# DISTRIBUTED OBJECTS AND REMOTE INVOCATION

# Introduction

This chapter is concerned with programming models for distributed applications . . .

Familiar programming models have been extended to apply to distributed programs:

- Remote procedure call (RPC)
- Remote method invocation (RMI)
- Event-based programming model

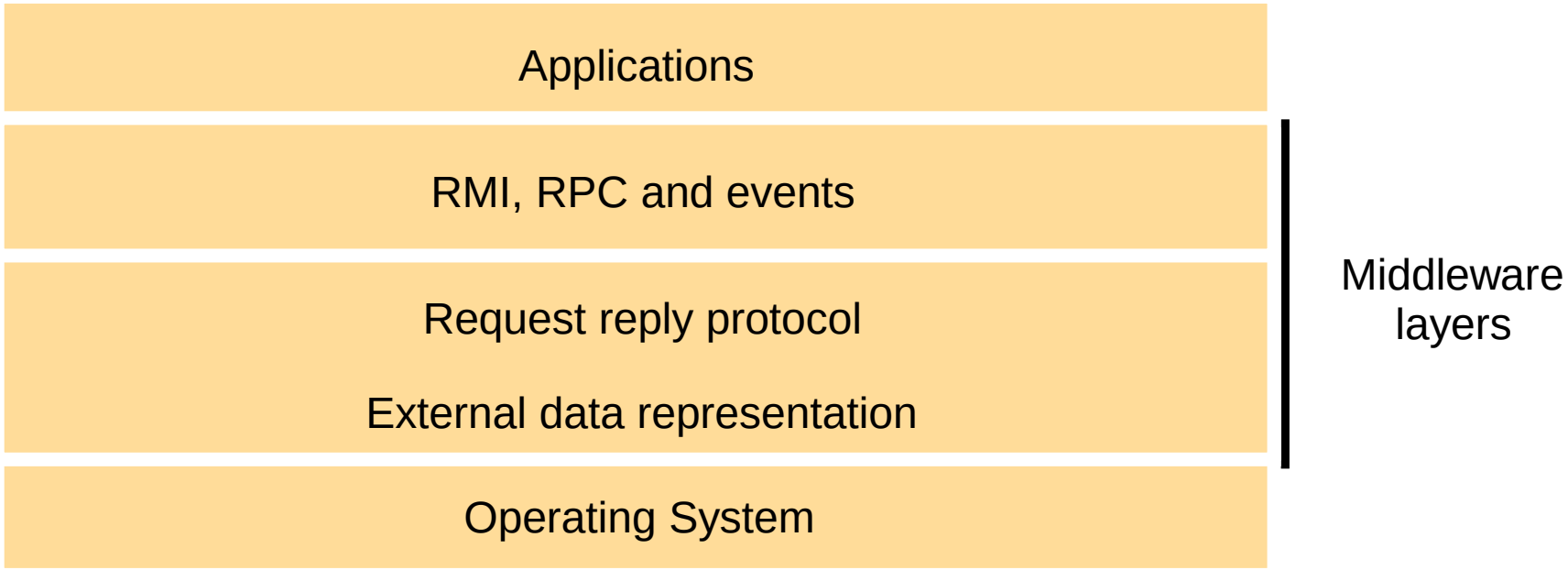
# Introduction

**Middleware**

The middleware layer uses protocols based on messages between processes to provide its higher-level abstractions. .

- Location transparency
- Communication protocols
- Computer hardware
- Operating systems
- Use of several programming languages.

# Middleware layers



# Interfaces

Communication between modules can be by procedure calls or by direct access to the variables . . .

Interface - to control the possible interactions between modules.

Interfaces in distributed systems - In a distributed program, the modules can run in separate processes. . .

# Interfaces

Parameter passing mechanisms . . .

Types of parameters in the specification of a method:

- *Input* parameters
- *Output* parameters
- parameter used for both input and output

Pointers cannot be passed . . .

# Interfaces

Client-server model: *Service interface*

- refers to the specification of the procedures offered by a server.

Distributed object model: *Remote interface*

- specifies the methods of an object that are available for invocation by objects in other processes.
- methods can pass objects as arguments and results . . .
- references to remote objects may also be passed.

# Interfaces

**Interface definition languages (IDLs)**:

An RMI mechanism can be integrated with a particular with a particular programming language if it includes a notation for defining interfaces, . . .

# Interfaces

**Interface definition languages (IDLs)**:

- designed to allow objects implemented in different languages to invoke one another.
- provides a notation for defining interfaces
- each of the parameters of a method may be described as for *input* or *output* in addition to having its type specified.

# CORBA IDL example

```
// In file Person.idl
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 IDL example

The interface named *PersonList* specifies the methods available for RMI in a remote object that implements that interface.

Ex: the method *addPerson* specifies its argument as *in*, meaning that it is an *input* argument;

the method *getPerson* that retrieves an instance of *Person* by name specifies its second argument as *out*, meaning that it is an *output* argument.

# Communication between distributed objects

The object-based model for a distributed system extends the model supported by object-oriented programming languages to make it apply to distributed objects.

Communication between distributed objects - using RMI

*The object model*

*Distributed objects*

*The distributed object model*

*Design issues for RMI*

*Implementation of RMI*

*Distributed garbage collection*

# The object model

An object-oriented program consists of a collection of interacting objects, . . .

Object references:

- to invoke a method in an object, the object reference and method name are given, together with any necessary arguments.

Interfaces:

- an interface defines the signatures of a set of methods
- an object will provide a particular interface if its class contains code that implements the methods of that interface
- an interface also defines types

# The object model

Actions:

- Action in an object-oriented program is initiated by an object invoking a method in another object.

An invocation of a method can have three effects:

1. the state of the receiver may be changed;
2. a new object may be instantiated; and
3. further invocations on methods in other objects may take place.

# The object model

Exceptions

- Programs can encounter many sorts of errors and unexpected conditions . . .
  - Exceptions provide a clean way to deal with error conditions without complicating the code.

Garbage collection

- When a language does not support garbage collection, the programmer has to deal with it.

# Distributed objects

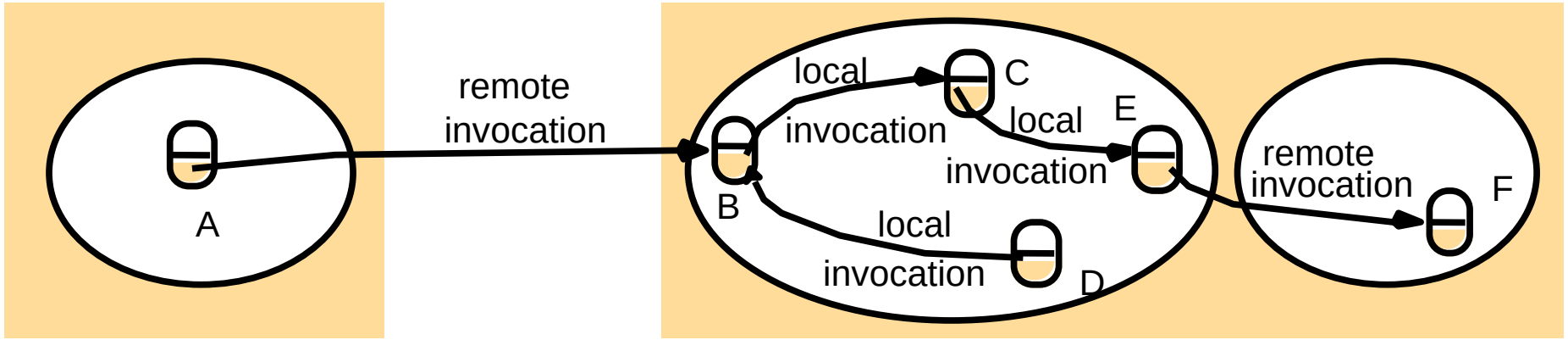
- The state of an object consists of the values of its instance variables. In the object-based paradigm the state of a program is partitioned into separate parts, each of which is associated with an object.
- Distributed object systems may adopt the client-server architecture. Objects are managed by servers . . .
- Distributed objects can assume other architectural models.
- Having client and server objects in different processes enforces encapsulation. The state of an object can be accessed only by the methods of the object . . .

# The distributed object model

- Each process contains a collection of objects
- some objects can receive both *local* and *remote* invocations

Method invocations between objects in different processes, whether in the same computer or not, are known as *remote method invocations*

# Remote and local method invocations



# The distributed object model

Remote objects: objects that can receive remote invocations.

Remote object reference: this should be available.

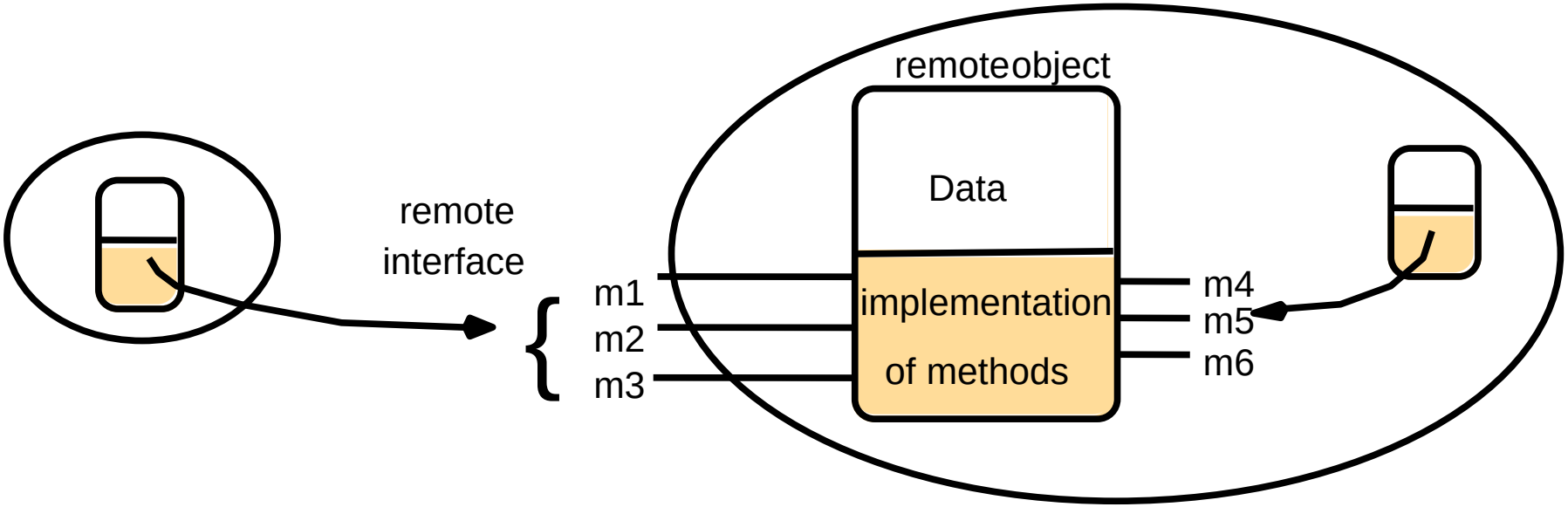
Remote interface: Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

# The distributed object model

Remote object reference: An identifier that can be used throughout a distributed system.

Remote interface: The class of a remote object implements the methods of its remote interface.
Objects in other processes can invoke only the methods that belong to its remote interface.
- CORBA provides an IDL used for defining remote interfaces.
- In Java RMI, remote interfaces are defined in the same way as any other Java interface.
They extend an interface named *Remote*.

# A remote object and its remote interface



# The distributed object model

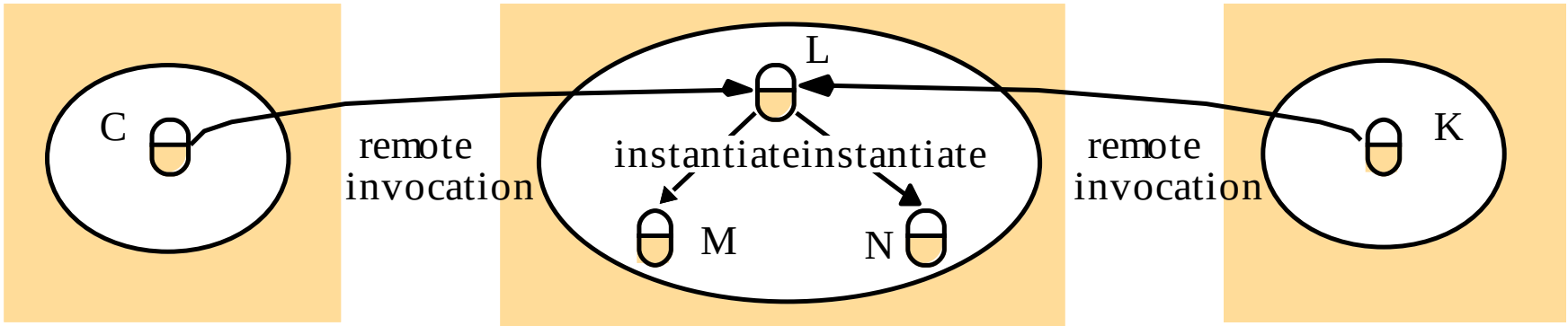
Actions in a distributed object system:

- is initiated by a method invocation, which may result in further invocations on methods in other objects. . .
- when an action leads to the instantiation of a new object, that object will normally live with in the process where instantiation is requested.

if the newly instantiated object has a remote interface, . . .

- distributed applications may provide remote objects with methods for instantiating objects which can be accessed by RMI.

# Instantiation of remote objects



# The distributed object model

Garbage collection in a distributed-object system:

- Distributed garbage collection is generally achieved by cooperation between the existing local garbage collector and an added module.

Exceptions:

- Remote method invocation should be able to raise exceptions that are due to distribution as well as those raised during the execution of the method invoked.

# Design Issues for RMI

Design issues in making RMI a natural extension of local method invocation:

- The choice of invocation semantics.
- The level of transparency that is desirable for RMI.

# Design Issues for RMI

RMI invocation semantics - the *doOperation* can be implemented in different ways to provide different delivery guarantees. The main choices are:

- *Retry request message*
- *Duplicate filtering*
- *Retransmission of results*

Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations.

# Invocation semantics

| *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* |

# Design Issues for RMI

Maybe invocation semantics:

- The remote method may be executed once or not at all.
- None of the fault tolerance measures is applied.
- Types of failure:
  - omission failures;
  - crash failures.
- if the result message has not been received after a timeout and there are no retries, . . .
- Useful only for applications in which occasional failed invocations are acceptable .

# Design Issues for RMI

At-least-once invocation semantics:

- The invoker receives either a result or an exception informing it that no result was received.
- Can be achieved by the retransmission of request messages.
- Types of failure:
  - crash failures;
  - arbitrary failures.
- May be used if all of the methods in the remote interfaces are idempotent operations.

# Design Issues for RMI

At-most-once invocation semantics:

- The invoker receives either a result or an exception informing it that no result was received.
- Can be achieved by using all of the fault tolerance measures.

# Design Issues for RMI

Transparency: Remote invocation can be made to look like local invocation. . . .

- Remote invocations are more vulnerable to failures than local ones

impossible to distinguish between failure of the network & of the remote server process

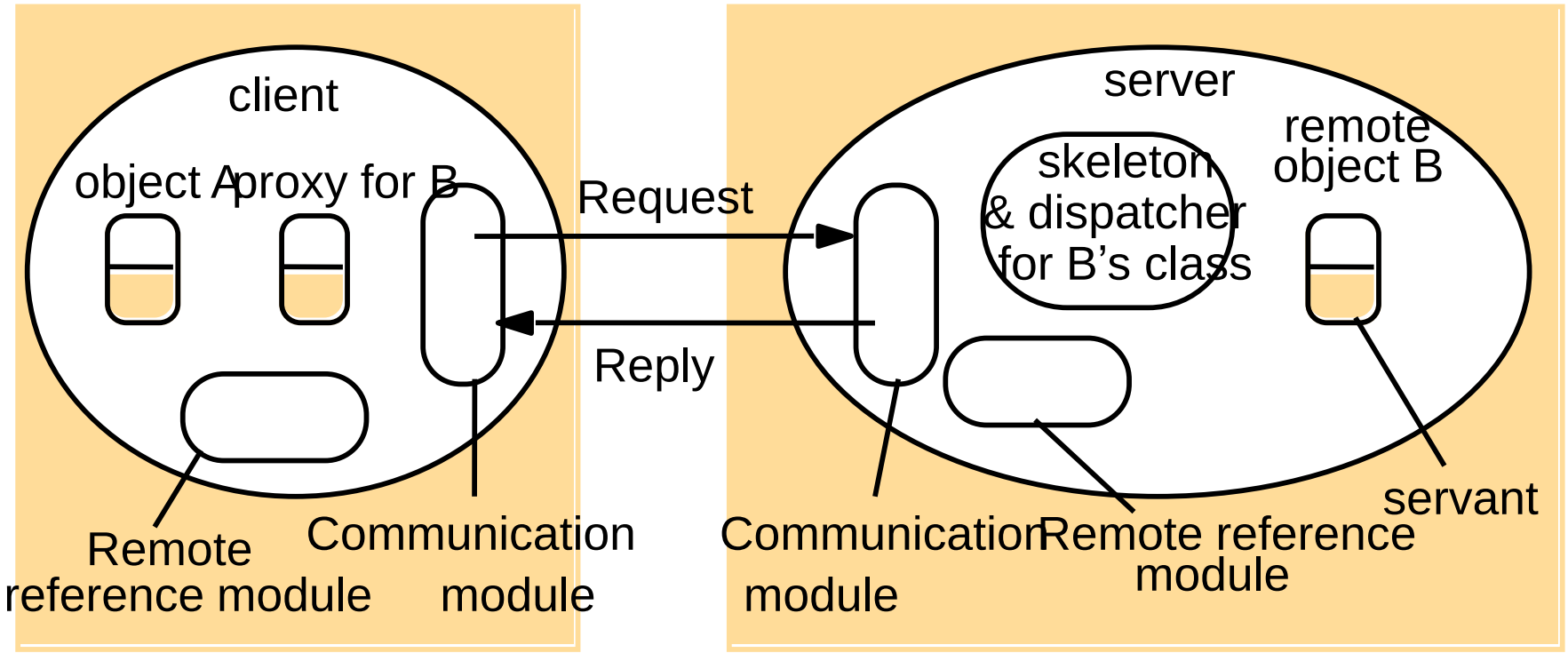
- The latency of a remote invocation is several orders of magnitude greater than that of a local one.

# Implementation of RMI

Several separate objects and modules are involved.

Illustration: an application-level object A invokes a method in a remote application-level object B for which it holds a remote object reference.

# The role of proxy and skeleton in remote method invocation



# Implementation of RMI

Communication module: The two cooperating communication modules carry out the request-reply protocol.

- This module uses only the first three items of the request-reply message - message type, requestId and object reference
- These modules are together responsible for providing a specified invocation semantics.
- The communication module in the server selects the dispatcher for the class of the object to be invoked, passing on its local reference, . . .

# Implementation of RMI

Remote reference module:

- is responsible for translating between local and remote object references and for creating remote object references.
- the remote reference module in each process has a *remote object table* that records the correspondence between local object references in that process and remote object references.

# Implementation of RMI

The remote object table includes:

- An entry for all the remote objects held by the process.
  Ex: the remote object B will be recorded in the table at the server.
- An entry for each local proxy.
  Ex: the proxy for B will be recorded in the table at the client.

# Implementation of RMI

Actions of the remote reference module:

- When a remote object is to be passed as argument or result for the first time, the remote reference module is asked to create a remote object reference, which it adds to its table.
- When a remote object reference arrives in a request or reply message, the remote reference module is asked for the corresponding local object reference.

# Implementation of RMI

Servants:

- It is an instance of a class which provides the body of a remote object.
- eventually handles the remote requests passed on by the corresponding skeleton.

# Implementation of RMI

The RMI software:

- A layer of software between the application-level objects and the communication and remote reference modules.
- Roles of the middleware objects:
  - Proxy
  - Dispatcher
  - Skeleton

# Implementation of RMI

*Proxy:*

- makes remote method invocation transparent to clients by behaving like a local object to the invoker.
- hides the details of the remote object reference, the marshalling of arguments, etc.
- There is one proxy for each remote object for which a process holds a remote object reference.

# Implementation of RMI

*Dispatcher:*

- A server has one dispatcher and skeleton for each class representing a remote object.
- It receives the *request* message from the communication module, uses the *methodId* to select the appropriate method in the skeleton & passes on the *request* message.

# Implementation of RMI

*Skeleton:*

- The class of a remote object has a *skeleton,* which implements the methods in the remote interface.
- A skeleton method unmarshals the arguments in the *request* message and invokes the corresponding method in the servant.
- It waits for the invocation to complete and then marshals the result, together with any exceptions, in a *reply* message to the sending proxy's method.

# Implementation of RMI

The binder:

- a separate service that maintains a table containing mappings from textual names to remote object references.
- used by servers to register their remote objects by name & by clients to look them up.

Server threads:

- to avoid the execution of one remote invocation delaying the execution of another, servers generally allocate a separate thread for the execution or each remote invocation.

# Implementation of RMI

Activation of remote objects:

- Some applications require that information survive for long periods of time. . .

- Servers that manage remote objects can be started whenever they are needed by clients.
- Processes that start server processes to host remote objects are called *activators*.

# Implementation of RMI

A remote object is described as *active* when it is available for invocation within a running process, whereas it is called *passive* if is not currently active but can be made active.

A passive object consists of two pans:

- the implementation of its methods; and
- its state in the marshalled form.

# Implementation of RMI

Activation

- create an active object from the corresponding passive object
- by creating a new instance of its class and initializing its instance variables from the stored state.

An activator is responsible for:

- Registering passive objects that are available for activation
- Starting named server processes and activating remote objects in them
- Keeping track of the locations of the servers for remote objects that it has already activated

# Implementation of RMI

**Persistent object stores**

- Persistent object: An object that is guaranteed to live between activations of processes.
- a persistent object store will manage very large numbers of persistent objects.
- Activation is normally transparent
- Persistent objects that are no longer needed in main memory can be passivated.

# Implementation of RMI

**Object location**

- In the simple case, a remote object reference can act as an address. . .

- A *location service* helps clients to locate remote objects from their remote object references.
It uses a database that maps remote object references to their current locations

# Implementation of RMI

**Distributed garbage collection**

- the aim is to ensure that if a local or remote reference to an object is still held anywhere, then the object will continue to exist.

# Implementation of RMI

Java distributed garbage collection algorithm

- based on reference counting.
- whenever a remote object reference enters a process, a proxy will be created.
- the server should be informed of the new proxy at the client.
- later when there is no longer a proxy at the client, the server should be informed.

# Implementation of RMI

- Each server process maintains a set of the names of the processes that hold remote object references for each of its remote objects;
  Ex: *B.holders* is the set of client processes that have proxies for object B.
- When a client C first receives a remote reference to a particular remote object, B, it makes an *addRef(B)* invocation to the server of that remote object and then creates a proxy;
  the server adds C to *B.holders.*

# Implementation of RMI

- When a client C's garbage collector notices that a proxy for remote object *B* is no longer reachable, it makes a *removeRef(B)* invocation to the corresponding server and then deletes the proxy;
  the server removes C from *B.holders.*
- When *B.holders* is empty, the server's local garbage collector will reclaim the space occupied by *B* unless there are any local holders.

# Implementation of RMI

This algorithm is intended to be carried out by means of pairwise request-reply communication with at-most-once invocation semantics between the remote reference modules in processes.

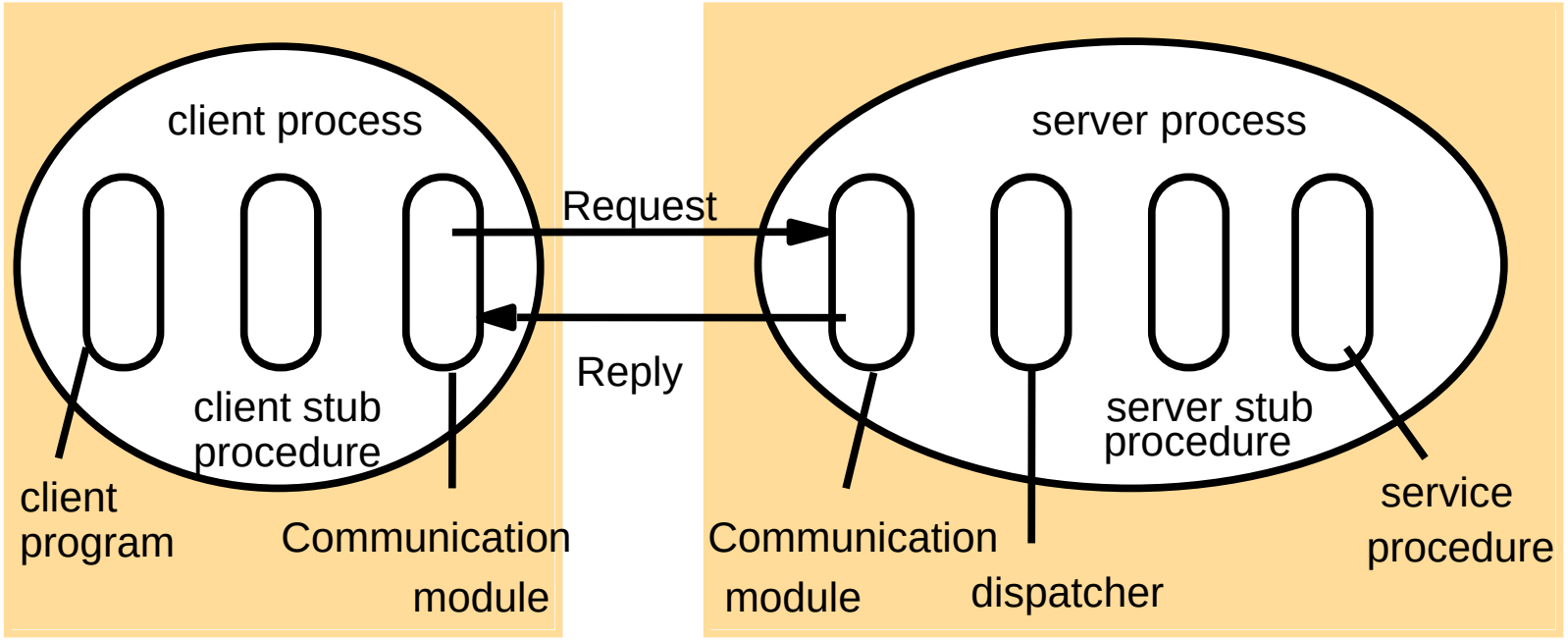
The algorithm tolerates communication failures:
- if addRef(B) call returns an exception . . .
- if removeRef(B) fails . . . leases

The algorithm tolerates failure of client processes . . . leases

# Remote procedure call

- A client program calls a procedure in another program running in a server process.
- Servers may be clients of other servers to allow chains of RPCs.
- A server process defines in its *service interface* the procedures that are available for calling remotely.
- May be implemented to have one of the choices of invocation semantics.

# Role of client and server stub procedures in RPC in the context of a procedural language



# Remote procedure call

The software that supports RPC

- no remote reference modules are required.
- Client that accesses a service includes one *stub procedure* for each procedure in the service interface.
- Role of a stub procedure is similar to that of a proxy method.

# Remote procedure call

The software that supports RPC

- Server process contains a dispatcher together with one server stub procedure and one service procedure for each procedure in the service interface.
- The dispatcher selects one of the server stub procedures according to the procedure identifier in the request message.
- A *server stub procedure* is like a skeleton method.
- The service procedures implement the procedures in the service interface.

# Events and notifications

Events - the actions that the user performs on objects that cause changes in the objects that maintain the state of the application.

Distributed event-based systems extend the local event model by allowing multiple objects at different locations to be notified of events taking place at an object

The *publish-subscribe* paradigm.

Notifications - Objects that represent events

- Notifications may be stored, sent in messages, queried and applied in a variety of orders to different things.

# Events and notifications

Two main characteristics of distributed event-based systems:

Heterogeneous:

- components in a distributed system that were not designed to interoperate can be made to work together.

Asynchronous:

- notifications are sent asynchronously by event-generating objects to all the objects that have subscribed to them.
- publishers and subscribers need to be decoupled.

# Events and notifications

**Simple dealing room system**

- The task is to allow dealers using computers to see the latest information about the market prices of the stocks they deal in.
- The market price for a single named stock is represented by an object with several instance variables.
- The information arrives in the dealing room from several different external sources . . .
- It is collected by processes called information providers.

# Events and notifications

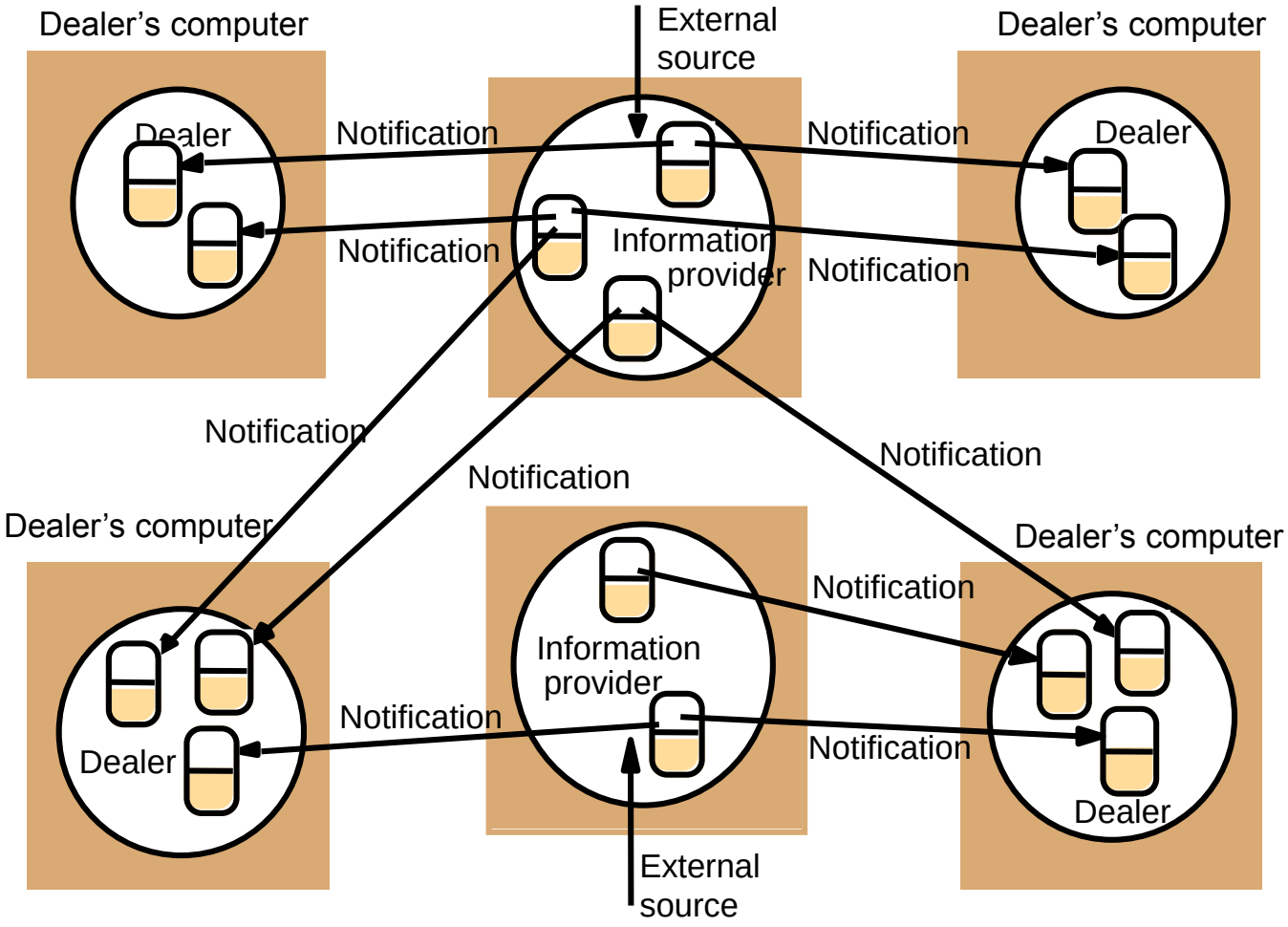
**Simple dealing room system**

The system can be modelled by

An information provider process that continuously receives new trading information from a single external source and applies it to the appropriate stock objects

A dealer process creates an object to represent each named stock that the user asks to have displayed.

# Dealing room system



# Events and notifications

**Event types**

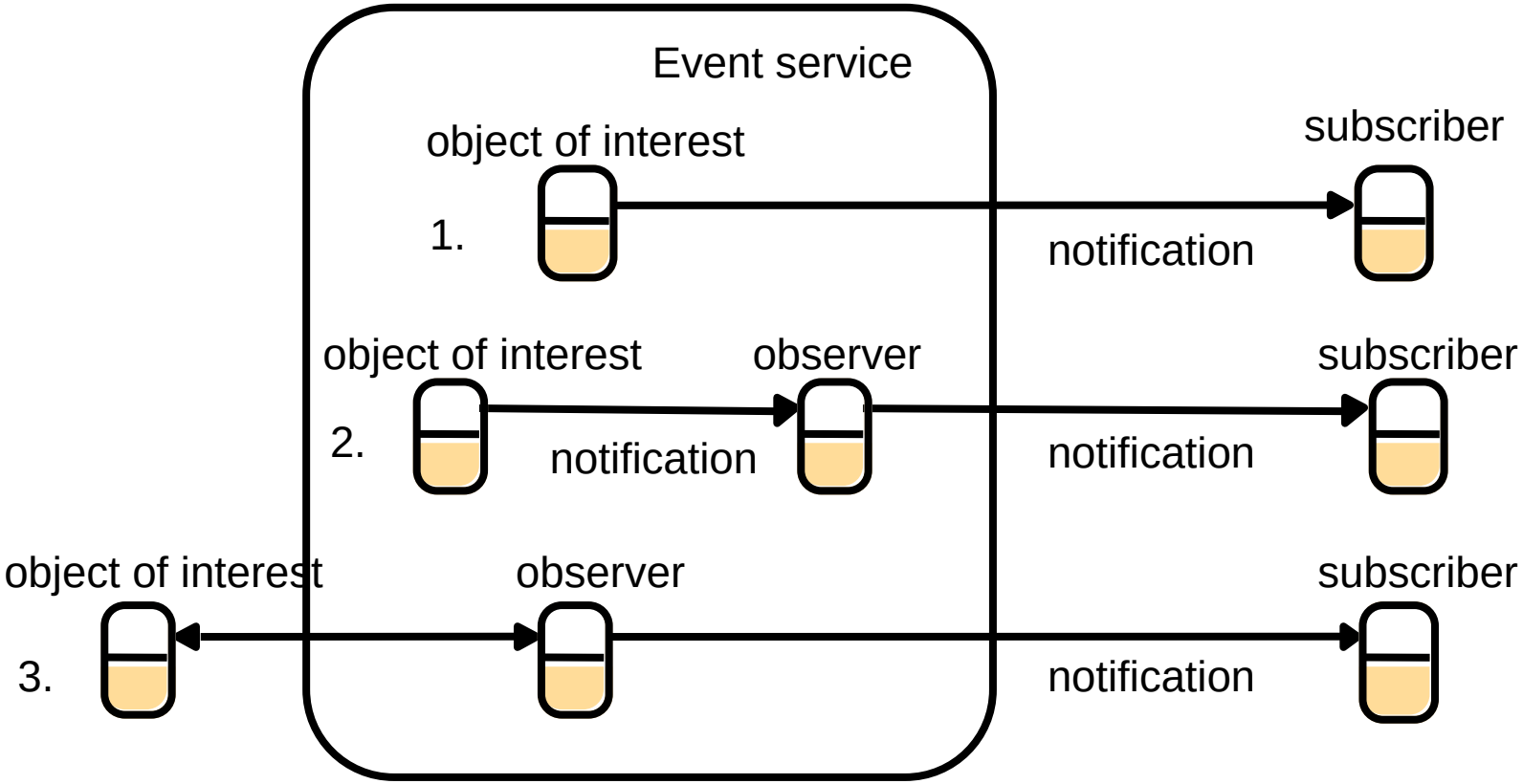
- an event source can generate events of one or more different *types*.
- each event has *attributes* that specify information about that event.
- types and attributes are used both in subscribing to events and in notifications.

# Participants in distributed event notification

An architecture designed to decouple the publishers from the subscribers.

Event service - maintains a database of published events and of subscribers' interests.

# Architecture for distributed event notification



# Architecture for distributed event notification

*The object of interest*

*Event*

*Notification*

*Subscriber*

*Observer objects*

*Publisher*

# Architecture for distributed event notification

Three cases:

1. An object of interest inside the event service without an observer.
2. An object of interest inside the event service with an observer.
3. An object of interest outside the event service.

# Architecture for distributed event notification

Delivery semantics

A variety of different delivery guarantees can be provided for notifications

- the one that is chosen should depend on the requirements of applications.

# Architecture for distributed event notification

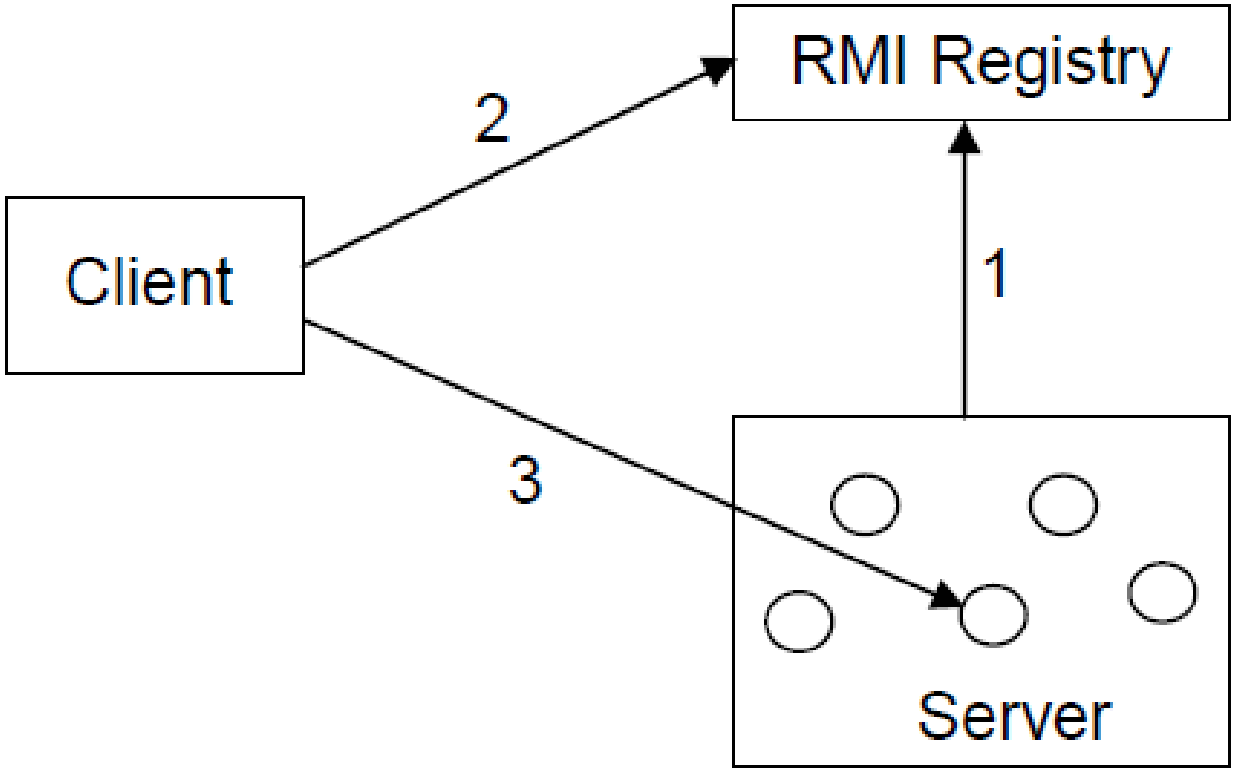
Roles for observers

- Forwarding
- Filtering of notifications
- Patterns of events
- Notification mailboxes

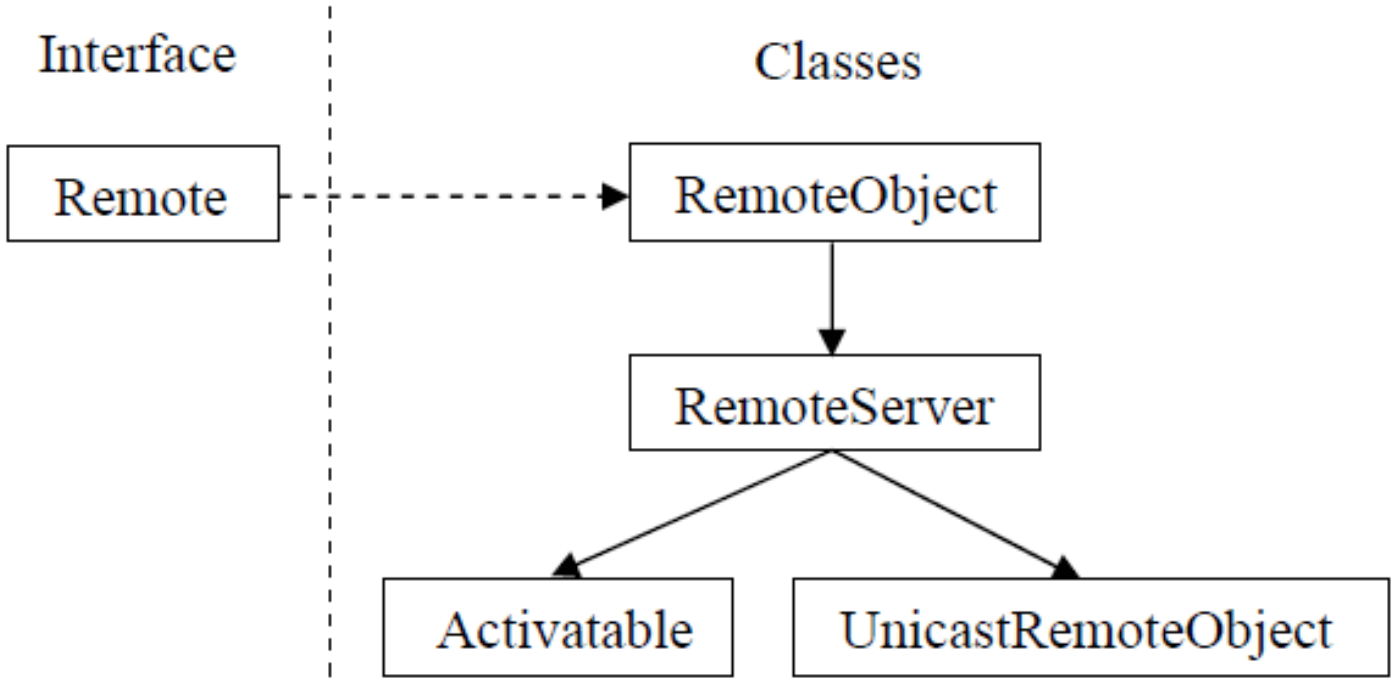
# Distributed object application. . .

- Locate remote objects
- Communicate with remote objects
- Load class bytecodes for objects that are passed as parameters or return values

# An Illustration of a distributed object application.



# Interfaces and Classes in the *java.rmi* package.



# Interfaces and Classes . . .

The *RemoteObject* class implements the *Remote* interface while the other classes extend *RemoteObject*.

Implementor of a remote object must implement the *Remote* interface, which must satisfy the following conditions:

1. It must extend the interface *Remote*.
2. Each remote method declaration in the remote interface must include the exception *RemoteException*

# Interfaces and Classes . . .

The *RemoteObject* class:

RMI server functions are provided by the class *RemoteObject* and its subclasses *RemoteServer*, *UnicastRemoteObject* and *Activatable*.

# RMIregistry

Binder for Java RMI.

An instance of RMIregistry must run on every server computer that hosts remote objects.

Maintains a table mapping textual, URL-style names to references to remote objects hosted on that computer.

Accessed by methods of the *Naming* class

# The *Naming* class of Java RMIregistry

*void rebind (String name, Remote obj)*

This method is used by a server to register the identifier of a remote object by name.

*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. A remote object reference is returned.

*String [] list()*

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

# Implementation of a simple RMI system

This RMI system contains the following files:

- HelloWorld.java: The remote interface.
- HelloWorldClient.java: The client application in the RMI system.
- HelloWorldServer.java: The server application in the RMI system.

# HelloWorld.java

```
import java.rmi.Remote;
import java.rmi.RemoteException;
//Classname: HelloWorld
//Comment: The remote interface.
public interface HelloWorld extends Remote {
    String helloWorld() throws RemoteException;
}
```

# HelloWorldClient.java

**Client program**: The client

- starts by using a binder to look up a remote object reference.
- continues by sending RMIs to that remote object or to others

# HelloWorldClient.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
/*
Classname: HelloWorldClient
Comment: The RMI client.
*/
```

# HelloWorldClient.java

```
public class HelloWorldClient {
    static String message = "blank";
    static HelloWorld obj = null;
```

# HelloWorldClient.java

```
public static void main(String args[])
{
   try {
      obj = (HelloWorld)Naming.lookup("//"+ "kvist.cs.umu.se"+
"/HelloWorld");
      message = obj.helloWorld();
      System.out.println("Message from the RMI-server was: \""
+ message + "\"");
   }
   catch (Exception e) {
      System.out.println("HelloWorldClient exception: "
+ e.getMessage());
      e.printStackTrace();
   }
}}
```

# HelloWorldServer.java

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
import java.rmi.server.UnicastRemoteObject;
```

# HelloWorldServer.java

```
public class HelloWorldServer extends nicastRemoteObject
implements HelloWorld {
   public HelloWorldServer() throws RemoteException {
      super();
   }
   public String helloWorld() {
      System.out.println("Invocation to helloWorld was
succesful!");
      return "Hello World from RMI server!";
}
```

# HelloWorldServer.java

```
  public static void main(String args[]) {
    try {
      // Create an object of the HelloWorldServer class.
      HelloWorldServer obj = new HelloWorldServer();
      // Bind this object instance to the name "HelloServer".
      Naming.rebind("HelloWorld", obj);
      System.out.println("HelloWorld bound in registry");
    }
    catch (Exception e) {
      System.out.println("HelloWorldServer error: " +
e.getMessage());
      e.printStackTrace();
    }
  }
}
```