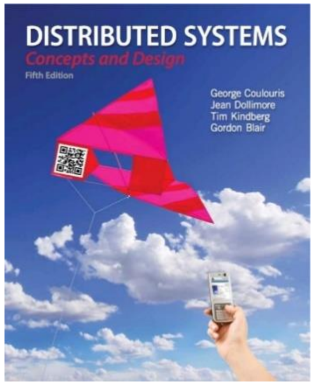
Distributed Objects Programming - Remote Invocation



Some concepts are drawn from Chapter 5

Rajkumar Buyya

Cloud Computing and **D**istributed **S**ystems (CLOUDS) Laboratory
School of Computing and Information Systems

The University of Melbourne, Australia
http://www.cloudbus.org/652

Co-contributors: Xingchen Chu, Rodrigo Calheiros, Maria Sossa, Shashikant Ilager

Sun Java online tutorials:

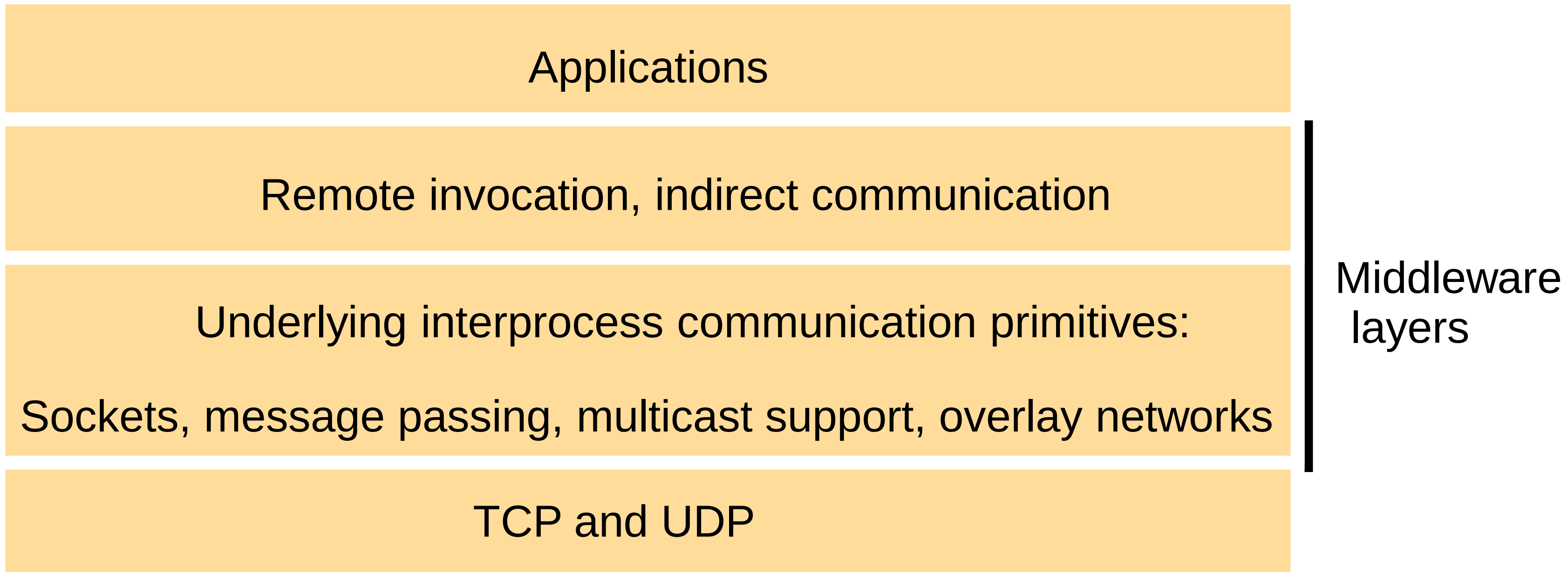
http://java.sun.com/docs/books/tutorial/rmi/

Outline

- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

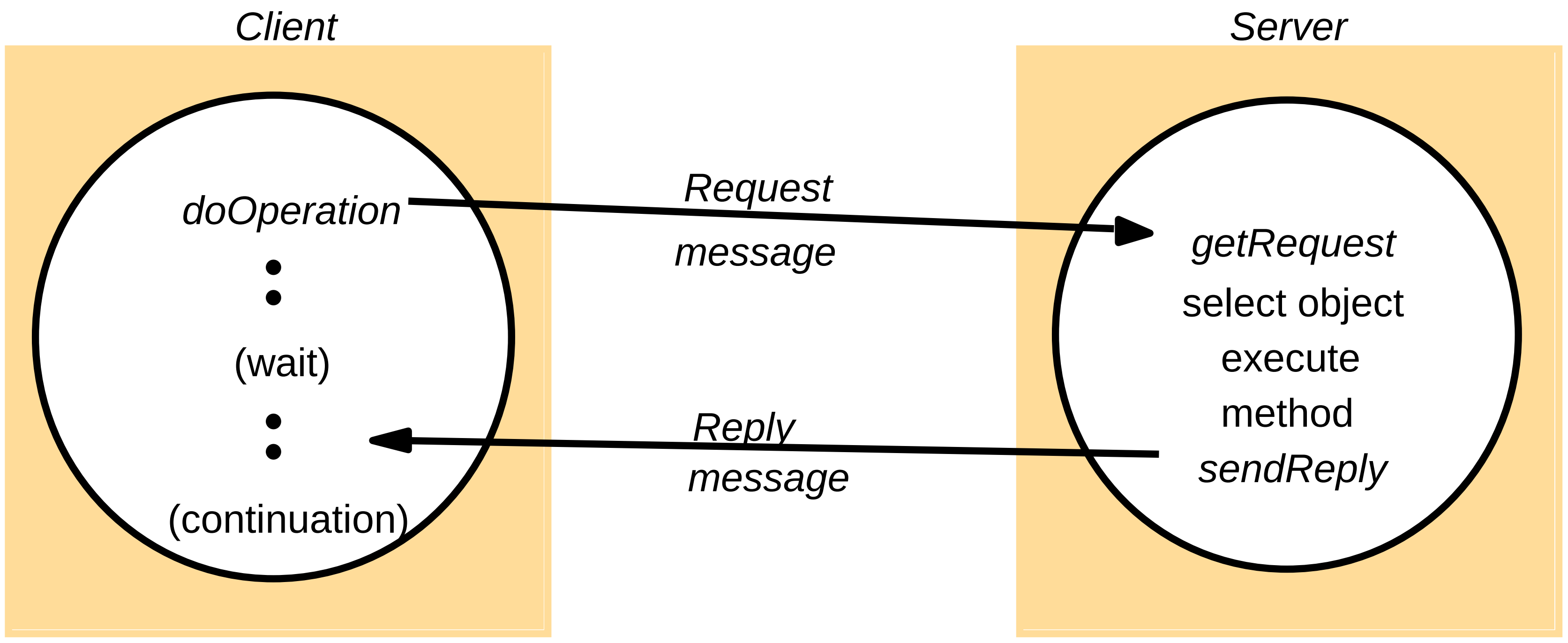
Introduction

- We cover high-level programming models for distributed systems. Two widely used models are:
  - *Remote Procedure Call (RPC)* - an extension of the conventional procedure call model
  - *Remote Method Invocation (RMI)* - an extension of the object-oriented programming model.


Applications
Remote invocation, indirect communication
Underlying interprocess communication primitives:
Sockets, message passing, multicast support, overlay networks
TCP and UDP
Middleware layers

Request-Reply Protocol

- Exchange protocol for the implementation of remote invocation in a distributed system.
- We discuss the protocol based on three abstract operations: doOperation, getRequest and sendReply



Request-Reply Operations

- **public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)**
  - Sends a request message to the remote server and returns the reply
  - The arguments specify the remote server, the operation to be invoked and the arguments of that operation
  - Note: The caller of **doOperation is blocked** until the server performs the requested operation and transmits a reply message to the client process

Request-Reply Operations

- **public byte[] getRequest ()**
  - Acquires a client request via the server port
- **public void sendReply (byte[] reply, InetAddress clientHost, int clientPort)**
  - Sends the reply message reply to the client at its Internet address and port

Failure Model of the Request/Reply Protocol

- If the three primitives doOperation, getRequest and sendReply are implemented over UDP datagrams, then they suffer from:
  - Omission failures
  - Messages are not guaranteed to be delivered in sender order.
- To allow for occasions when a server has failed or a request or reply message is dropped, doOperation uses a **timeout** when it is waiting to get the server's reply message.

Failure Model of the Request/Reply Protocol
Timeouts

- What happens if a timeout occurs?
  - Return failure of doOperation to the client
  - Resend the request **repeatedly** to the server until a reply is received

Failure Model of the Request/Reply Protocol
Discarding Duplicate Messages

- What happens if a request is repeatedly resent?
  - In cases when the request message is retransmitted, the server may receive it more than once.
  - This can lead to the server executing an operation more than once for the same request
  - To avoid this, the protocol is designed to recognize successive messages (from the same client) with the same request identifier and to filter
out duplicates.

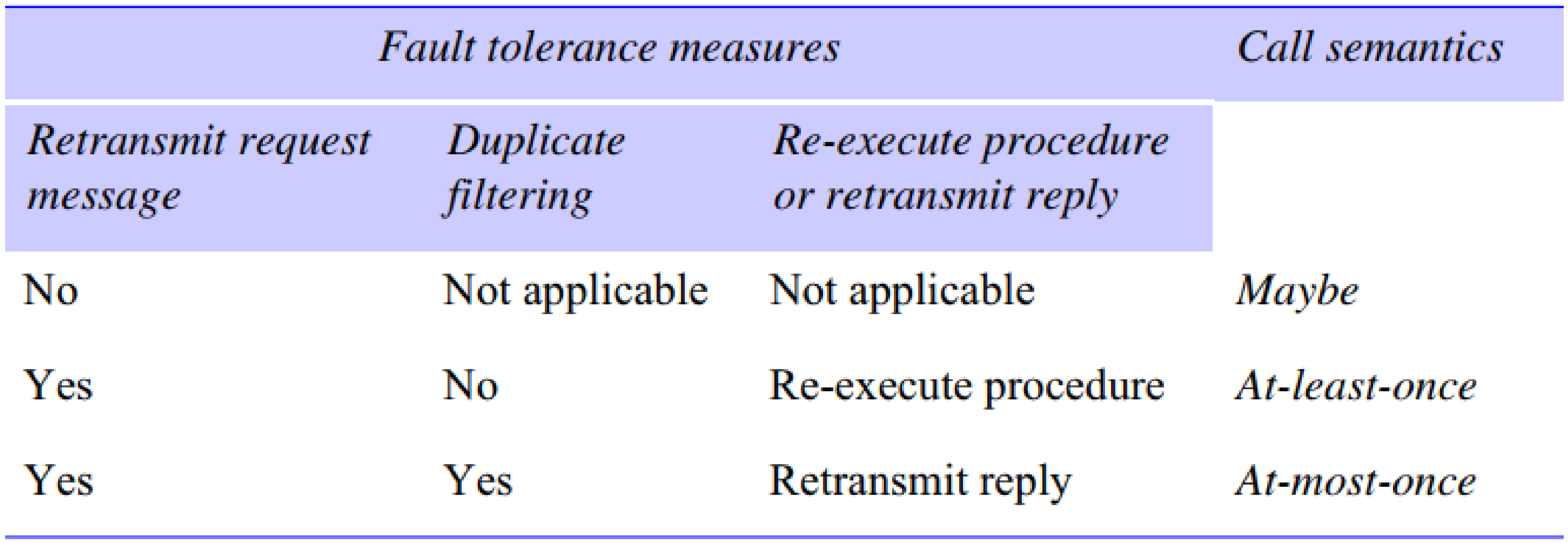
Failure Model of the Request/Reply Protocol
Lost Reply Messages

- What happens if a reply is lost?
  - If the server has already sent the reply when it receives a duplicate request it will need to execute the operation again to obtain the result
  - Some servers may re-execute the same operation again (**Idempotent operations?**)

RPC Call Semantics

- As 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** as seen by the invoker.

RPC Call Semantics



Fault tolerance measures			*Call 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*

What about local procedure calls semantics?

Invocation Semantics

- Middleware that implements remote invocation generally provides a certain level of semantics:
  - **Maybe**: The remote procedure call may be executed once or not at all. Unless the caller receives a result, it is unknown as to whether the remote procedure was called.
  - **At-least-once**: Either the remote procedure was executed at least once, and the caller received a response, or the caller received an exception to indicate the remote procedure was not executed at all.

Invocation Semantics

- Middleware that implements remote invocation generally provides a certain level of semantics:
  - **At-most-once**: The remote procedure call was either executed exactly once, in which case the caller received a response, or it was not executed at all and the caller receives an exception.
- Java RMI (Remote Method Invocation) supports at-most-once invocation.
  - It is supported in various editions including J2EE.
- Sun RPC (Remote Procedure Call) supports at-least-once semantics

Service Interface

- The term **service interface** is used to refer to the specification of the procedures offered by a server, defining the types of the arguments of each of the procedures.
- ***Interface definition languages* (IDLs)** are designed to allow procedures implemented in different languages to invoke one another.

Service Interface

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();
};
```

External Data Representation

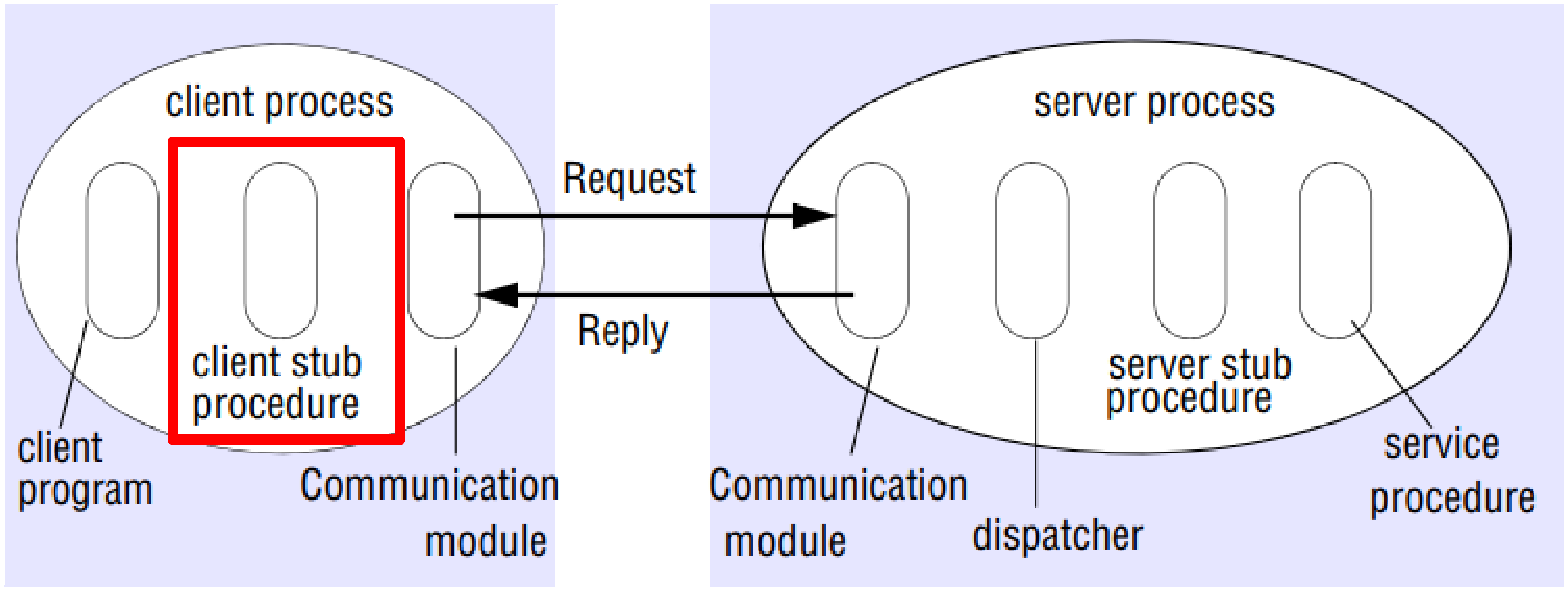
- The information stored in running programs is represented as data structures – for example, by sets of interconnected objects – whereas the information in messages consists of sequences of bytes.
- The individual primitive data items transmitted in messages can be data values of many different types, and not all computers store primitive values such as integers in the same order. (Recall Big Endian!)
- Another issue is the set of codes used to represent characters

Marshalling

- How can we enable any two computers to exchange binary data values then?
- How about marshalling?

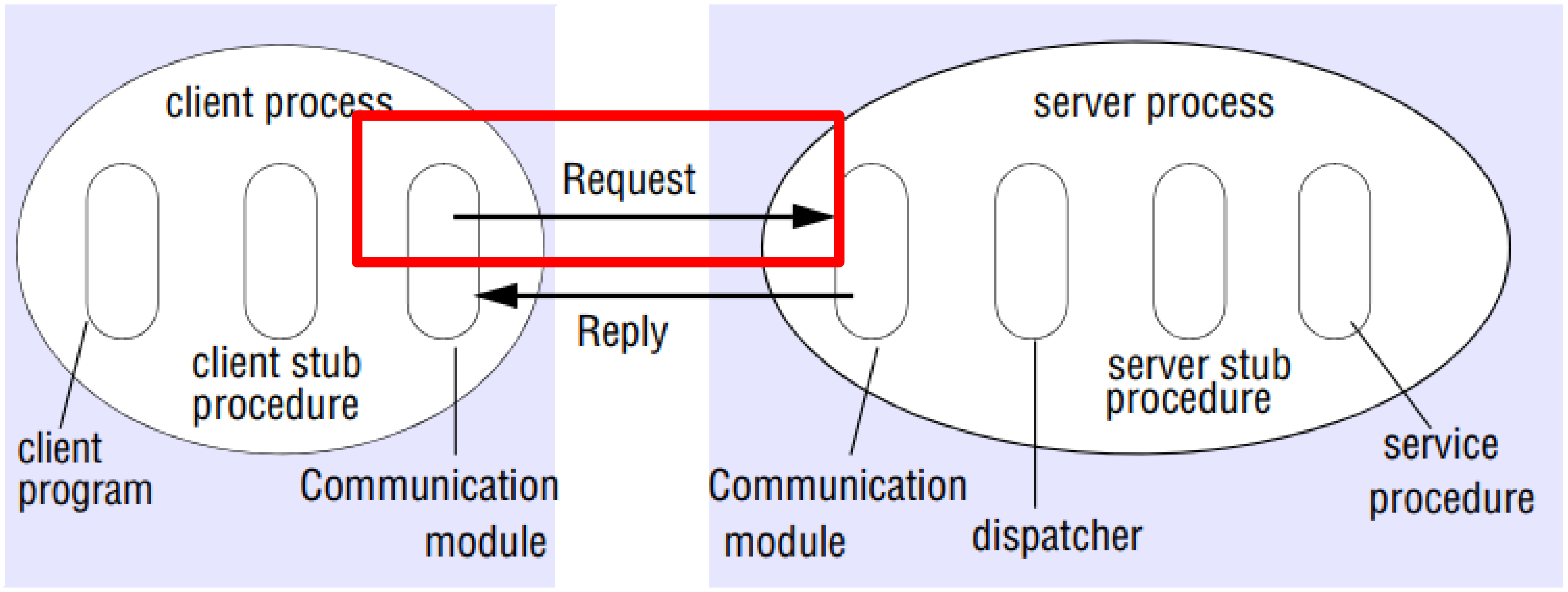
Implementation of RPC

Role of client and server stub procedures in RPC



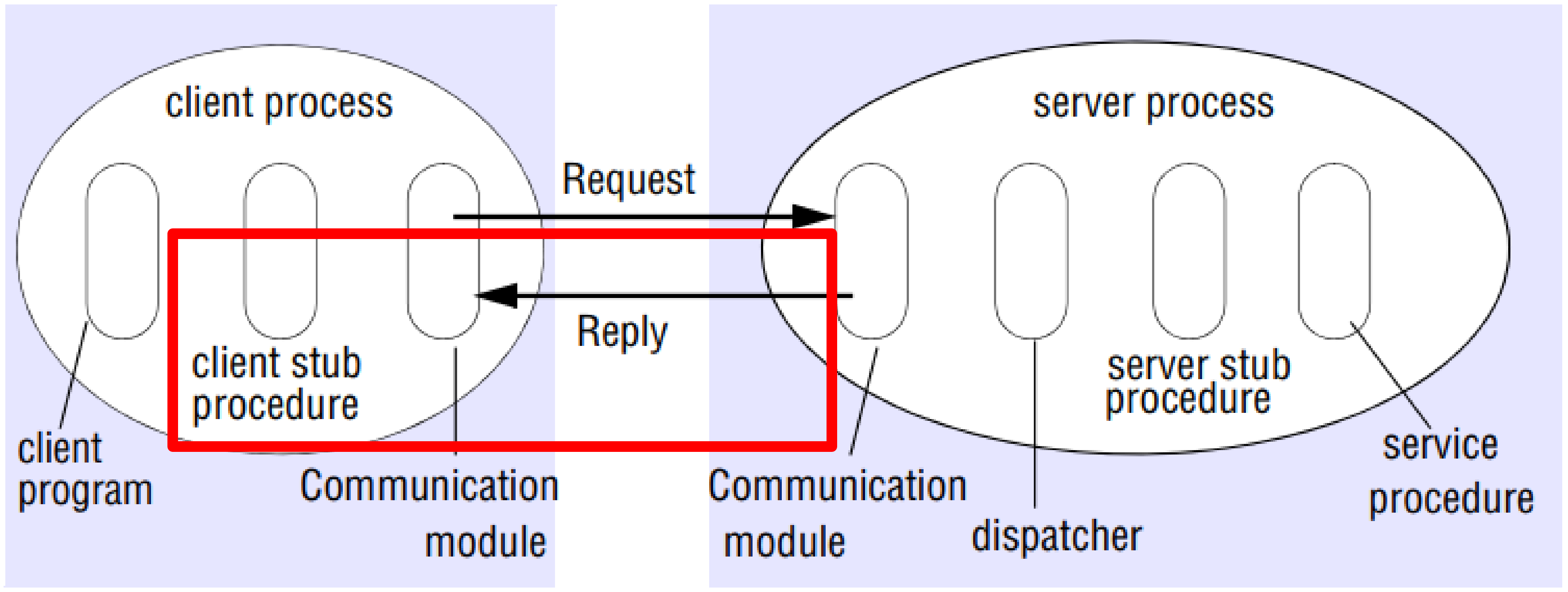
Implementation of RPC

Role of client and server stub procedures in RPC



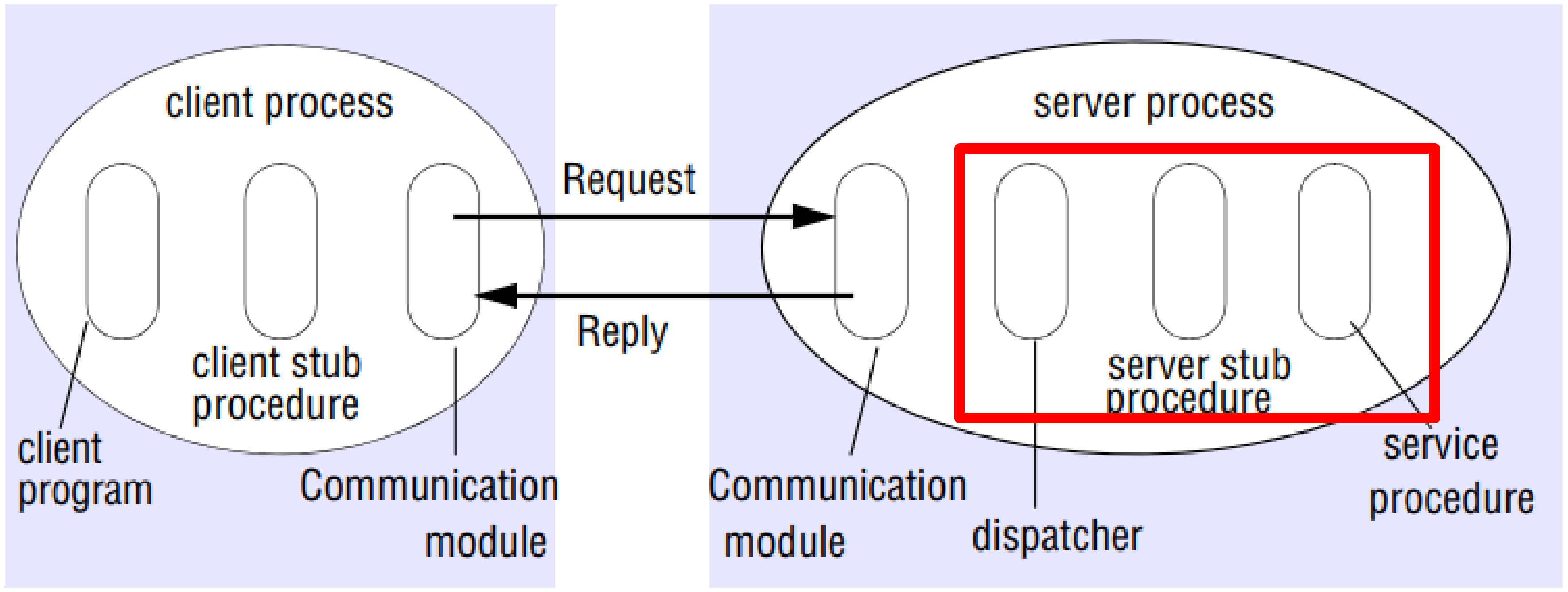
Implementation of RPC

Role of client and server stub procedures in RPC



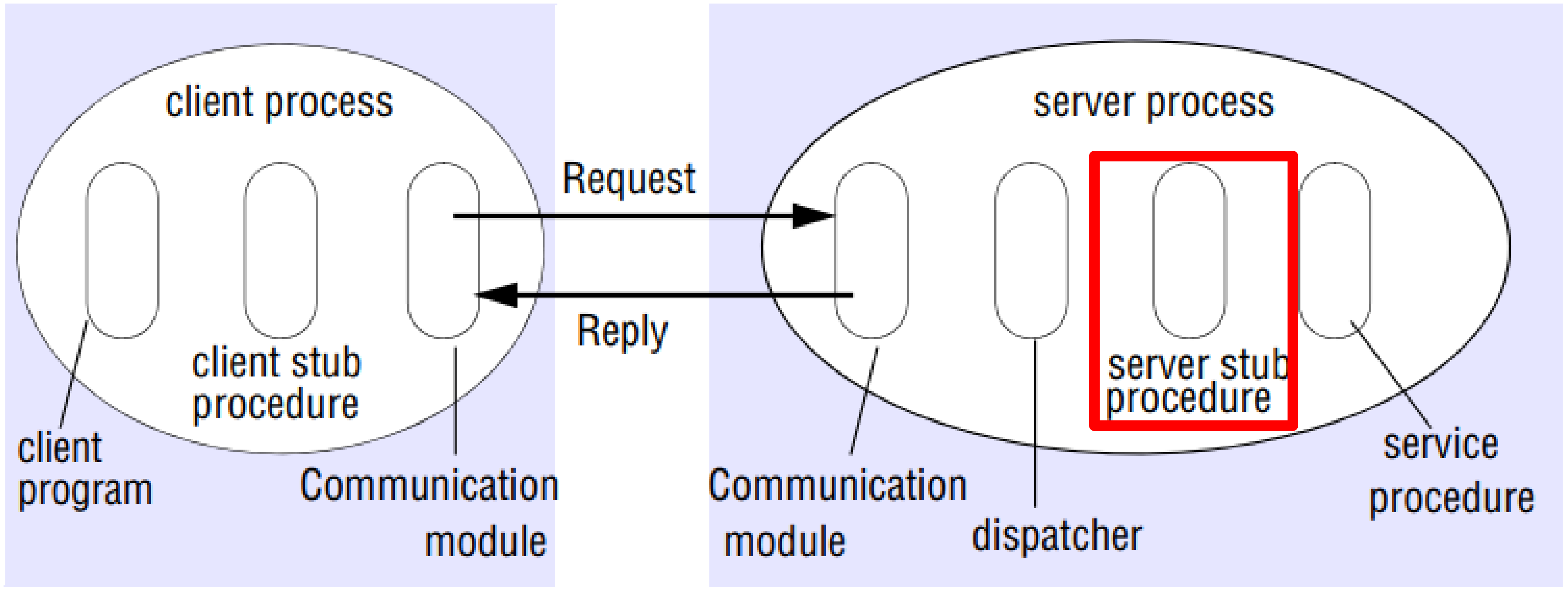
Implementation of RPC

Role of client and server stub procedures in RPC



Implementation of RPC

Role of client and server stub procedures in RPC



Distributed Objects

- A programming model based on Object-Oriented principles for distributed programming.
- Enables reuse of well-known programming abstractions (Objects, Interfaces, methods…), familiar languages (Java, C++, C#...), and design principles and tools (design patterns, UML…)
- Each process contains a collection of objects, some of which can receive both remote and local invocations:
  - Method invocations between objects in *different processes* are known as **remote method invocation,** *regardless the processes run in the same or different machines*.
- Distributed objects may adopt a client-server architecture, but other architectural models can be applied as well.

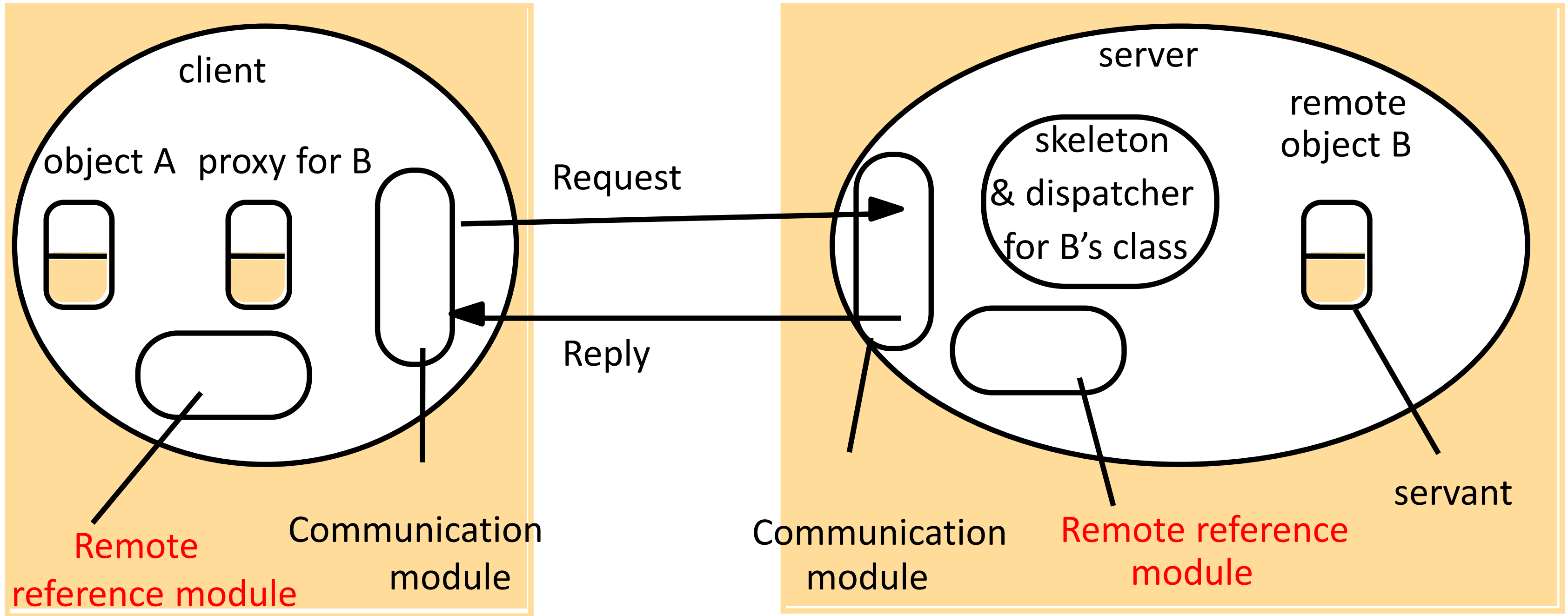
Outline

Java RMI

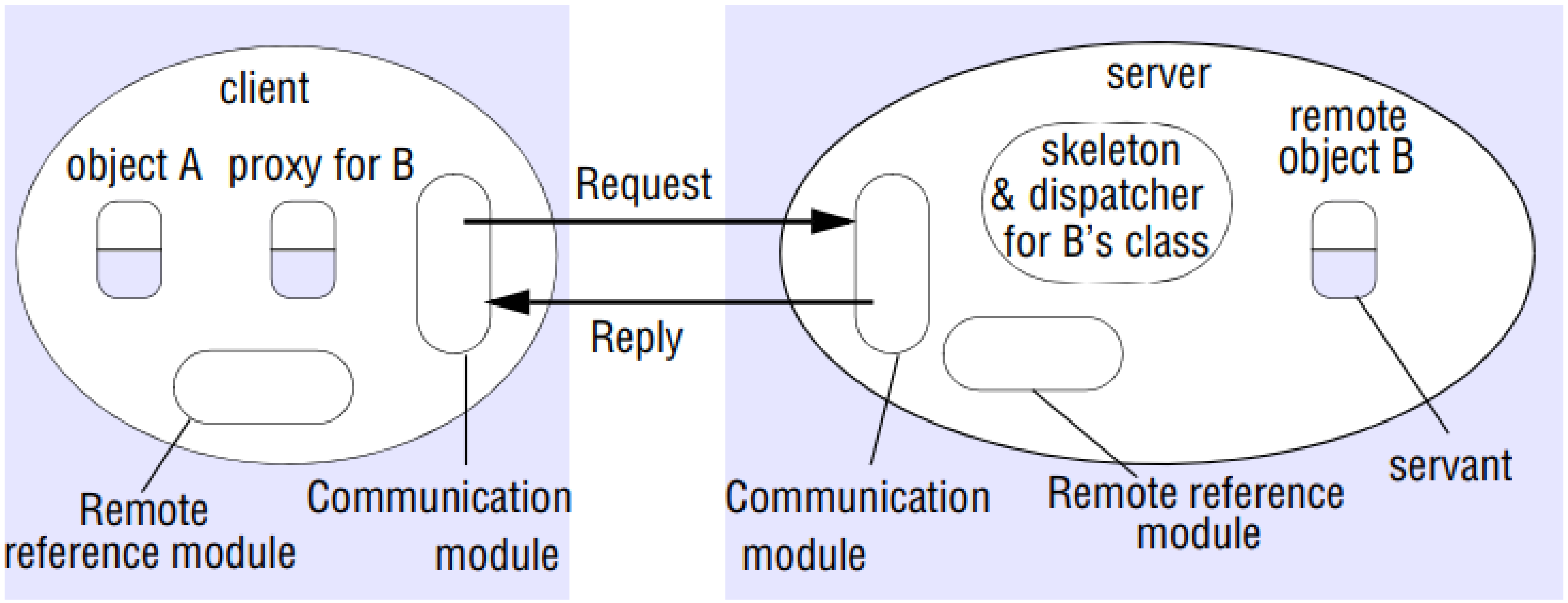
- Java Remote Method Invocation (Java RMI) is an extension of the Java object model to support distributed objects
  - methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts
- RMI uses object serialization to marshal and unmarshal
  - Any serializable object can be used as parameter or method return
- RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference.
- **Releases of Java RMI**
  - Java RMI is available for Java Standard Edition (JSE), Java Micro Edition (JME), and Java Enterprise Edition (Java EE)

RMI Architecture and Components

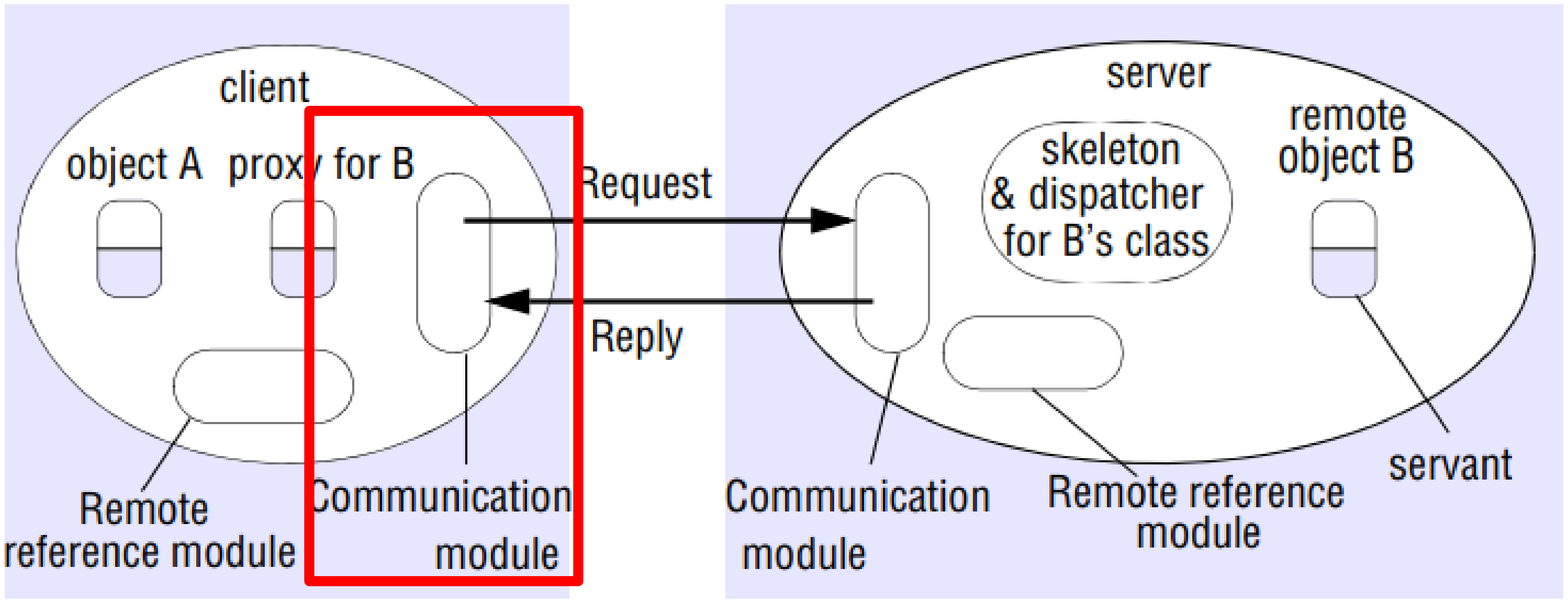
- Remote reference module (at client & server) is responsible for providing addressing to the proxy (stub) object
- Proxy is used to implement a stub and provide transparency to the client. It is invoked directly by the client (as if the proxy itself was the remote object), and then marshal the invocation into a request
- Communication module is responsible for networking
- Dispatcher selects the proper skeleton and forward message to it
- Skeleton un-marshals the request and calls the remote object



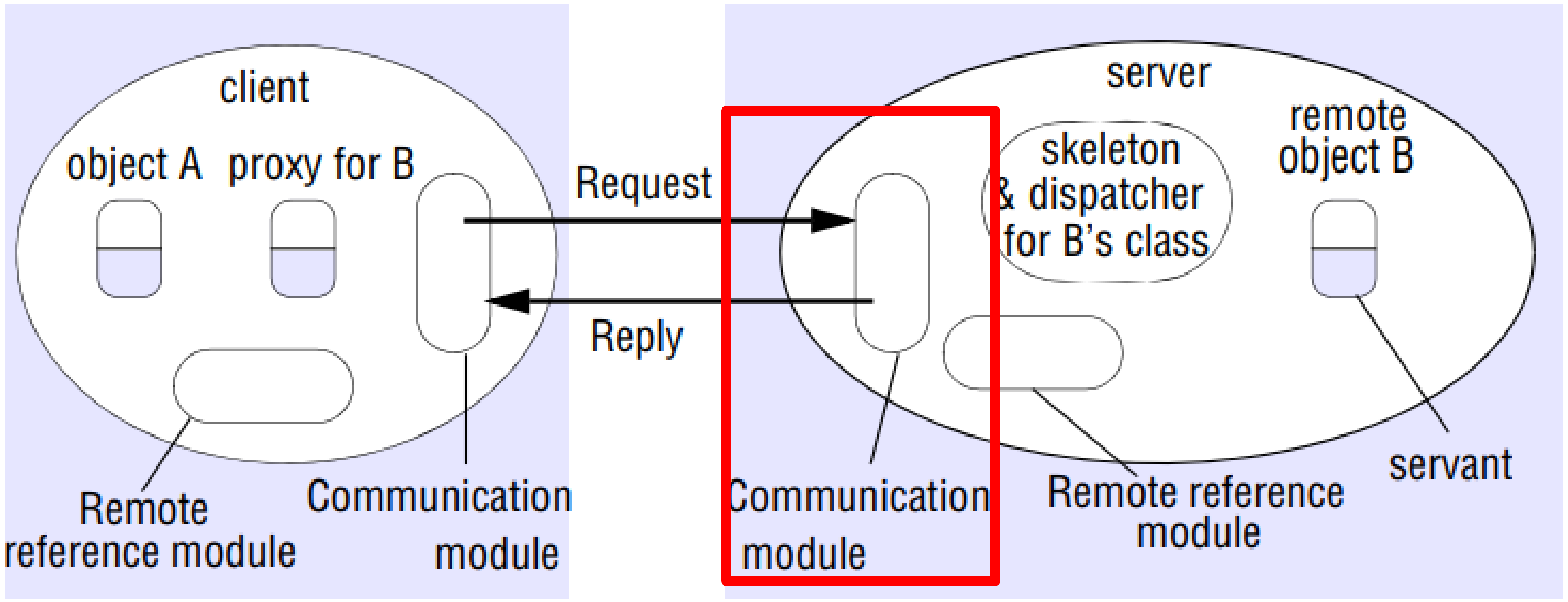
RMI Architecture and Components



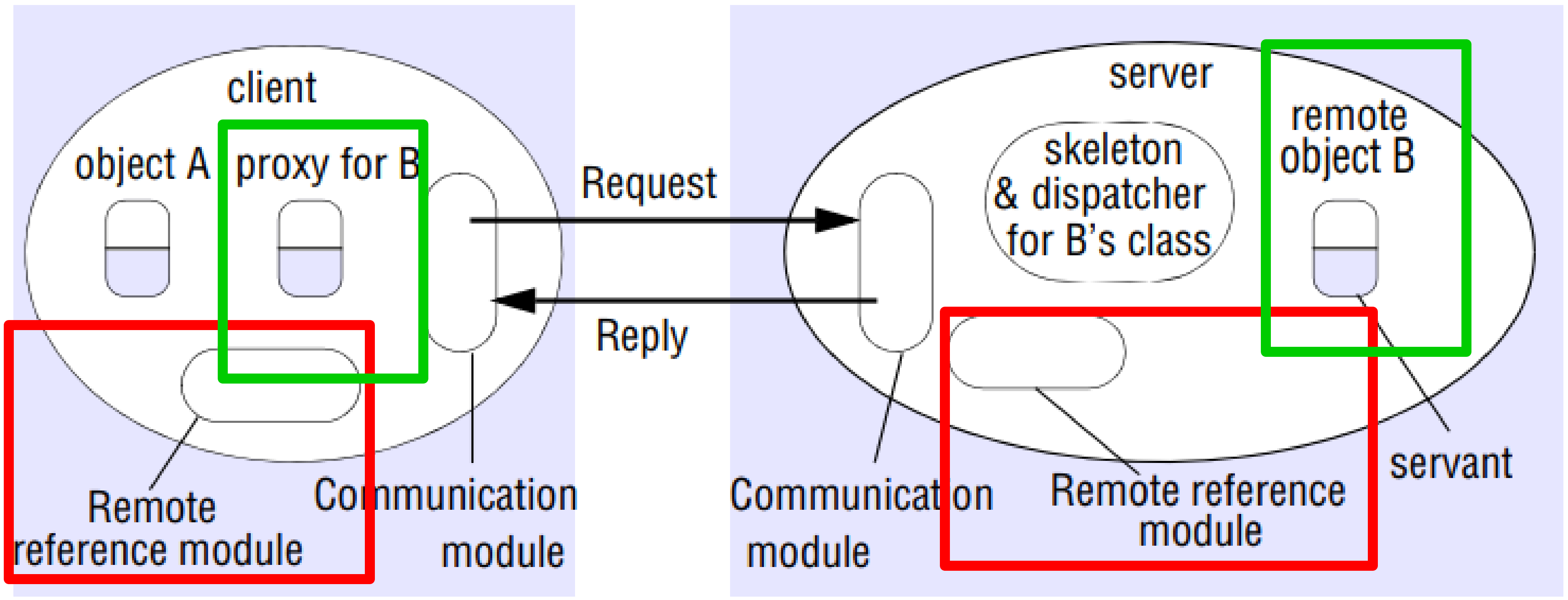
RMI Architecture and Components



RMI Architecture and Components

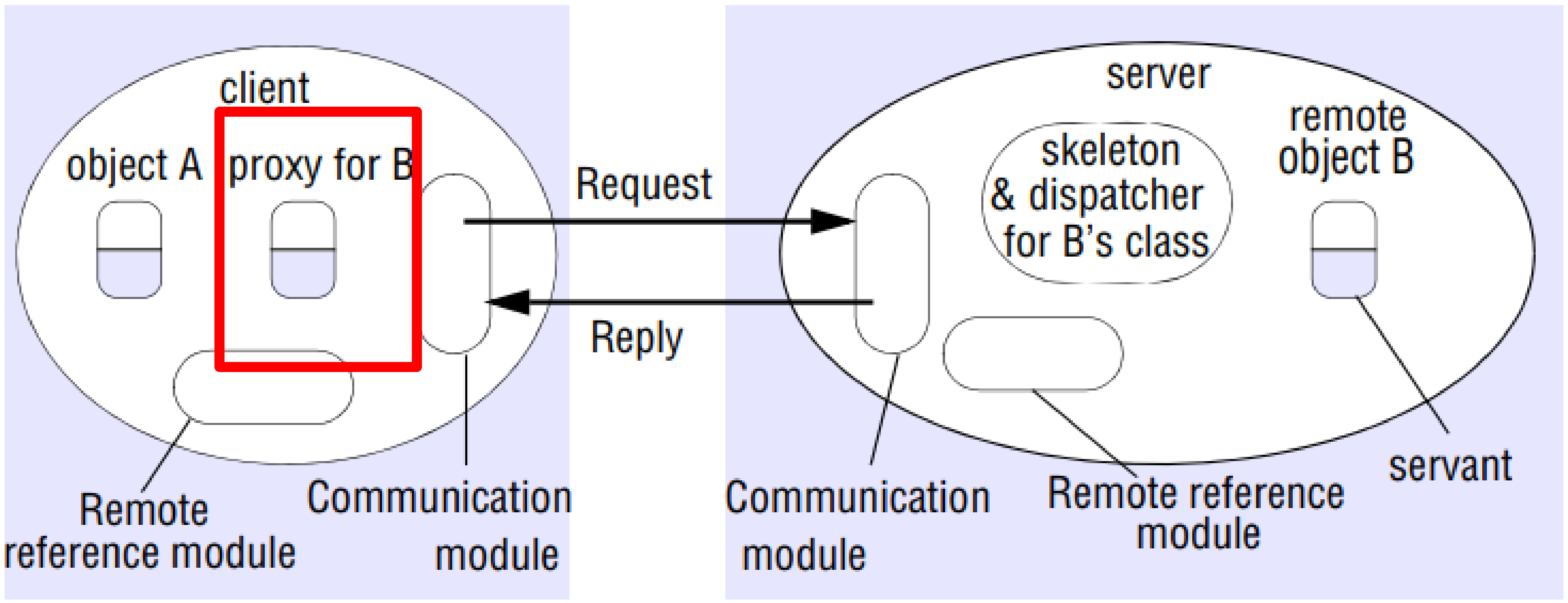


RMI Architecture and Components

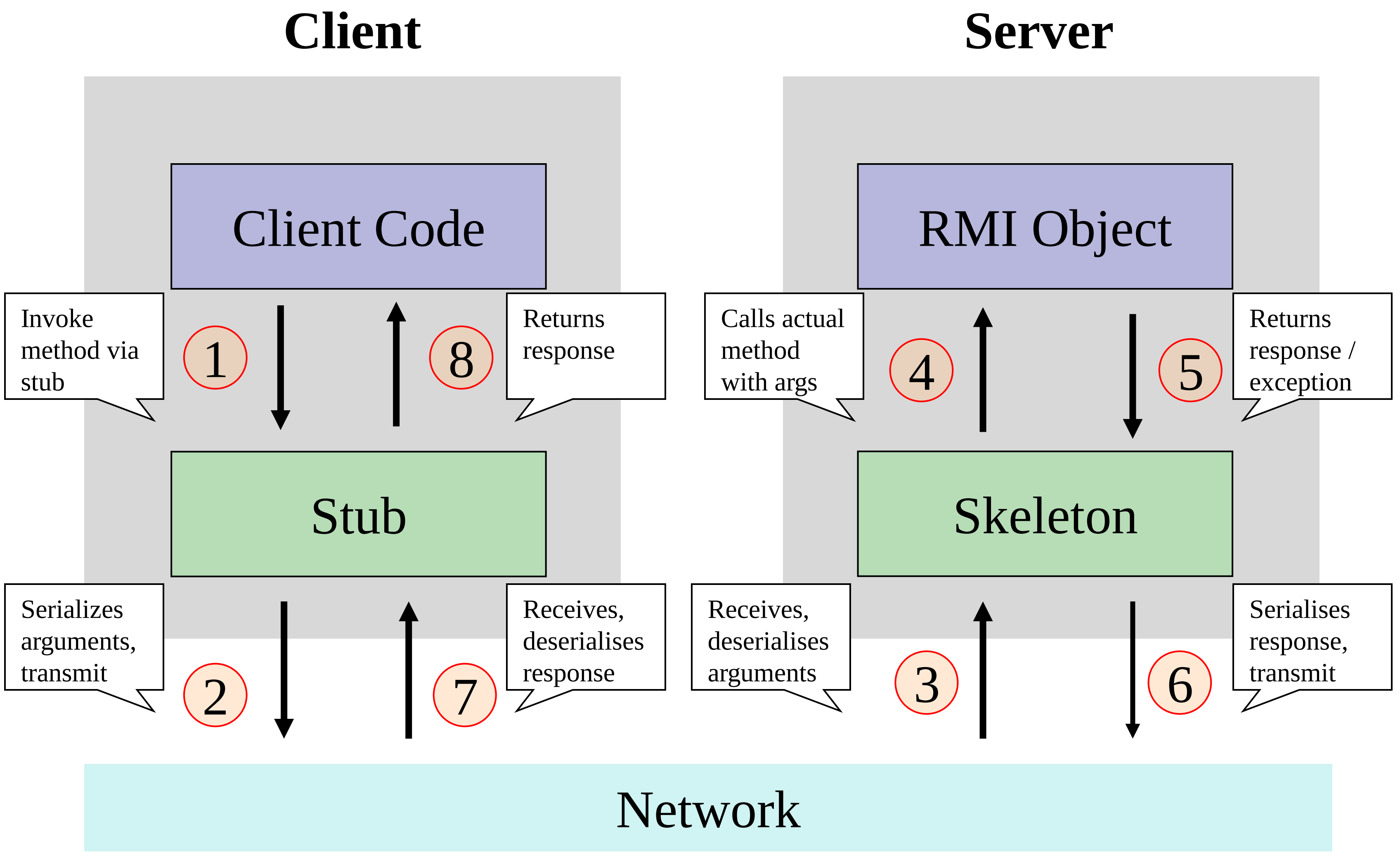


Remote Object table?

RMI Architecture and Components



Invocation Lifecycle


Client
Server
Client Code
RMI Object
Invoke method via stub
1
8
Returns response
Calls actual method with args
4
5
Returns response / exception
Stub
Skeleton
Serializes arguments, transmit
2
7
Receives, deserialises response
Receives, deserialises arguments
3
6
Serialises response, transmit
Network

Outline

- Introduction to Distributed Objects
- Remote Method Invocation (RMI) Architecture
- RMI Programming and a Sample Example:
  - Server-Side RMI programming
  - Client-Side RMI programming
- Advanced RMI Concepts
  - Security Policies
  - Exceptions
  - Dynamic Loading
- A more advanced RMI application
  - Math Server
- RPC and Summary

Steps for implementing an RMI application

- Design and implement the components of your distributed application
  - Remote interface
  - Servant program
  - Server program
  - Client program
- Compile source code and generate stubs
  - Client proxy stub
  - Server dispatcher and skeleton
- Make classes network accessible
  - Distribute the application on server side
- Start the application

RMI Programming and Examples

- Application Design
  - Remote Interface
    - Exposes the set of methods and properties available
    - Defines the contract between the client and the server
    - Constitutes the root for both stub and skeleton
  - Servant component
    - Represents the remote object (skeleton)
    - Implements the remote interface
  - Server component
    - Main driver that makes available the servant
    - It usually registers with the naming service
  - Client component

Example application – Hello World

- Server side
  - Create a HelloWorld interface
  - Implement HelloWorld interface with methods
  - Create a main method to register the HelloWorld service in the RMI Name Registry
  - Generate Stubs and Start RMI registry
  - Start Server
- Client side
  - Write a simple Client with main to lookup HelloWorld Service and invoke the methods

1. Define Interface of remote method

```
//file: HelloWorld.java
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloWorld extends Remote {
   public String sayHello(String who) throws RemoteException;
}
```

2. Define RMI Server Program

```
// file: HelloWorldServer.java
import java.rmi.Naming;
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloWorldServer extends UnicastRemoteObject
implements HelloWorld {
        public HelloWorldServer() throws RemoteException
                super();
        }
        public String sayHello(String who) throws
RemoteException {
                return "Hello "+who+" from your friend RM
433-652 :-)";
```

2. Define RMI Server Program

```
public static void main(String[] args) {
        String hostName = "localhost";
        String serviceName = "HelloWorldService";
        if(args.length == 2){
                hostName = args[0];
                serviceName = args[1];
         }
         try{
                HelloWorld hello = new HelloWorldServer();
                 Naming.rebind("rmi://"+hostName+"/"+serviceName,
                hello);
                System.out.println("HelloWorld RMI Server is
                running...");
                   }catch(Exception e){
                           e.printStackTrace();
                   }
          }
```

3. Define Client Program

```
// file:  RMIClient.java
import java.rmi.Naming;
public class RMIClient {
   public static void main(String[] args) {
      String hostName = "localhost";
      String serviceName = "HelloWorldService";
      String who = "Raj";
      if(args.length == 3){
         hostName = args[0];
         serviceName = args[1];
          who = args[2];
       }
       else if(args.length == 1){
           who = args[0];
       }
       try{
        HelloWorld hello =
 (HelloWorld)Naming.lookup("rmi://"+hostName+"/"+serviceName);
         System.out.println(hello.sayHello(who));
       }catch(Exception e){
           e.printStackTrace();
```

Required Reading

- Chapter 5 from the text book: Coulouris, George F., Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Fifth Edition