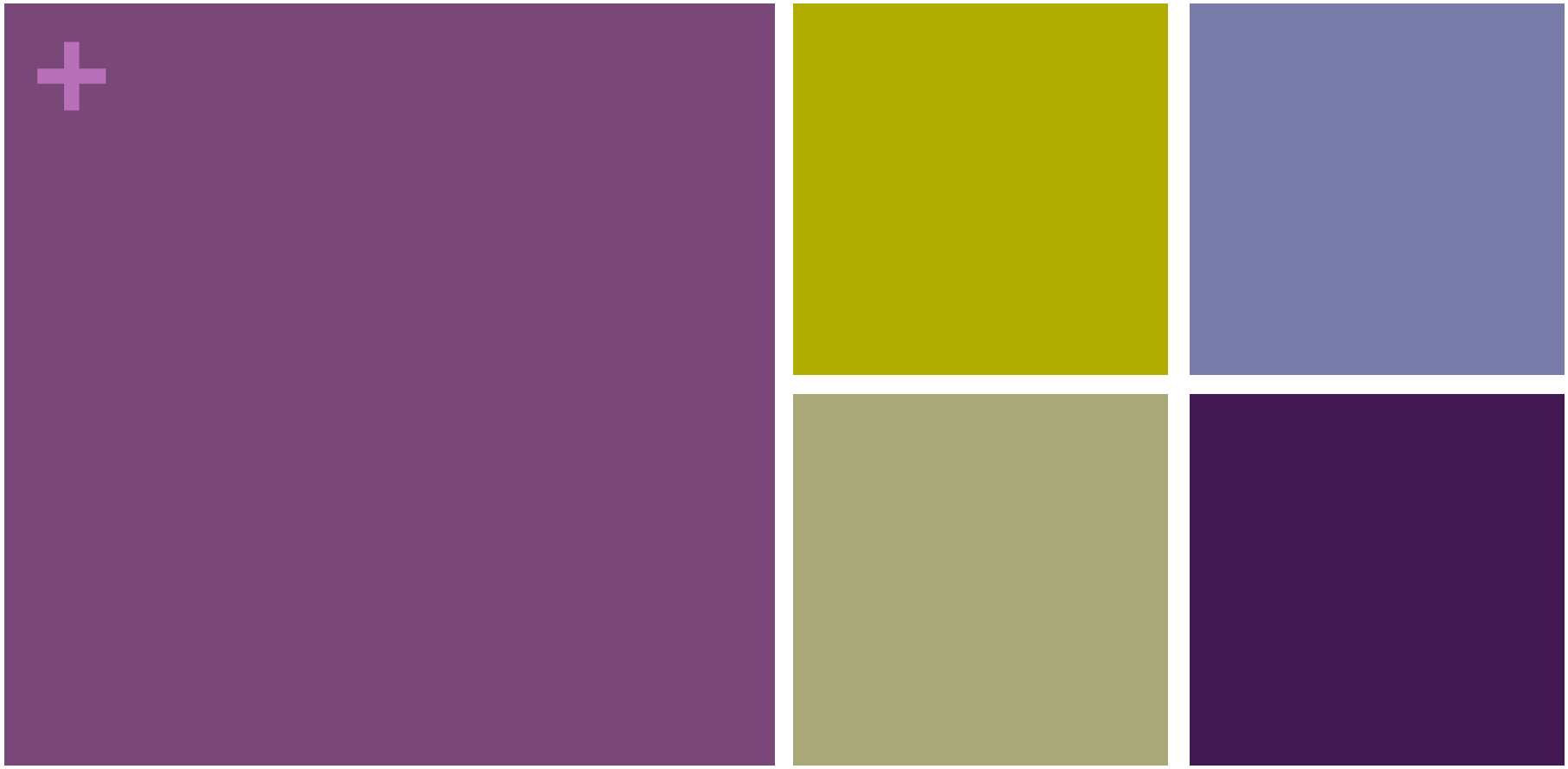


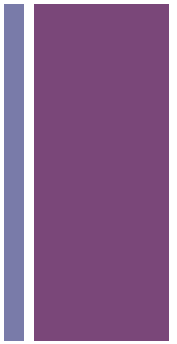
Remote Invocation

Seamless programming with remote procedures and remote object's methods

Course update

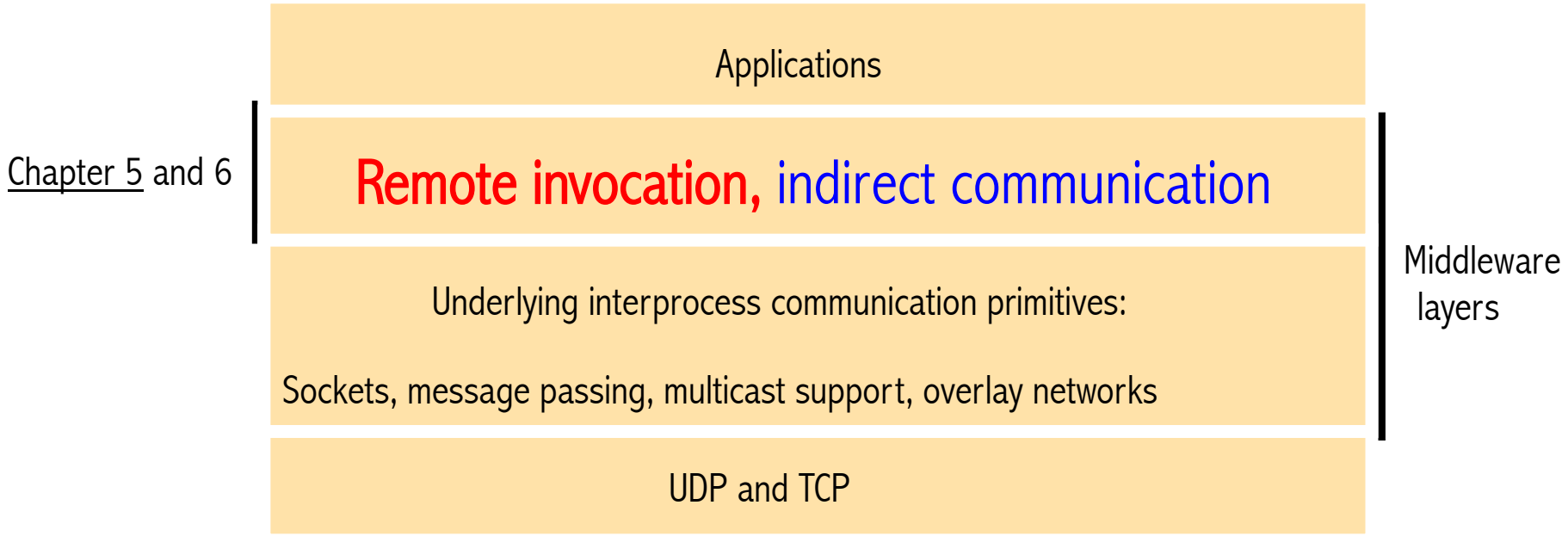
- Characterization of DS (Ch. 1)
- Distributed System Models (Ch. 2)
- Interprocess Communication (Ch. 4)
  - Sockets
  - Marshalling
- **Remote Invocation (Ch. 5)**
  - **Case study: Java RMI**

Chapter 5, structure



1. Introduction
2. Request-reply protocols
3. RPC, Remote Procedure Call
4. RMI, Remote Method Invocation
5. Case study: Java RMI

Figure 5.1
Middleware layers



Introduction to Remote Invocation

- **Request-reply protocols**
  - Based on message passing
  - Used in Client/Server computing (C/S)
    - http is a RR protocol

- **RPC**: Extension of procedural programming to distributed systems

- **RMI:** Extension of local method invocation that allows an object living in one process to invoke the methods of an object *living* in another process
  - The term RMI is used here in a generic way
  - Java RMI is a particular kind of RMI

Request-reply protocols

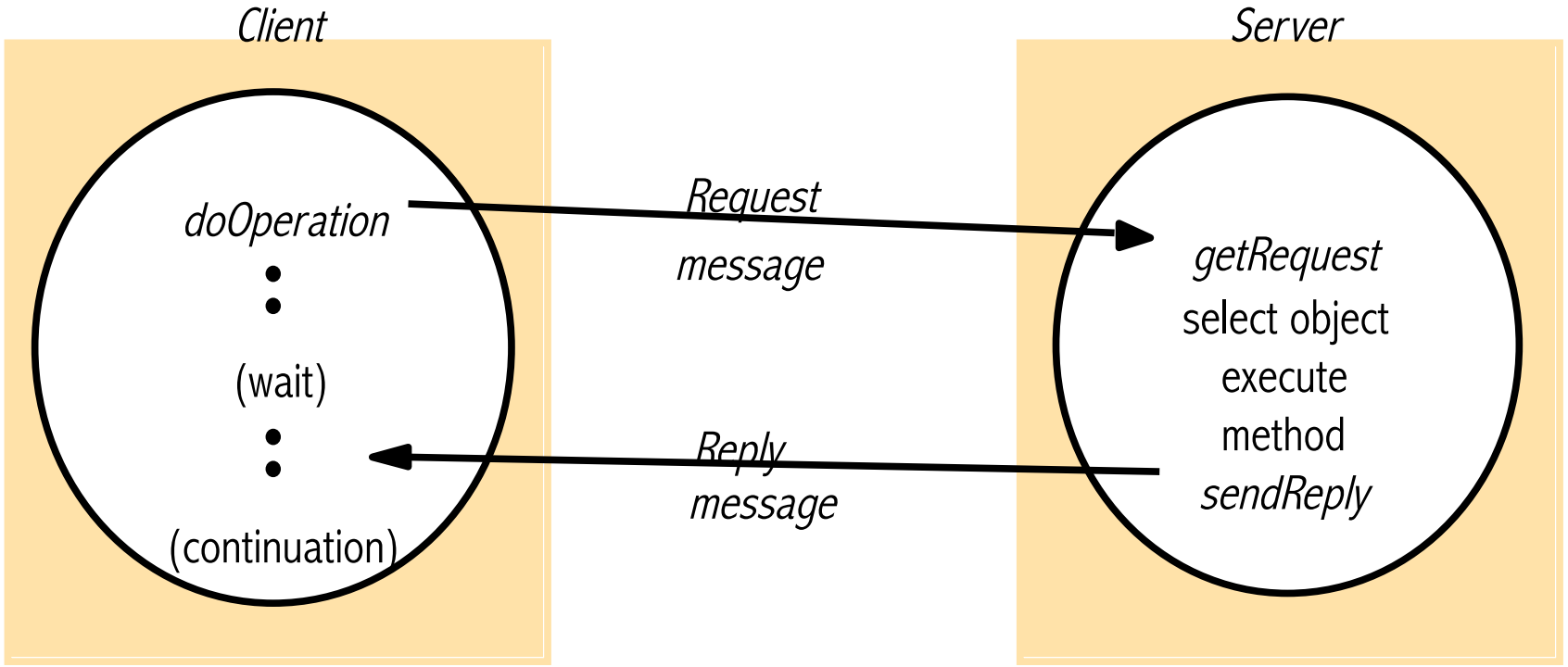
The most primitive service

Request-reply protocols

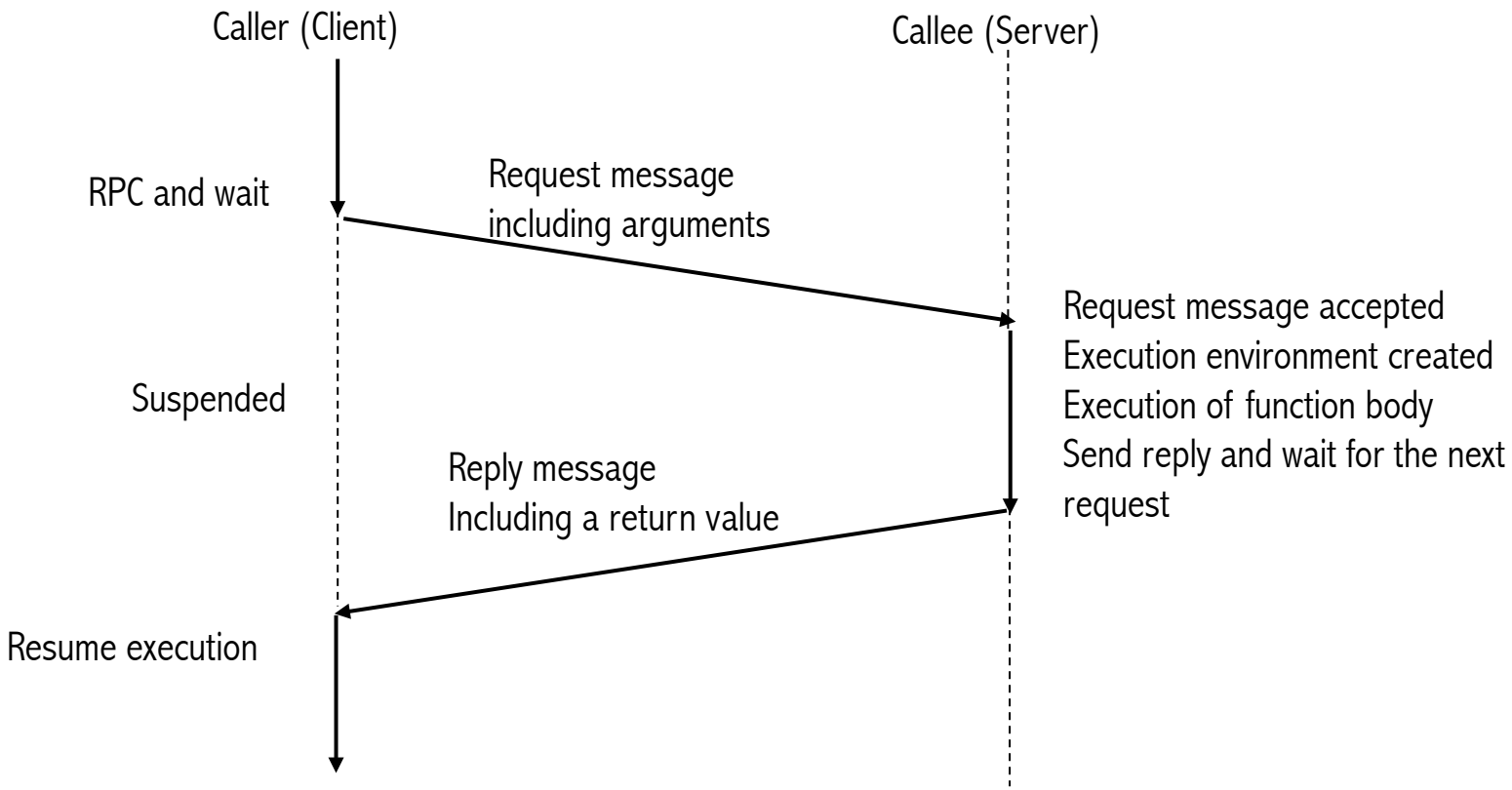
- Used in Client/Server interactions
- Synchronous: The client blocks until it receives the response
- Reliable: The fact that a reply has been received constitutes an acknowledgement
- Asynchronous: In situations where the client can in fact defer the reception of the reply
- May be built over UDP or over TCP
  - UDP: avoids overhead
  - TCP: simplifies the solution

Request-reply protocols over UDP

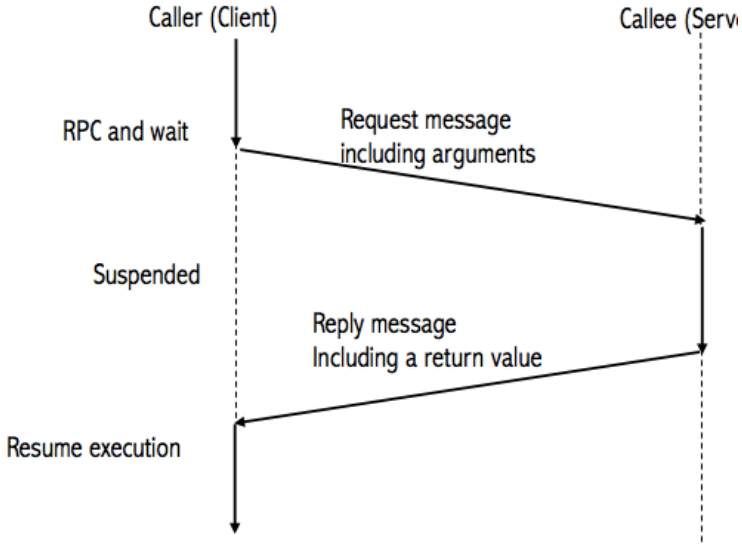
- Delivery guarantees are provided by the RR protocol implementation itself
  - The server reply is the acknowledgement



Request-reply protocols
Concept



Request-reply protocols
Properties



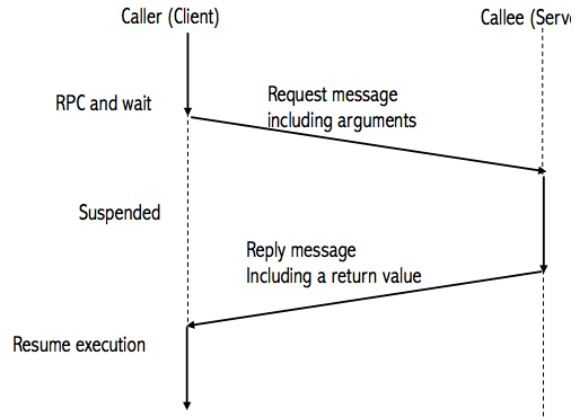
- Message identifiers
- Failure model
  - Omission
  - Lack of ordering guarantees
  - Failure of processes (Not Byzantine)
- Timeouts:
  - RR uses a timeout in case a request or reply message got lost
  - RR produces a request repeatedly until it gets a response
- Discards repeated reponses
- Lost reply messages: idempotent operations
- Journals: A record of request or reply messages

Figure 5.5 Request-response exchange protocols

Protocol Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

Request-reply protocols
Over TCP

- If arguments and result can be any size, then, the byte-stream orientation of TCP is a perfect fit
- TCP cares that the RR messages are properly sequenced and delivered
- Simplifies implementation of RR
- End-to-end link utilization can be improved by pipelining RR
  - RR Operations must be idempotent



Request-reply protocols
http (Over TCP)

- Uses *persistent* TCP connections
- Supports content negotiation
- Supports authentication
- Requests and replies are marshalled as ASCII text
- MIME (Multipurpose Internet Mail Extensions) are used to identify contents
  - text/plain, text/html, image/gif etc.
- http Methods
  - GET: Requests a resource identified by a URL, all the information about the resource is in the URL
  - HEAD: Returns no data, only the information about the data (Headers)
  - POST: URL of resource that can deal with data included in the body
  - PUT, DELETE, OPTIONS, TRACE, CONNECT

Figure 5.6
HTTP request message

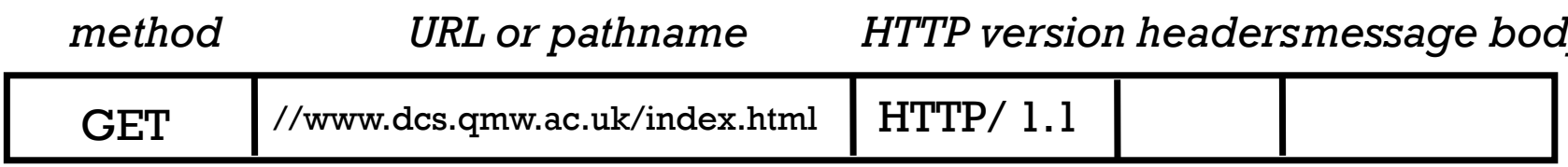
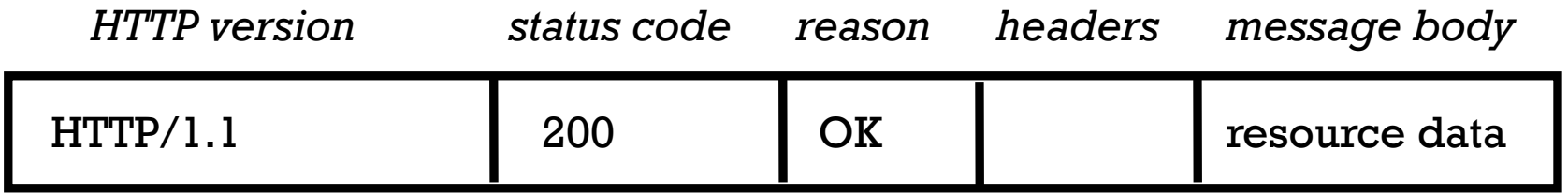


Figure 5.7
HTTP *Reply* message



Remote Procedure Call (RPC)

Programming of distributed systems looks like conventional programming

Remote Procedure Call

- Making programming of distributed systems look alike conventional programming
- A major intellectual breakthrough !!!
- Procedures (functions) can be called on remote machines as though they were residing on the local machine
- Encoding/decoding parameters and return values and the semantics of procedure calls remains hidden
- Birrell and Nelson 1984

Remote Procedure Call
Design issues

- Style of programming promoted by RPC
  - *Programming with interfaces*
- Call semantics
  - *Equivalent to a local procedure*
- Transparency
  - Location and access

Remote Procedure Call
Programming with interfaces

- Each module has its interface
- Modules are implemented so that they hide everything internal but the interface
- Implementation may be changed IF it does not affect the interface
- Service interface (CN)
  - *Specification of procedures offered by a server*
  - Cannot specify direct access to variables
  - Call by reference is not supported
    - Separated addressing spaces
    - Addresses are not reusable
- Interface Definition Languages (IDL)
  - Benefit programs written in different languages
  - Devised so that interoperation among different languages is guaranteed
  - Applies to RPC and to RMI as well
  - CORBA IDL, Sun XDR, WSDL (WS), Google protocol buffers

Figure 5.9
RPC Call semantics

- In Local Procedure Calls the semantics is

exactly once

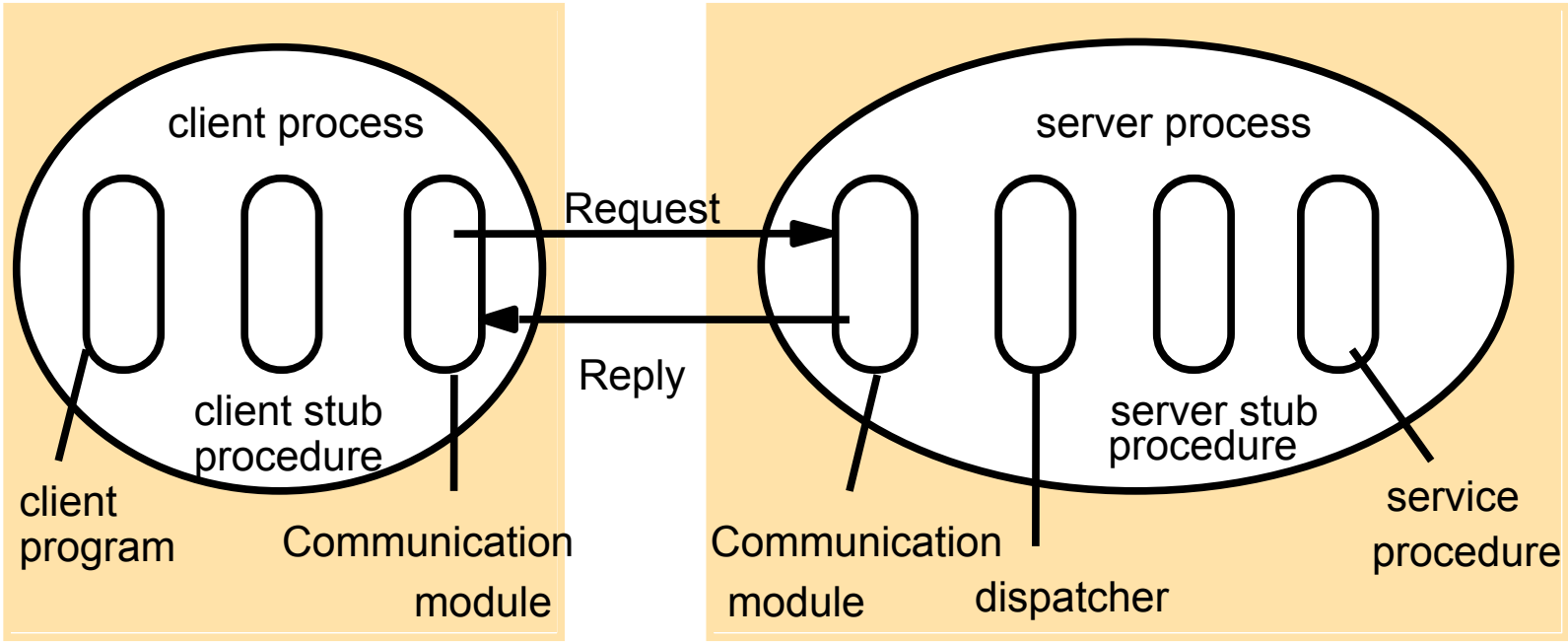
Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

Figure 5.9
RPC Transparency

- Creators of RPC aimed to make RPC calls very much like local calls
- i.e. RPC aims to offer:
  - Location transparency
  - Access transparency
- RPC calls, however, are more vulnerable than local calls
  - Network failures
  - Computers, processes, etc.
  - Network latency (Several orders of magnitude higher)
- RPC calls must recover from those failures

Figure 5.10

Role of client and server **stub** procedures in RPC



Remote Method Invocation

Programming of distributed systems looks like OO programming

Remote Method Invocation

Object that calls a method of a remote object

- Interfaces
- Transparency: access, location
- Object-oriented
  - Objects, classes and inheritance, methodologies, tools
  - All RMI objects have a unique object references (Local or remote)
  - More powerful parameter passing semantics than in RPC
- Parameter passing: By value or by object-reference

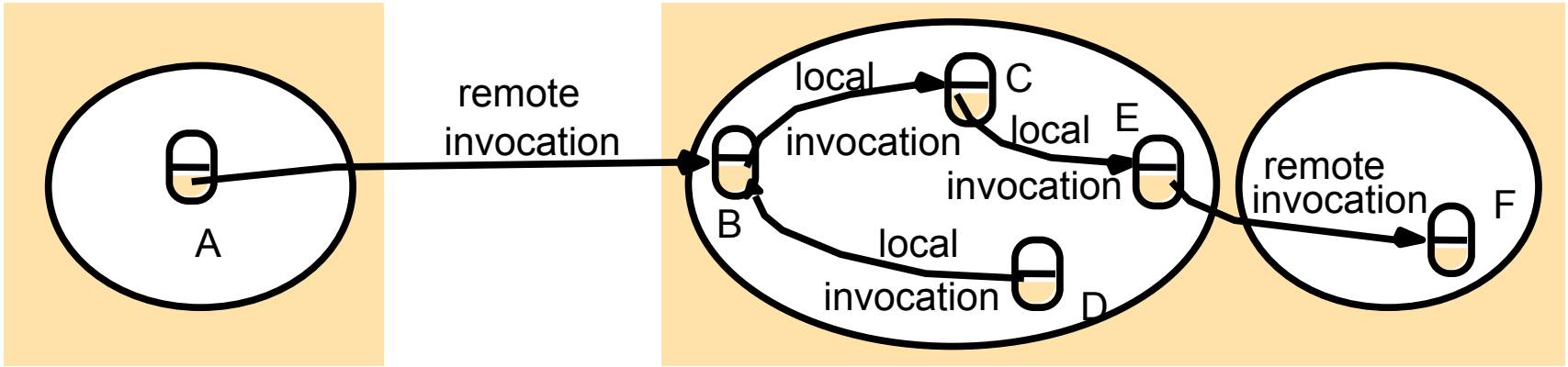
Remote Method Invocation
Design issues for RMI (I/II)

- Object model
  - Object references
  - Interfaces
  - Actions: An object invokes a method on another object
  - Exceptions
  - Garbage collection
- Distributed objects
  - May adopt a C/S architecture
  - C/S: Servers and clients invoke remote methods
  - Can be replicated: fault tolerance, better performance, objects may migrate
  - Distributed 'encapsulation': The state of an object in RMI can only be accessed via method invocation

Remote Method Invocation
Design issues for RMI (II/II)

- Distributed object model
  - A **remote object** is an object that can receive remote invocations
- Remote object references: may be passed as arguments
  - May be returned as rmi return values
- Remote interfaces: The methods exported by a remote object

Figure 5.12
Remote and local method invocations



Remote method invocations

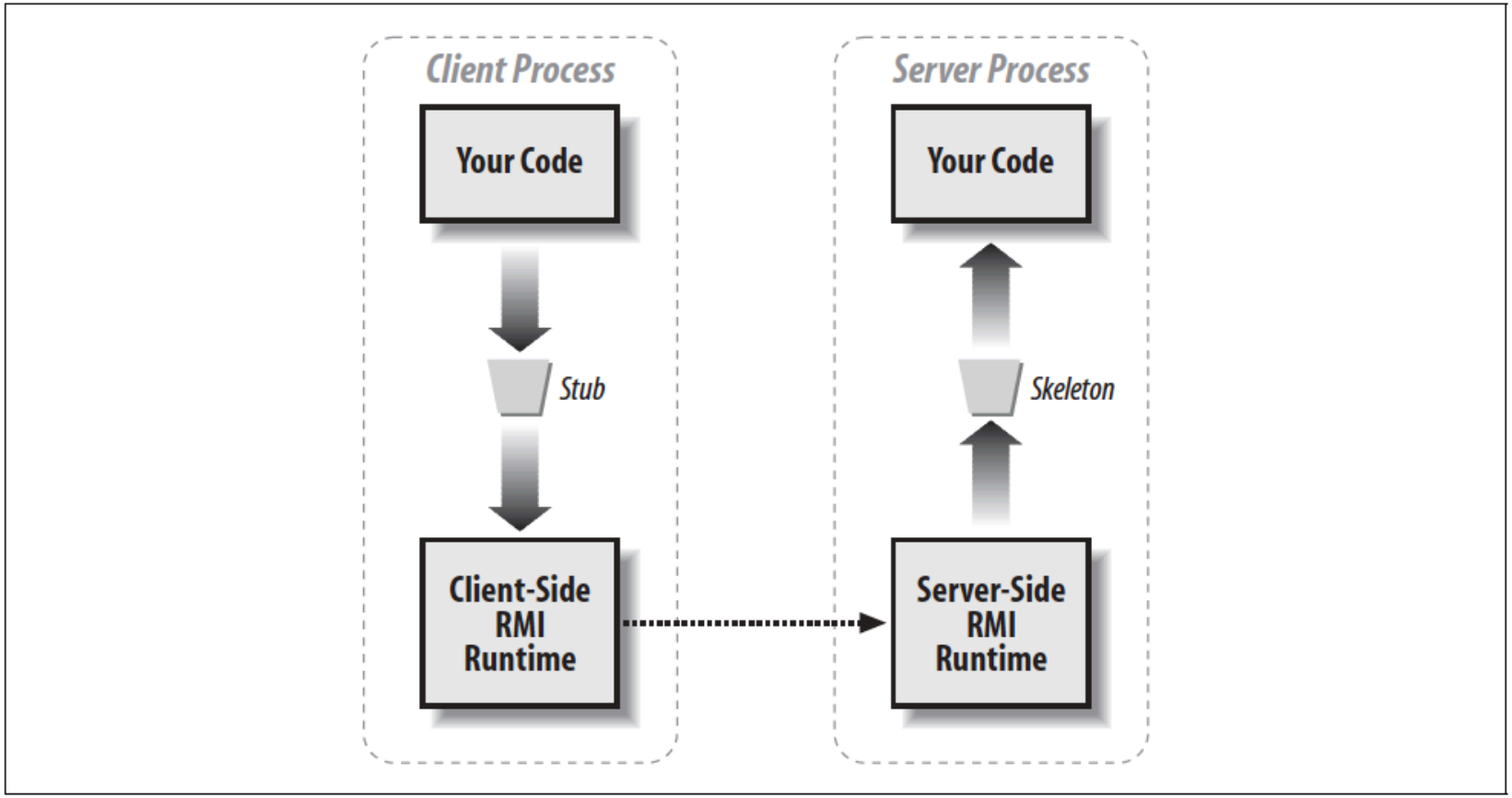
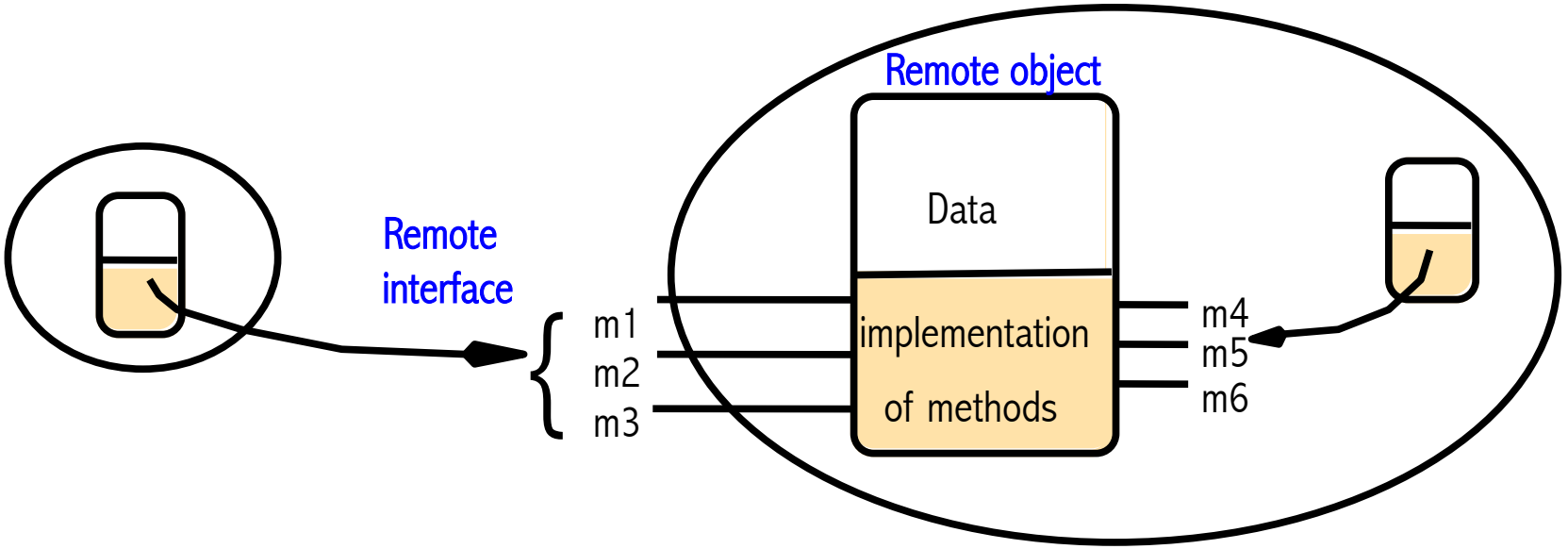


Figure 16-1. The layers in a remote call

Java RMI (O'Reilly)

Figure 5.13

A remote object and its remote interface



Implementation of RMI

Constituent parts

- Communication module: RR protocol, marshalling and unmarshalling, invocation semantics (*at-most-once*)
- Remote reference module: Translating between local to remote references
  - Proxy (= Stub class): Client-side stand-in for the remote interface
- Servant: Instance of a class that provides the body of a **remote object**

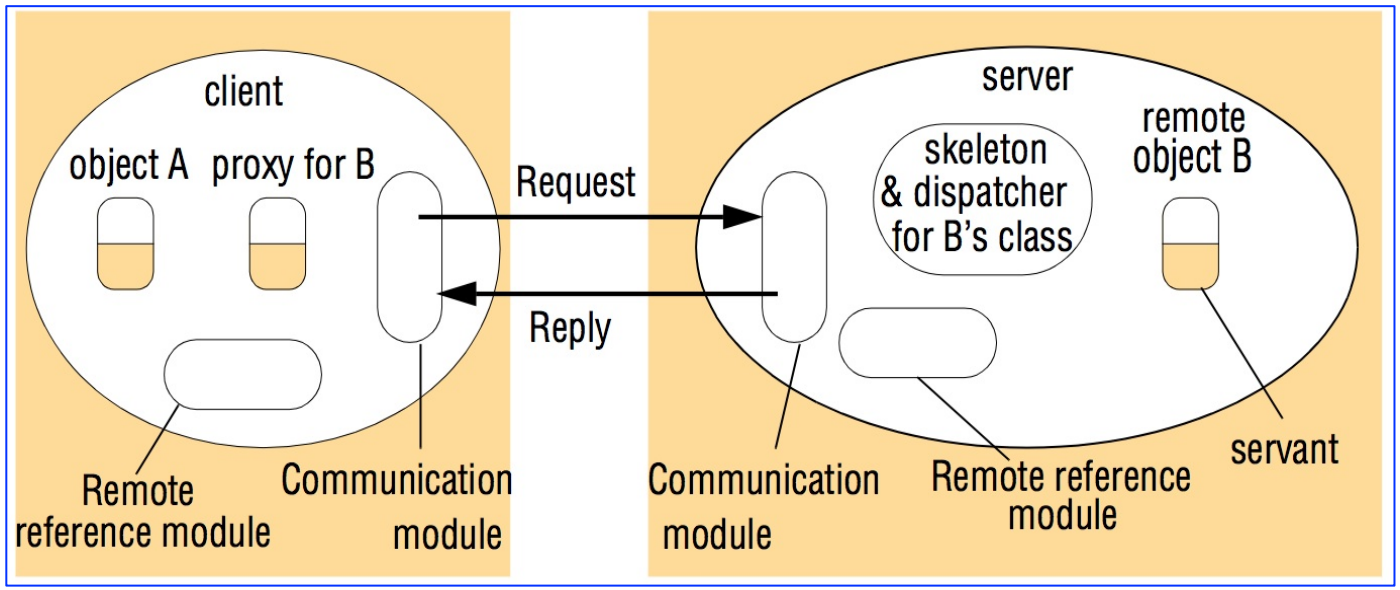
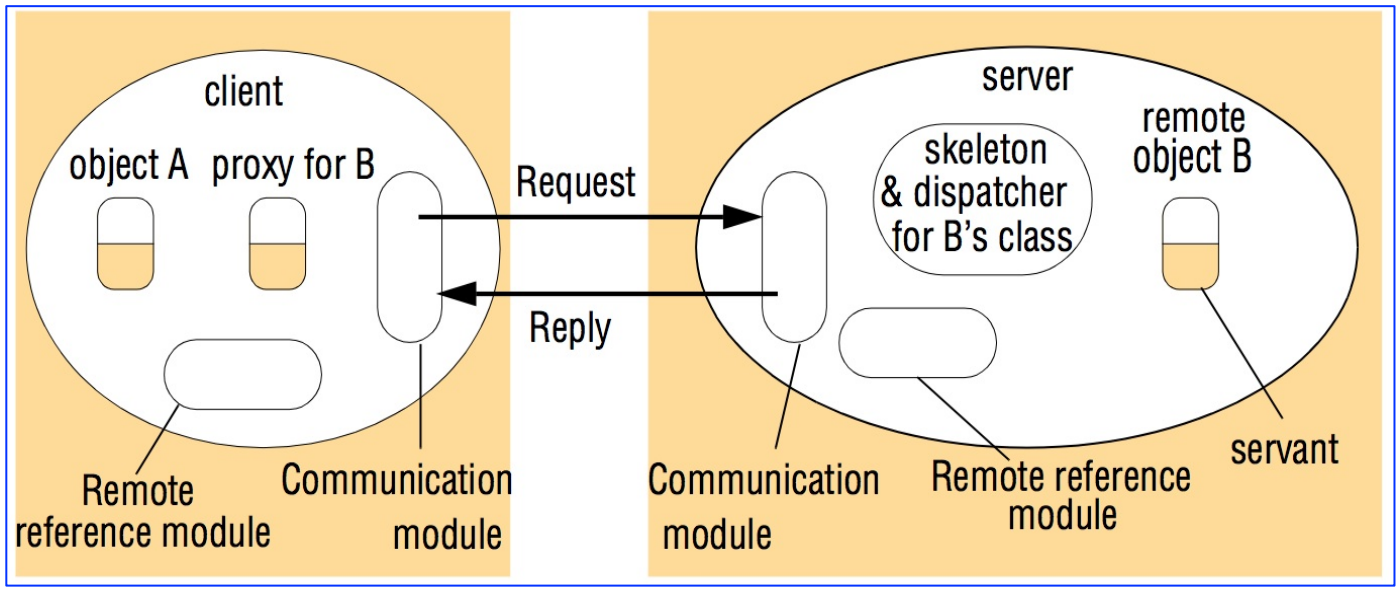


Figure 5.15

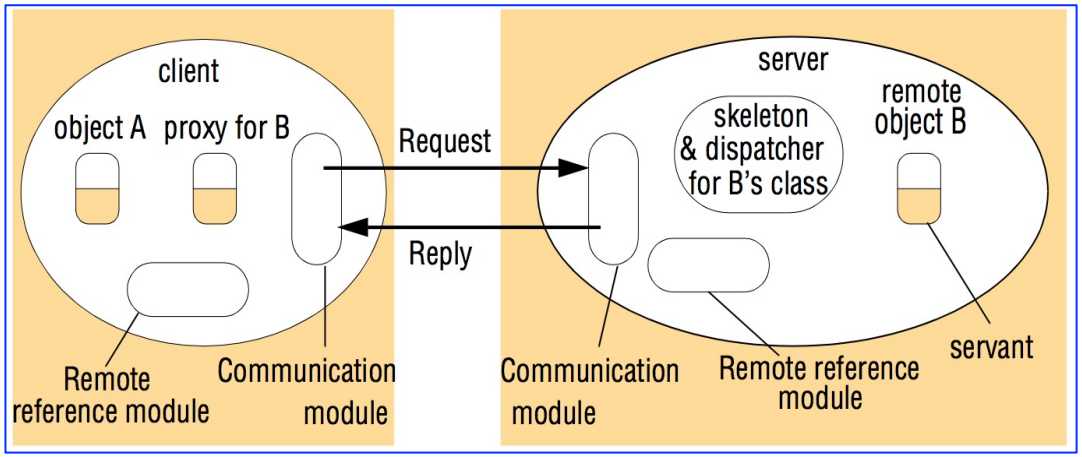
The role of proxy and skeleton in remote method invocation

- Proxy (Stub): Make remote invocation transparent for clients, a stand-in for a remote object's interface
- Dispatcher: A server has one dispatcher and one skeleton for each class representing a remote object
- Skeleton (Stub): Implements the methods on the remote interface
  - Marshalls and unmarshalls the arguments in the request message and invokes the corresponding method in the servant



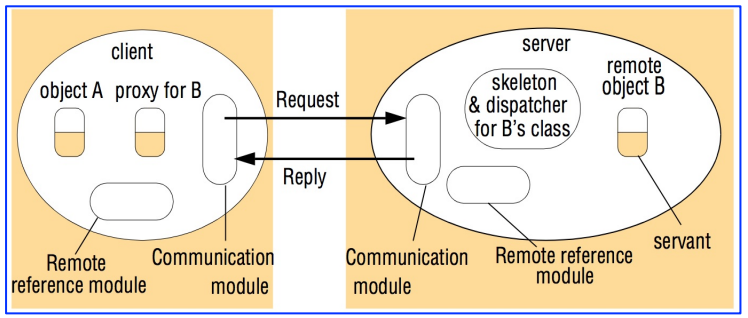
Remote Method Invocation
Other aspects (I/III)

- Generation of proxies, dispatchers and skeletons
  - JAVA RMI: **$ rmic** *after compiling the classes*
  - *Almost transparent in Java 1.6*
- Java 1.6 and 1.7:
  - Automatic generation of proxies and skeletons (Stubs)
  - Automatic client-download of proxies by way of the **rmiregistry** in cooperation with a web server
    - Java Stubs



Remote Method Invocation
Other aspects (II/III)

- Factory Methods and Factory Objects
  - Remote interfaces cannot include constructors
  - Servants cannot be created by remote invocation on constructors ☺
- Factory method: An object's method that is used to make objects
  - A method that creates servants
  - If a client requests the creation of an object it resorts to a remote factory method
- Threads: Servants use them to increase throughput and guarantee server uptime
  - Be careful with global variables
- Objects may be dormant (Not activated)
  - They get activated upon firts request in an automatic way
  - Java Activation Framework



Remote Method Invocation
Other aspects (III/III)

- Distributed Garbage Collection
  - Reference counting
- Remote object reference enters a process:
  - A proxy is created
  - If the proxy leaves because the remote object has left, the hosting process must be informed
- Works in cooperation with the Local Garbage Collector
  - This protocol is RR with at-most-once call semantics

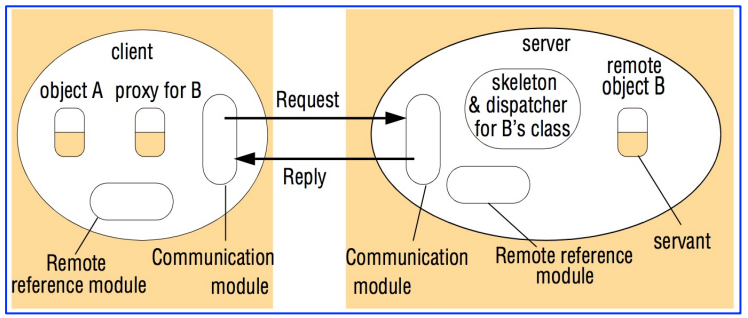


Figure 5.17

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, as shown in Figure 15.18, line 3.

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.

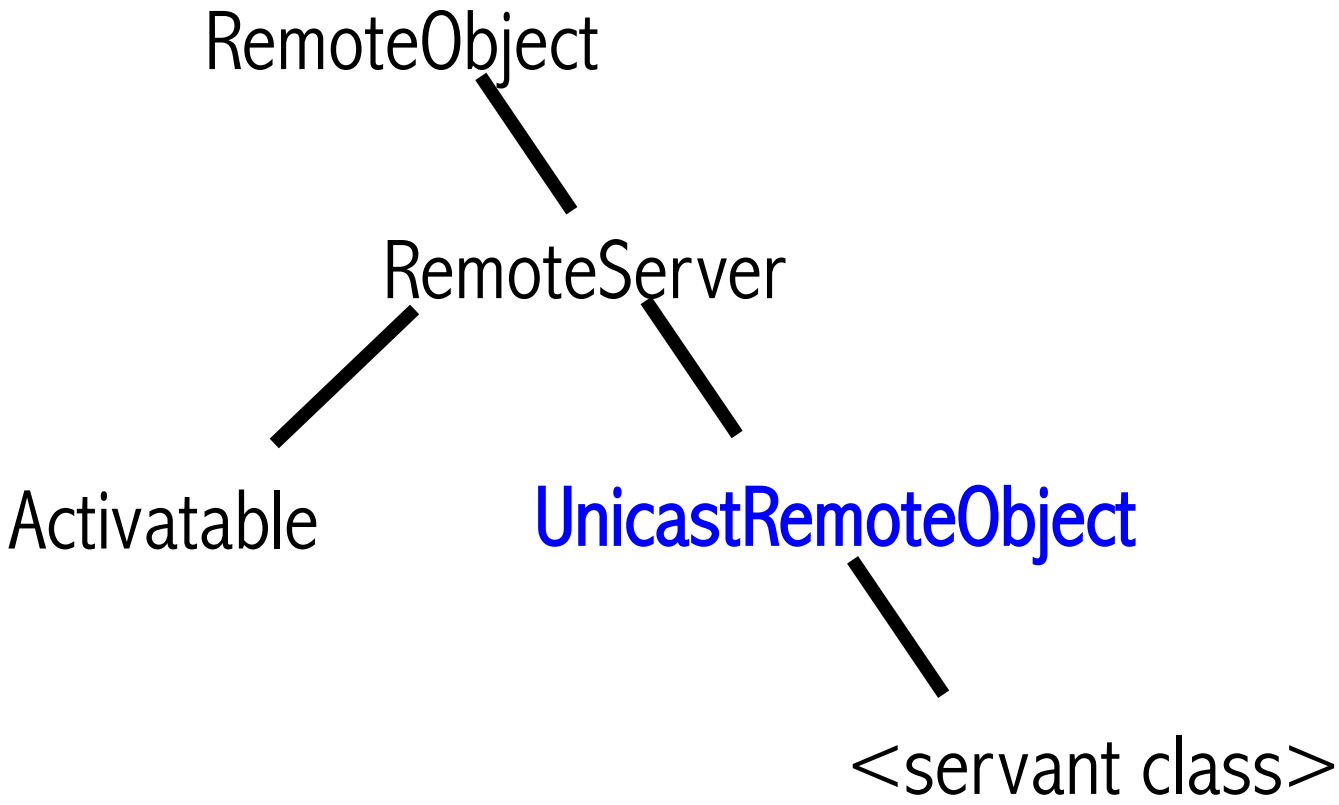
Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure 5.20 line 1. A remote object reference is returned.

String [] list()

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

Figure 5.21
Classes supporting Java RMI



```
package basicrmi;

import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.logging.Level;
import java.util.logging.Logger;

public class Server {
    public static void main(String[] args) {

        //An RMI registry
        Registry registry;

        try {

            //The object we are exporting to the network -the remote object
            basicrmi.common.ExampleMethodsImpl emi = new basicrmi.common.ExampleMethodsImpl();

            //Create an rmi registry instance to listen on TCP port 50001
            registry = LocateRegistry.createRegistry(50001);

            //Export the newly created object on TCP port 50002
            basicrmi.common.ExampleMethods ccstub = (basicrmi.common.ExampleMethods) UnicastRemoteO

            //The registry name of this remote server is "aslan"
            String sn = "aslan";

            //Register the exported object into the RMI registry with key "aslan"
            registry.rebind(sn, emi);
            ...
```

```
            //Register the exported object into the RMI registry with key "aslan"
            registry.rebind(sn, emi);

            System.out.println("------------------------");
            System.out.println(sn +" Server running");
            System.out.println("------------------------");

            System.out.println("Listing of current registry entries:");

            String names[] = registry.list();

            for (int i = 1; i < names.length + 1; i++) {
                System.out.println(i + ": " + names[i - 1]);
            }

            System.out.println("------------------------");

        } catch (RemoteException ex) {
            Logger.getLogger(Server.class.getName()).log(Level.SEVERE, null, ex);
            return;
        }
    }
}
```

```
package basicrmi;

/**
 * ************************************************************************
 * Universidad de León, EIII Grado en Ingeniería Informática Asignatura de
 * Arquitectura de Sistemas Distribuidos (C) 2013 José María Foces Morán,
 * instructor.
 *
 * Client.java, v01.03
 *
 * This class is the client that will execute the example remote methods.
 *
 * PRECONDITIONS FOR EXECUTION OF THIS PROGRAM
 * ------------------------------------------
 *
 * This program needs that a security policy file be specified in the java command
 * line, that file should contain a java security policy file that allows the
 * run time to dynamically load java code from anywhere, specifically, it needs
 * to download and execute the proxy classes from the remote server by way of
 * the remote registry. In order for the user to express this permission, a
 * command line like the one following would be fine:
 *
 * $ java -Djava.security.policy="all.policy" basicrmi.Client \
 *                                            protocol.unileon.es 50001 aslan
 *
 * The policy file (all.policy) contents required in this case would be:
  grant{
        permission java.security.AllPermission;
  };
 ************************************************************************/
```

```
public class Client {
private static final int HOSTNAMEIDX = 0;
private static final int REGISTRYPORTIDX = 1;
private static final int RMINAMEIDX = 2;

    public static void main(String args[]) {
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new SecurityManager());
        }

        try {

            Registry reg = LocateRegistry.getRegistry(args[HOSTNAMEIDX],
                                                      Integer.parseInt(args[REGISTRYPORTIDX]));

            String names[] = reg.list();

            System.out.println("Remote objects registered at "+args[HOSTNAMEIDX]+",
            port "+args[REGISTRYPORTIDX]+":");
            for(int i = 0; i < names.length; i++){
                System.out.println("\t" + names[i]);
            }

            ExampleMethods cc = (ExampleMethods) reg.lookup(args[RMINAMEIDX]);

            String ts = "Hello world from Java RMI!!!";

            System.out.println("Hash of " + ts +" = "+ cc.longStringHash(ts));
            System.out.print("Computing 5!  = " + cc.factorial(5));
            System.out.println(". Received from: "+args[HOSTNAMEIDX]+"/"+args[RMINAMEIDX]);

        } catch …
```

FIN

Focus of chapter 4
Middleware layers

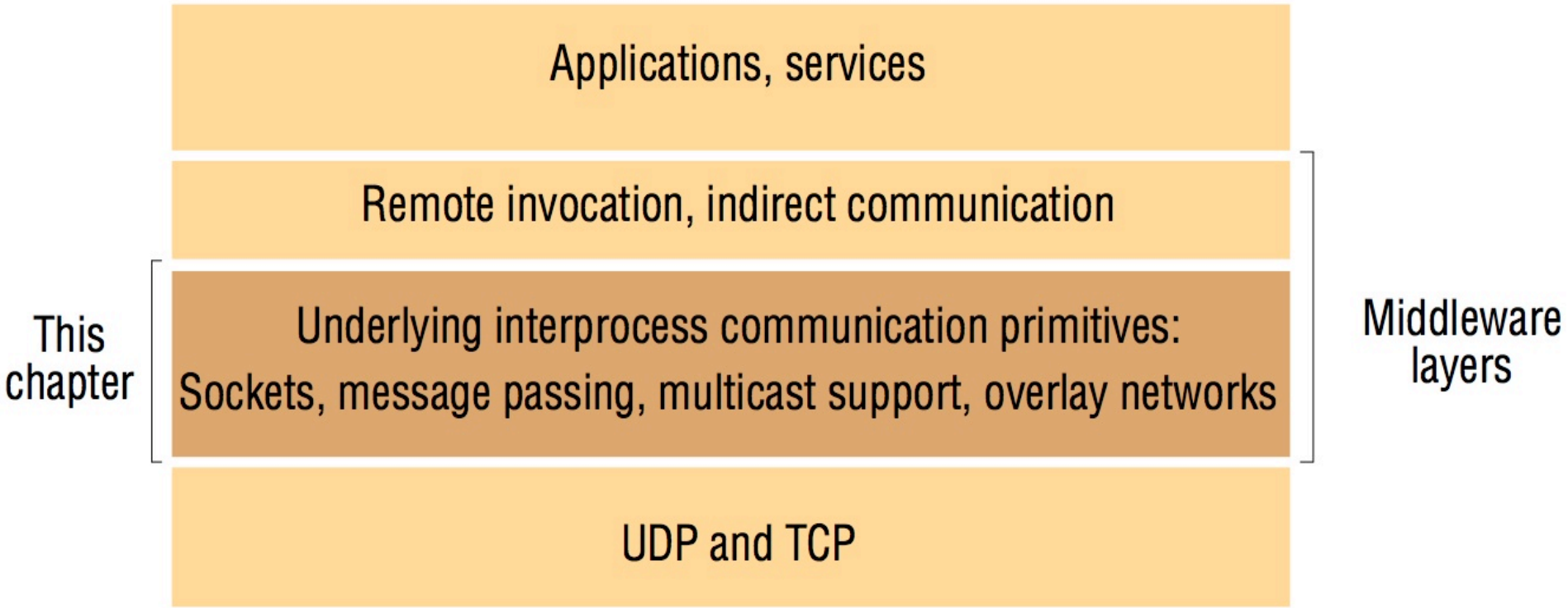


Figure 5.3 Operations of the request-reply protocol

- Each request is identified by a requestId:
  - doOperation() in the client generates a requestId for each message
  - The server copies the requestId into the corresponding reply
  - Thus, doOperation() can check for delayed earlier calls

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.

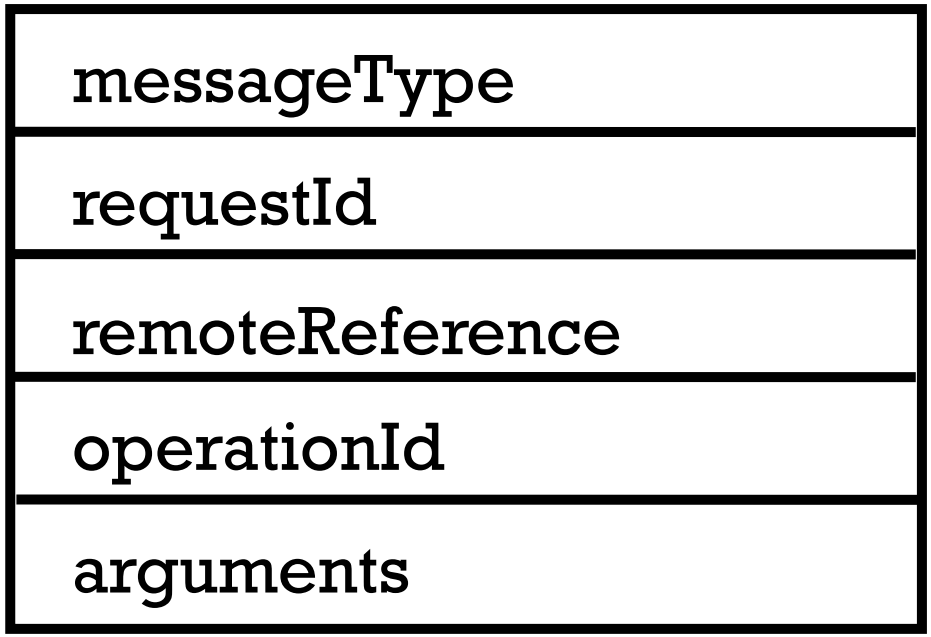
public byte[] getRequest ();

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.

Figure 5.4
Request-reply message structure



int (0=Request, 1= Reply)

int

RemoteRef

int or Operation

array of bytes

Figure 5.8
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();
};
```

Figure 5.1: Files interface in Sun XDR.

An IDL and type definition language

```
const MAX = 1000;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;
struct Data {
        int length;
        char buffer[MAX];
};
struct writeargs {
        FileIdentifier f;
        FilePointer position;
        Data data;
};
```

```
struct readargs {
        FileIdentifier f;
        FilePointer position;
        Length length;
};

program FILEREADWRITE {
   version VERSION {
        void WRITE(writeargs)=1;
        Data READ(readargs)=2;
   }=2;
} = 9999;
```

Figure 5.14
Instantiation of remote objects