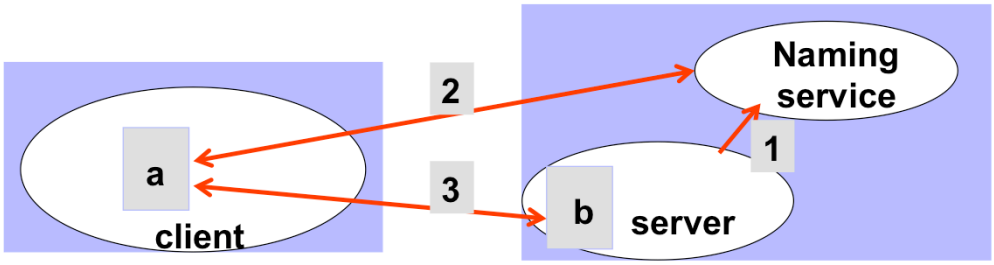
Session Tracking

For developing applications it is important that middleware services are available to the software developer. For instance, (i) a naming service helps to register and lookup remote object references by a textual name instead of dealing directly with the remote object references, (ii) an event service helps clients to register for certain events instead of having to program callbacks to all interested clients, (iii) a session tracking service can help tracking user sessions without having to manually program it, (iv) transaction service for implementing transactions, without the need for manually programming the two-phase commit protocol and (v) persistence service for automatically storing and retrieving the state of objects, without having to program the manual SQL statements. 10 middleware services exist and are listed above.

The following middleware services will be detailed here:

- Naming Service
- Event and Notification Service
- Activation Service
- Dynamic Invocation Service

CORBA Naming Service



Registration of Object References

Names are structured in a hierarchy

- cfr directories in file system
- files and directories can be assigned a “Kind” id

In the case of CORBA, the naming service has an IDL interface which provides following methods:

bind: for servers to register the remote object references of CORBA objects by name (e.g. bind (path, Object)), an exception is generated when the name is already bound

rebind: same as bind, except that when the name is already bound, the remote object reference is overwritten

resolve: for clients to look up the remote object references by name (e.g. Object = resolve(path))

These methods belong to an interface called NamingContext. The CORBA names are structured in a hierarchy (cfr directories in a file system). Files and Directories can be assigned a kind id. A path is defined as an array of NameComponents (a struct containing a name). The path starts from an initial context provided by CORBA.

CORBA server

Java class with *Main* method

Creates and initializes the ORB

Creates an instance of Servant class

Registers it to the ORB (through the *connect* method)

Gets a reference for the Naming Service

Registers the server to the Naming Service

Waits for incoming client requests

When using the CORBA naming service, the following extra steps are required in the implementation, detailed in the previous chapter:

CORBA server implementation: i.e. the class with the main method, after an instance of the servant class is created, a reference to the Naming Service has to be obtained, and the server has to be registered to the naming service

Client implementation: a reference to the Naming Service has to be obtained and a resolve has to be invoked on this reference, from the returned result the remote object reference can be created.

Running server and client with NS

Start orbd :

- UNIX command shell : $ orbd -ORBInitialPort 1050 -ORBInitialHost *nameserverhost* &
- MS-DOS system prompt : start orbd -ORBInitialPort 1050 -ORBInitialHost *nameserverhost*

Start the Echo server :

- UNIX command shell :
  $ java echoServer -ORBInitialPort 1050 -ORBInitialHost *nameserverhost* &
- MS-DOS system prompt :
  start java echoServer -ORBInitialPort 1050 -ORBInitialHost *nameserverhost*
- Result: echoServer ready and waiting ...

1050 = port on which Naming Service listens

Run the client application :

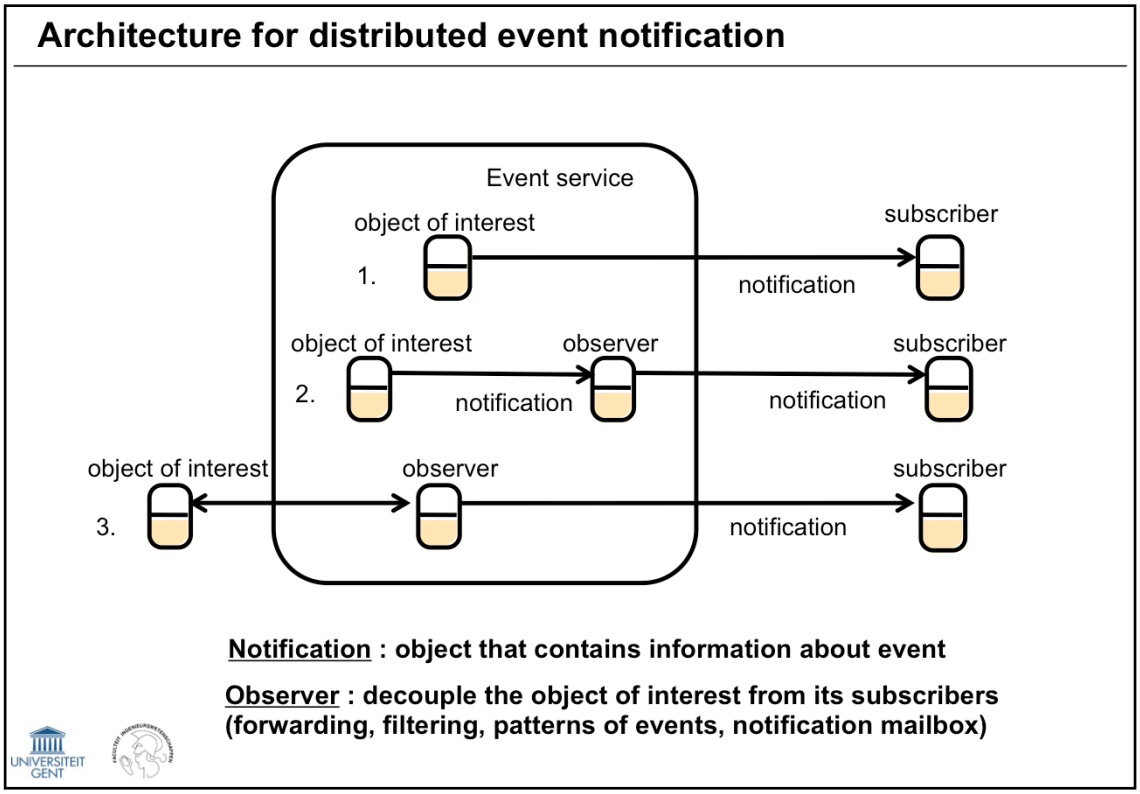
- $ java echoClient -ORBInitialPort 1050 -ORBInitialHost localhost
- Result: I said, "Hello!". The Object said, "Hello!"

When running the server and client: the name and port of the naming server has to provided either as command line arguments or in configuration file.

CORBA Trading Service

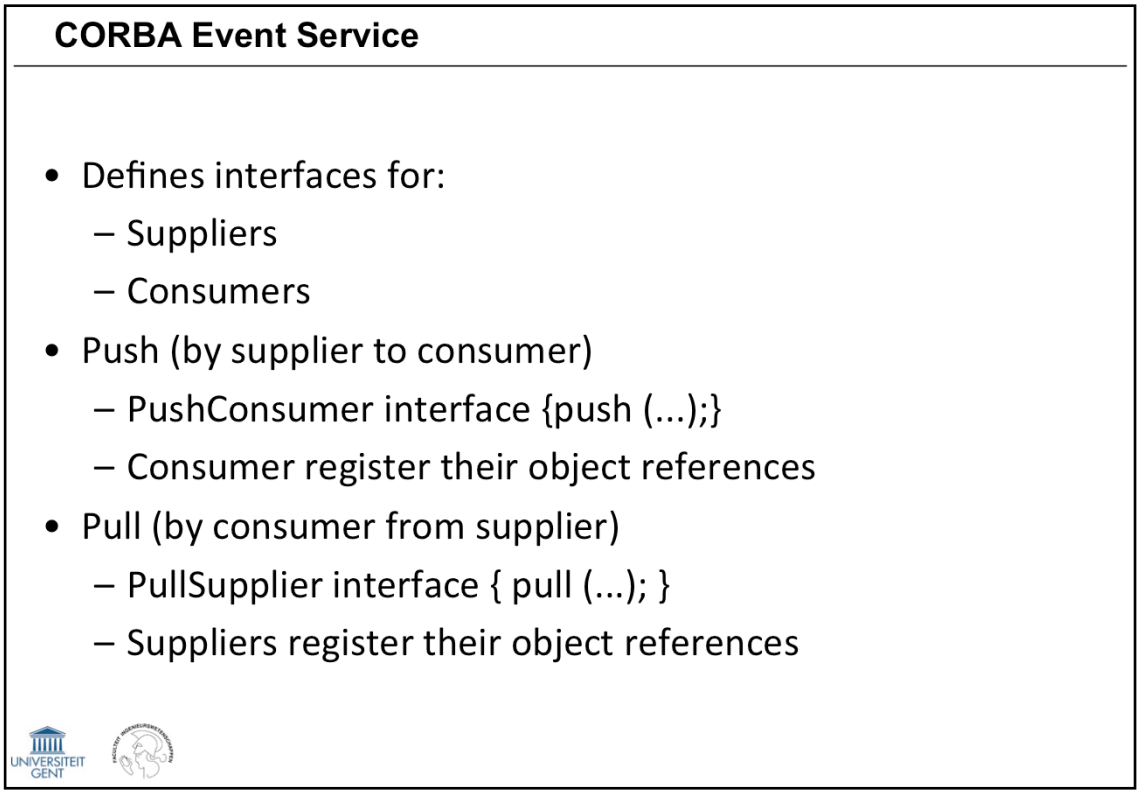
- Trading Service:
  - Comparable to Naming Service
  - Allows objects to be located by attribute : directory service
  - Database: service types and attributes -> remote object references
  - Clients specify :
    - Constraints on the values of attributes
    - Preferences for the order in which to receive matching offers

The Trading Service is comparable to the Naming Service. It is a directory service, which allows objects to be located by one or multiple attributes. Clients can specify constraints on the values of the attributes and preferences for the order in which to receive matching offers. Multiple trading services can be federated, i.e. can refer to each other and form virtually one trading service.



The above figure shows 3 options for distributed event notification. The third option is the most flexible one and is referred to as the observer pattern: the observer object decouples the object of interest from its subscribers. The observer object can take care of following tasks:

- Forwarding of the events to the subscribers,
- Filtering of the events based on the specified criteria,
- Detection of patterns in the events,
- Notification mailbox functionality, i.e store the events until they can be delivered or retrieved



CORBA provides an Event Service, based on the publish-subscribe principle. A CORBA Event Service defines IDL interfaces for the suppliers, i.e the objects of interest (cfr Figure 3.1) and the consumers, i.e. the subscribers. Following methods are available:

Push : invoked by the suppliers on the PushConsumer interface, consumers register their object references with suppliers

Pull: invoked by the consumer on the PullSupplier interface, in this model suppliers register their object references with consumers

Corba Event Service (2)

- Event Channels
  - Allow multiple suppliers to communicate with multiple consumers
  - Buffer between suppliers and consumers
- *EventChannel* interface
  - implemented by objects in event server
- Chains of event channels

Event channels are used for complying with the observer pattern. They allow multiple suppliers to communicate with multiple consumers and are a buffer between the suppliers and consumers. The push-pull model to the event channels is followed: suppliers push events to the channel and consumers either pull events from it or get events pushed from it. Chains of event channels can also be constructed.

Notification Service

- Extends Event Service with filters
  - Notifications have datatype (<-> any)
  - Event Consumers may use filters
    - Specify events they are interested in
    - Proxies forward notifications to consumers according to constraints specified in filters
  - Event Suppliers can discover the events the consumer are interested in
  - Event Consumers can discover a set of event types (subscribe to new event)
  - Configure properties of Event Channel
    - Reliability, priority of events, required ordering, policy for discarding stored events
  - Event type repository

UNIVERSITEIT GENT

The Notification Service extends the Event Service. Besides the event service functionality, it allows to use filters. Notifications have a datatype (in contrast to the any datatype of the Event Service). Event consumers can use the filters to specify the events they are interested in. Proxies forward the notifications to consumers according to constraints specified in filters. Event suppliers can discover the events the consumer are interested in and event consumers can discover a set of event types (subscribe to new events).

The Notification Service allows to configure the properties of the event channel, in terms of:

- reliability

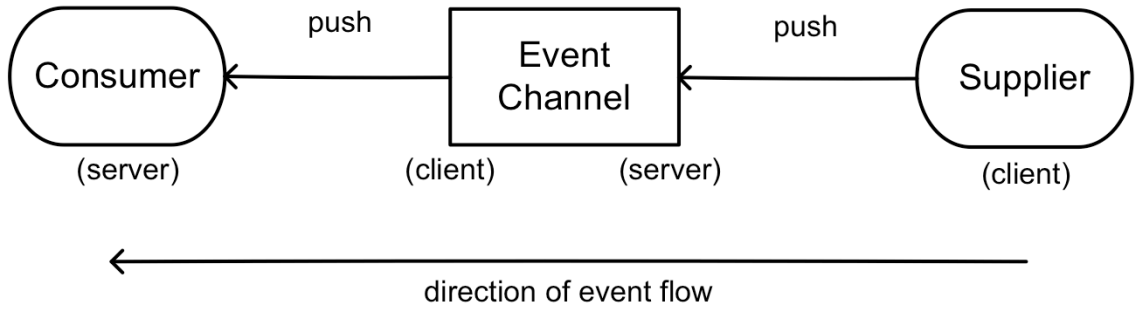
- priority of events

- required ordering

- policy for discarding stored events

A structured event type consists of an event header and an event body. The event header specifies the domain type, the event type, the event name, and the requirements. The event body consists of <name, value> pairs.

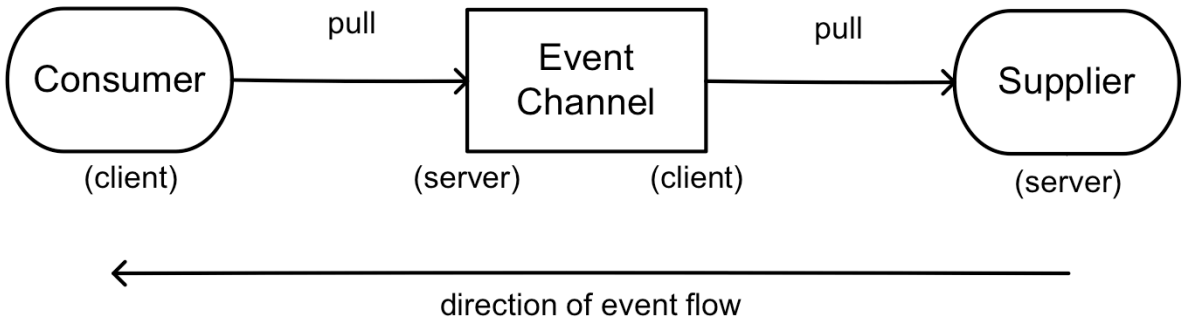
Corba Event service (1)



direction of event flow

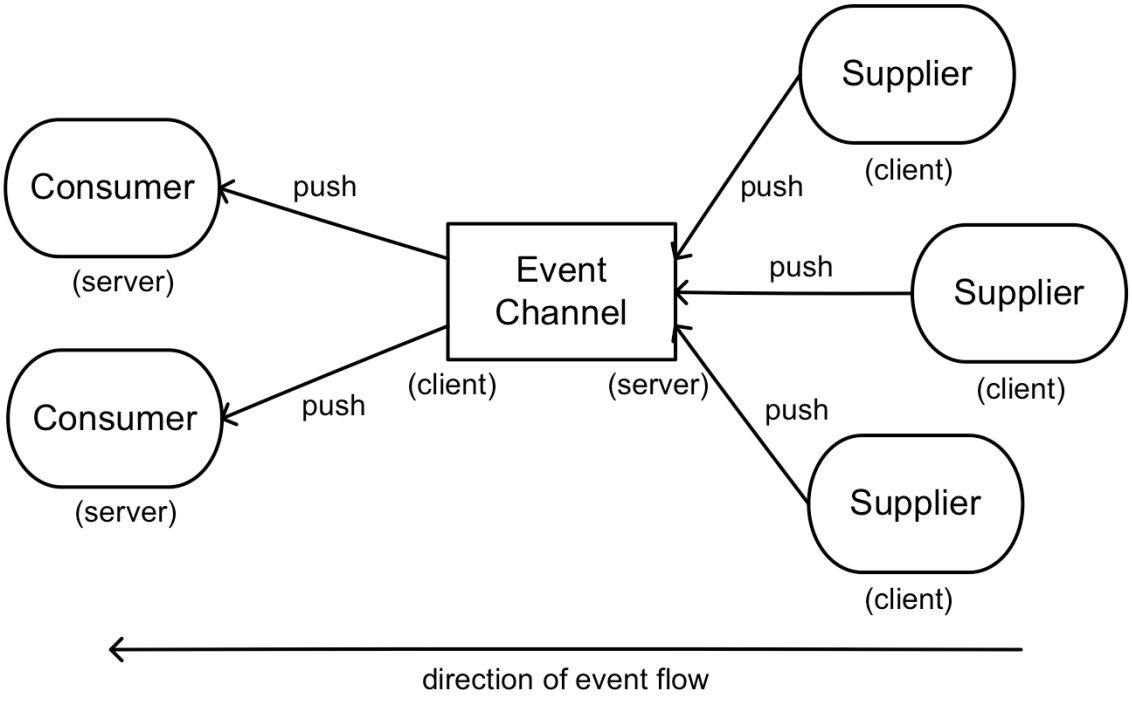
Push event delivery model

Corba Event service (2)

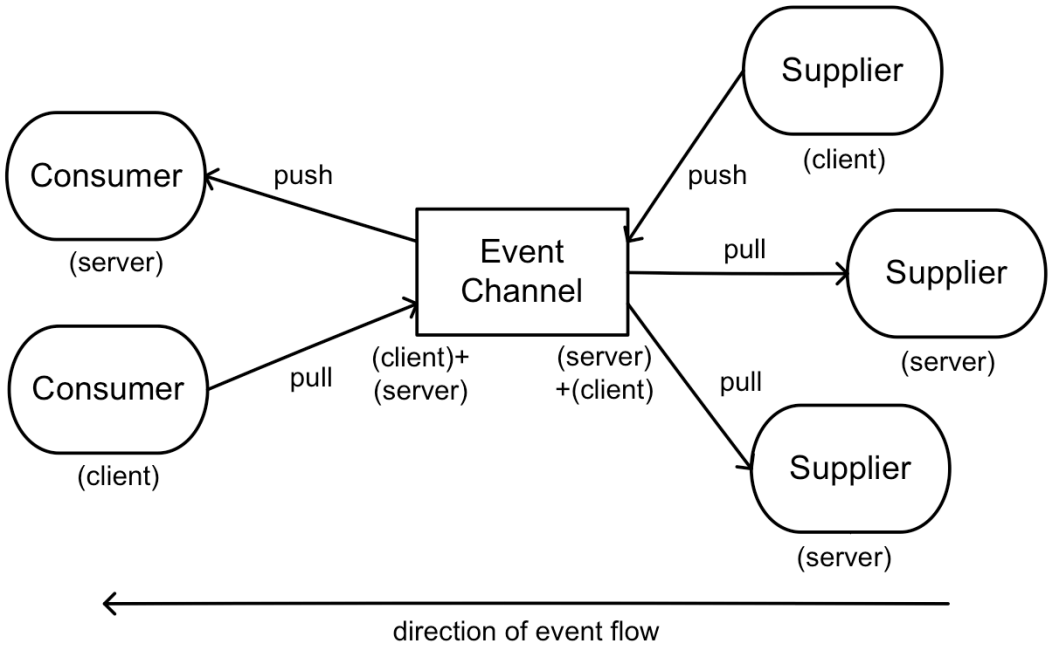


Pull event delivery model

Corba Event service (3)

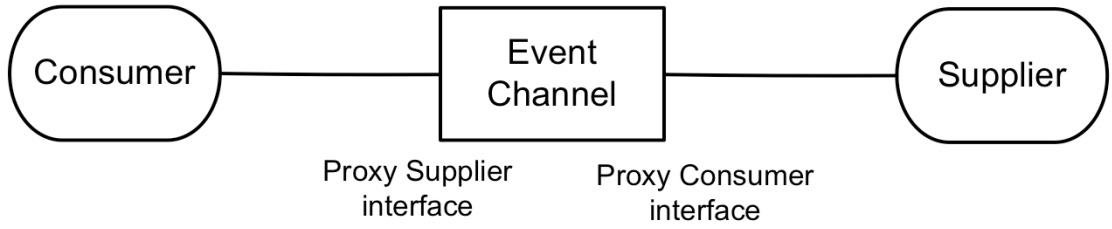


Corba Event service (4)



Hybrid Push/Pull event delivery model

Corba Event service (5)

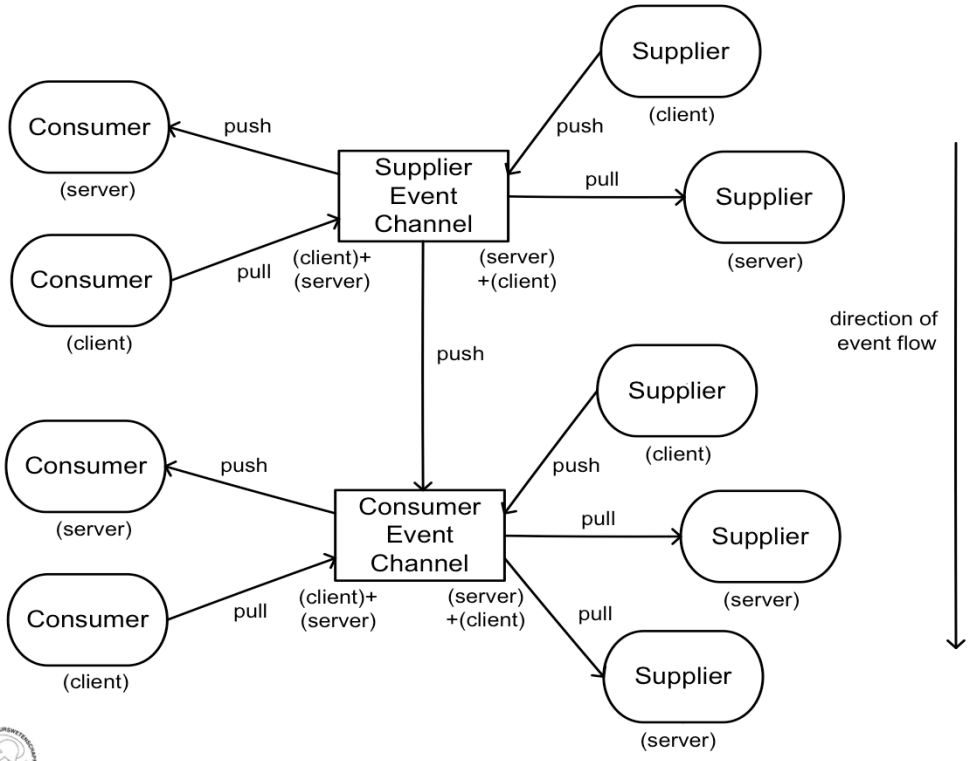


Proxy Supplier interface

Proxy Consumer interface

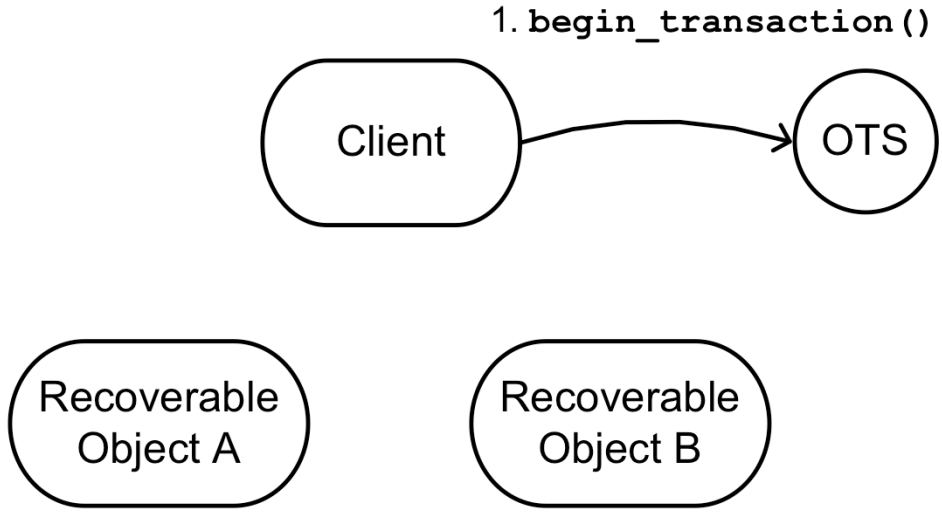
Event Channel Interfaces

Corba Event service (6)



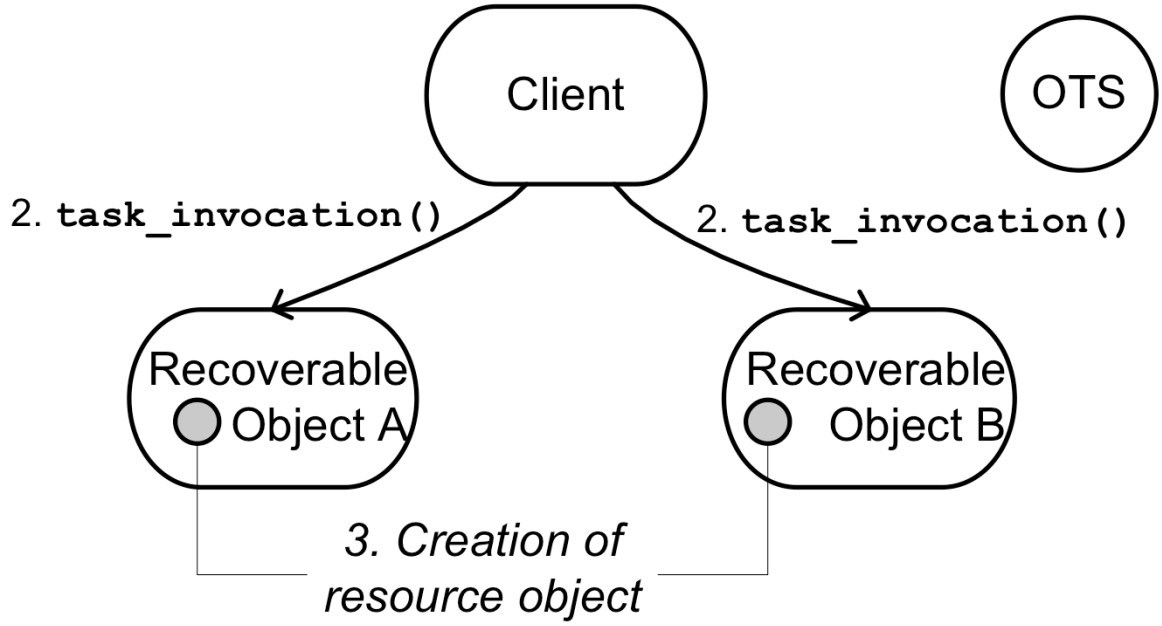
Chaining of event channels (federation)

Corba Transaction service (1)



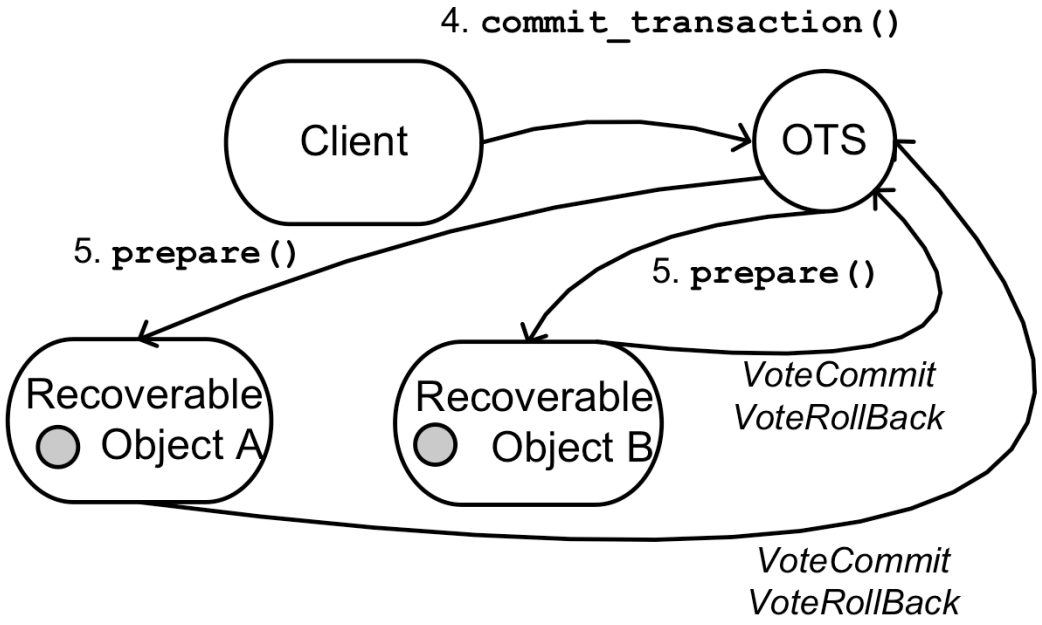
Transaction scenario – part I

Corba Transaction service (2)



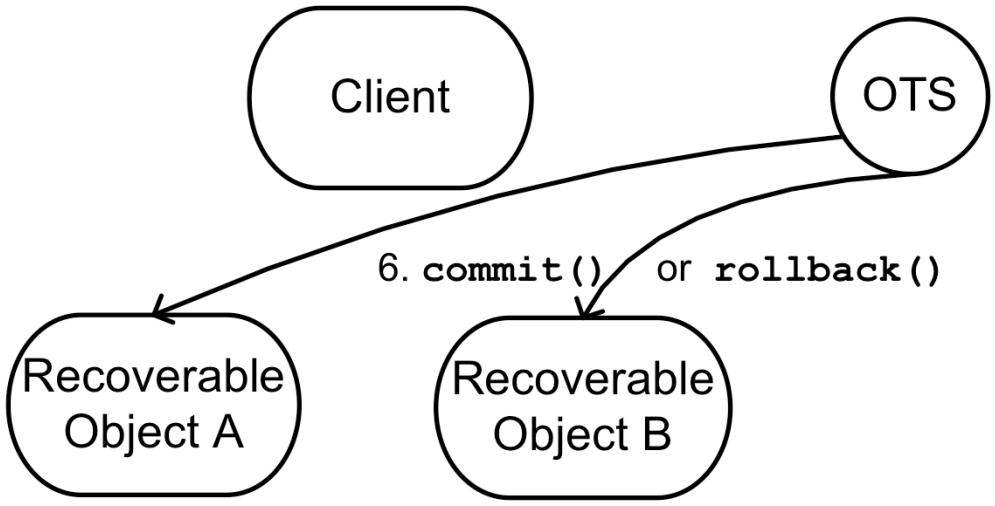
Transaction scenario – part II

Corba Transaction service (3)



Transaction scenario – part III

Corba Transaction service (4)



Transaction scenario – part IV

Persistence service

Automated storing and retrieving data
to/from databases

-> next chapter

Activation Service

Activation service

Activation

= on demand execution of services to reduce server processes/threads

activate objects when requests arrive

passivate objects if in consistent state

Activator

- keeps table of passive (activatable) objects
- start server processes on demand
- create active server objects from passive state (e.g. serialized state)
- keeps table of activated objects and their location

In practice ...

CORBA : activator service

Java RMI : 1 activator on each server computer

Implementation Repository

- Responsible for
  - activating registered servers on demand
  - locating servers that are currently running
- Object Adapter name is used for registration and activation
- Implementation Repository entry:

Object Adapter Name
Pathname of Object Implementation
Host name and port number of server

- Extra information : access control information

The CORBA Object Adapter name is used for registration and activation. An implementation repository is a database, containing for each registered Object Adapter Name: (i) the pathname of the object implementation binary and (ii) the host name and port number of the computer, which can run the object. Access control rights for each object can be stored as well.

Implementation Repository

- On ImR machine: start ImR

  $ imr –p 2446 –f /etc/imr.db –i /etc/imr/ior_file

- On participating hosts: startup daemon

  $ imr_ssd

- Register a program:
  $ imr_mg add “echoServer” –c “java echoServer”
  $ imr_mg add “echoServer” –c “xterm –e java echoServer”

- Automatic creation of new entries when activating objects:
  $ imr –p 2446 –f /etc/imr.db –i /etc/imr/ior_file –a

When using the Implementation Repository service, the repository binary is started on one selected computer. On each participating host of the distributed system, a daemon needs to be started. When registering a binary on a participating host, this information is passed to the Implementation Repository and the object startup information is known in the entire distributed system. Some CORBA products also allow the automatic creation of new implementation repository entries when activating objects (i.e. without the need for manual registration of the binaries).

Loadbalancing service

Automatic distribution of load between different servers

Often a broker is used:
same interface as servers
forwards load to “most appropriate” server

(either prediction of load or retrieving load information)

Often in combination with Naming service

Dynamic Invocation service

Allowing the invocation of objects, whose interface is not known at compile time, but discovered at run time

Java: Reflection API

CORBA: Interface repository

CORBA Interface Repository

- Provide information about registered IDL interfaces
- Can supply
  - the names of the methods
  - the corresponding names and types of arguments and exceptions
- Reflection facility
- Useful when client has no proxy for object

- Not required when static invocation with client stubs and IDL skeletons

The *CORBA Interface Repository* provides information about registered IDL interfaces and can supply:

- the names of the methods

- the corresponding names and types of arguments and exceptions

It offers the reflection facility, which is available in Java. This is useful when a client has no proxy for a remote object. It is not required for static invocation with client stubs and IDL skeletons.

At compile time, the Dynamic Invocation Interface is used to obtain from the interface repository the necessary information, construct an invocation with the required arguments and sent it to the server. The Dynamic Skeleton Interface at server side allows to accept invocations on an interface for which there is no skeleton. It inspects the request, discovers the target object and invokes the target.

Other services

Security
Session tracking

-> next chapter