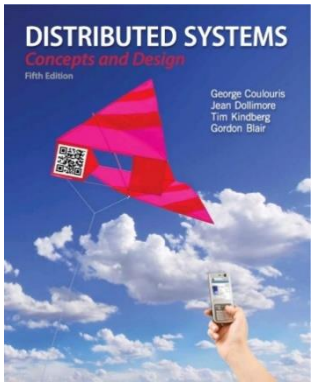
Distributed Objects Programming - Remote Invocation



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

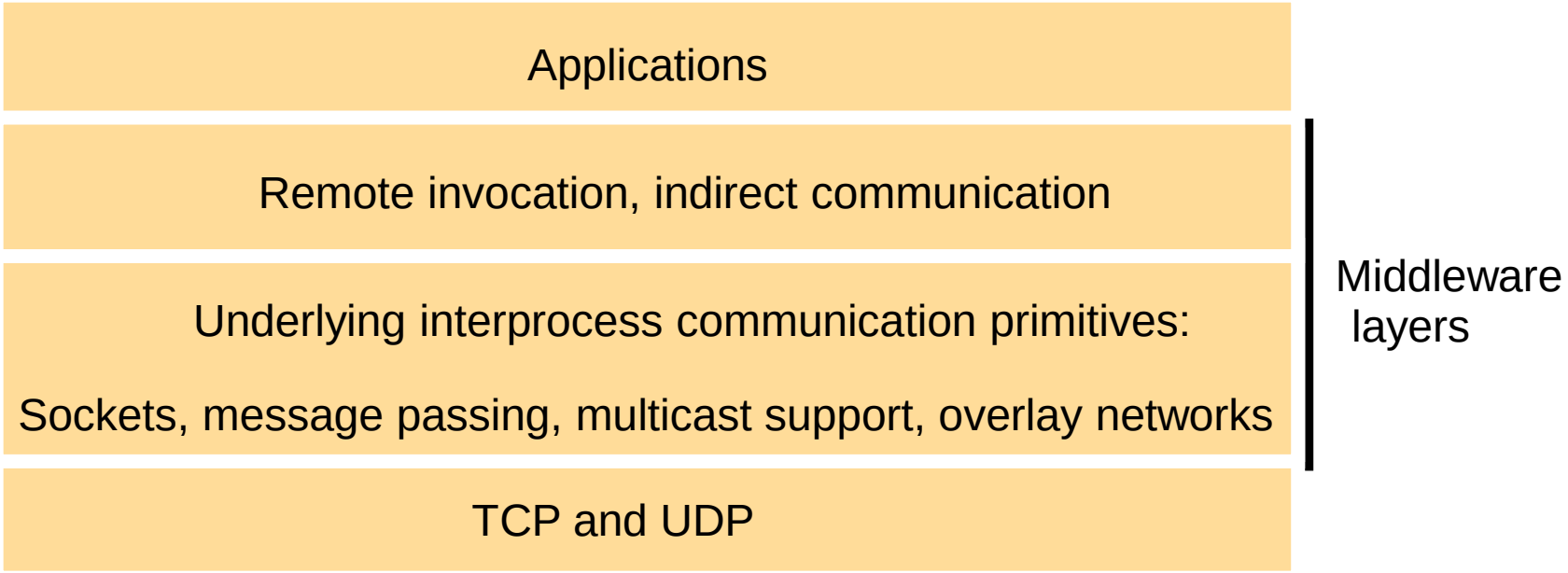
Co-contributors: Xingchen Chu, Rodrigo Calheiros, Maria Sossa, Shashikant Ilager

Sun Java online tutorials:

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

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.



Introduction

- We discuss how processes (or entities at a higher level of abstraction such as objects or services) communicate in a distributed system, examining the following **remote invocation paradigms**:
  - Request reply protocol
  - Remote Procedure Call (RPC): One implementation of the request-reply protocol
  - Remote Method Invocation (RMI): Another implementation of the request-reply protocol
  - Any additional suggestions?

Outline

- Request reply protocol
- Remote procedure call
- Remote method invocation
- Case Study: Java RMI

Outline

- **Request reply protocol**
- Remote procedure call
- Remote method invocation
- Case Study: Java RMI

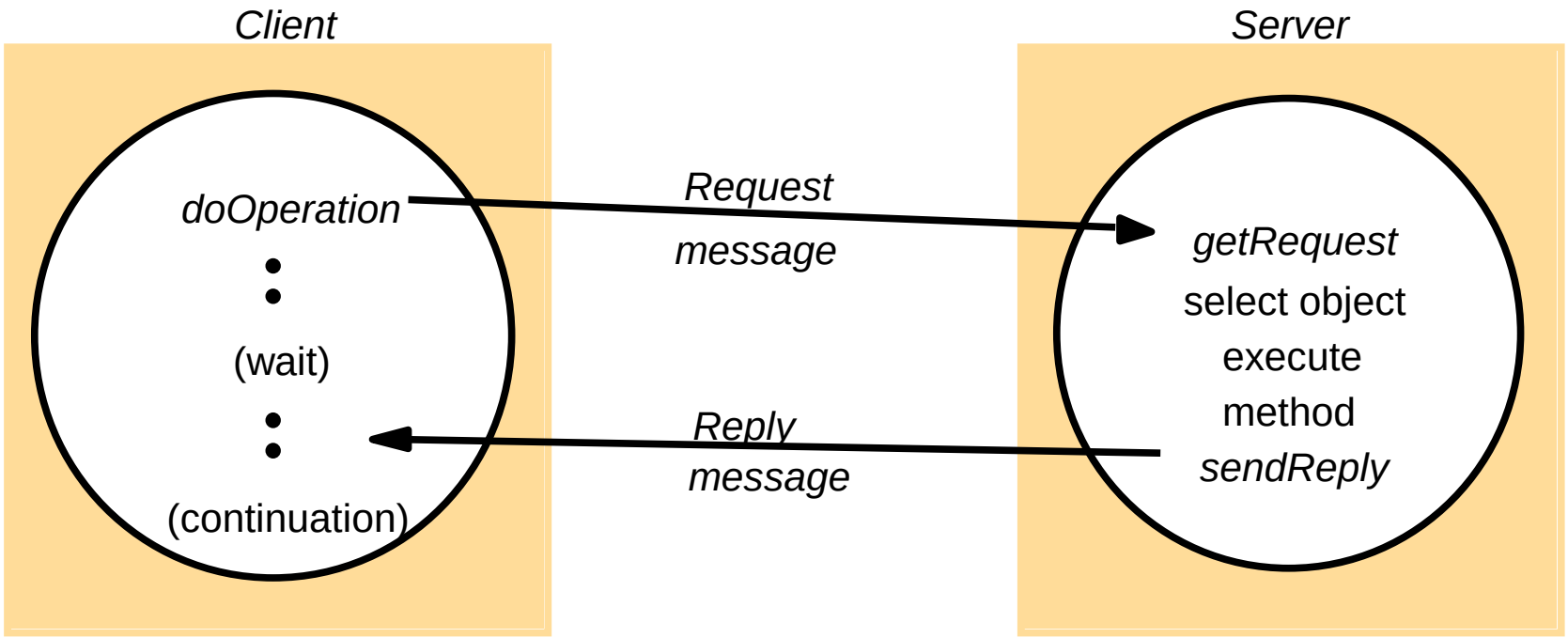
Request-reply Protocol

- Supports low-level client-server interactions
  - Usually use UDP datagrams, could use TCP streams
- Three primitives
  - `doOperation`: client sends request message to server
  - `getRequest`: server receives request msg, selects+invokes operation
  - `sendReply`: server sends reply message back to (blocked) client

6

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

Request-Reply Operations

- **public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)**
- The *doOperation* method sends a request message **to the server whose Internet address and port are specified in the remote reference given as an argument**.
- Its result is a byte array containing the reply.
- The first argument of *doOperation* is an instance of the class *RemoteRef*,

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

Partial Failures

- The components of distributed systems communicate over a network.
- In communications technology terminology, the shared local and wide area networks that our systems communicate over are known as *asynchronous* networks.
- With such asynchronous network, what kind of scenarios could happen?

Partial Failures **Scenarios**

- Nodes can choose to send data to other nodes at any time.
- The network is *half-duplex*, meaning that one node sends a request and must wait for a response from the other. These are two separate communications.
- The time for data to be communicated between nodes is variable, due to reasons like network congestion, dynamic packet routing, and transient network connection failures.

Partial Failures **Scenarios**

- The receiving node may not be available due to a software or machine crash.
- Data can be lost. In wireless networks, packets can be corrupted and hence dropped due to weak signals or interference. Internet routers can drop packets during congestion.
- Nodes do not have identical internal clocks; hence they are not synchronized.

Partial Failures Example

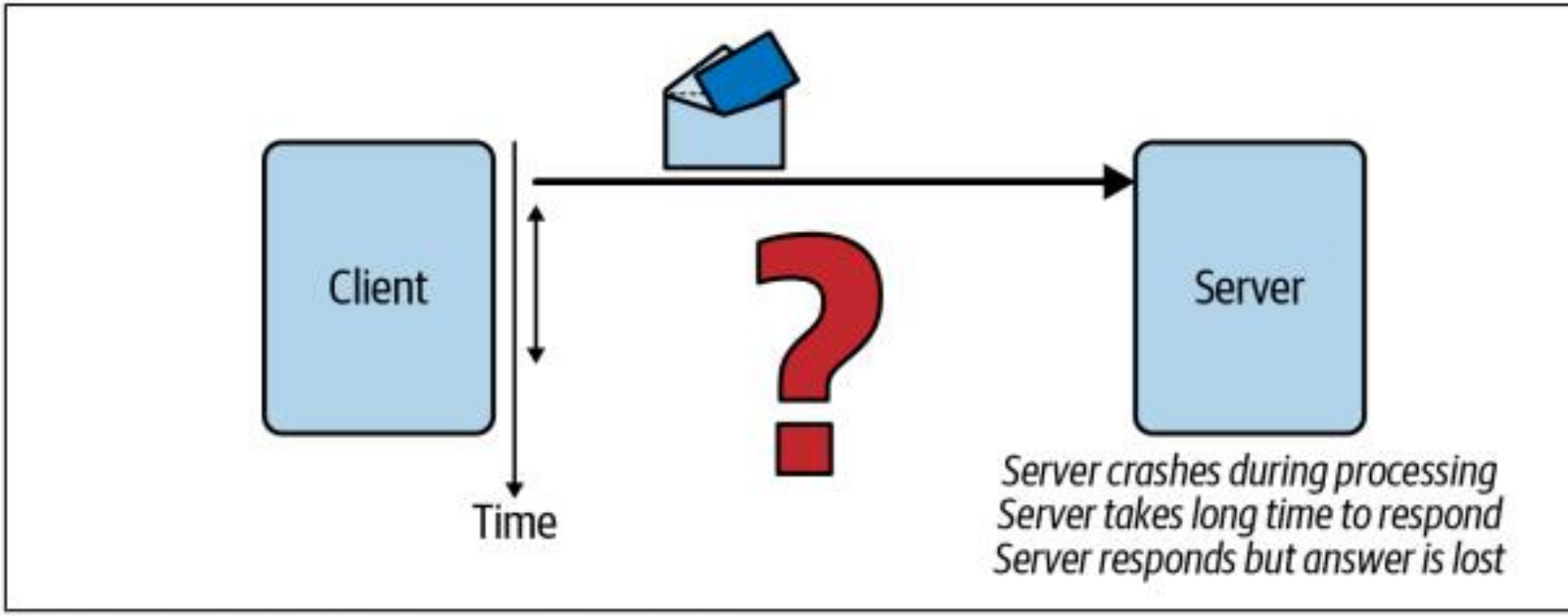


Figure 3-5. Handling partial failures

Partial Failures Example

- The request succeeds and a rapid response is received. All is well. (In reality, this outcome occurs for almost every request. *Almost* is the operative word here.)
- The destination IP address lookup may fail. In this case, the client rapidly receives an error message and can act accordingly.
- The IP address is valid but the destination node or target server process has failed. The sender will receive a timeout error message and can inform the user.

Partial Failures Example

- The request is received by the target server, which fails while processing the request and no response is ever sent.
- The request is received by the target server, which is heavily loaded. It processes the request but takes a long time (e.g., 34 seconds) to respond.
- The request is received by the target server and a response is sent. However, the response is not received by the client due to a network failure.

Crash Faults?

- The last three problems are considered crash faults.
- From the client's perspective, these three outcomes look exactly the same. The client cannot know without waiting (potentially forever) whether the response will arrive eventually or never arrive.
- How about re-sending the request?

Crash Faults?

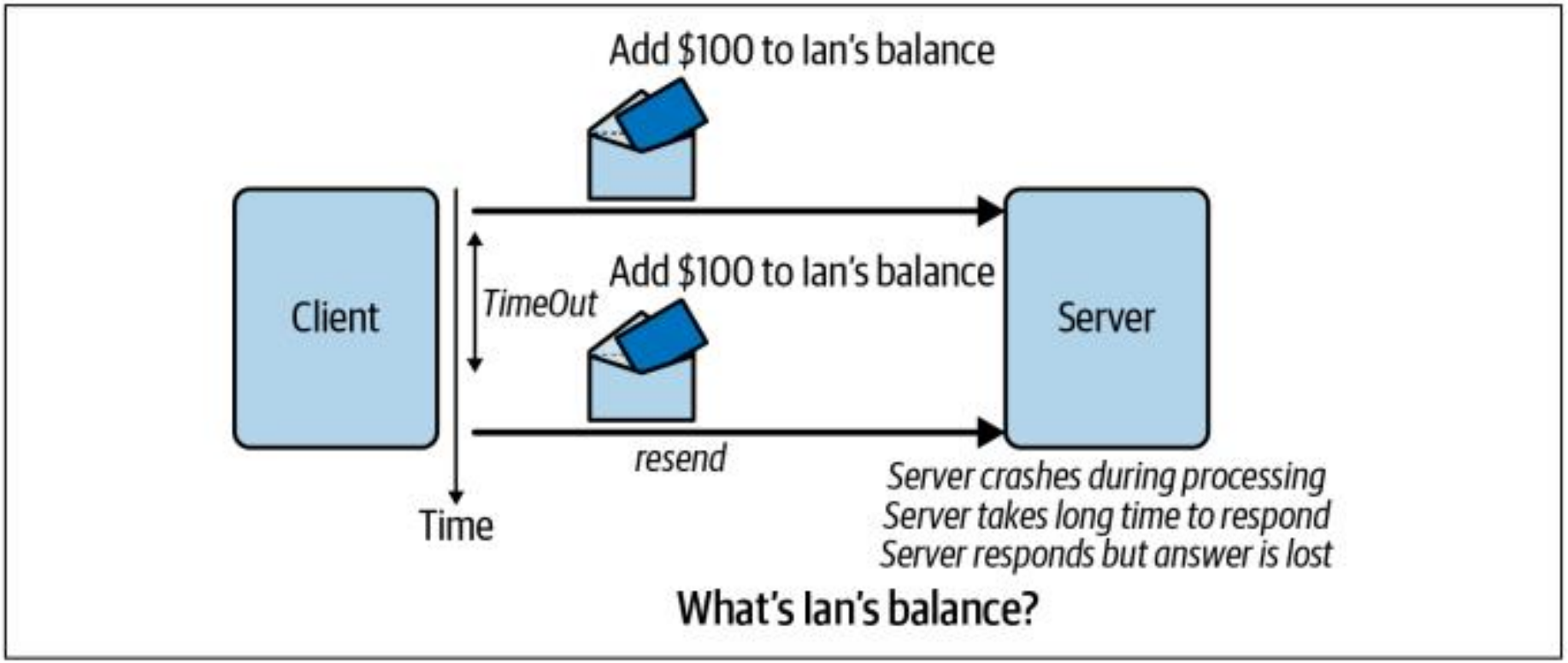


Figure 3-6. Client retries a request after timeout

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 aga to obtain the result
  - Some servers may re-execute the same operation again (**Idempotent operations?**)
  - An idempotent operation is an operation that can be performed repeatedly with the same effect as if it had been performed exactly once.
- How should a distributed system behave if all its operations
- are idempotent?
- How should the system behave if some operations are not idempotent?

Outline

- Request reply protocol
- **Remote procedure call**
- Remote method invocation
- Case Study: Java RMI

Remote Procedure Call (RPC)

- The concept of a remote procedure call (RPC) aims to make the programming of distributed systems look similar, if not identical, to conventional programming
- In RPC, procedures on remote machines can be called as if they are procedures in the local address space
- The underlying RPC system hides aspects of distribution (e.g., encoding/decoding of parameters and results, passing of messages, used semantics in case of failures).

Design Issues for RPC

- Three issues we will look into
- the style of programming promoted by RPC
- the call semantics associated with RPC
- the key issue of transparency and how it relates to remote procedure calls

Design Issues for RPC
1. Style of Programming

- Three issues we will look into
- the style of programming promoted by RPC
- the call semantics associated with RPC
- the key issue of transparency and how it relates to remote procedure calls

Design Issues for RPC
1. Style of Programming (Programming with Interfaces)

Explicit interface

- Hide a lot of implementation details
- Tell exactly how a client can access the server

Keeping implementation separate from interface

- Good idea? Why?

Differences from local procedure interface

- Can't access shared memory variables between client and server
- Call by reference does not make sense for RPC
  - Parameters are **`in`**, **`out`**, or **`inout`**
- Can't pass pointers (Why not?)

24

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();
};
```

Design Issues for RPC
2. 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.

Design Issues for RPC
2. 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?

Design Issues for RPC
2. RPC Call 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.

Design Issues for RPC
2. RPC Call 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

Design Issues for RPC
3. Transparency

Do you recall transparency from week 1 lecture?

- You want to move your business and computers to the Caribbean (because of the weather or low tax)?
- Your client moves to the Caribbean (more likely)?

Let us review some types of transparency

- *Access transparency* enables local and remote resources to be accessed using identical operations
- *Location transparency* enables resources to be accessed without knowledge of their physical or network location (for example, which building or IP address).

Design Issues for RPC
3. Transparency

- **How does RPC achieve transparency?**
- All the necessary calls to marshalling and message-passing procedures were hidden from the programmer making the call.
- Although request messages are retransmitted after a timeout, this is transparent to the caller to make the semantics of remote procedure calls like that of local procedure calls.

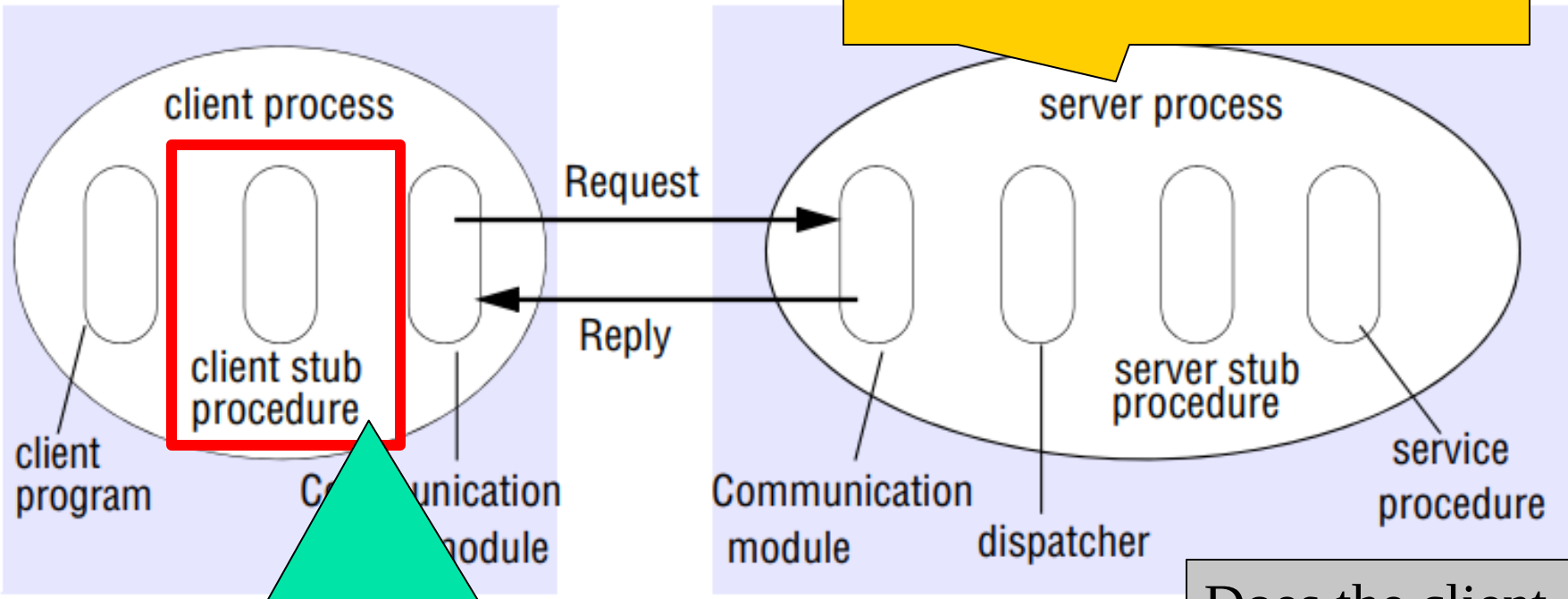
Implementation of RPC

- Assume that the server provide a service interface that includes several services

Implementation of RPC

provide a service interface that includes several services

Role of client and server stub procedures in RP

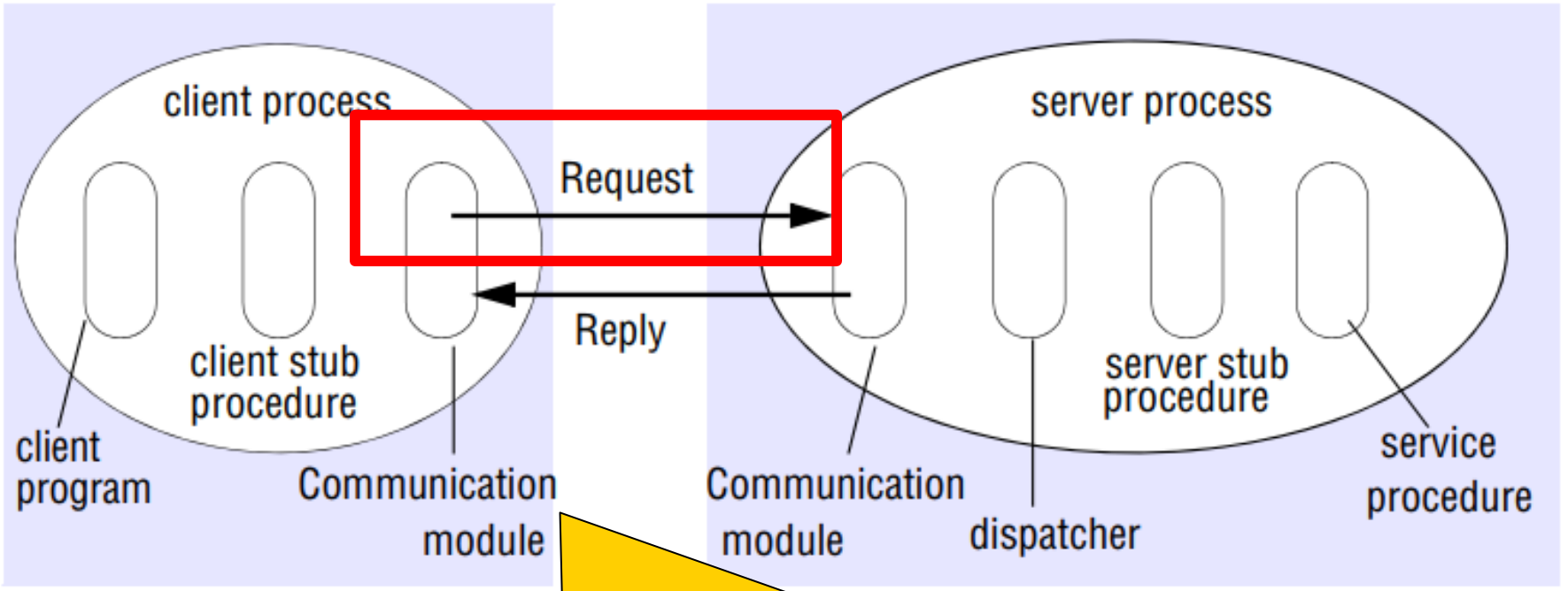


One stub for each procedure
Behaves like a local procedure to the client

Does the client stub execute the call?

Implementation of RPC

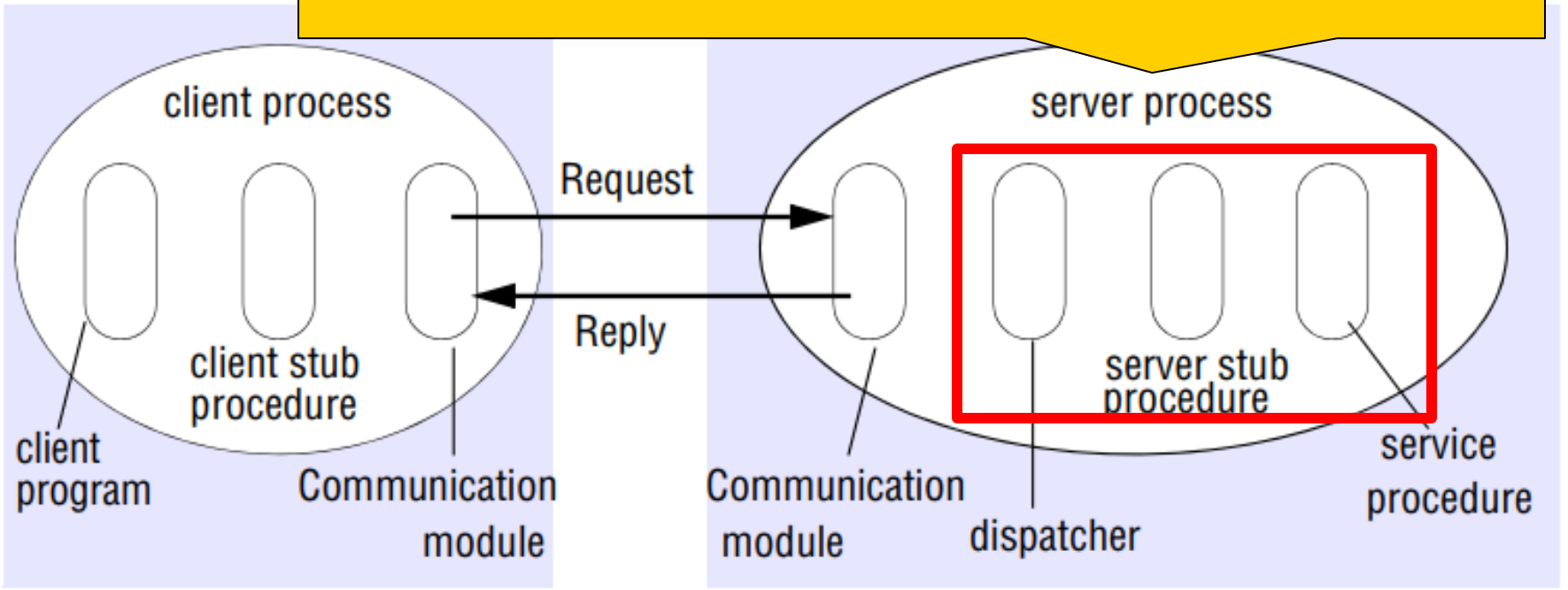
Role of client and server stub procedures in RPC



The communication module is used to send the marshalled info to the server

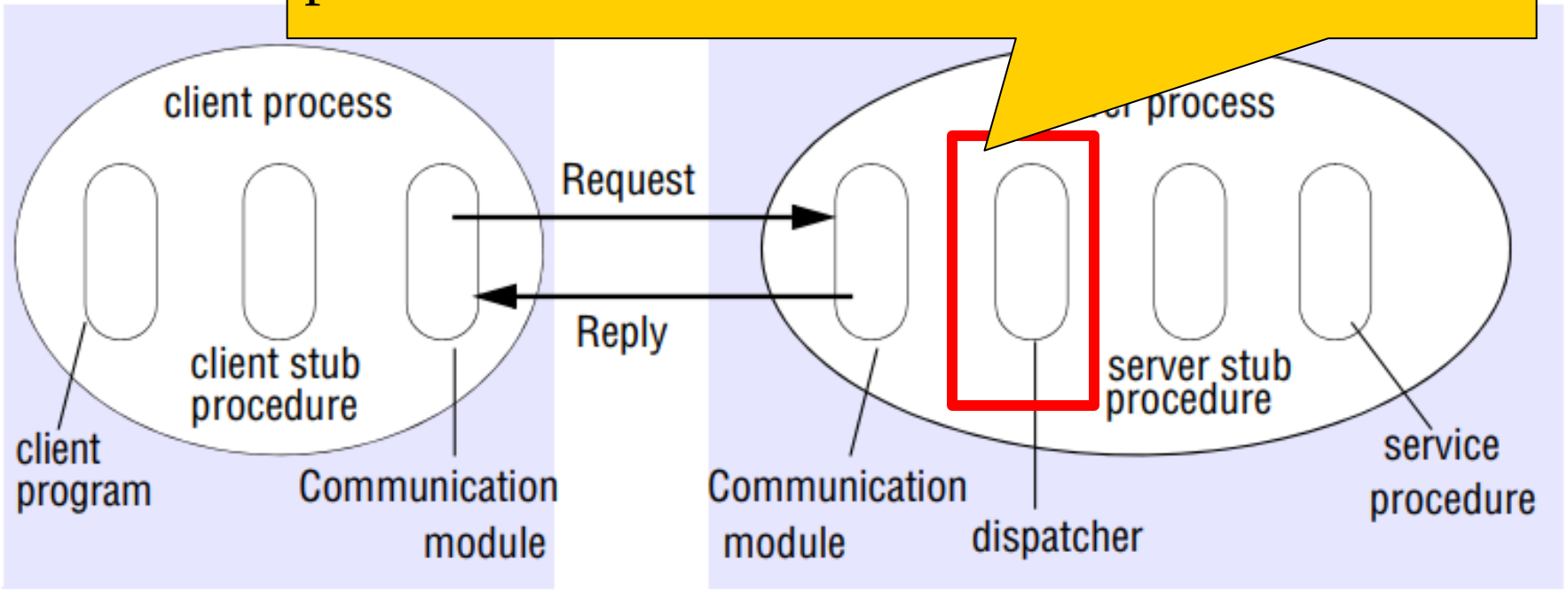
The server process contains a **dispatcher** together with **one server stub procedure** and **one service procedure for each procedure** in the service interface.

Role of client and



Dispatcher: Based on the received request message (that includes a procedure ID), it selects the corresponding server stub procedure

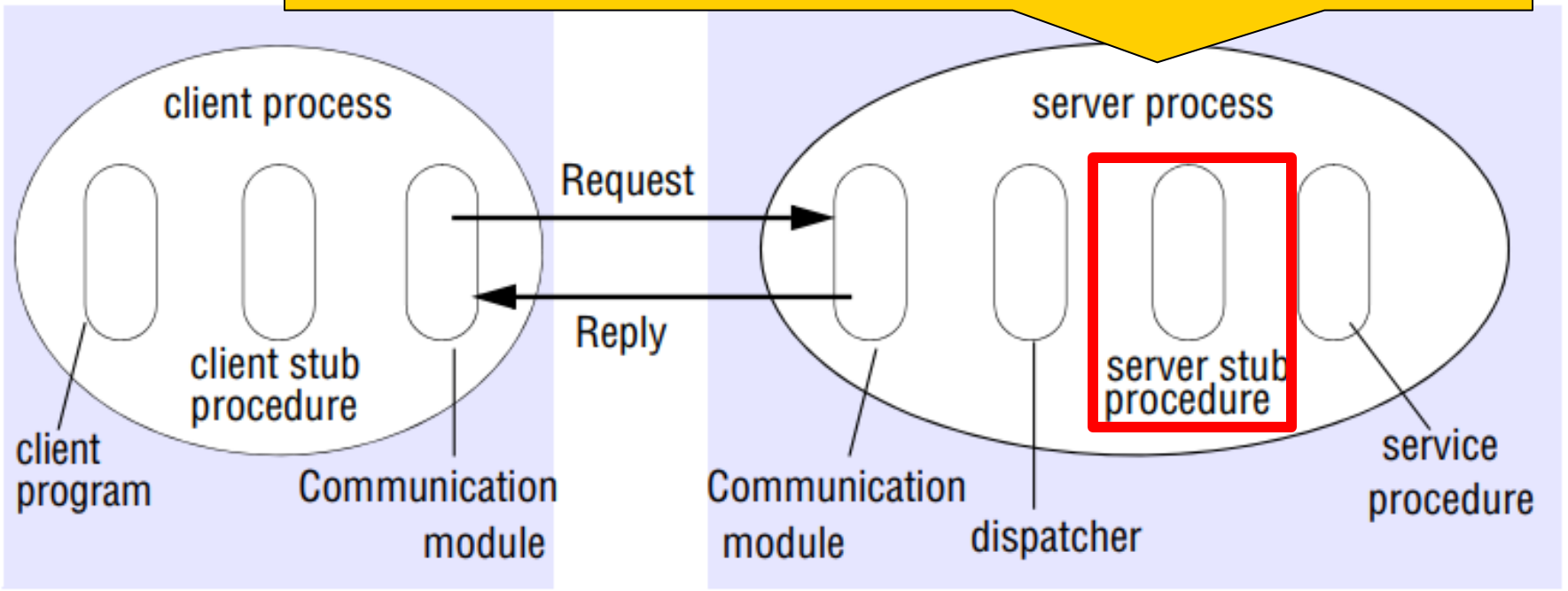
Role of client an



Implementation of RPC

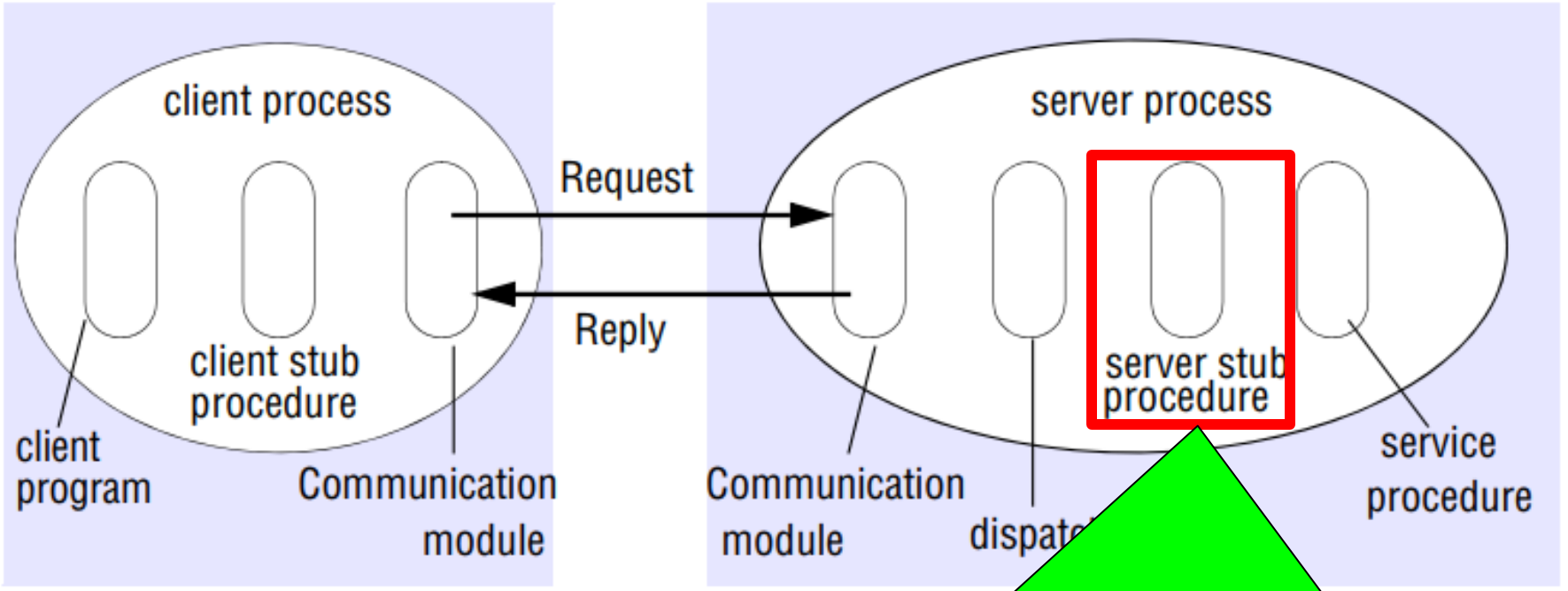
Server stub: unmarshalls the arguments and calls the corresponding service procedure

Role of client an



Implementation of RPC

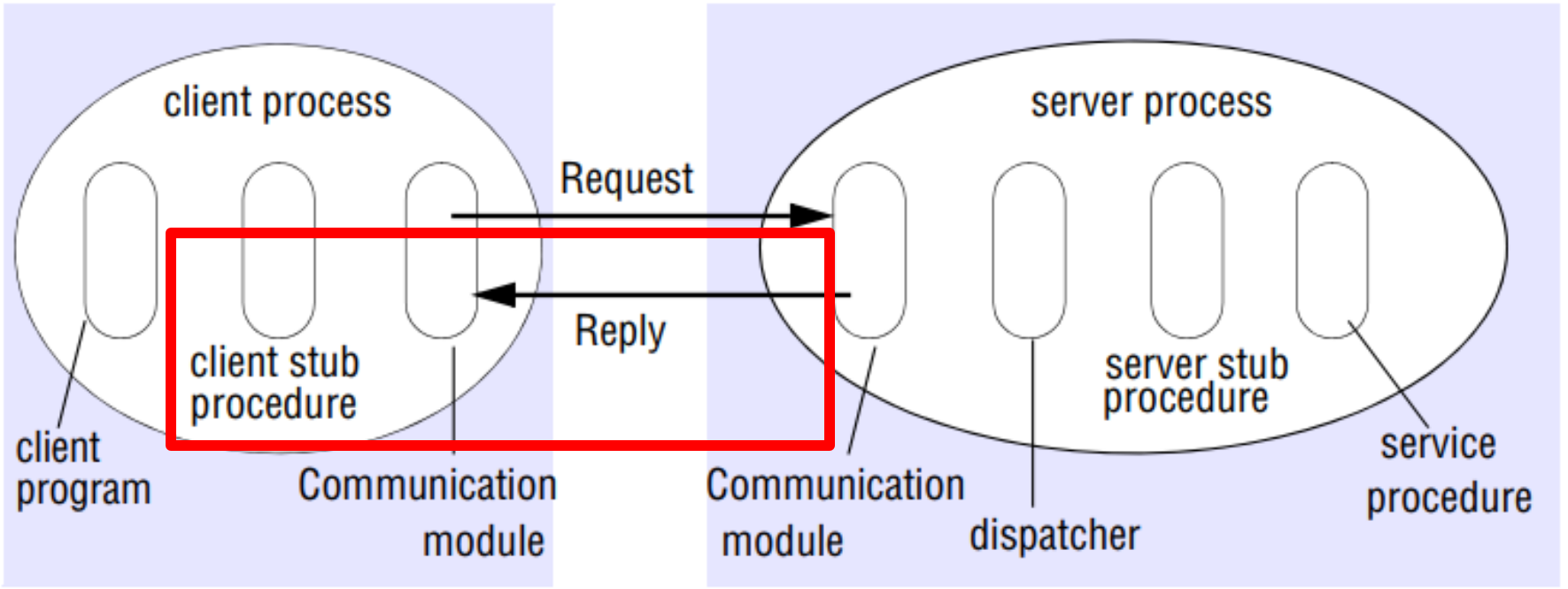
Role of client and server stub procedures in RPC



Server stub: After execution, marshalls the return values for the reply message

Implementation of RPC

Role of client and server stub procedures in RPC



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

41

Marshalling

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

42

Outline

- Request reply protocol
- Remote procedure call
- **Remote method invocation**
- Case Study: Java RMI

Remote method invocation

- Similarities between RPC and RMI
  - Programming with interfaces
  - Both constructed on top of some RR protocol and have same choices in call semantics
  - Similar level of transparency
- Differences providing added expressiveness in RMI
  - Full expressive power of OO programming (not just a “fad”…)
  - Can cleanly pass object references as parameters

44

Passing Objects?

- Parameter passing is particularly important in distributed systems.
- RMI allows the programmer to pass parameters not only by value, as input or output parameters, but also by object reference.
- Passing references is particularly attractive if the underlying parameter is large or complex.
- The remote client can access an object using remote method invocation, **instead of having to transmit the object value across the network.**

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.

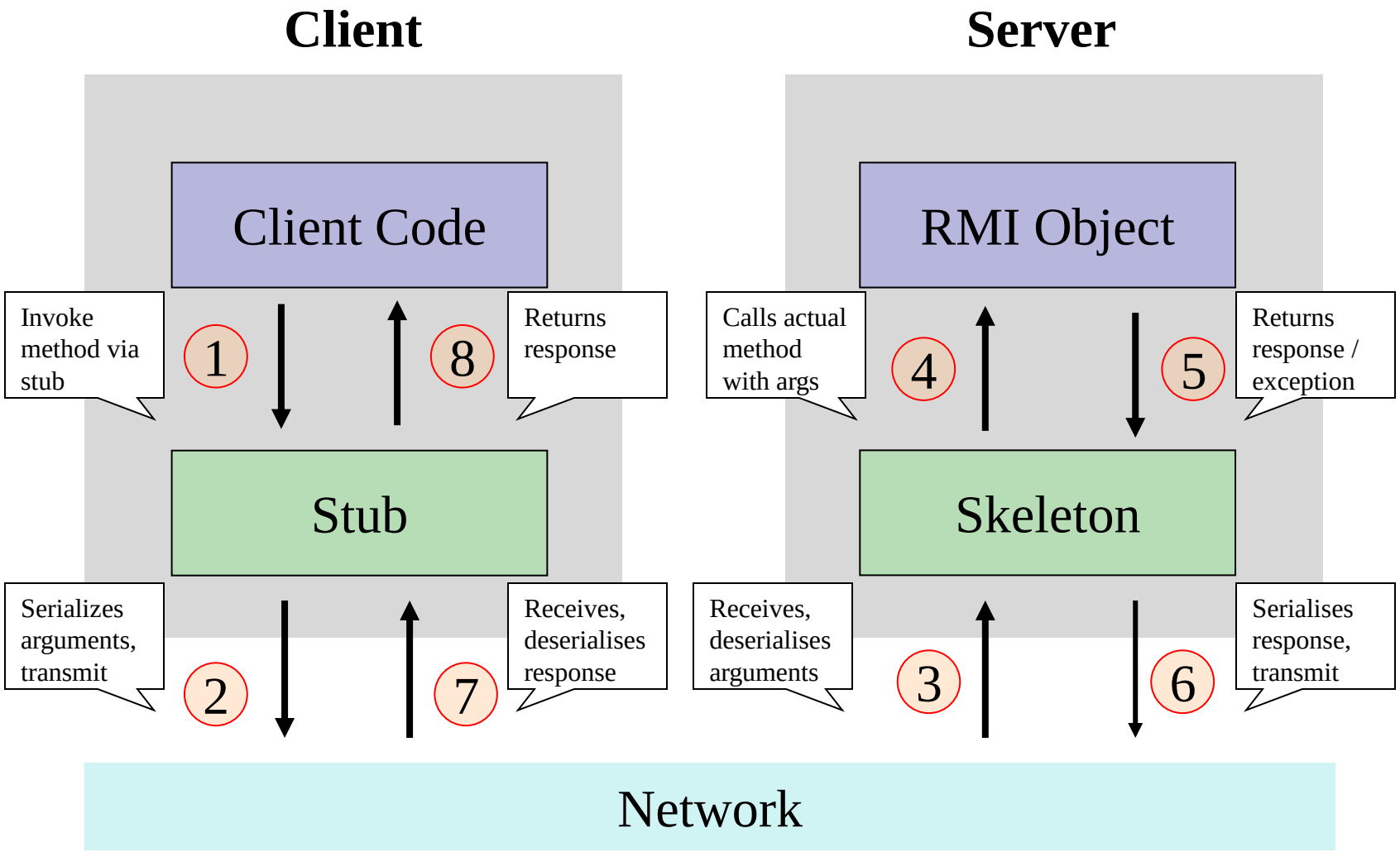
Outline

- Request reply protocol
- Remote procedure call
- Remote method invocation
- **Case Study: Java RMI**

Java RMI

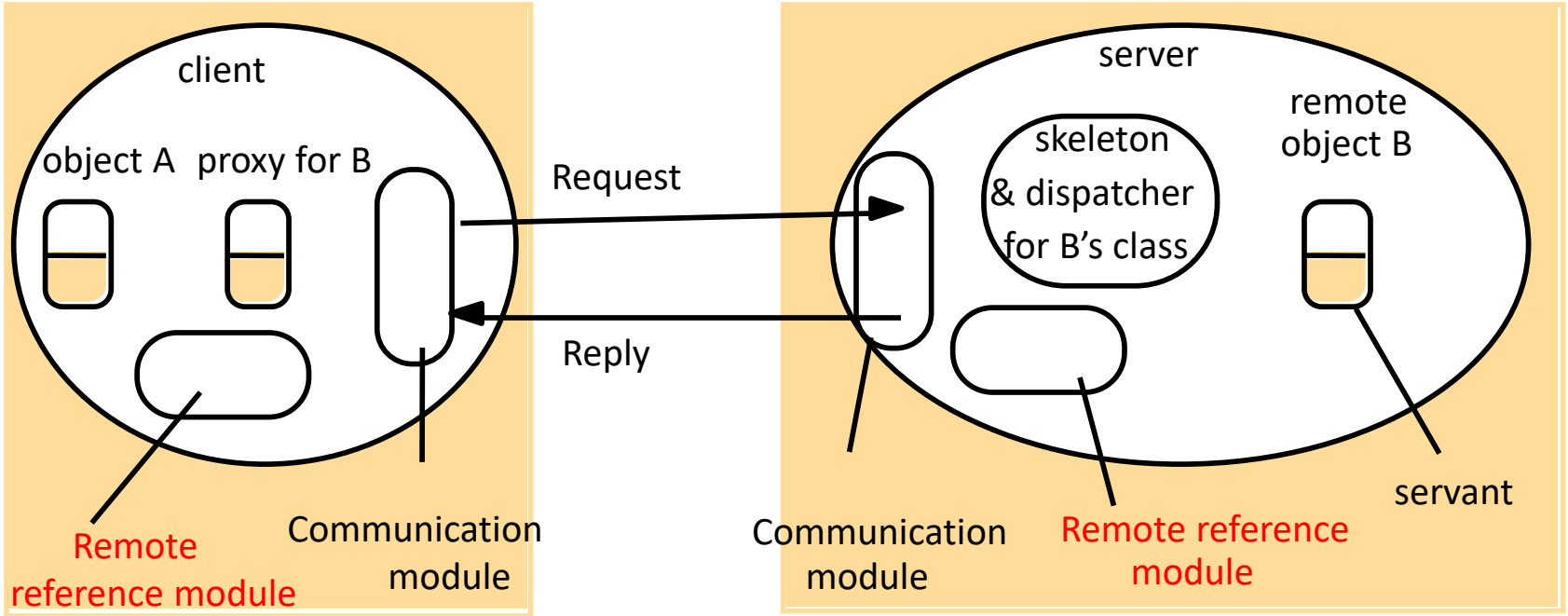
- Java Remote Method Invocation (Java RMI) is an extension of the Java object model to support distributed objects
  - methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts
- RMI uses object serialization to marshal and unmarshal
  - Any serializable 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 Invocation Lifecycle

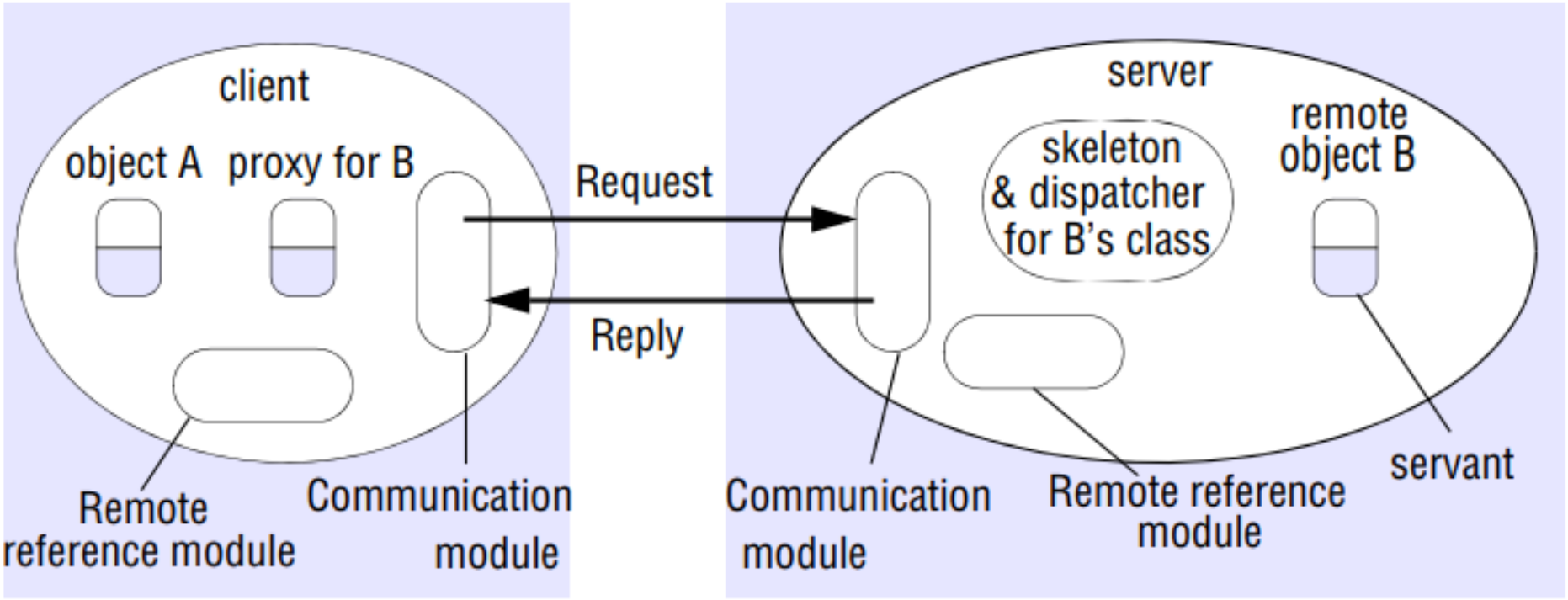


RMI Architecture and Components

- 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

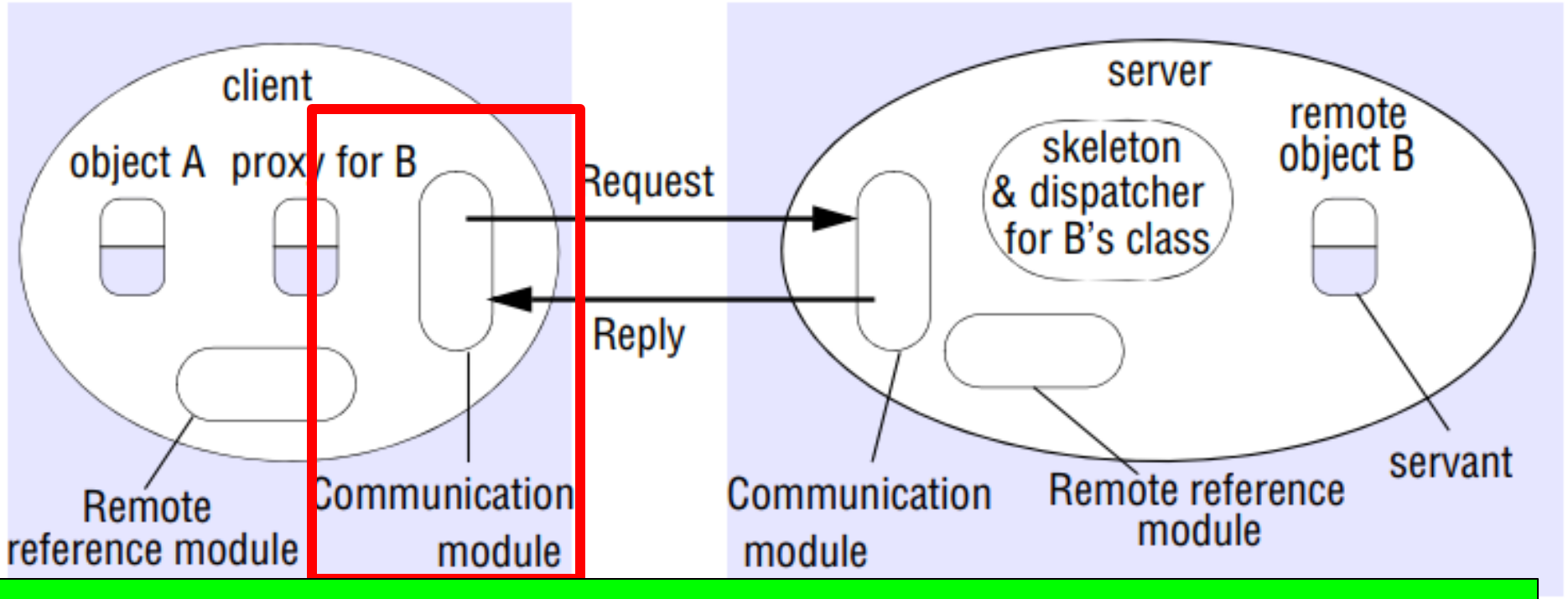


RMI Architecture and Components



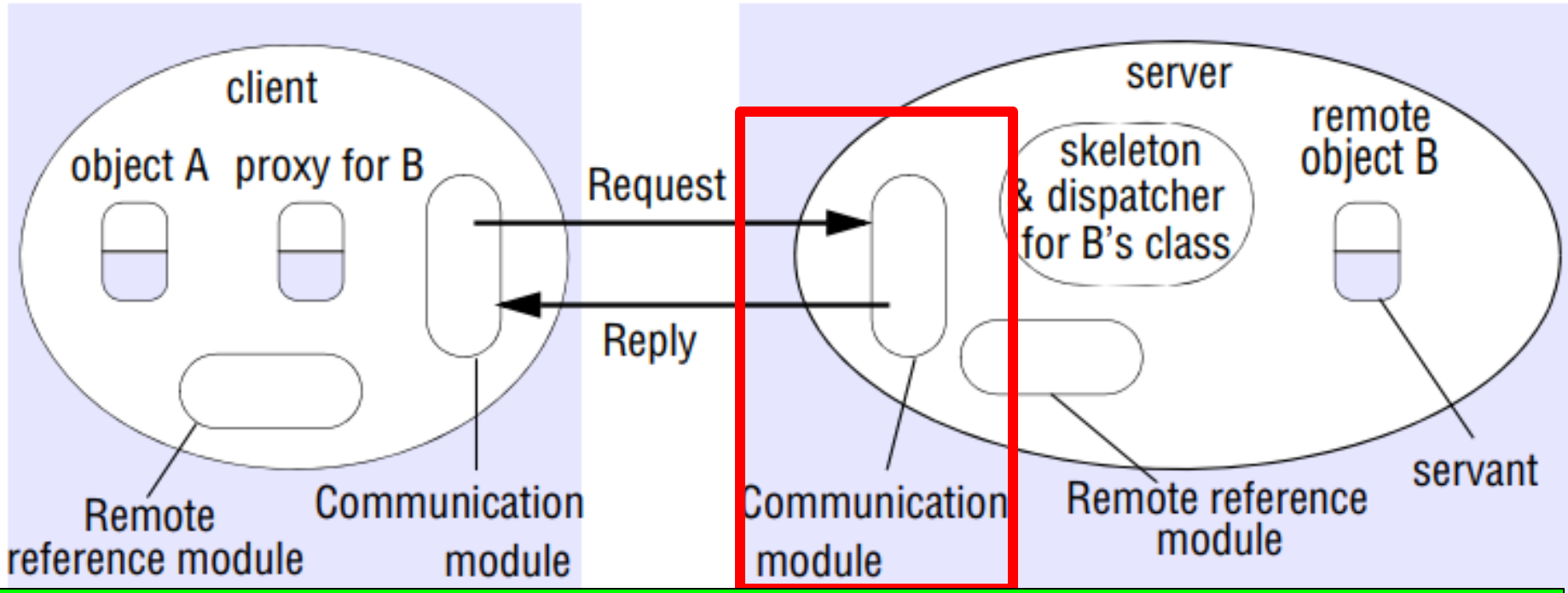
Assume that an **application-level object A** invokes a method in a **remote application-level object B** for which it holds a remoote object reference (Proxy B).

RMI Architecture and Components



- The communication modules carry out the request-reply protocol, which transmits request and reply messages.
- The communication module of the client sends the message to the server

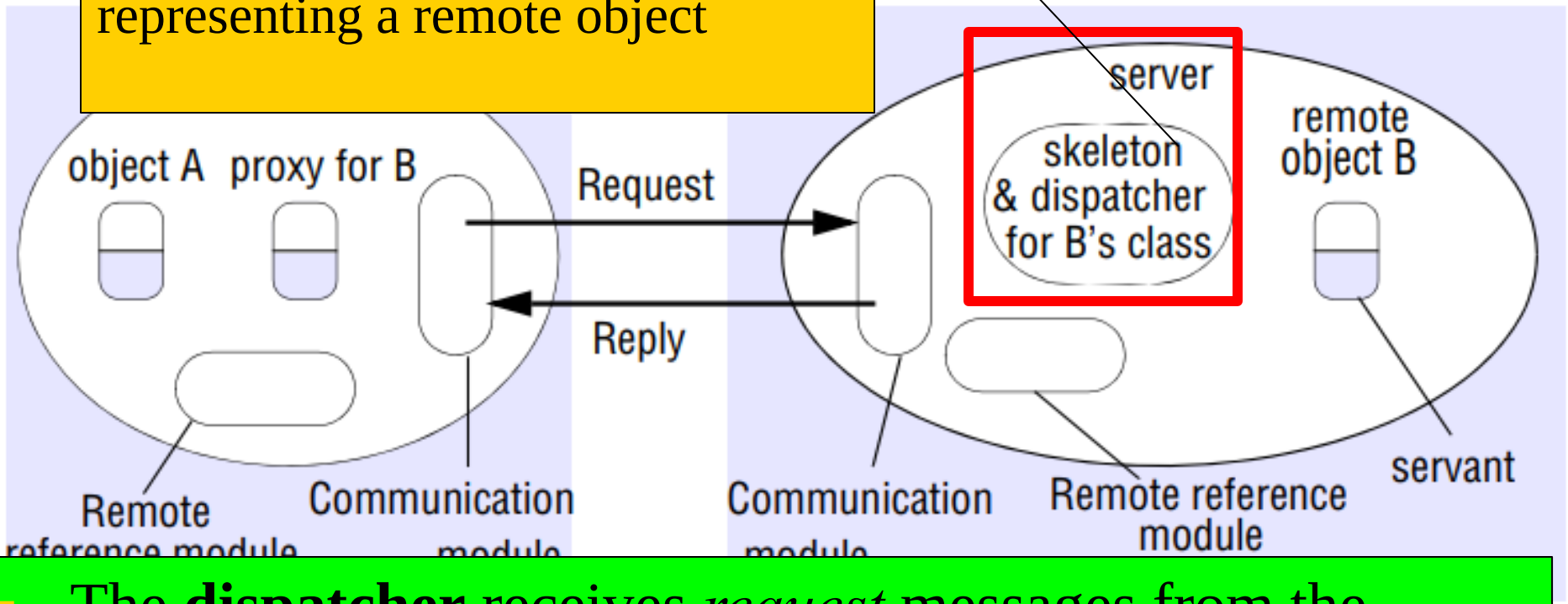
RMI Architecture and Components



- The **communication module in the server** selects the **dispatcher** for the class of the object to be invoked, passing on **its local reference**, which it gets from the **remote reference module** in return for the remote object identifier in the *request* message.

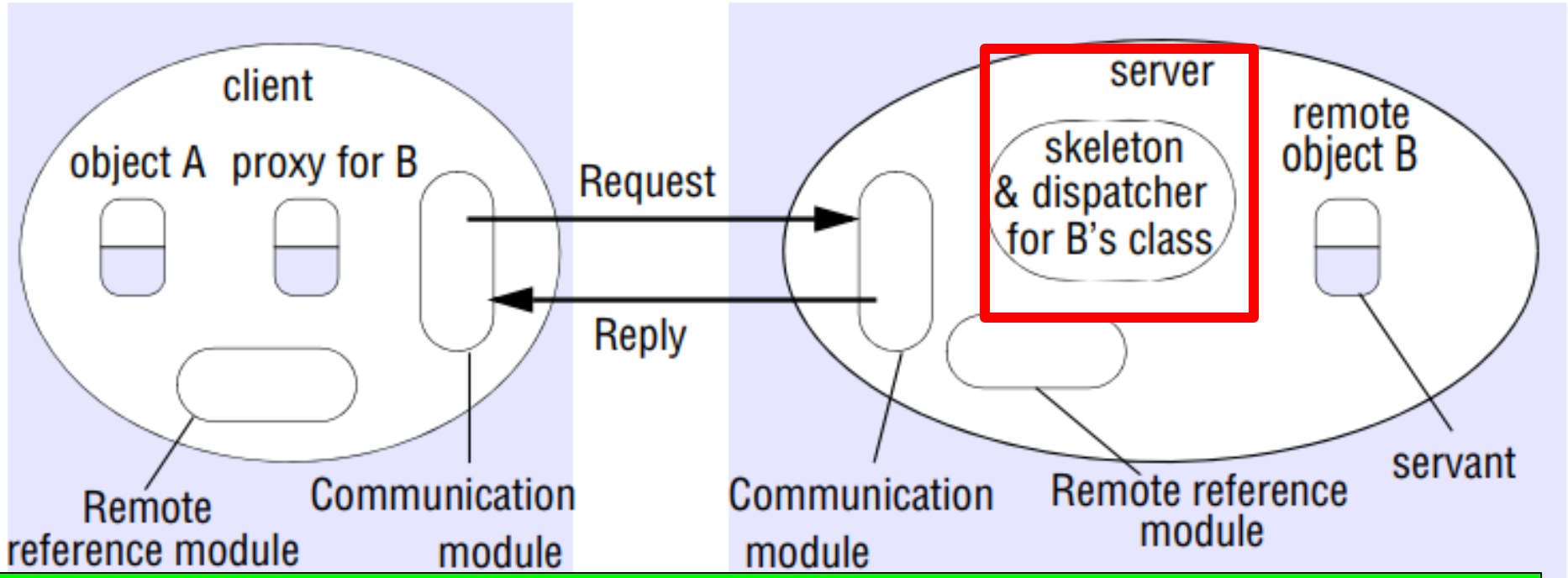
A server has one **dispatcher** and one **skeleton** for each class representing a remote object

d Components



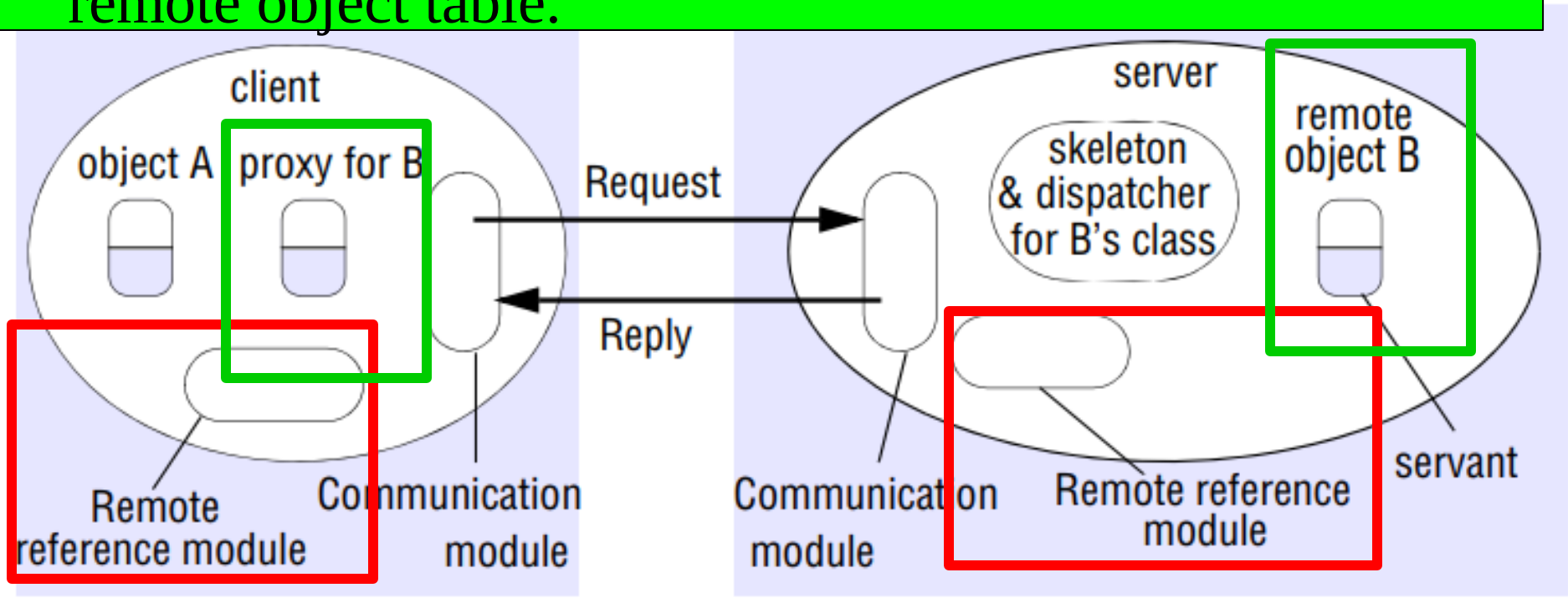
- The **dispatcher** receives *request* messages from the communication module. It uses the *operationId* to **select the appropriate method in the skeleton**, passing on the *request* message.

RMI Architecture and Components



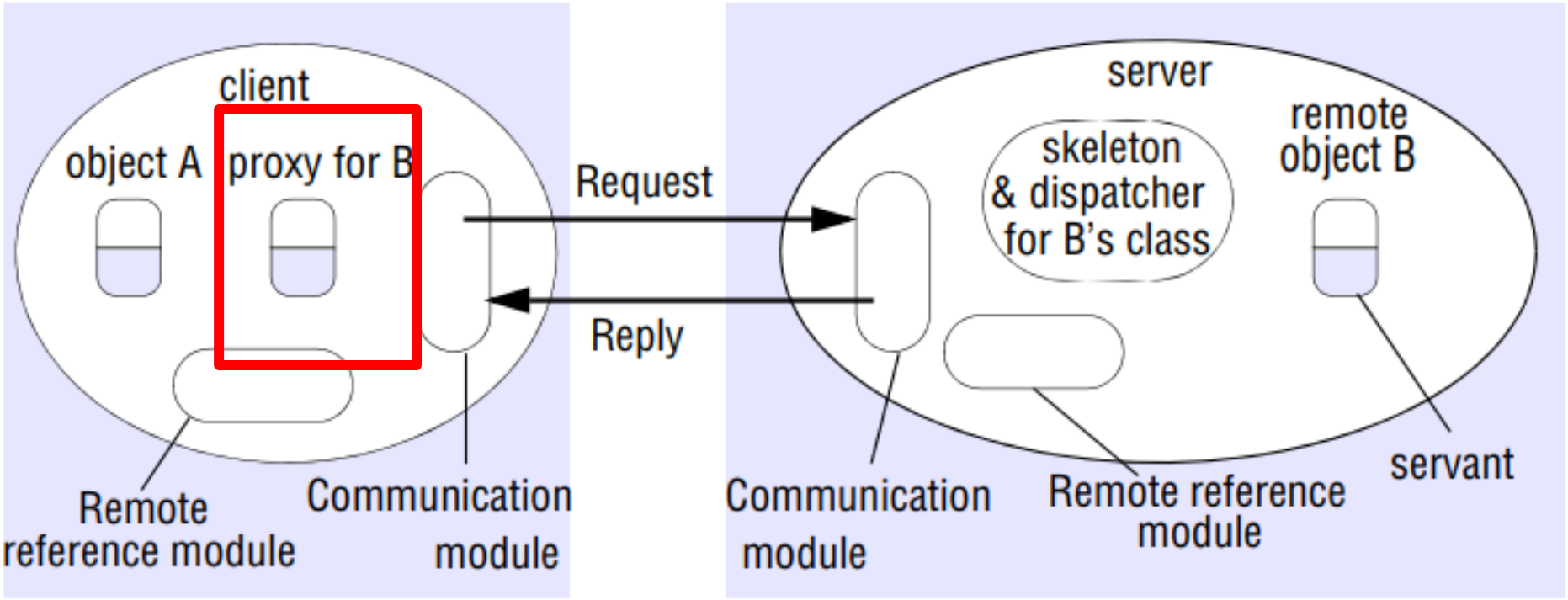
- The class of a remote object has a *skeleton*, which implements the methods in the remote interface.
- A **skeleton method** unmarshals the arguments in the *request* message and invokes the corresponding method in the servant (A *servant* is an instance of a class)

- **A remote reference module** is responsible for translating between local and remote object references.
- The remote reference module in each process has a remote object table.



Remote Object table?

RMI Architecture and Components



Outline

- Request reply protocol
- Remote procedure call
- Remote method invocation
- Case Study: Java RMI
- **Running Examples**

EXAMPLE A

The compute engine server accepts tasks from clients, runs the tasks, and returns any results.

Compute Server

1. Designing a Remote Interface

```
package compute;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface Compute extends Remote {
    <T> T executeTask(Task<T> t) throws RemoteException;
}
```

Compute Server

2. Implementing a Remote Interface

- Declare the remote interfaces being implemented
- Provide an implementation for each remote method in the remote interfaces
-
-

Compute Server

2. Implementing a Remote Interface

```
public class ComputeEngine implements Compute {

    public ComputeEngine() {
        super();
    }

    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }
```

Compute Server

3. Making the Remote Object Available to Clients

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
            (Compute) UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
```

Compute Server

3. Making the Remote Object Available to Clients

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
            (Compute) UnicastRemoteObject.exportObject(engine, 0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
```

Compute Server

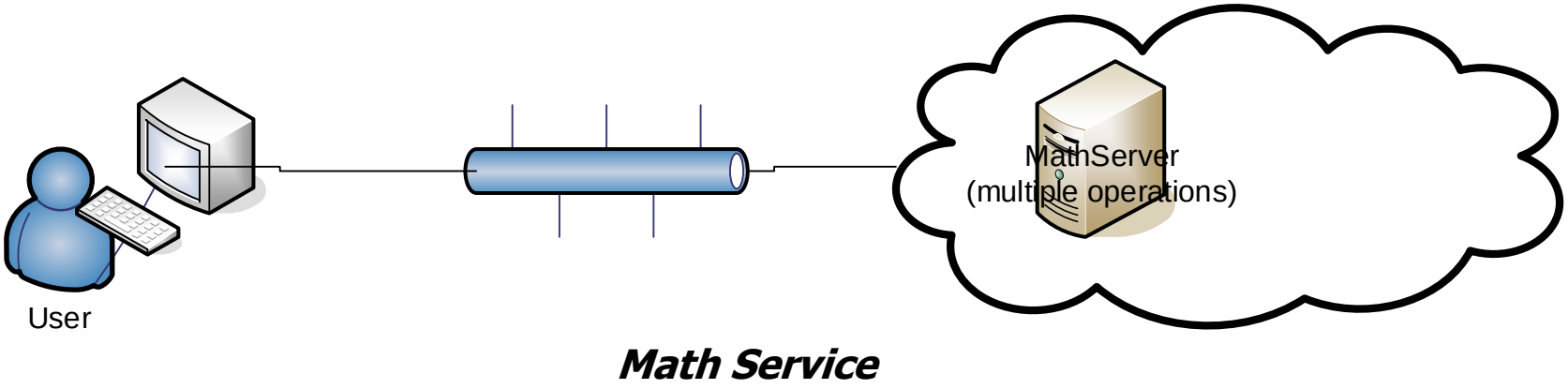
5. Client side

```
public class ComputePi {
    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }
        try {
            String name = "Compute";
            Registry registry = LocateRegistry.getRegistry(args[0]);
            Compute comp = (Compute) registry.lookup(name);
            Pi task = new Pi(Integer.parseInt(args[1]));
            BigDecimal pi = comp.executeTask(task);
            System.out.println(pi);
        } catch (Exception e) {
            System.err.println("ComputePi exception:");
            e.printStackTrace();
        }
    }
}
```

EXAMPLE B

Re-implement the MathServer using RMI

A Simple Math Server in RMI



Java RMI Example

- Specify the Remote Interface

```
public interface IRemoteMath extends Remote
{
      double add(double i, double j) throws
      RemoteException;

      double subtract(double i, double j)
      throws RemoteException;
}
```

Java RMI Example

- Implement the Servant Class

```
public class RemoteMathServant extends
UnicastRemoteObject implements IRemoteMath {
    public double add ( double i, double j )
throws RemoteException {
        return (i+j);
    }

    public double subtract ( double i, double
j ) throws RemoteException {
        return (i-j);
    }
```

Java RMI Example

- Implement the server

```
public class MathServer {
    public static void main(String args[]){
      IRemoteMath remoteMath = new
      RemoteMathServant();
      Registry registry =
      LocateRegistry.getRegistry();
      registry.bind("Compute", remoteMath );
      System.out.println("Math server ready");
}
}
```

Java RMI Example

- Implement the client program

```
public class MathClient {
    public static void main(String[] args) {
        try {
            LocateRegistry.getRegistry("localhost");
            IRemoteMath remoteMath = (IRemoteMath)
  registry.lookup("Compute");
  System.out.println( "1.7 + 2.8 = ” +
  remoteMath.add(1.7, 2.8) );
  System.out.println( "6.7 - 2.3 = ” +
  remoteMath.subtract(6.7, 2.3) );
}
catch( Exception e ) {
  System.out.println( e );
```

References/Required Readings

- Chapter 5 from the textbook: Coulouris, George F., Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Fifth Edition
- Chapter 3 “Distributed Systems Essentials” from the textbook: Gorton, Ian. *Foundations of Scalable Systems*. " O'Reilly Media, Inc.", 2022.
- Java RMI tutorial https://docs.oracle.com/javase/tutorial/rmi/index.html