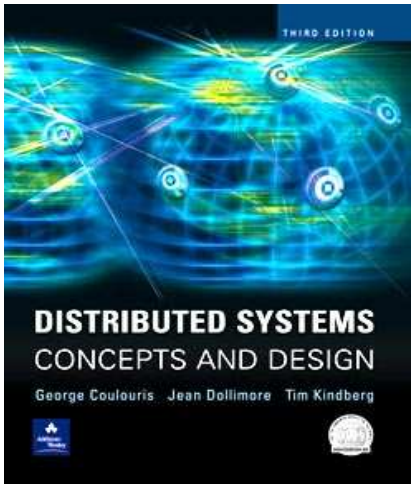
Chapter 5: Distributed Objects and Remote Invocation



From **Coulouris, Dollimore and Kindberg**

Distributed Systems: Concepts and Design

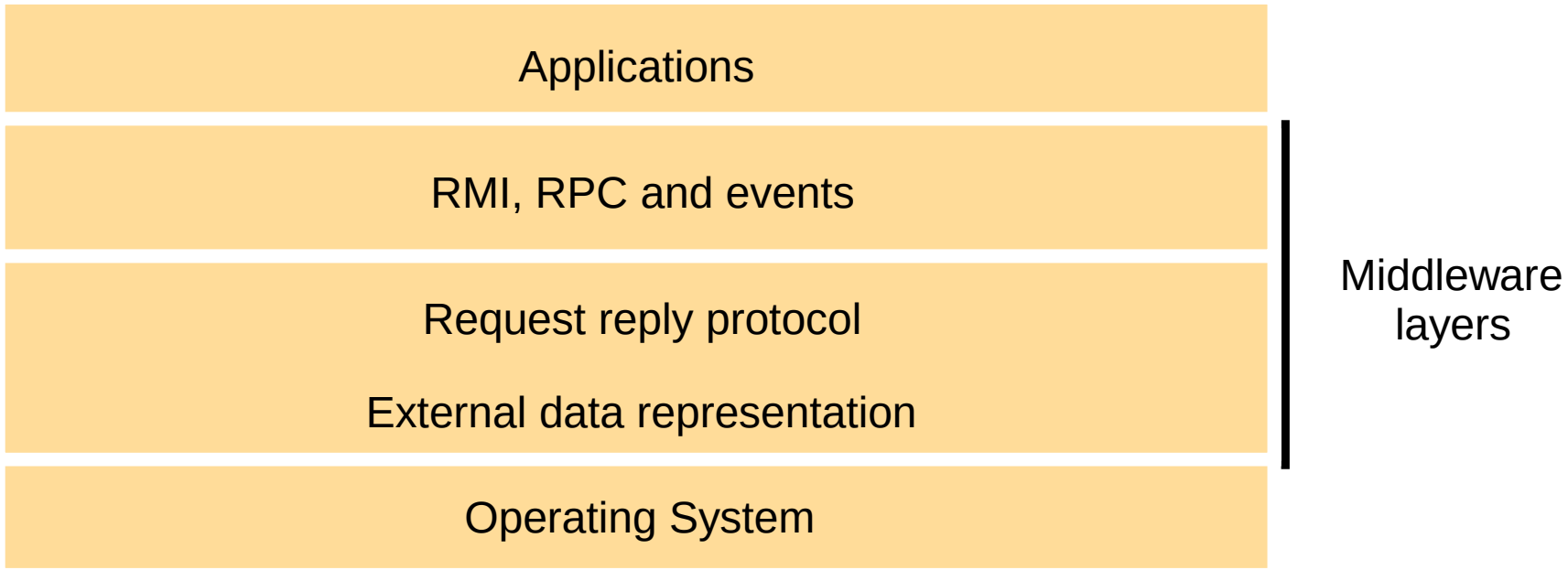
Programming Models for Distributed Application

- Remote procedure call – client calls the procedures in a server program that is running in a different process
- Remote method invocation (RMI) – an object in one process can invoke methods of objects in another process
- Event notification – objects receive notification of events at other objects for which they have registered
- These mechanism must be location-transparent.

Role of Middleware

- Middleware Roles
  - provide high-level abstractions such as RMI
  - enable location transparency
  - free from specifics of
    - communication protocols
    - operating systems and communication hardware
  - interoperability

Figure 5.1
Middleware layers



Interfaces

- Interface – skeleton of public class
  - Interaction specification
  - useful abstraction that removes dependencies on internal details
  - examples
    - procedure interface
    - class interface
    - module interface
- Distributed system interfaces
  - What is the main difference from single processor situation?
    - processes at different nodes
    - can only pass accessible information

Service Interface (RPC)

- Service interface
  - specifies set of procedures available to client
  - input and output parameters
  - Remote Procedure Call
    - arguments are marshaled
    - marshaled packet sent to server
    - server unmarshals packet, performs procedure, and sends marshaled return packet to client
    - client unmarshals the return
    - all the details are transparent

Remote Interface (RMI)

- Remote interface
  - specifies methods of an object available for remote invocation
  - input and output parameters may be objects
  - Remote Method Invocation
    - communication actual arguments marshaled and sent to server
    - server unmarshals packet, performs procedure, and sends marshaled return packet to caller
    - client unmarshals return packet
    - common format definition for how to pass objects (e.g., CORBA IDL or Java RMI)

Interface Definition Language

- Interface Definition Language
  - notation for language independent interfaces
    - specify type and
    - kind of parameters
  - examples
    - CORBA IDL for RMI
    - Sun XDR for RPC
    - DCOM IDL
  - IDL compiler allows interoperability

CORBA IDL Example

```
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();
};
```

Figure 5.2

CORBA has a struct

remote interface

remote interface defines methods for RMI

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

- Remote interface:
  - specifies the methods of an object available for remote invocation
  - an interface definition language (or IDL) is used to specify remote interfaces. E.g. the above in CORBA IDL.
  - Java RMI would have a class for *Person*, but CORBA has a *struct*

The Object Model

- An object encapsulates encapsulates both data and methods.
- Objects can be accessed via **object references**.
- An **interface** provides signatures of methods without implementation
- Actions are performed by **method invocations**.
  - The state of receiver may be changed.
  - Further invocations of methods on other objects may take place.

The Object Model

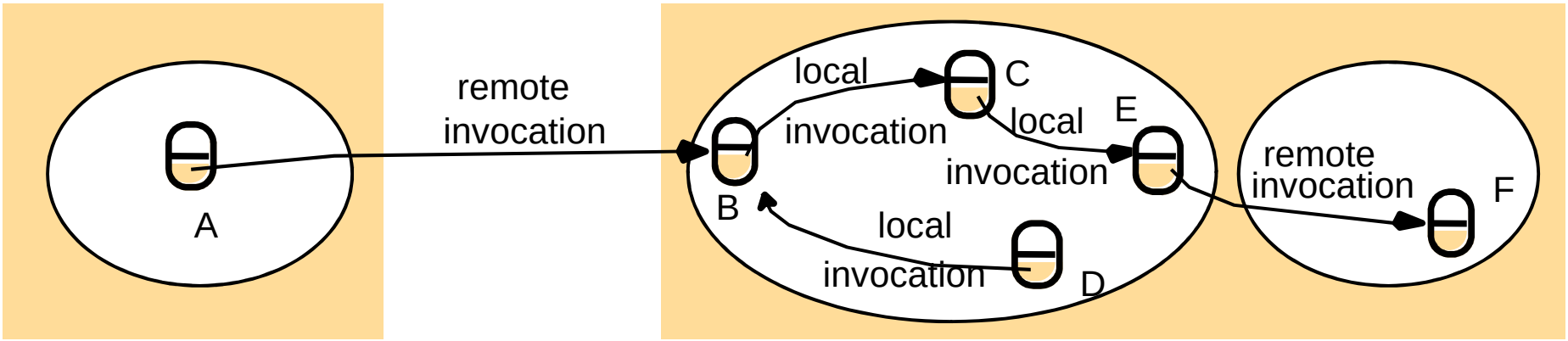
- **Exceptions** may be thrown to caller when an error occurs.
- **Garbage collection** frees the space occupied by objects when they are no longer needed.

The Distributed Objects Model

- Remote method invocation – Method invocations between objects in different processes, whether in the same computer of not.
- Local method invocation – Method invocations between objects in the same process.
- Remote object – Objects that can receive remote invocations.
- Remote and local method invocations are shown in Figure 5.3.

Distributed Object Model

Figure 5.3



- each process contains objects, some of which can receive remote invocations, others only local invocations
- those that can receive remote invocations are called *remote objects*
- objects need to know the *remote object reference* of an object in another process in order to invoke its methods. How do they get it?
- the *remote interface* specifies which methods can be invoked remotely

Distributed Object Model

- Remote object reference
  - An object must have the remote object reference of an object in order to do remote invocation of an object
  - Remote object references may be passed as input arguments or returned as output arguments
- Remote interface
  - Objects in other processes can invoke only the methods that belong to its remote interface (Figure 5.4).
  - CORBA – uses IDL to specify remote interface
  - JAVA – extends interface by the **Remote** keyword.

Remote object reference

- Construct unique remote object reference
  - IP address, port, interface name
  - time of creation, local object number (new for each object)
- Use in the same way as local object references
- If used as address

  ➔cannot support relocation

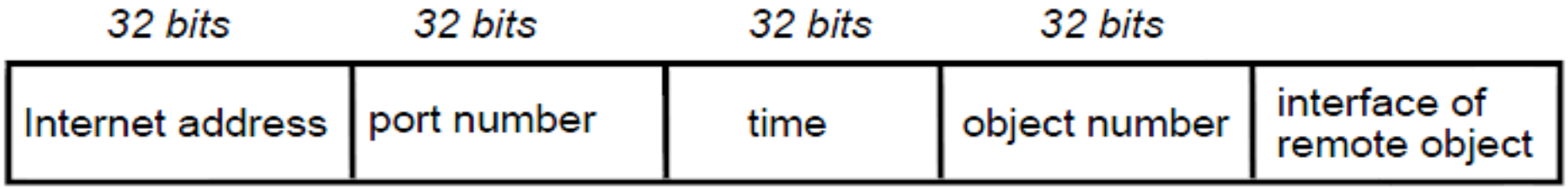
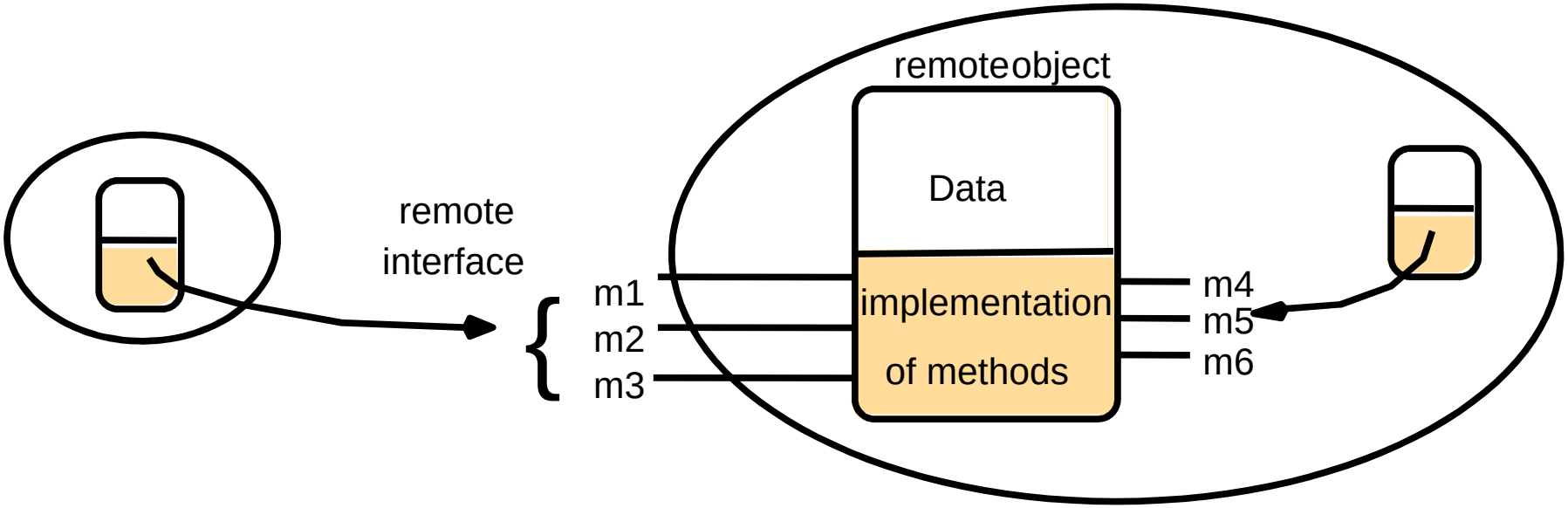


Figure 5.4
A remote object and its remote interface



Design Issues for RMI

- Two important issues in making RMI natural extension of local method: (These problems won't occur in the local invocation.)
  - **Number of times of invocations** are invoked in response to a single remote invocation
  - **Level of location transparency**
- **Exactly once invocation semantics** - Every method is executed exactly once. (Ideal situation)

Invocation Semantics Properties

- **Exactly once invocation semantics** - Every method is executed exactly once. (Ideal situation).

Invocation Semantics Properties

- **Maybe invocation semantics**:
  - The invoker can not determine whether or not the remote method has been executed.
  - Types of failures:
    - Omission failures if the invocation or result message is lost.
    - Crash failures when the server containing the remote object fails.
  - Useful for applications where occasional failed invocation are acceptable.

Invocation Semantics Properties

- **At-least-once invocation semantics**:
  - The invoker either receives a result (in which case the user knows the method was executed **at least once**) or an exception.
  - Types of failures:
    - Retransmitting request masks omission failures.
    - Crash failures when the server containing the remote object fails.
    - Arbitrary failure when the remote method is invoked more than once, wrong values are stored or returned. An **idempotent method** : the result of a successful performed request is independent of the number of times it is executed.
  - Useful if the objects in a server can be designed to have idempotent operations.

Invocation Semantics Properties

- **At-most-once invocation semantics**:
  - The invoker either receives a result (and the user knows the the method was executed exactly **at most once**) or an exception.
  - All fault tolerance methods executed.
  - Omission failures can be eliminated by retransmitting request.
  - Arbitrary failures can be prevented by ensuring that no method is executed more than once.
- JAVA RMI and CORBA use at-most-once semantics. CORBA also maybe semantics for methods that do not return results. SUNRPC provides at-least-once semantics.

Invocation Semantics

- **To provide a more reliable request-reply protocol, these fault-tolerant measures can be employed:**
  - **Retry request message**: whether to retransmit the request message until either a reply is received or the server is assumed to have failed.
  - **Duplicate filtering**: when retransmissions are used, whether to filter out duplicate requests at the server.
  - **Retransmission of results**: whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

Invocation Semantics

- Combinations of these measures lead to a variety of possible semantics for the reliability of remote invocations.
- Figure 5.5 shows the measures, with corresponding names for the invocation semantics that they produce.

Invocation Semantics

- Local invocations are executed exactly once
- Remote invocations cannot achieve this. Why not? :
  - The Request-reply protocol can apply fault-tolerance measure.

Figure 5.5 *Fault tolerance measures*

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

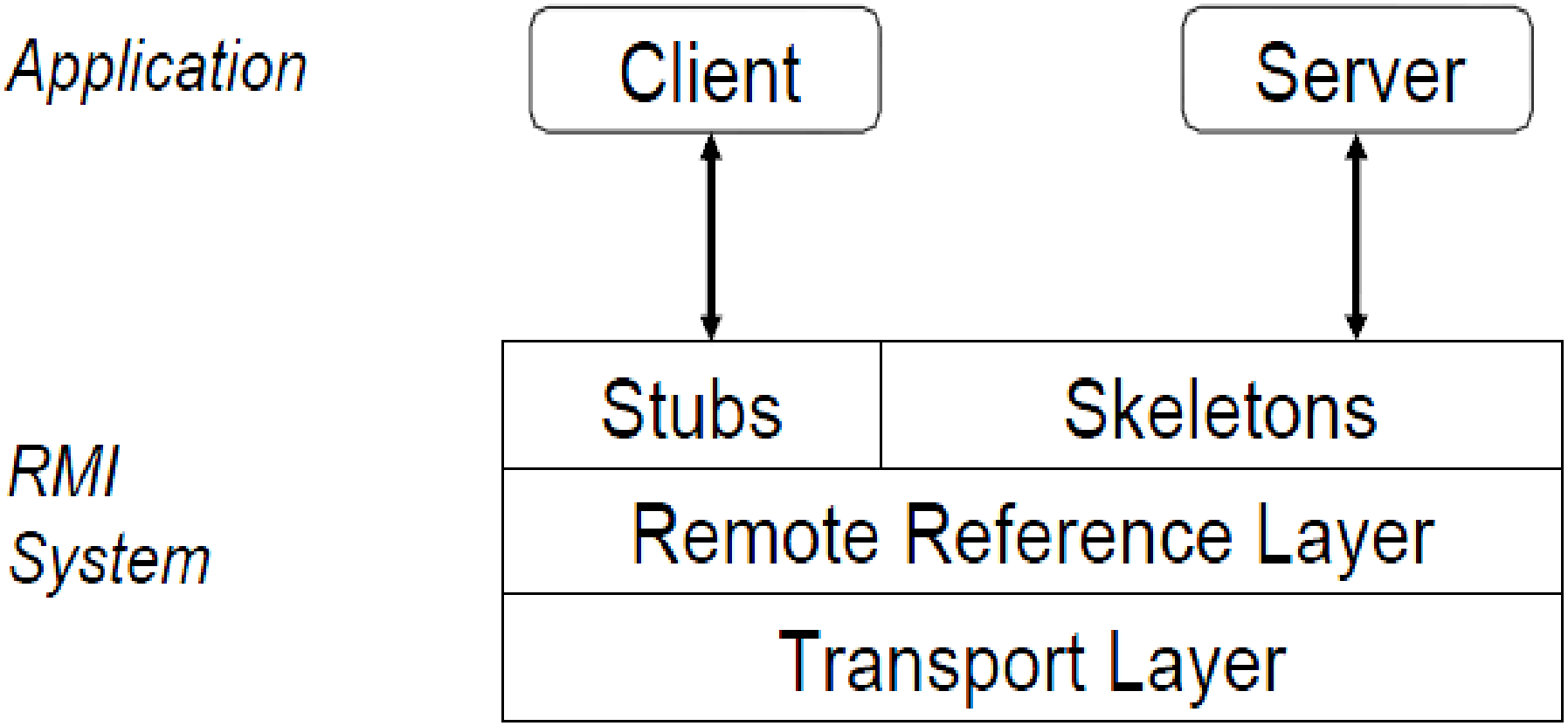
(Location) Transparency Issues

- Goal is to make a remote invocation as similar as possible a local invocation.
- Issues (Differences from the local invocation)
  - Syntax may be made identical but behavioral differences exists. The cause could be **failure** and **latency**.
  - Interface specification
    - Exceptions and exception handling are needed.
    - Different invocation semantics can be employed.

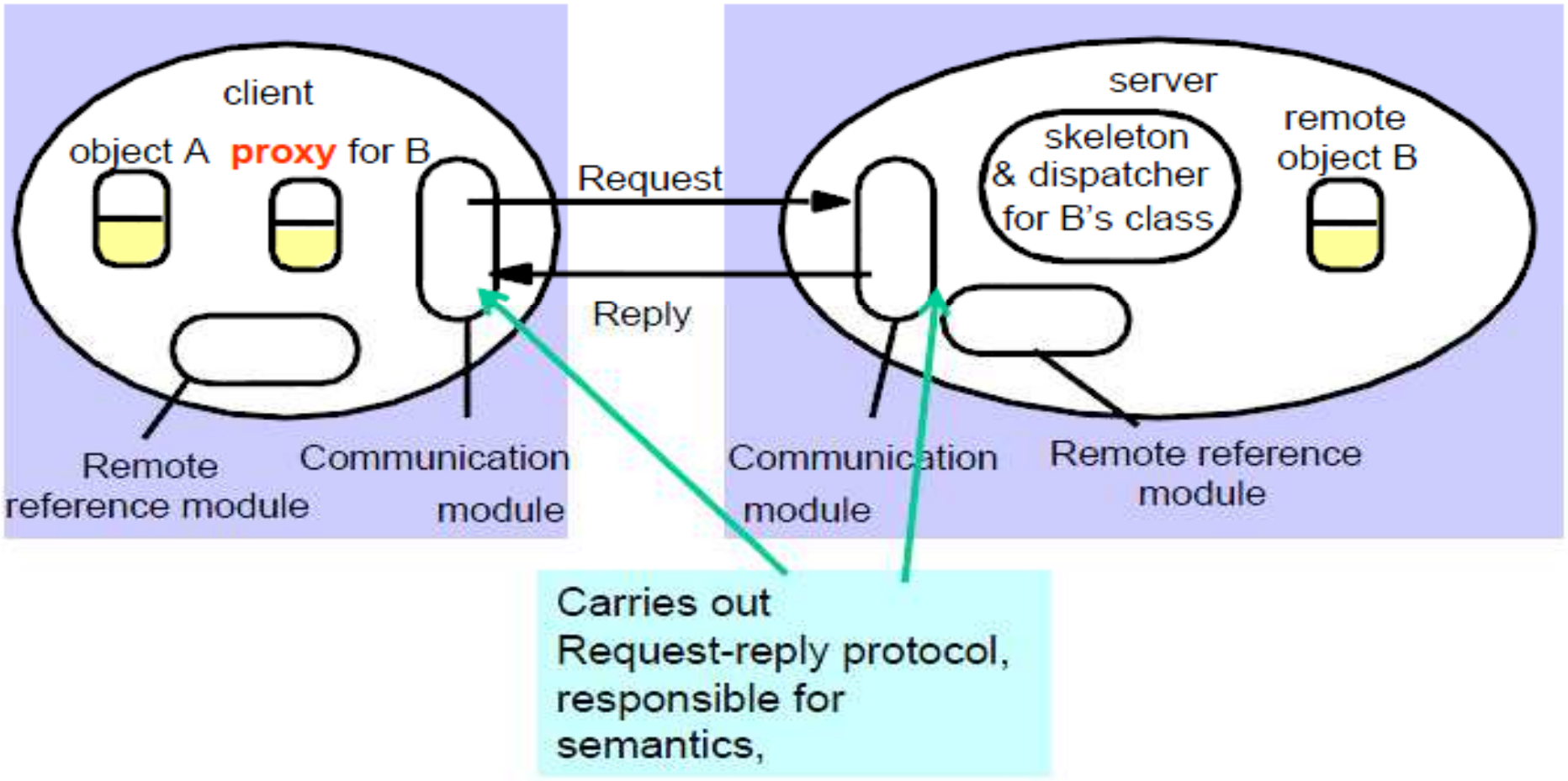
Implementation of RMI

- Figure 5.6 shows an object A invokes a method in a remote object B.
- Communication module:
  - Request-Reply Protocol
  - Responsible for providing selected invocation semantics
  - The server selects the dispatcher for the class of the object to be invoked.

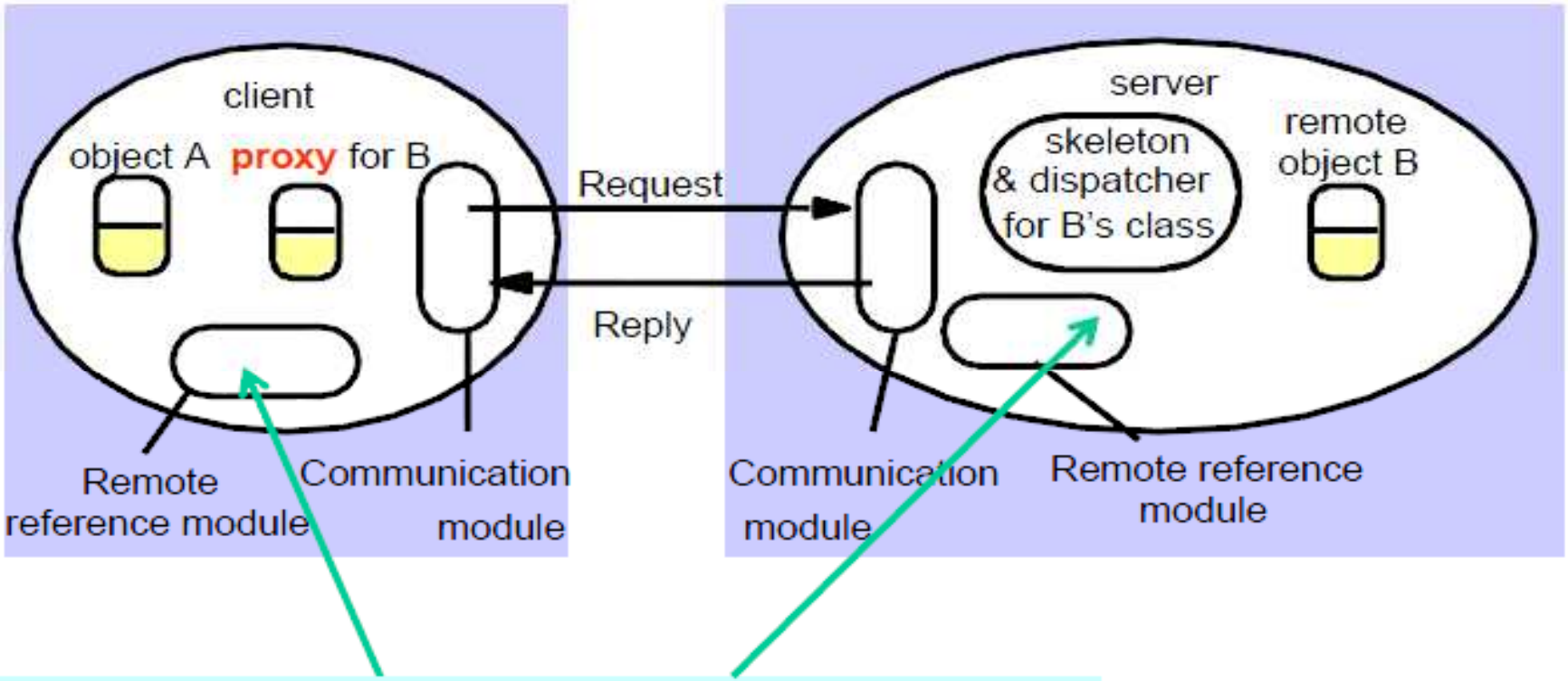
RMI Architecture



RMI Implementation

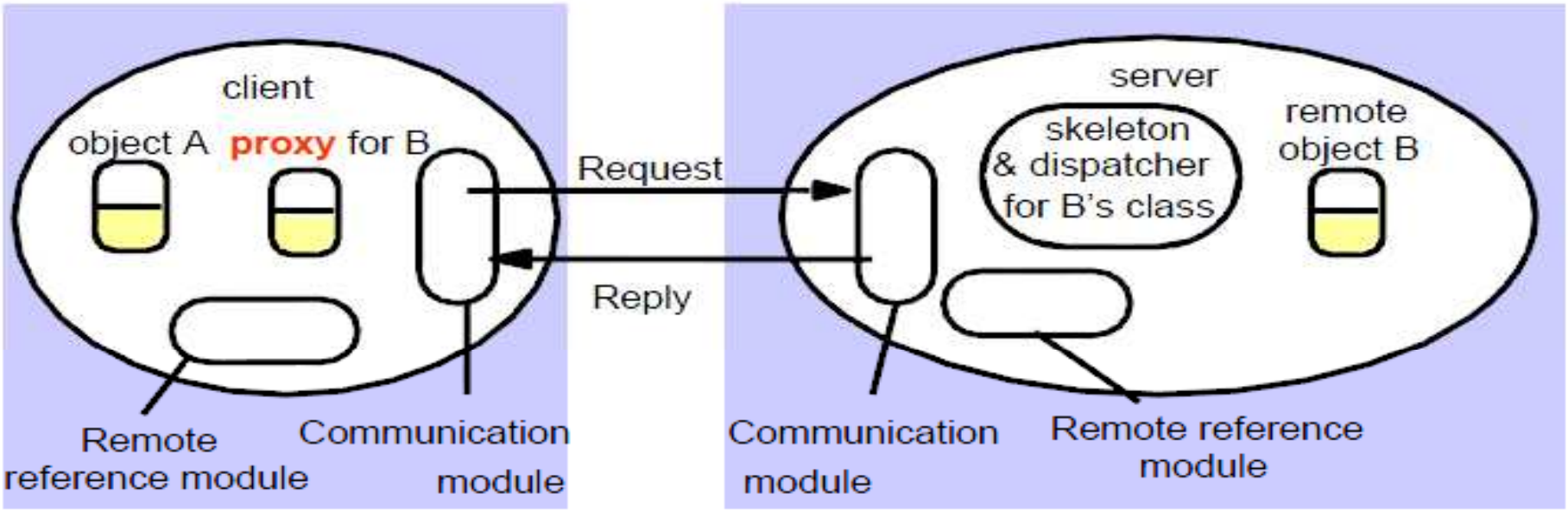


RMI Implementation



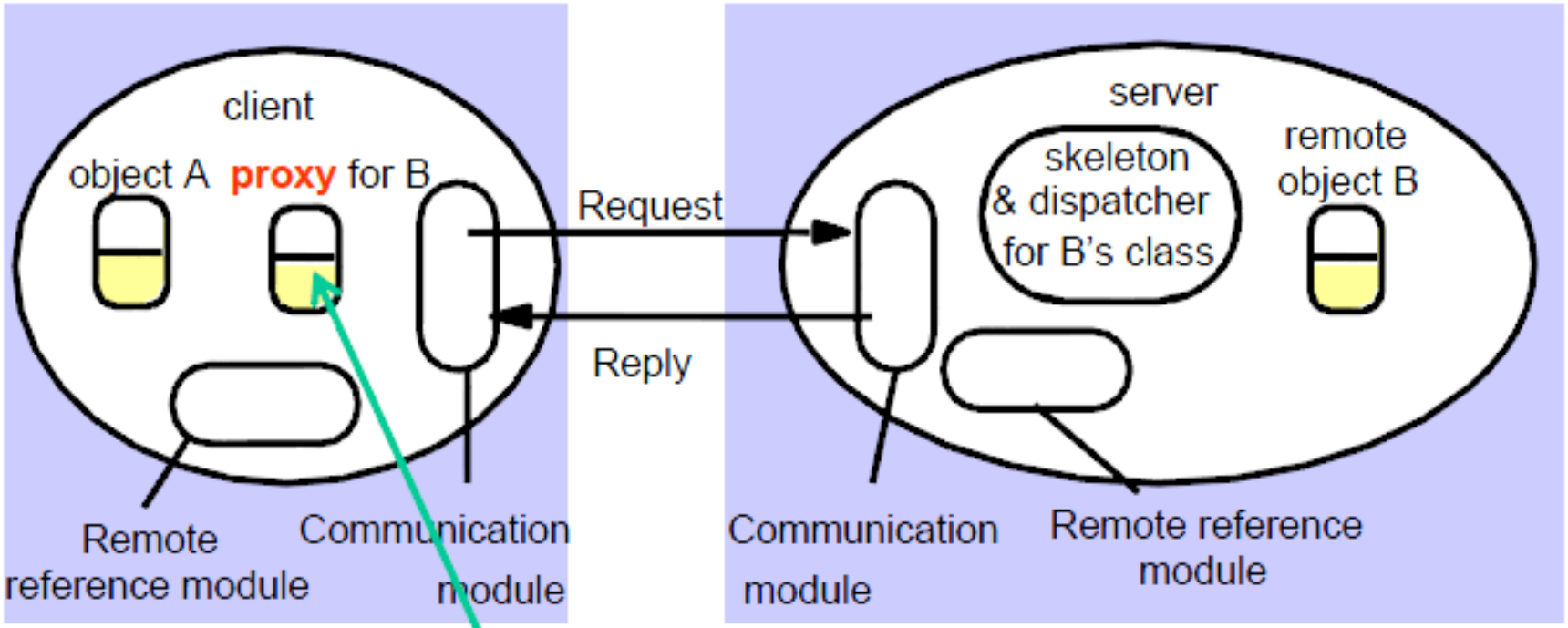
Translates between local and remote object references,
creates remote object references.
Uses remote object table
(relating remote and local objects references, plus proxies)

RMI Implementation



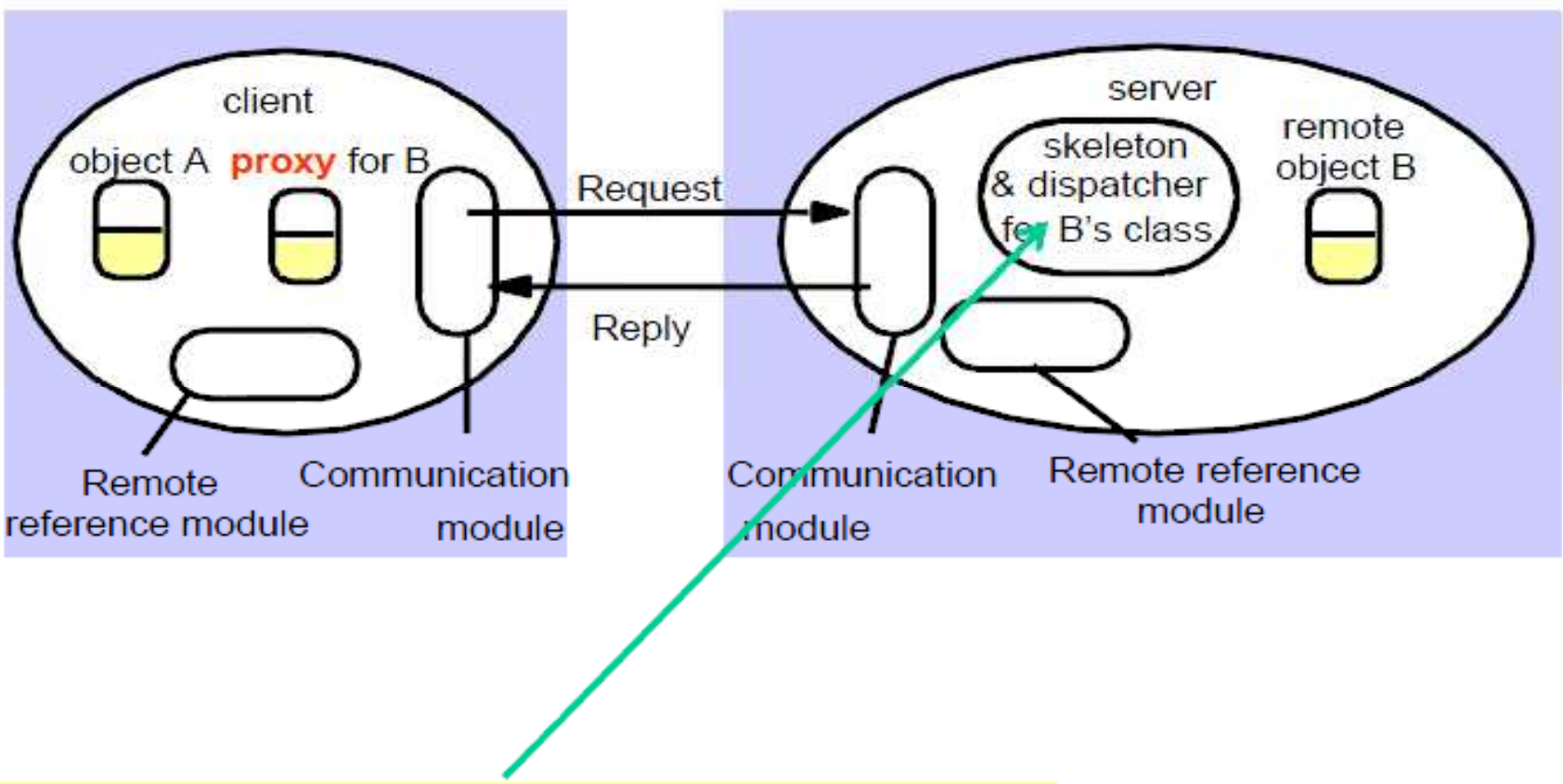
RMI software - between application level objects and communication and remote reference modules
(according to JRMP v1.1)

RMI Implementation



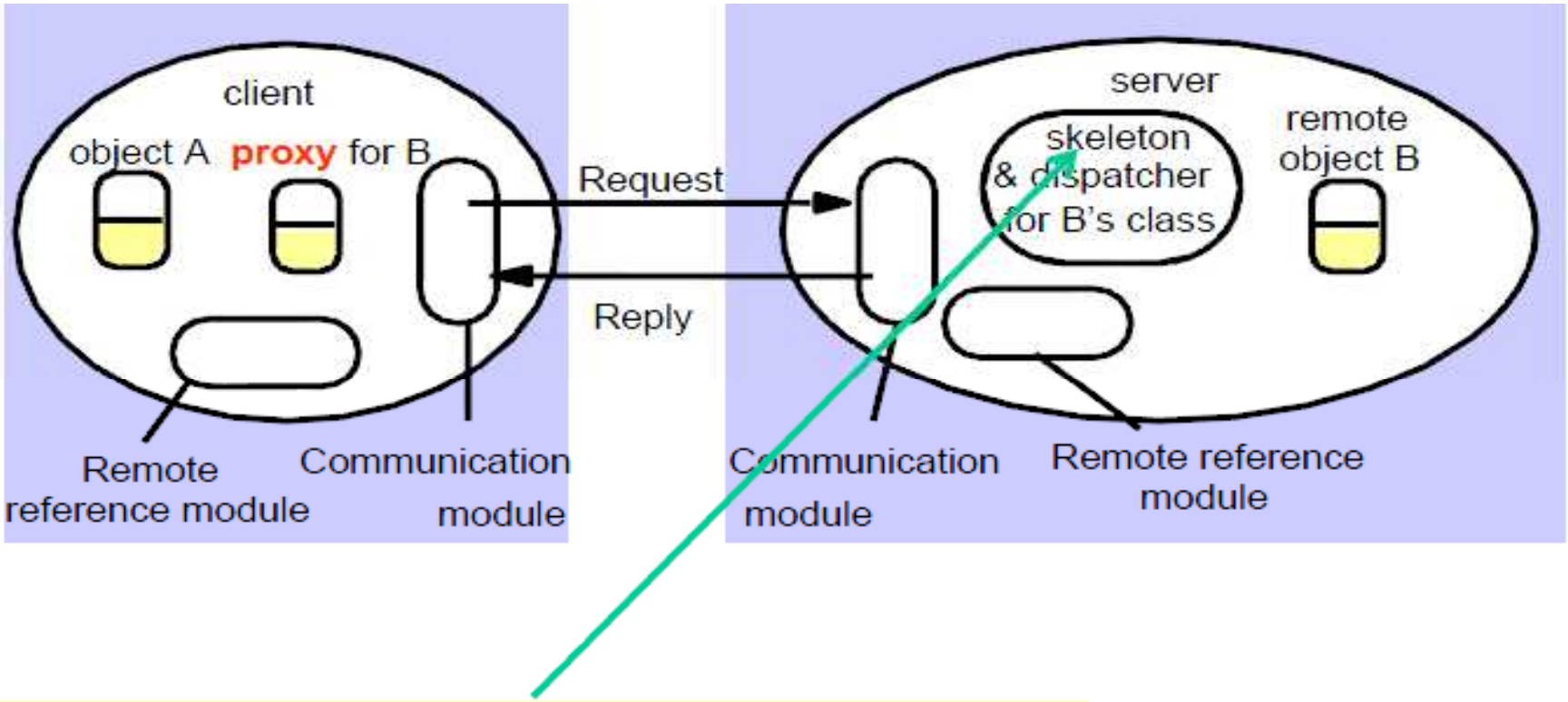
Proxy - makes RMI transparent to client. Class implements remote interface. Marshals requests and unmarshals results. Forwards request.

RMI Implementation



Dispatcher - gets request from communication module and invokes method in skeleton (using *methodID* in message).

RMI Implementation



Skeleton - implements methods in remote interface. Unmarshals requests and marshals results. Invokes method in remote object.

Remote Reference Module

- Responsibilities
  - translation between local and remote object references
    - remote object table
      - entry for each remote object held by process
      - entry for each local proxy
    - arriving remote object reference - creation of remote object references
  - creation of remote object references
    - need to pass a remote object - look up in remote object table (create new remote object reference and add entry if necessary)

RMI Software

- **Proxy** - provides remote invocation transparency
  - marshal arguments, unmarshal results, send and receive messages
- **Dispatcher** - handles transfer of requests to correct method
  - receive requests, select correct method, and pass on request message
- **Skeleton** - implements methods of remote interface
  - unmarshal arguments from request, invoke the method of the remote object, and marshal the results

RMI Server and Client Programs

- **Server**
  - classes for dispatchers, skeletons and remote objects
  - initialization section for creating some remote objects
  - registration of remote objects with the binder
- **Client**
  - classes for proxies of all remote objects
  - binder to look up remote object references
  - cannot create remote objects by directly calling constructors - provide factory methods instead

RMI Binder and Server Threads

- A **binder** in a distributed system is a separate service that maintains a table containing mappings from textual names to remote object references.
- Server threads
  - sometimes implemented so that remote invocation causes a new thread to be created to handle the call
  - server with several remote objects might also allocate separate threads to handle each object

RMI Binder and Server Threads

- Activation of remote objects
  - A remote object is described as **active** when it is a running process.
  - A remote object is described as **passive** when it can be made active if requested.
- An object that can live between activations of processes is called a **persistent object**.
- A **location service** helps clients to locate remote objects from their remote references.

RMI Distributed Garbage Collection

- Aim - recover memory if no reference to an object exist. If there is a reference object should still exists.
- Java distributed algorithm - based on reference counting
- The distributed garbage collector works in cooperation with the local garbage collector.
  - Each server has a table (**B.holders**) that maintains list of references to an object.
  - When the client C first receives a reference to an object B, it invokes **addRef(B)** and then creates a proxy. The server adds C to the remote object holder **B.holders**.

RMI Distributed Garbage Collection

- The distributed garbage collection (continued):
  - When remote object B is no longer reachable, it deletes the proxy and invokes **removeRef(B)**.
  - When **B.holders** is empty, the server reclaim the space occupied by B.
- Leases in Jini
  - To avoid complicated protocols to discover whether a resource are still used, the resource is leased for use for a period of time.
  - An object representing a lease implements the **Lease** interface.

Thank You