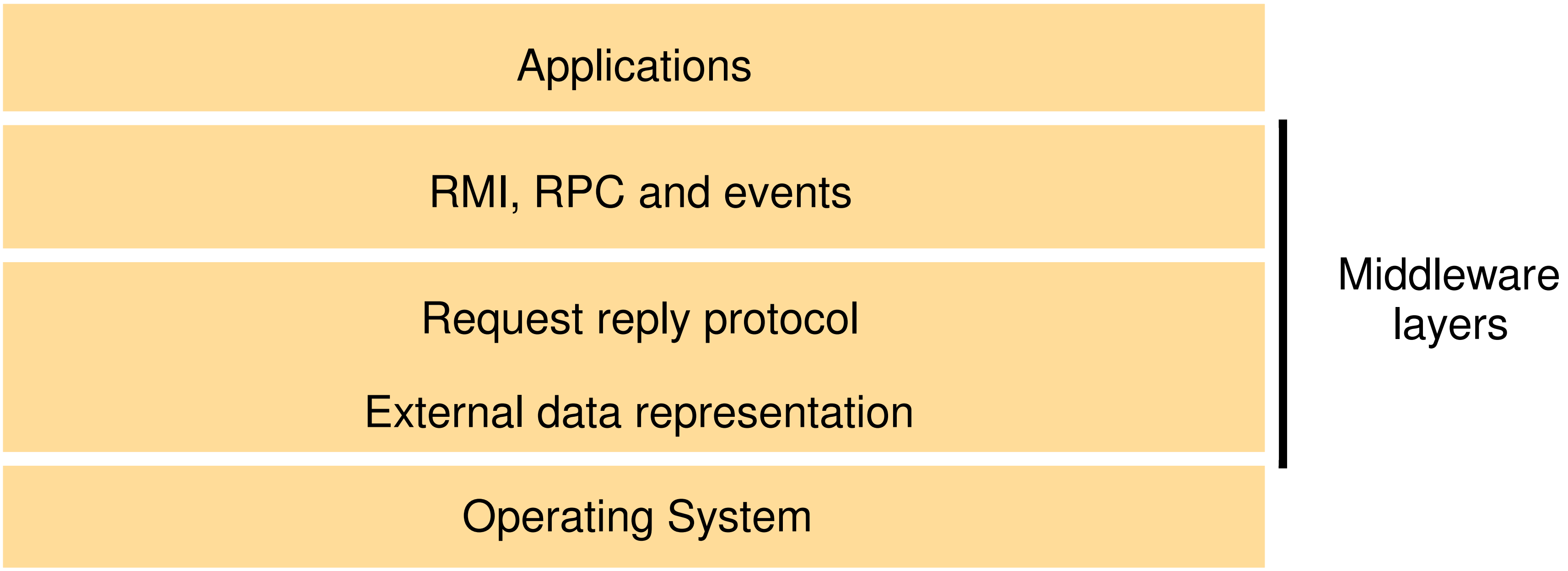
Chapter 5

Distributed Objects and Remote Invocation

Distributed Programming Model

- Middleware provides a programming model above the basic building blocks of processes and message passing.
- Applications are composed of cooperating programs in different processes, and often in different computers.
- Programming model is either
  - **Remote procedure call (RPC)** – Extension of conventional procedure call model
  - **Remote method invocation (RMI)** – Extension of object-based model
  - Distributed event-based model – Extension of event-based model (We will not cover it in this course.)

Middleware Layers (1)



Applications

RMI, RPC and events

Request reply protocol

External data representation

Operating System

Middleware layers

Middleware Layers (2)

- RPC and RMI provide location transparency.
    - The client cannot tell whether the invoked procedure/method is in the same or different process.
    - The client does not need to know the location of the server.
- The protocols that support middleware are independent of the underlying transport protocols.
    - RMI is based on request-reply protocol which can be implemented over UDP or TCP.
- Agreed standard for external data representation is used to hide difference in hardware architecture.
- Abstraction provided by middleware is independent of the underlying operating system.

clip: Powers of Ten

- Some middleware (e.g. CORBA) allows distributed applications to use more than one programming language.
    - This is achieved by using an **interface definition language (IDL).**

Interface

- The interface of a module specifies the procedures methods and variables that can be accessed from other modules.
  - Variables are not accessed directly but by getter and setter procedures that will be added automatically to the interface.
  - Pointers (memory locations) cannot be passed as arguments or returned as results of calls to remote modules. WHY ??
  - For RMI, objects can be passed as arguments or returned as results of methods. Moreover, remote object references may also be passed. WHY ??

IDL

- IDLs are designed to allow objects implemented in different languages to invoke one another.

```
struct Person {
            string name;
            string place;
            long year;
} ;
interface PersonList {
            readonly attribute string listname;
            void addPerson(in Person p) ;
            void getPerson(in string name, out Person p);
            long number();
};
```

CORBA has a struct

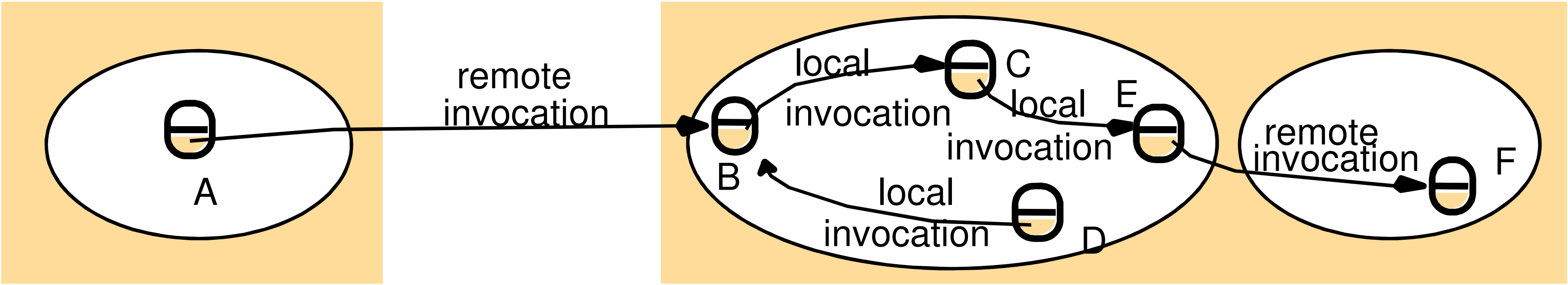
string getlistname();

remote interface

remote interface defines methods for RMI

parameters are *in*, *out* or *inout*

Distributed Object Model



- Each process contains objects, some of which can receive remote invocations (i.e. remote objects), others only local invocations.
- Objects need to obtain the remote object reference (e.g. from a name service or parameter passing) in order to invoke the remote methods.

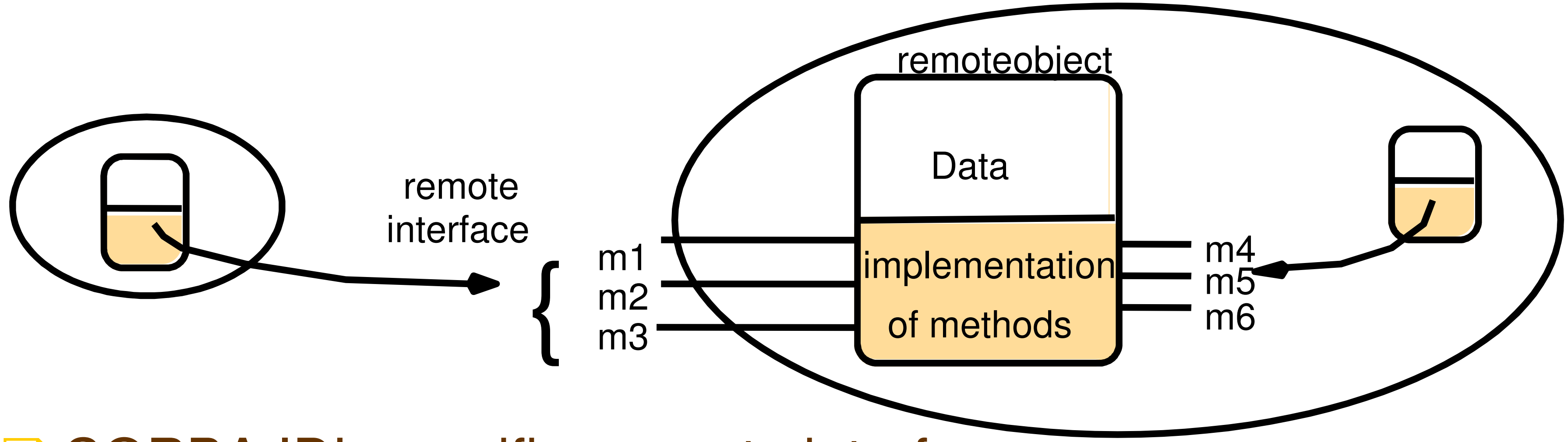
```
s = new PersonListImpl(…);   //remote object created on server side
bind(s, "MyPL") to name service //ROR is bound to service name
```

```
lookup name service for "MyPL"   //client side
s.getPerson(…);
```

- Remote interface specifies which methods can be invoked remotely.

Remote Interface

class PersonListImpl implements PersonList {…}

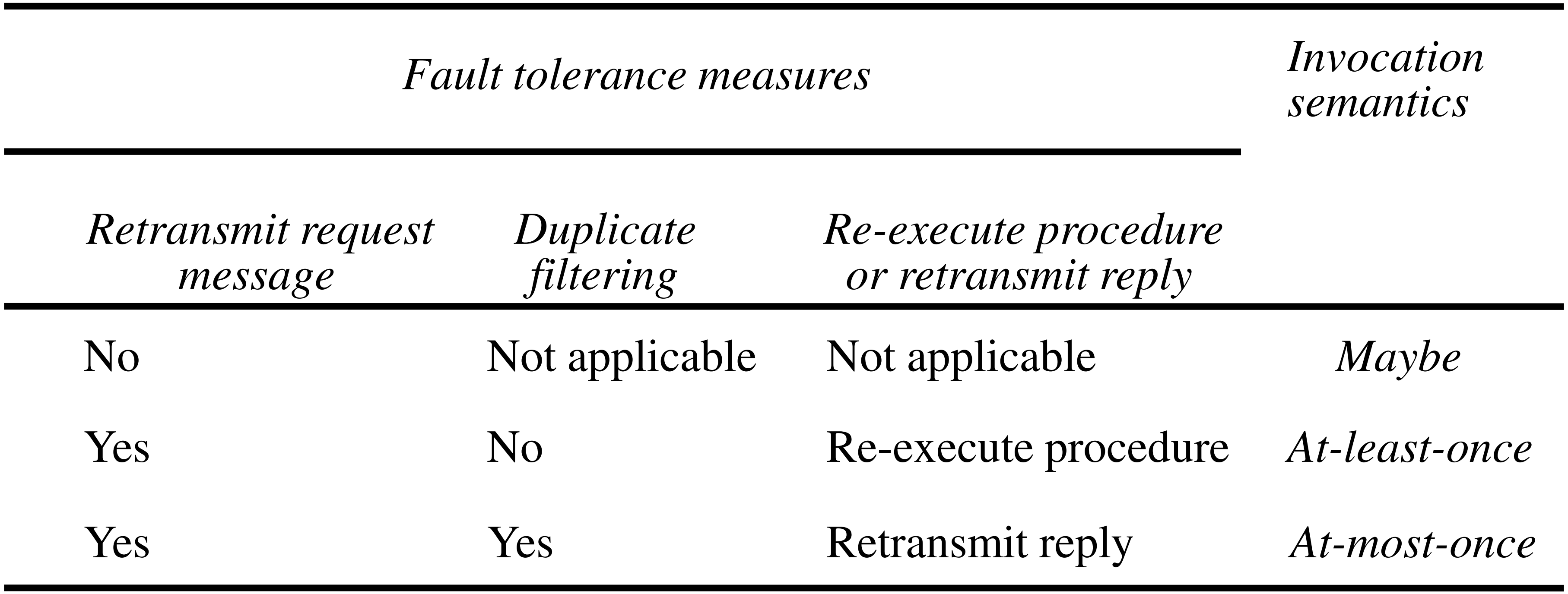


- CORBA IDL specifies remote interface.
  - Classes of remote objects and client programs may be implemented in any language for which an IDL compiler is available.
- Remote interface in Java RMI is as any Java interface but it extends an interface named Remote.
- Remote interface supports exceptions that are due to distribution (e.g. on timeouts) and that are raised during method execution (e.g. read beyond end of file).

*** Very Important ***

RMI Invocation Semantics (1)

- Local invocation has exactly-once semantics.
- RMI has several semantics due to fault-tolerance measures.



Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

RMI Invocation Semantics (2)

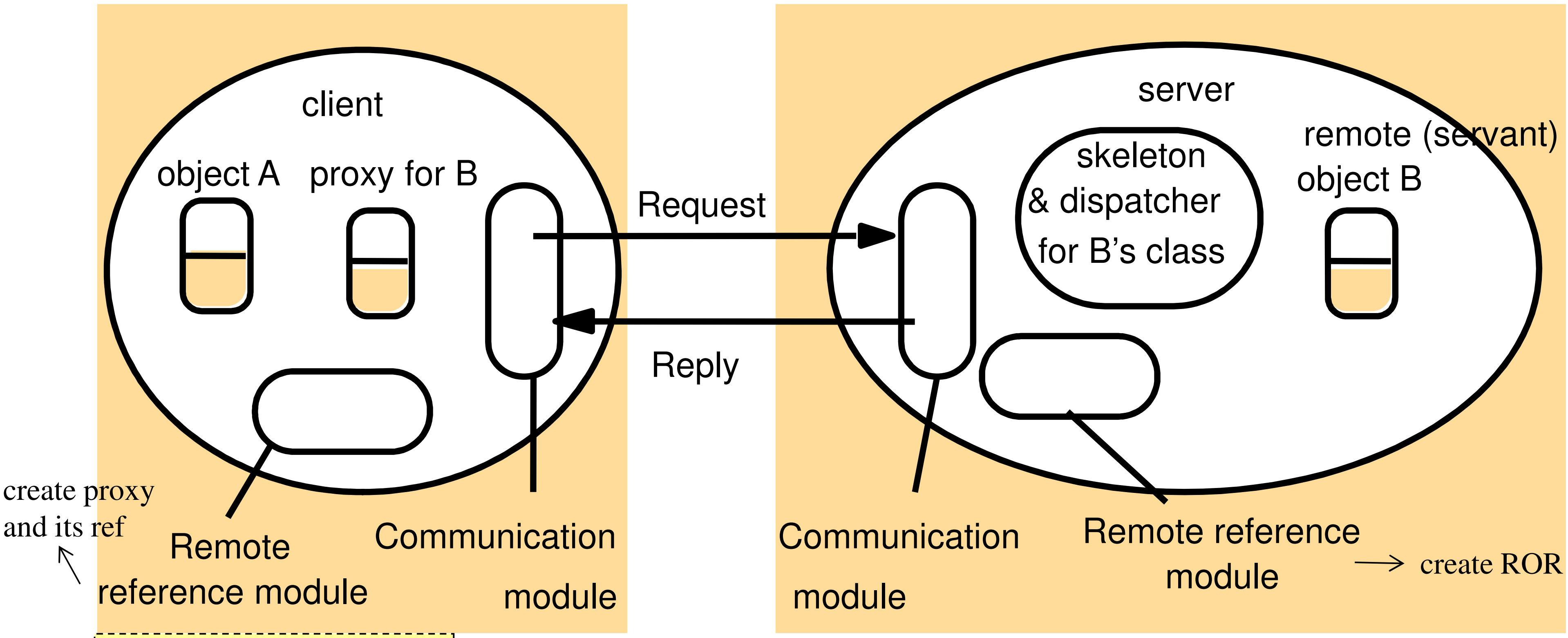
- Maybe semantics suffers from omission failure and crash failure.
  - It is useful only for applications in which occasional failed invocations are acceptable.
- At-least-once semantics suffers from crash failure and arbitrary failure (i.e. wrong value for non-idempotent operation).
  - It is useful if methods in the remote interface can be designed to be idempotent.
- At-most-once semantics suffers from crash failure.
- CORBA and Java RMI have at-most-once semantics (but CORBA allows maybe semantics if requests do not return results). Sun RPC has at-least-once semantics.

Transparency in Remote Invocation

- Syntax of remote invocation should be the same as that of a local invocation.
- All necessary calls to marshalling and message passing procedures are hidden from the programmer.
- But the difference between local and remote objects should be expressed in their interfaces.
  - CORBA and Java RMI interface can throw remote exceptions.
  - Some IDL may allow call semantics of a method to be specified (e.g. at-least-once may be used for idempotent operation in order to avoid overheads of at-most-once).

*** Very Important ***

RMI Implementation



client

object A

proxy for B

Request

Reply

server

skeleton & dispatcher for B's class

remote (servant) object B

create proxy and its ref

Remote reference module

Communication module

Communication module

Remote reference module

create ROR

Remote Object Table:
{<ROR, proxy ref>}

Remote Object Table:
{<ROR, servant ref>}

Communication Module

- Each process has a communication module.
- Two communication modules carry out request-reply protocol and provide invocation semantics.
- It uses message type, requestID, and remote object reference parts of a message.
- It selects a dispatcher for the remote object to be invoked.

Remote Reference Module

- ❑ Each process has a remote reference module.
- ❑ The process has a remote object table that records
  - Local proxy object references and corresponding remote object references (for client role)
  - Local servant object references and corresponding remote object references (for server role)
- ❑ The remote reference module creates a remote object reference for a remote object to be passed as an argument for the first time.
  - The reference is added to the table.
- ❑ It looks up the table for a local reference (of proxy or servant) when a remote object reference arrives in a request or reply.
  - For client role, if that remote object reference is not in the table, a new proxy is created (based on the interface name specified in the reference) and its reference is added to the table.

Components of RMI Software (1)

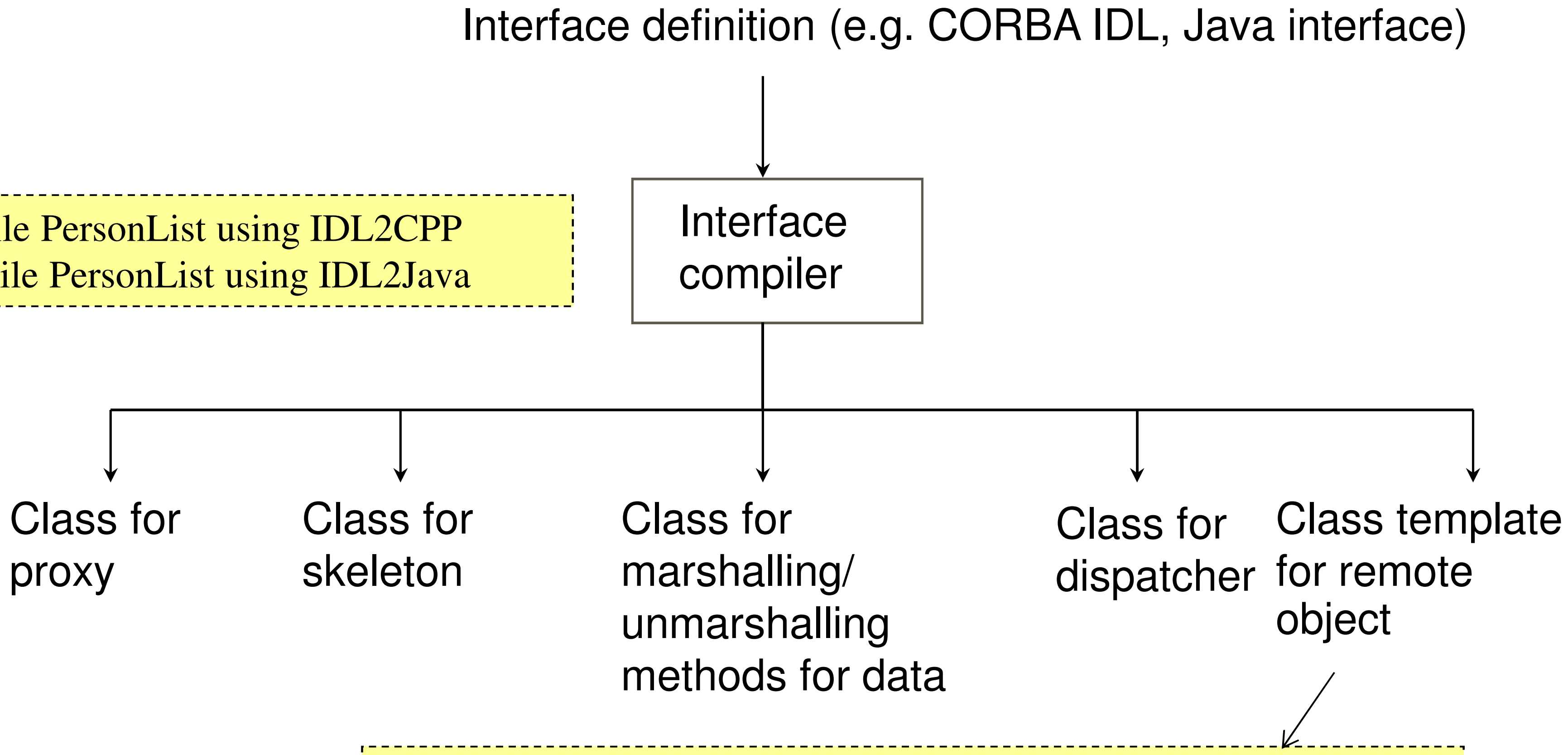
- It is between application-level objects and the communication and remote reference modules.
- Proxy
  - One proxy for each remote object class.
  - It behaves as a servant object that is local to the client, implementing the methods in the remote interface.
  - But it only marshals the request (i.e. the reference to the remote object, its own methodID, and arguments), sends it, receives and unmarshals reply, and returns to the client.

Components of RMI Software (2)

- Dispatcher
  - One for each remote object class on a server.
  - It receives the request from communication module.
  - It calls a method of a skeleton based on the methodID.
- Skeleton
  - One for each remote object class on a server.
  - It behaves as a client that is local to the remote object, implementing the methods in the remote interface.
  - But it only unmarshals the request, invokes the method in the remote object, marshals the result (together with exceptions), and sends it to the proxy’s method.

Interface Processing

Java RMI



Interface definition (e.g. CORBA IDL, Java interface)

Interface compiler

Class for proxy

Class for skeleton

Class for marshalling/ unmarshalling methods for data

Class for dispatcher

Class template for remote object

client: compile PersonList using IDL2CPP
server: compile PersonList using IDL2Java

```
public class PersonListImpl implements PersonList {
public void addPerson(Person p) throws RemoteException { }
…
```

Server and Client Programs in RMI

- ❑ Server program contains classes of dispatcher, skeleton, and remote object.
- ❑ Its initialization section creates remote object and registers it with a binder. ← register ROR with name service
- ❑ Client program contains class of proxy.
- ❑ It uses a binder to look up the remote object reference.
- ❑ A binder maintains a table containing mappings from textual names to remote object references (e.g. CORBA naming service, RMIregistry).

Java RMI

Java RMI

- It extends Java object model to support distributed objects in Java language.
- Client and server are aware of remoteness.
  - A remote object must implement Remote interface.
  - A client object must handle RemoteExceptions.
- Remote interface definition is defined using Java interface; no additional IDL.

Java Remote Interfaces for Distributed Whiteboard

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException;
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

methods

ROR

methods

- Clients draw graphical objects and inform the server.
- Server assigns a version number to each new shape that arrives and to itself.
- Clients can poll the server for the latest shapes drawn by other users.

Parameter Passing in Java RMI

- Any objects that implement Serializable interface can be passed (by value) as parameters (e.g. GraphicalObject)

deserialized

  - A new object will be created in the receiver process and its methods can be invoked locally.
- Primitive types are serialized when being passed.
- Remote object references are passed if parameters are remote objects (e.g. Shape)

RMIregistry

- It is the binder of Java RMI; its instance must run on every computer that hosts remote objects.
- It maintains a table that maps a URL-style name (//computerName:port/objectName) to a remote object reference.

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name, as shown in Figure 15.13, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 15.15 line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.

Server Program in Java RMI (1)

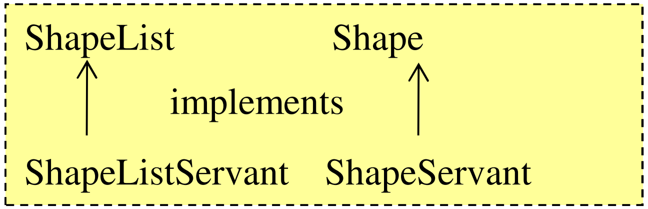
- Class ShapeListServer with main method

```
import java.rmi.*;
public class ShapeListServer{
    public static void main(String args[]){
        System.setSecurityManager(new RMISecurityManager());
         try{
            ShapeList aShapeList = new ShapeListServant();
        Naming.rebind("ShapeList", aShapeList );
            System.out.println("ShapeList server ready");
            }catch(Exception e) {
            System.out.println("ShapeList server main " + e.getMessage());}
    }
}
```

Server Program in Java RMI (2)

- Class ShapeListServant implements interface ShapeList.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
     private Vector theList;              // contains the list of Shapes
     private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {
            version++;
            Shape s = new ShapeServant( g, version);
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```



Client Program in Java RMI (1)

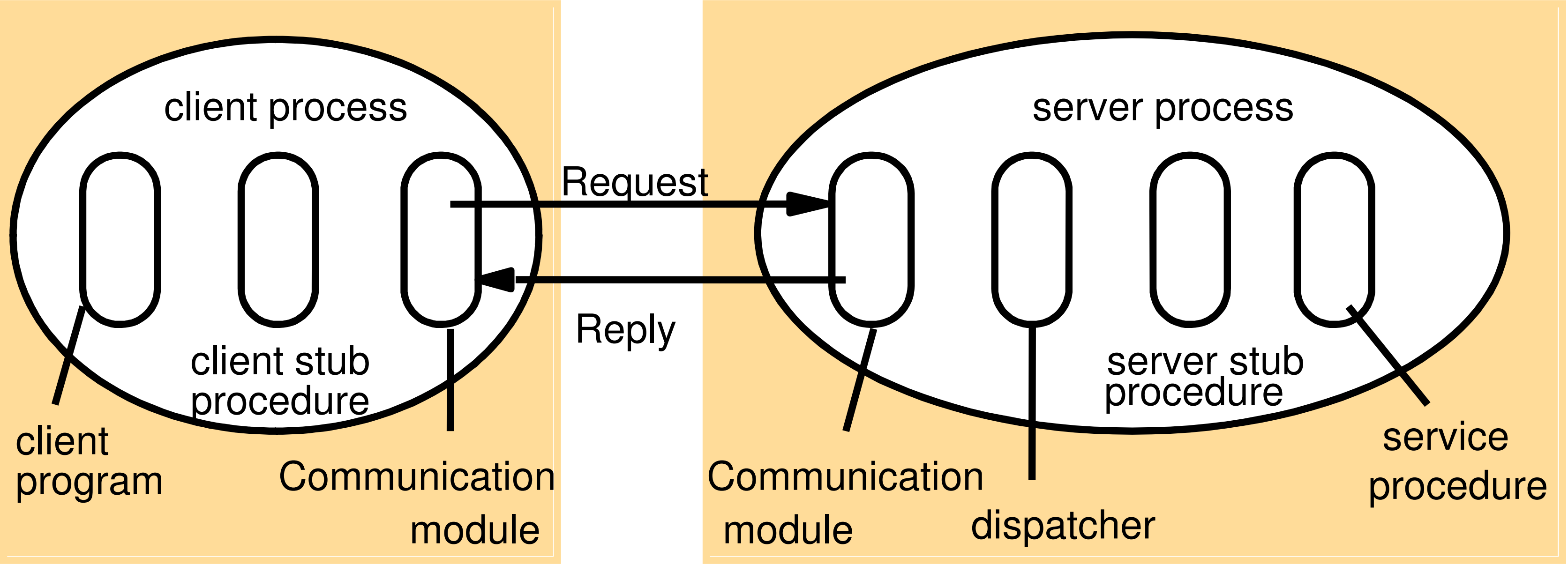
- Class ShapeListClient with main method

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
   public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{        ROR
          aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList");
          Vector sList = aShapeList.allShapes();
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

Remote Procedure Call (RPC)

- Example: Sun RPC which is designed for Sun NFS
- Since procedure call is not concerned with objects, remote reference module is not required.
- Remote object reference is omitted from request message.

client creates handle (socket)


client process
Request
Reply
client stub procedure
client program
Communication module
server process
server stub procedure
service procedure
Communication module
dispatcher