Remote distributed objects

- Data and operations **encapsulated** in an object
- Operations implemented as **methods** grouped into **interfaces**
- Object offers only its **interface** to clients
- **Object server** is responsible for a collection of objects
- **Client stub (proxy)** implements interface
- **Server skeleton** handles (un)marshaling and object invocation

![Diagram of distributed object architecture](image)
Remote distributed objects

Types of objects I

- **Compile-time objects**: Language-level objects, from which proxy and skeletons are automatically generated.
- **Runtime objects**: Can be implemented in any language, but require use of an **object adapter** that makes the implementation *appear* as an object.

Types of objects II

- **Transient objects**: live only by virtue of a server: if the server exits, so will the object.
- **Persistent objects**: live independently from a server: if a server exits, the object’s state and code remain (passively) on disk.
Example: Enterprise Java Beans (EJB)

What is it
Java object hosted by special server that allows for different means of calling the object by remote clients.
Types of EJBs

Four different types

- **Stateless session bean**: Transient object, called once, does its work and is done. **Example**: execute an SQL query and return result to caller.

- **Stateful session bean**: Transient object, but maintains client-related state until the end of a session. **Example**: shopping cart.

- **Entity bean**: Persistent, stateful object, can be invoked during different sessions. **Example**: object maintaining client info on last number of sessions.

- **Message-driven bean**: Reactive objects, often triggered by message types. Used to implement publish/subscribe forms of communication.
**Observation**

Most distributed objects are not distributed at all: state is kept at a single node. **Alternative:** Globe objects, which are physically distributed across multiple machines.
Globe distributed objects

Note
To make DSOs generic, we need to separate function from distribution support

Same interface as implemented by semantics subobject
Globe distributed objects

Note
Replication subobject essentially decides how and when the local semantics subobject will be invoked.
Processes: Object servers

Servant
The actual implementation of an object, sometimes containing only method implementations:
- Collection of C or COBOL functions, that act on structs, records, database tables, etc.
- Java or C++ classes

Skeleton
Server-side stub for handling network I/O:
- Unmarshalls incoming requests, and calls the appropriate servant code
- Marshalls results and sends reply message
- Generated from interface specifications
Processes: Object servers

Object adapter

The “manager” of a set of objects:

- Inspects (as first) incoming requests
- Ensures referenced object is activated (requires identification of servant)
- Passes request to appropriate skeleton, following specific activation policy
- Responsible for generating object references
Observation
Object servers determine how their objects are constructed
Example: Ice

```cpp
main(int argc, char* argv[]) {
    Ice::Communicator ic;
    Ice::ObjectAdapter adapter;
    Ice::Object object;
    ic = Ice::initialize(argc, argv);

    adapter = ic->createObjectAdapterWithEndPoints
                 ("MyAdapter","tcp -p 10000");
    object = new MyObject;

    adapter->add(object, objectID);
    adapter->activate();

    ic->waitForShutdown();
}
```

Note

Activation policies can be changed by modifying the properties attribute of an adapter. Ice aims at simplicity, and achieves this partly by putting policies into the middleware.
Client-to-object binding

Object reference
Having an object reference allows a client to bind to an object:
- Reference denotes server, object, and communication protocol
- Client loads associated stub code
- Stub is instantiated and initialized for specific object

Two ways of binding
- Implicit: Invoke methods directly on the referenced object
- Explicit: Client must first explicitly bind to object before invoking it
Client-to-object binding: implicit/explicit

Distr_object* obj_ref;
obj_ref = ...;
obj_ref->do_something();

Distr_object* obj_ref;
Local_object* obj_ptr;
obj_ref = ...;
obj_ptr = bind(obj_ref);
obj_ptr->do_something();

Some remarks
- Reference may contain a URL pointing to an implementation file
- (Server,object) pair is enough to locate target object
- We need only a standard protocol for loading and instantiating code

Observation
Remote-object references allow us to pass references as parameters. This was difficult with ordinary RPCs.
The Java Remote Method Invocation (RMI) system allows an object running in one Java Virtual Machine (VM) to invoke methods on an object running in another Java VM.

Java RMI provides applications with transparent and lightweight access to remote objects. RMI defines a high-level protocol and API.

Programming distributed applications in Java RMI is simple.

- It is a single-language system.
- The programmer of a remote object must consider its behavior in a concurrent environment.
Remote Method Invocation (RMI)

RMI Architecture

- **Stub**: lives on the client; pretends to be the remote object
- **Skeleton**: lives on the server; talks to the true remote object
- **Reference Layer**: determines if referenced object is local or remote
- **Transport Layer**: packages remote invocations; dispatches messages between stub and skeleton
Java RMI Applications

A Java RMI application needs to do the following:

- **Locate remote objects:** Applications can use one of two mechanisms to obtain references to remote objects:
  - An application can register its remote objects with RMI’s simple naming facility, the `rmiregistry`, or
  - the application can pass and return remote object references as part of its normal operation.

- **Communicate with remote objects:** Details of communication between remote objects are handled by RMI; to the programmer, remote communication looks like a standard Java method invocation.

- **Load class bytecodes for objects that are passed around:** Because RMI allows a caller to pass objects to remote objects, RMI provides the necessary mechanisms for loading an object’s code, as well as for transmitting its data.
Remote Method Invocation (RMI)

Java RMI Applications

RMI is supported by two Java packages, `java.rmi` and `java.rmi.server`. An application that uses RMI has 3 components:

- an interface that declares headers for remote methods;
- a server class that implements the interface; and
- one or more clients that call the remote methods.
Remote Method Invocation (RMI)

Basics

(Assume client stub and server skeleton are in place)

- Client invokes method at stub
- Stub marshals request and sends it to server
- Server ensures referenced object is active:
  - Create separate process to hold object
  - Load the object into server process
  - ...

- Request is unmarshaled by object’s skeleton, and referenced method is invoked
- If request contained an object reference, invocation is applied recursively (i.e., server acts as client)
- Result is marshaled and passed back to client
- Client stub unmarshals reply and passes result to client application
RMI: Parameter passing

Object reference

Much easier than in the case of RPC:
- Server can simply bind to referenced object, and invoke methods
- Unbind when referenced object is no longer needed

Object-by-value

A client may also pass a complete object as parameter value:
- An object has to be marshaled:
  - Marshall its state
  - Marshall its methods, or give a reference to where an implementation can be found
- Server unmarshals object. Note that we have now created a copy of the original object.
- Object-by-value passing tends to introduce nasty problems
RMI: Parameter passing

Note
Systemwide object reference generally contains server address, port to which adapter listens, and local object ID. Extra: Information on protocol between client and server (TCP, UDP, SOAP, etc.)
RMI: Parameter passing

Question
What’s an alternative implementation for a remote-object reference?
RMI Registry

- The RMI registry is a simple server-side bootstrap naming facility that allows remote clients to get a reference to a remote object.
- Servers name and register their objects to be accessed remotely with the RMI Registry.
- Clients use the name to find server objects and obtain a remote reference to those objects from the RMI Registry.
- A registry service is a background program that maintains a list of registered server names on a host. It is invoked by: `rmiregistry port &`
- The registry service is provided by a Naming object that provides two key methods:
  - `bind` to register a name and server
  - `lookup` to retrieve the server bound to a name
RMI: Programming Applications

RMI Inheritance Hierarchy

```
Java.rmi.RemoteObject
    
Java.rmi.UnicastRemoteObject
    
extends

MyServer

Java.rmi.Remote
    
MyServerInterface
    
implements
```
Programming RMI Applications

The steps involved in programming an RMI application are:

- Write a java interface that extends `Remote`. Each method in the interface needs to declare that they throw `RemoteException`.
- Write a server class that extends `UnicastRemoteObject` and implements the methods in the interface.
- Write code that instantiates the server and registers its name with a registry service. This code can be in that main method of the server class or within another class.
- Write a client class that interacts with the server. The client then calls `Naming.lookup` to get a reference to a server object from the registry service. Once it has this reference the client can then invoke the remote server’s methods.
An Example RMI Application

The files needed for creating a Java RMI application are:

- A remote interface defines the remote interface provided by the service. Usually, it is a few single line statements specifying the service functions (DatabaseInterface.java). (An interface is the skeleton for a public class.)
- A remote object implements the remote service. It contains a constructor and required functions. (Database.java)
- The server offers the remote service, installs a security manager and contacts rmiregistry with an instance of the service under the name of the remote object. (DatabaseServer.java)
- A client that invokes the remote method. (DatabaseClient.java)
import java.rmi.*;
import java.rmi.server.*;

public interface DatabaseInterface extends Remote {
    public int read() throws RemoteException;
    public void write(int value) throws RemoteException;
}
import java.rmi.*;
import java.rmi.server.*;

public class Database extends UnicastRemoteObject
    implements DatabaseInterface {

    private int data = 0; // the database

    public Database(int value) throws RemoteException {
        data = value;
    }

    public int read () throws RemoteException {
        return data;
    }

    public void write (int value) throws RemoteException {
        data = value;
        System.out.println ("New value is: " + data);
    }
}

RMI: Programming Applications

Database.java
DatabaseServer.java

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

public class DatabaseServer {

    public static void main (Strings[] args) {
        try {
            // create Database Server Object
            Database db = new Database(0);

            // register name and start serving
            String name = "rmi://fuji:9999/DB";
            Naming.bind(name,db);
            System.out.println (name + " is running");
        } catch (Exception ex) {
            System.err.println (ex);
        }
    }
}
```
DatabaseClient.java

import java.rmi.*;

public class DatabaseClient {

    public static void main (String[] args) {
        try {
            // set RMI Security Manager
            System.setSecurityManager(new RMISecurityManager() {
                public void checkConnect(String host, int port) {} 
                public void checkConnect(String host, int port, Object Context) {} 
            });
            // get database object
            String name = "rmi://fuji:9999/DB";
            DatabaseInterface db = (DatabaseInterface)Naming.lookup(name);
            int value, rounds = Integer.parseInt(args[0]);
            for (int i = 0; i < rounds; i++) {
                value = db.read();
                System.out.println("read: " + value);
                db.write(value+1);
            }
        } catch (Exception ex) {
            System.err.println (ex);
        }
    }
}
Building the Application

- Compile the code:
  javac Database.java DatabaseClient.java DatabaseInterface.java DatabaseServer.java
- Generate stub and skeleton class files:
  rmic Database (note: not needed for Java 5 or later)
- Start the RMI registry:
  rmiregistry 9999 &
- Start the server:
  java -Djava.security.policy=java.policy DatabaseServer
- Run the client:
  java -Djava.security.policy=java.policy DatabaseClient 10
OMG

The Object Management Group (OMG) was formed in 1989 and now has over 800 members. Its aims were:

- to make better use of distributed systems
- to use object-oriented programming
- to allow objects in different programming languages to communicate with one another
- The object request broker (ORB) enables clients to invoke methods in a remote object
- CORBA (Common Object Request Broker Architecture) is a specification of an architecture supporting this (rather than an implementation such as RMI).
CORBA: Overview

- **ORB**: the runtime system responsible for communication between objects and their clients
- **CORBA facilities**: collections of classes and objects that provide general-purpose capabilities that are useful in many applications.
  - **Horizontal facilities**: user interfaces, information management, system management, task management
  - **Vertical facilities**: e.g., electronic commerce, banking, manufacturing, healthcare, telecommunications
CORBA: Components

Main CORBA Components

- **An Interface Definition Language (IDL)** and its mapping onto the implementation language (C, C++, Java, Smalltalk, Ada, COBOL)
  - specifies the syntax of objects and services
  - cannot be used to describe semantics
  - object can be implemented by a language without classes

- **An Interface Repository (IR)** representing the interfaces of all objects available in the distributed system.

- A fully dynamic calling mechanism allowing:
  - run-time discovery of objects
  - discovery of available interfaces from an IR
  - construction of message requests
  - sending of message requests

- **Object Adapters**: an abstraction mechanism for removing implementation details from the message substrate.
**ORB offers:**
- Basic communication
- Object references
- Initial finding of services

**Interfaces:**
- Static (described in IDL)
- Run-time creation (Dynamic Invocation Interface)
### CORBA: Communication

<table>
<thead>
<tr>
<th>Request type</th>
<th>Failure semantics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronous</td>
<td>At-most-once</td>
<td>Caller blocks until a response is returned or an exception is raised</td>
</tr>
<tr>
<td>One-way</td>
<td>Best effort delivery</td>
<td>Caller continues immediately without waiting for any response from the server</td>
</tr>
<tr>
<td>Deferred Synchronous</td>
<td>At-most-once</td>
<td>Caller continues immediately and can later block until response is delivered</td>
</tr>
</tbody>
</table>
CORBA: Event and Notification Services

- **Push-style model:**
  - Push event to consumers

- **Pull-style model:**
  - Ask suppliers for new event
CORBA: Persistent Communication

**CORBA messaging service:**

- Object-oriented approach to communication
- All communication takes place by invoking an object
- May not be possible to get an immediate reply
- **Callback model:** A client features two interfaces: method and callback
- **Polling model:** A client is offered a collection of operations to poll its ORB for incoming results
- Messages sent are stored by the underlying system in case the client or server is not yet running
CORBA: Messaging

1. Call by the application
   - Client application
   - Client proxy
   - Client RTS

2. Request to server

3. Response from server
   - Callback interface
   - Polling interface

4. Call by the RTS
   - RTS

4. Call by the application
A Simple IDL Interface

For Callback/Polling, use special IDL compiler to transform original (synchronous) method invocation into an asynchronous one e.g.:

```c
int add(in int i, in int j, out int k)
```

- **Two methods produced for Callback**
  - `void sendcb_add(in int i, in int j)` // called by client’s app
  - `void replycb_add(in ret_val, in int k)` // called by client’s RTS/orb
  - i.e. replace both output args from `add` with 2 inputs to the callback.

- **Two methods produced for Polling**
  - `void sendpoll_add(in int i, in int j)` // called by client’s app
  - `void replypoll_add(out ret_val, out int k)` // called by client’s app
**Observation**

In order to invoke remote objects, we need a means to uniquely refer to them.
CORBA: Object references

Observation
It is not important how object references are implemented per object-based system, as long as there is a standard to exchange them between systems.

Solution
Object references passed from one RTS to another are transformed by the bridge through which they pass (different transformation schemes can be implemented)
Observation

Passing an object reference \textit{refA} from RTS A to RTS B circumventing the A-to-B bridge may be useless if RTS B doesn’t understand \textit{refA}
Interoperability is achieved by the introduction of a standard inter-ORB protocol (General Inter-ORB Protocol GIOP) which:

- Defines an external data representation, called the Common Data Representation (CDR)
- Specifies formats for the messages in a request-reply protocol including messages for enquiring about the location of an object, for canceling requests and for reporting errors.

The Internet Inter-ORB protocol (IIOP) defines a standard form for remote object references.

- IIOP is GIOP implemented in TCP/IP
CORBA: Clients and Servers

Clients:
- The IDL specification is compiled into a proxy
- Proxy connects a client application to the underlying ORB

Servers:
- The Portable Object Adapter (POA) makes code appear as CORBA objects to clients
- Creates remote object references for CORBA objects
- Activates objects
- Implements a specific activation policy (e.g., thread-per-request, thread-per-connection, thread-per-object policy,...)
- Methods are implemented by means of servants.
Mapping of CORBA object identifiers to servants.

a) The POA supports multiple servants.
b) The POA supports a single servant.
Programming CORBA Applications

The steps involved in programming a CORBA application are:

- Write the IDL interface file
- Compile the IDL file to generate stubs and skeletons
- Write and compile the remote object implementation
- Write and compile the remote server
- Write and compile the client
Example: a CORBA object with operation `sayHello()` returning a string.

**Defining the Interface: Hello.IDL**

```idl
module HelloApp
{
    interface Hello
    {
        string sayHello();
    }
};
```

To compile the IDL file to generate the stubs and skeletons:

```
idlj -fall Hello.idl
```
Files generated by idlj

Below is a UML diagram showing the files autogenerated by IDL.
This generates the following files:

- `_HelloImplBase.java` or `HelloPOA.java`: this abstract class is the server skeleton, providing basic CORBA functionality for the server. It implements the `Hello` interface. The server class `HelloServant` extends `_HelloImplBase`.
- `_HelloStub.java`: this class is the client stub, providing CORBA functionality for the client. It implements the `Hello` interface.
- `HelloHelper.java`: this final class provides auxiliary functionality, notably the `narrow` method required to cast CORBA object references to their proper types.
- `HelloHolder.java`: this final class holds a public instance member of type `Hello`. It provides operations for out and inout arguments, which CORBA has but which do not map easily to Java’s semantics.
HelloOperations.java: this interface contains the single method sayHello(). The IDL-to-Java mapping puts all of the operations defined on the IDL interface into this file, which is shared by both the stubs and skeletons.

Hello.java: this interface contains the Java version of our IDL interface. It contains the single method sayHello. The Hello interface extends org.omg.CORBA.Object, providing standard CORBA object functionality as well. Here is the generated Hello.java file:

```java
/* Hello.java as generated by idlj */
package HelloApp;
public interface Hello extends org.omg.CORBA.Object {
    String sayHello();
}
```
Implementing the Remote Object: **HelloServant.java**

```java
public class HelloServant extends _HelloImplBase {
    public String sayHello() {
        return "\nHello world!!\n";
    }
}
```
Implementing the Remote Server: `HelloServer.java`

```java
// The package containing our stubs.
import HelloApp.*;

// HelloServer will use the naming service.
import org.omg.CosNaming.*;

// The package containing special exceptions thrown by the name service.

// All CORBA applications need these classes.
import org.omg.CORBA.*;

public class HelloServer {
    public static void main(String args[]) {
        try {
            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Create the servant and register it with the ORB
            HelloServant helloRef = new HelloServant();
            orb.connect(helloRef);
        }
    }
}
```
// Get the root naming context
org.omg.CORBA.Object objRef =
    orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);

// Bind the object reference in the root naming context
NameComponent nc = new NameComponent("Hello", ";
NameComponent path[] = nc;
ncRef.rebind(path, helloRef);

// Wait for invocations from clients
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
    sync.wait();
}
} catch(Exception e) {
    System.err.println("ERROR: "+ e);
e.printStackTrace(System.out);
}
Implementing the Client: **HelloClient.java**

```java
import HelloApp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class HelloClient {
    public static void main(String args[]) {
        try {
            // Create and initialize the ORB
            ORB orb = ORB.init(args, null);

            // Get the root naming context
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
        }
    }
}
```
// Resolve the object reference in naming
NameComponent nc = new NameComponent("Hello", "");
NameComponent path[] = nc;
Hello helloRef = HelloHelper.narrow(ncRef.resolve(path));

// Call the Hello server object and print results
String hello = helloRef.sayHello();
System.out.println(hello);
} catch(Exception e) {
    System.out.println("ERROR : " + e);
e.printStackTrace(System.out);
}
Building the Application

- Compile the .java files, including the stubs and skeletons (which are in the directory HelloApp):
  
  \texttt{javac *.java HelloApp/\_*.java}

- Start the Java IDL name server:
  
  \texttt{tnameserv -ORBInitialPort 1050}

- Start the Hello server:
  
  \texttt{java HelloServer -ORBInitialPort 1050}

- Run the Hello application client:
  
  \texttt{java HelloClient -ORBInitialPort 1050}

- The client output:
  
  \texttt{Hello world!!}
RMI vs CORBA

- CORBA was designed for language independence whereas RMI was designed for a single language where objects run in a homogeneous environment.
- CORBA interfaces are defined in IDL, while RMI interfaces are defined in Java.
- CORBA objects are not garbage collected because they are language independent and they have to be consistent with languages that do not support garbage collection, on the other hand RMI objects are garbage collected automatically.