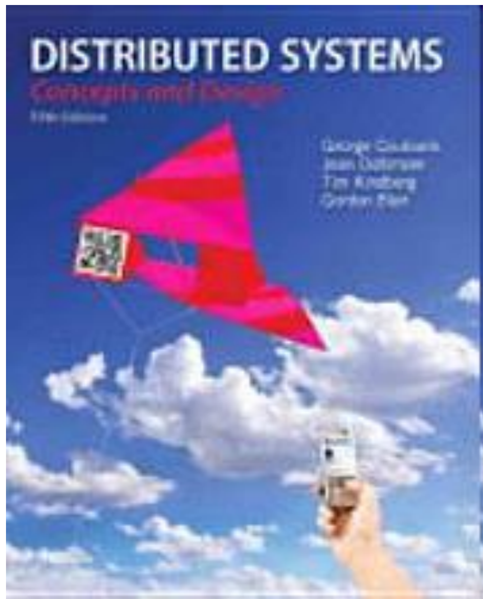
Distributed Objects and Remote Invocation



From **Coulouris, Dollimore and Kindberg**

Distributed Systems: Concepts and Design

Edition 5, © Addison-Wesley 2011

Objectives

- To study communication between distributed objects and the integration of remote method invocation into a programming language
- To be able to use Java RMI to program applications with distributed objects
- To study the extension of the event-based programming model to apply to distributed event-based programs

© Pearson Education 2005

Three Widely Used Programming Models

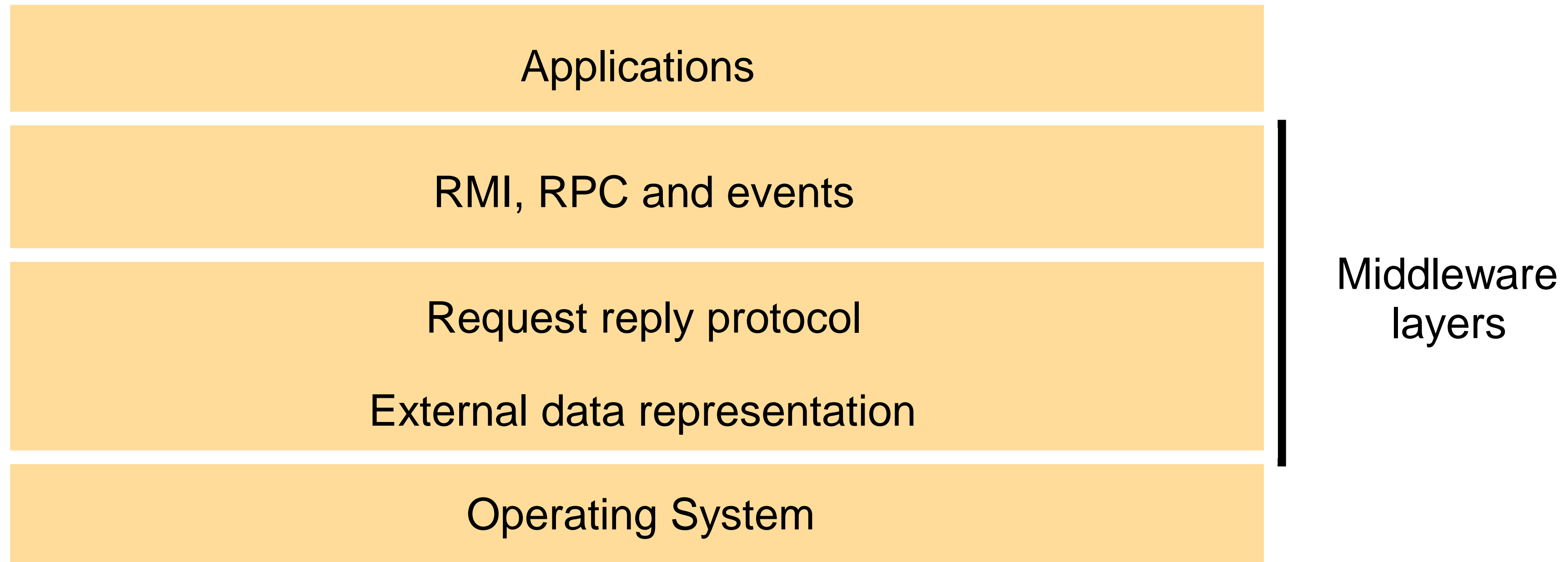
- *Remote Procedure Call model* - This is an extension of the conventional procedure call model.

- *Remote Method Invocation model* - This is an extension of the object-oriented programming model.

- *Event Based model* - This is based on the event based model which allows objects to receive notification

© Pearson Education 2005

Middleware layers



© Pearson Education 2005

Specific Abstractions Supported By Middleware Are:

- **Location transparency** - Clients do not have to know if the procedures and methods being called remote or local. Similarly in event notification, the object generating and receiving messages do not have to be aware of each others location.
- **Communication protocols** - Underlying protocols used for communication are hidden from the developer.
- **Computer Hardware** - Heterogeneity of hardware (e.g. different data representation formats) are hidden from the programmer (e.g. marshalling and unmarshalling to take care of differences in data representation formats)
- **Operating System** – Independent of the underlying OS.
- **Programming Language** - Certain middleware provides programming language independence while others do not (e.g. CORBA provides language independence while Java RMI does not).

© Pearson Education 2005

Interfaces

- An interface between two modules defines the types of interaction that can take place
- Parameter passing across external interfaces have the following restrictions:
  - Call by value and call by reference are not valid
  - Parameters passed across processes are defined as *input*, *output* or *both*
  - Pointers cannot be passed

© Pearson Education 2005

Types of Interfaces

- **Service interface:** Defines the procedures available from a server for RPC (e.g., a file server)
- **Remote interface:** Defines the methods of an object that can be invoked by objects of another processes using RMI
- **Interface Definition Languages (IDLs):** These provide a generic way of defining interfaces. Both client and server code stubs can be generated by compiling the interface specifications in IDL. IDL compilers allow code stubs to be generated in different languages, allowing objects implemented in one language to interact with objects implemented in a different language.

© Pearson Education 2005

Communication Between Distributed Objects

- Object-oriented concepts
- Distributed object concepts
- Design Issues
- Implementation
- Distributed Garbage Collection

© Pearson Education 2005

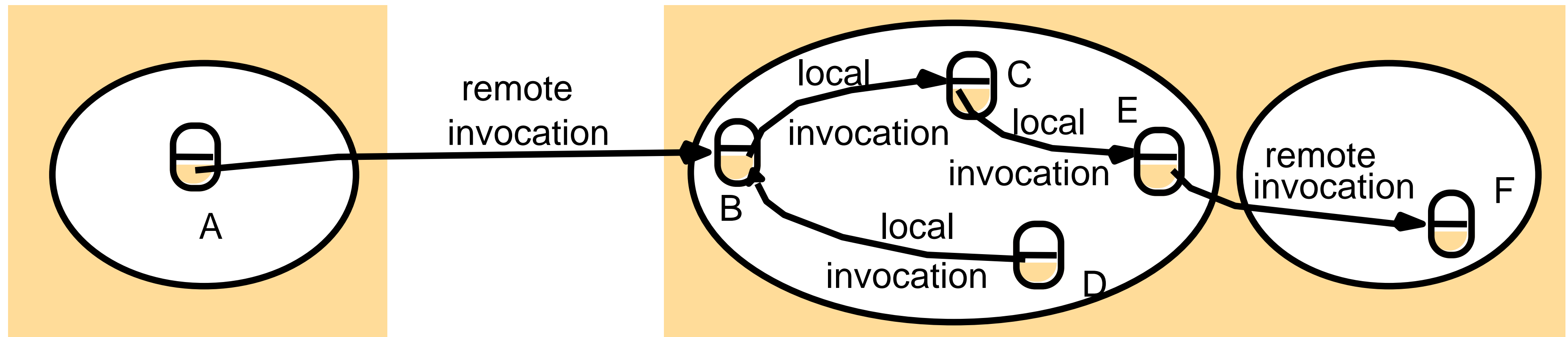
Object Communication (1):

- Distributed objects
  - state: values of its instances variables
  - actions: accessed only by its own methods

© Pearson Education 2005

Object communication (2):

- Local: within the same process
- Remote: difference processes (could be on different machines)
- Remote object: A remote object is one that can be invoked from another process



© Pearson Education 2005

Two Questions About Remote Objects

- A remote object can receive remote invocations as well as local invocations. True or False?

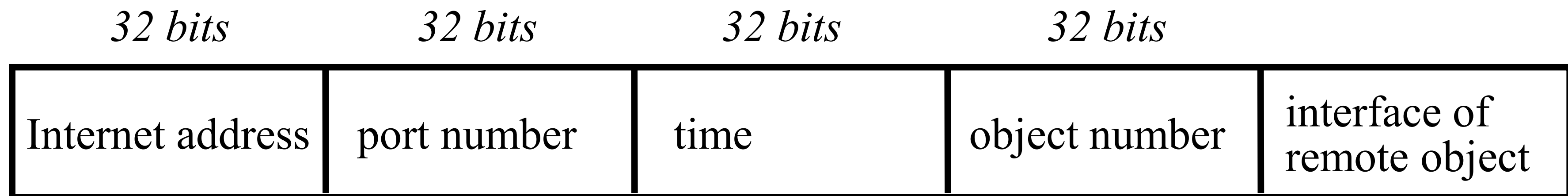
- Remote objects can invoke methods in local objects as well as other remote objects. True or False?

1. Remote object reference: Other objects can invoke the methods of a remote object if they have access to its remote object reference.
2. Remote interface: Every remote object has a remote interface that specifies which of its methods can be invoked remotely.

© Pearson Education 2005

Object communication (3):

- Remote object reference
  - accessing the remote object
  - identifier throughout a distributed system
  - can be passed as arguments
  - Definition: A remote object reference is an identifier for a remote object which is used to refer to it as the target of a remote invocation and can be passed as an argument or result of a remote invocation.

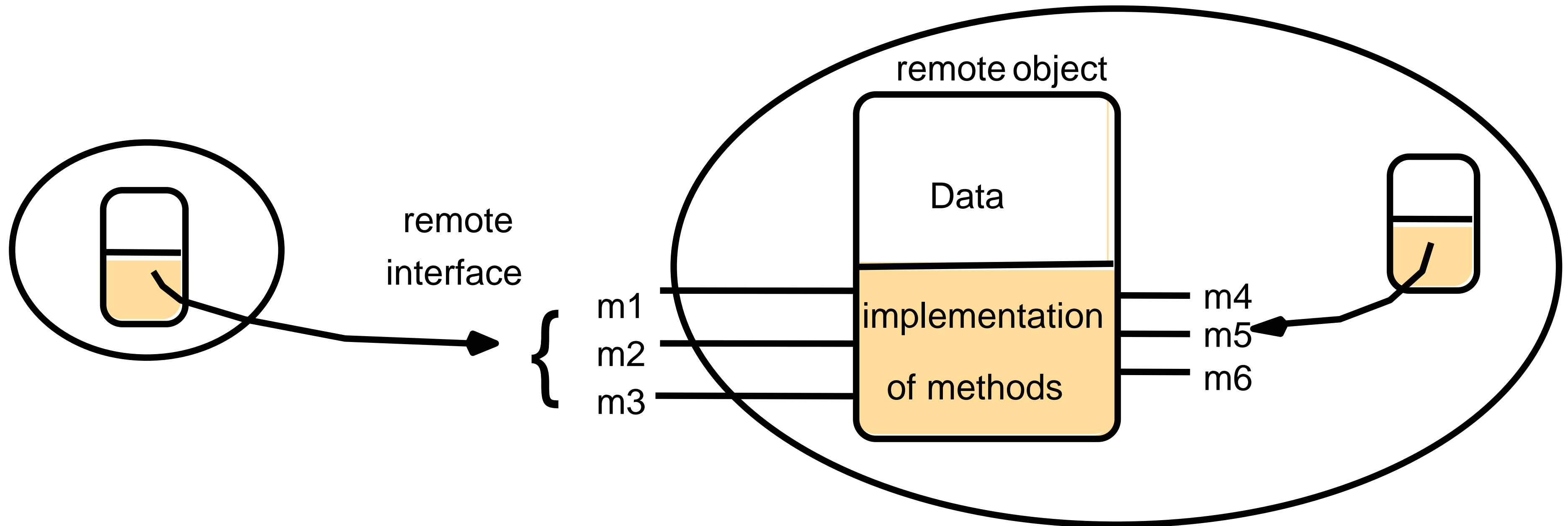


© Pearson Education 2005

Object communication (4):

- Remote interface
  - specifying which methods can be invoked remotely
  - name, arguments, return type
  - IDL
- Who defines the remote interface ?
- Who implements the methods of a object's remote interface?

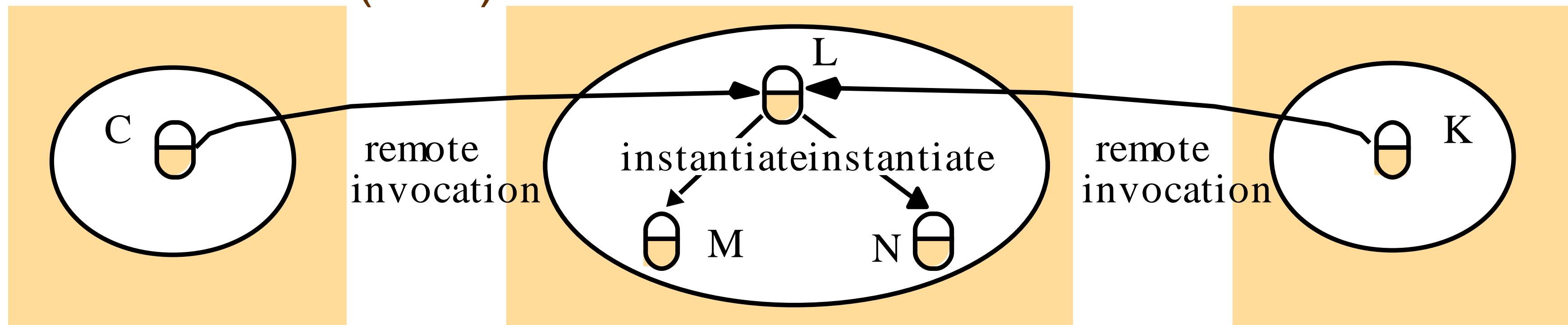
IDL is used for defining remote interfaces.



The class of a remote object implements the methods of its remote interface.

Object communication (5):

- Actions
  - Action is initiated by an object invoking a method in another object
  - Needs remote object reference
  - Calling of methods of objects in another process/host
  - Remote objects might have methods for instantiation (hence remote instantiation). Actions are invoked using Remote Method Invocation (RMI)



© Pearson Education 2005

Three Effects of An Invocation

- The state of the receiver may be changed.
- A new object may be instantiated.
- Further invocation on methods in other objects may take place.

© Pearson Education 2005

Object communication (6):

- Garbage collection
  - achieved through reference counting
  - local garbage collector
  - additional module to coordinate
- Exceptions
  - unexpected events or errors
  - similar to local invocations, but special exceptions related to remote invocations are available

© Pearson Education 2005

RMI Design Issues (1): Invocation Semantics

- **RMI invocation semantics:** Three main design decisions related to implementation of the request/reply protocols are
  - Strategy to retry request message
  - Mechanism to filter duplicates
  - Strategy for results retransmission: history

- Semantics
  - Maybe
  - At least once
  - At most once

© Pearson Education 2005

RMI Design Issues (2): 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*

© Pearson Education 2005

Choices Of RMI Semantics

- ***Maybe:*** the remote method may be executed once or not at all. It's useful only for applications in which occasional failed invocations are acceptable.
- ***At-least-once invocation semantics:*** the invoker receives either a result (the method was executed at least once) or an exception informing if no result was received. It's useful in idempotent situations.
- ***At-most-once:*** the method at most executed once. Java RMI uses at-most-once.

© Pearson Education 2005

Failures Associated With The 3 Invocation Semantics

- ***Maybe:*** omission failure, crash failures, no fault-tolerance at all
- ***At-least-once invocation semantics:*** crash failures, arbitrary failures
- ***At-most-once:*** complete fault tolerance

© Pearson Education 2005

RMI Design Issues (3): Transparency

Although location and access transparency are goals for middleware, in some cases complete transparency is not desirable due to:

- remote invocations being more prone to failure due to network and remote machines
- latency of remotes invocations is significantly higher than that of local invocations
- Therefore, many implementations of RMI provide access transparency but not complete location transparency.

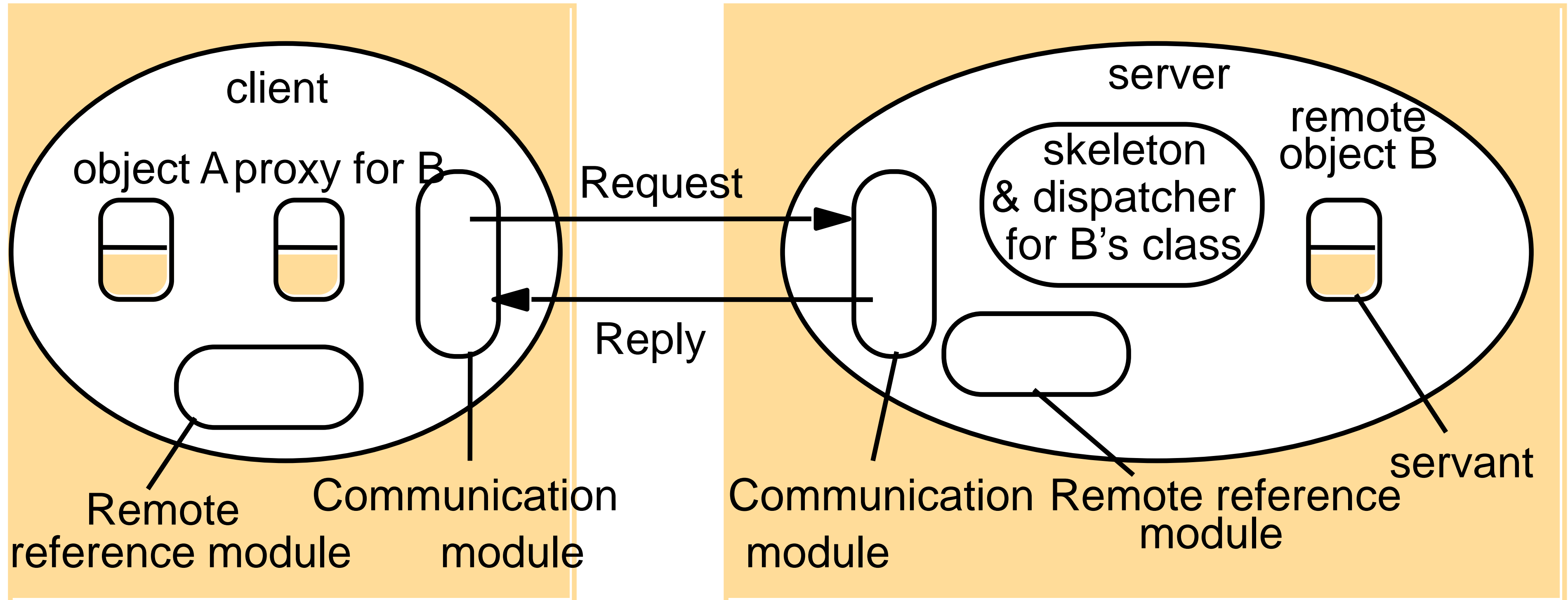
© Pearson Education 2005

RMI Design Issues (4): Transparency

- syntax might need to be different to handle different local vs remote errors/exceptions (e.g. Argus)
- affects IDL design
- current consensus
  - syntax is transparent
  - different interfaces (e.g., Java: implement Remote interface, RemoteExceptions)

© Pearson Education 2005

RMI Implementation (1):



© Pearson Education 2005

RMI Implementation (3):

Communication Module is Responsible for communicating messages (requests and replies) between the client and the server. It uses three fields from the message:

- message type
- request ID
- remote object reference

It is responsible for implementing a specified invocation semantics, for example, *at-most-once*.

© Pearson Education 2005

RMI Implementation (3):

Remote reference module Is responsible for:

- Creating remote object references
- Maintaining the remote object table which is used for translating between local and remote object references
  - remote object table
    - remote objects held by the process (B on server)
    - local proxy (B on client)
  - Entries are added to the remote object table when:
    - A remote object reference is passed for the first time
    - When a remote object reference is received and an entry is not present in the table

© Pearson Education 2005

The Actions Of The Remote Reference Module

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

- When a remote object reference arrives in a request, the module is asked for the corresponding local object reference, which may refer to a remote object.

© Pearson Education 2005

RMI Implementation (4):

- RMI software
  - Proxy: behaves like a local object, but represents the remote object
  - Dispatcher: look at the methodID and call the corresponding method in the skeleton
  - Skeleton: implements the method
- Generation of proxies, dispatchers and skeletons
  - IDL (RMI) compiler
- Dynamic invocation
  - Proxies are static—interface complied into client code
  - Dynamic—interface available during run time
    - Generic invocation; more info in “Interface Repository” (COBRA)
    - Dynamic loading of classes (Java RMI)

© Pearson Education 2005

RMI Software

This is a software layer that lies between the application and the communication and object reference modules. Following are the three main components.

- *Proxy:* Plays the role of a local object to the invoking object. There is a proxy for each remote object which is responsible for:
  - Marshalling the reference of the target object, its own method id and the arguments and forwarding them to the communication module.
  - Unmarshalling the results and forwarding them to the invoking object
- *Dispatcher:* There is one dispatcher for each remote object class. Is responsible for choosing appropriate method in the skeleton based on the method ID.
- *Skeleton:* Is responsible for: (server side)
  - Unmarshalling the arguments in the request and forwarding them to the servant.
  - Marshalling the results from the servant to be returned to the client.

© Pearson Education 2005

RMI Implementation (5):

- Server initialization: create the first object for remote access
  - Usually clients are not allowed to create servers
- Binder: is separate service that maintains a table containing mappings from textual names to remote object references.
  - Table mapping for names and remote object references
- Server threads: concurrency to avoid delaying the execution of another remote method invocation

© Pearson Education 2005

The binder

Client programs require a way to obtain the remote object reference of the remote objects in the server

- A **binder** is a service in a distributed system that supports this functionality
- A binder maintains a table containing mappings from textual names to object references.
- Servers register their remote objects (by name) with the binder. Clients look them up by name.

© Pearson Education 2005

Server and Client programs

A server program contains:

- classes for dispatchers and skeletons
- an *initialization* section for creating and initializing at least one of the servants
- code for registering some of the servants with the binder

A client program will contain the classes for all the proxies of remote objects

© Pearson Education 2005

RMI Implementation (6):

Activation of remote objects

- A remote object is *active* if it is available for invocation in the process.
- A remote object is *passive* if it is not currently active but can be made active. A passive object contains:
  - the implementation of the methods
  - its state in marshalled form

© Pearson Education 2005

RMI Implementation (7):

- persistent object stores
  - stored in marshaled form on disk for retrieval
  - saved those that were modified
  - persistent or not:
    - persistent root: any descendent objects are persistent (persistent Java, PerDiS)
    - some classes are declared persistent (Arjuna system)

© Pearson Education 2005

Assignment#1 (Chapter 5)

- 5.22

- Please note that Homework1 is available online and it will be due on Feb. 12 in class.

© Pearson Education 2005