Remote invocation

Programming locally vs programming remotely

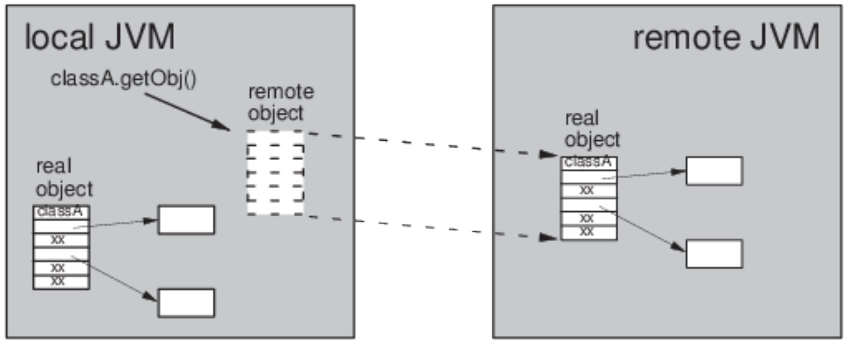
Real concurrency (vs interleaving)

Remote variables?

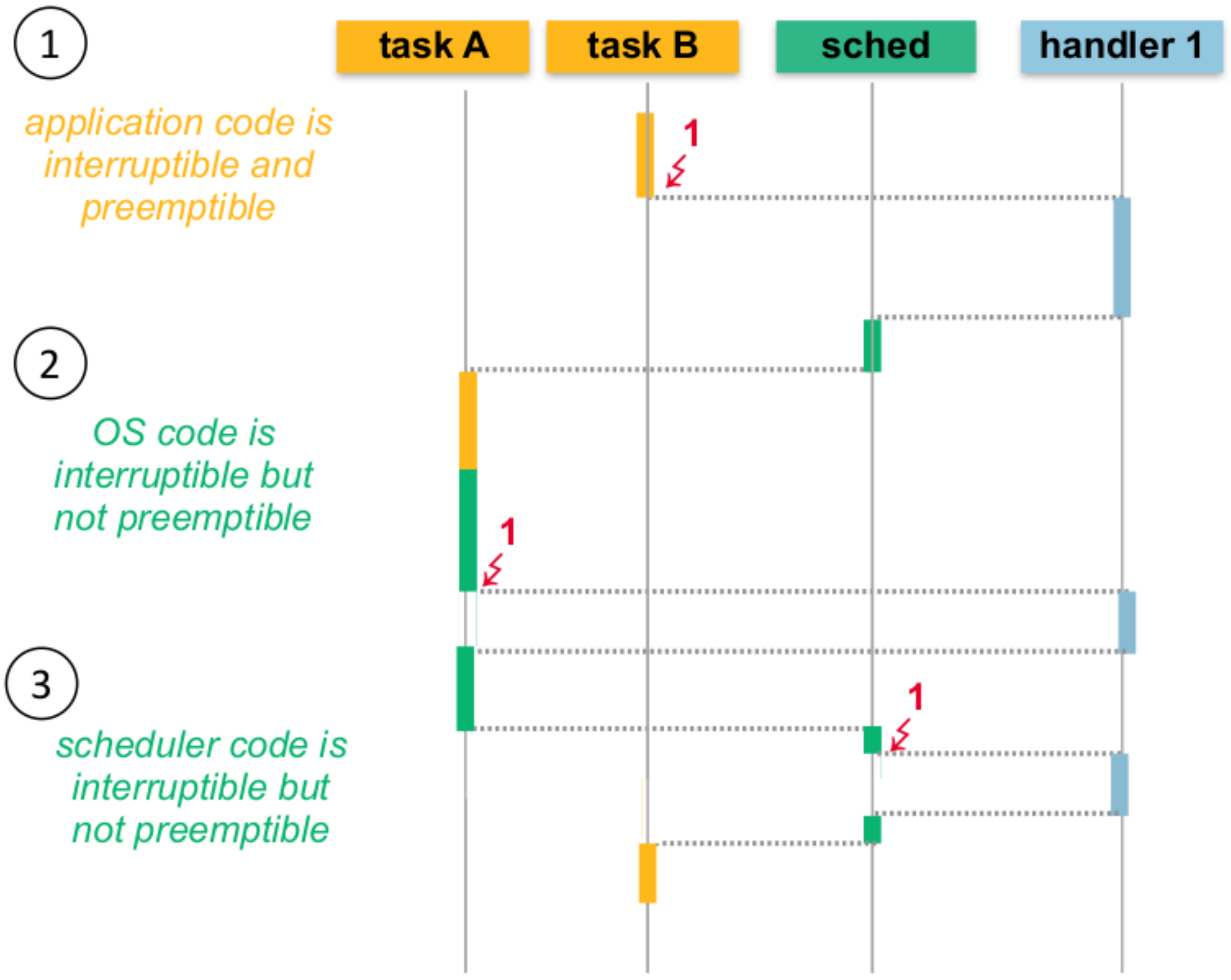
Remote procedures?

Remote data?

Mobile code, data, objects?



Concurrency by Interleaving


1
application code is interruptible and preemptible
2
OS code is interruptible but not preemptible
3
scheduler code is interruptible but not preemptible
task A
task B
sched
handler 1
1
1
1

Process interaction

Awareness	Relationship	Influence that a process has on the other	Potential control problems
Processes are unaware of each other	Competion (over resources)	• No dependency • Timing maybe affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware (eg. Shared object)	Cooperation by sharing	• dependency producer-consumer • Timing maybe affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• dependency producer-consumer • Timing maybe affected	• Deadlock (consumable resource) • Starvation

Agenda

Request-reply protocols

Remote procedure call

Remote method invocation

Case study: Java RMI

RPC and RMI

There are two main remote invocation techniques for communication in distributed systems:

- The remote procedure call (RPC) approach extends the programming abstraction of the procedure call to distributed environments, allowing a calling process to call a procedure in a remote node as if it were local
- Remote method invocation (RMI) is similar to RPC but for **distributed objects**, with added benefits in terms of using oo programming concepts: extending the concept of an object reference to distributed environments, and allowing the use of object references as parameters in remote invocations.

Implementation issues

Transparency property

- Syntactic transparency
- Semantic transparency

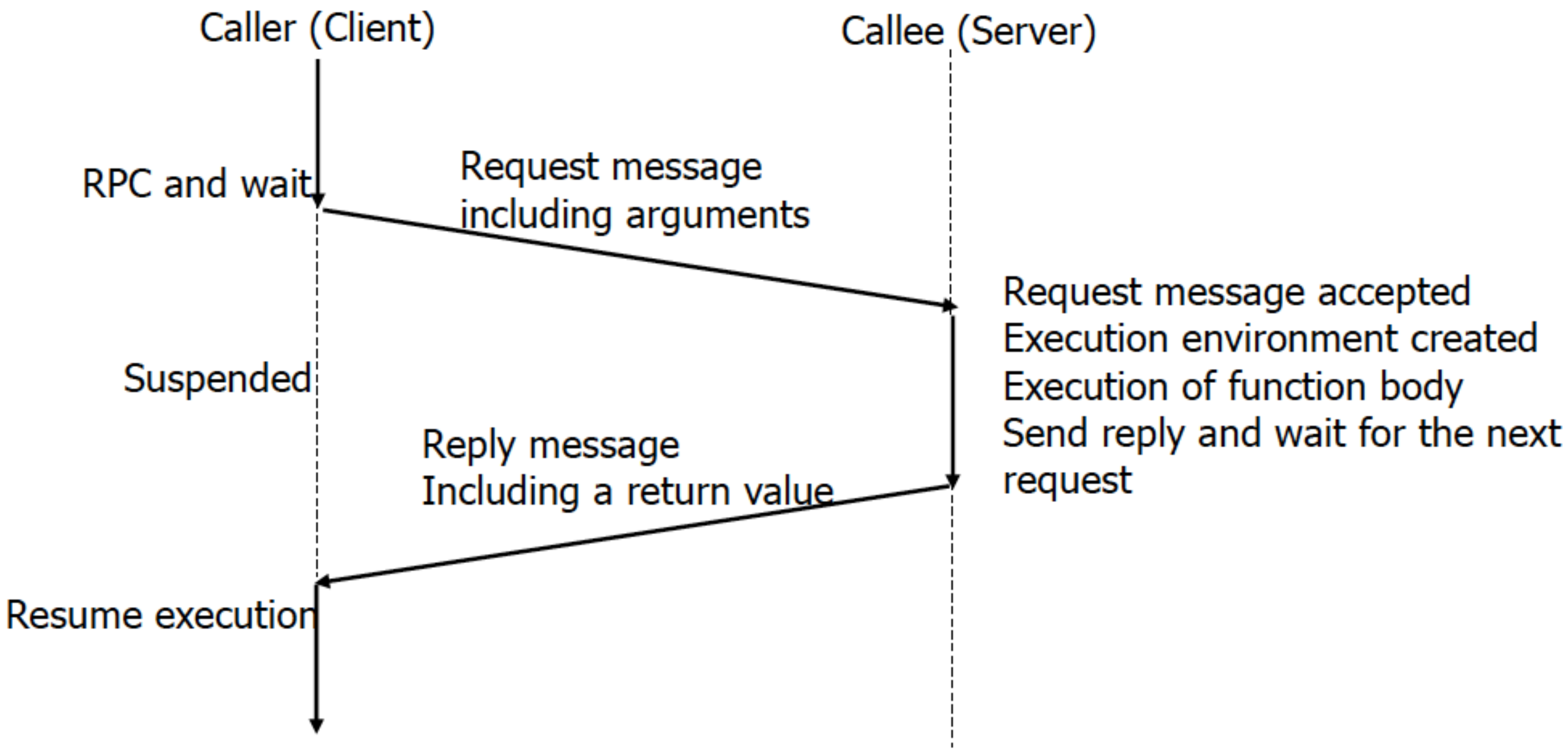
Analogy in semantics between local and remote procedure calls

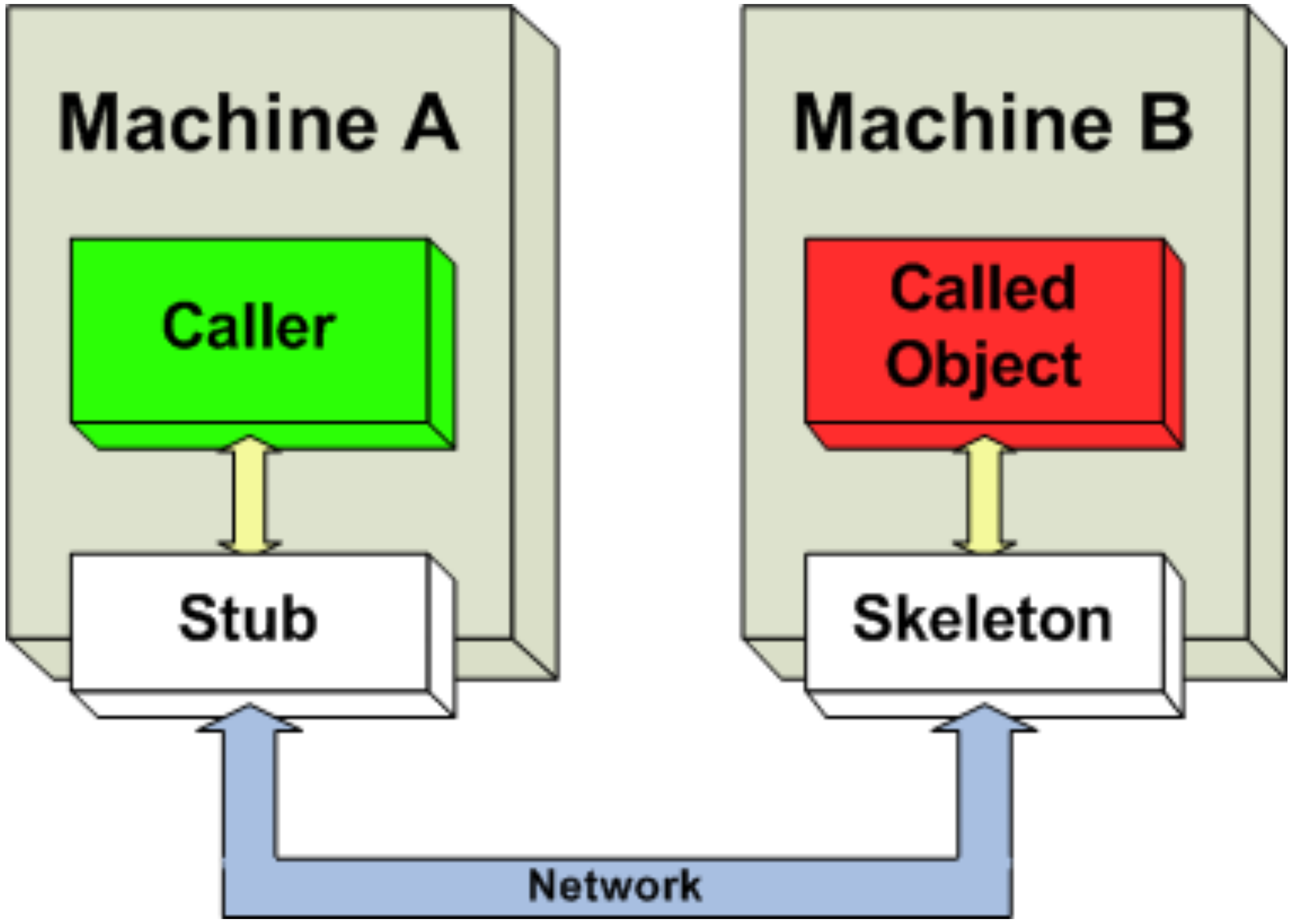
- Caller capable of passing arguments (Automatic marshalling)
- Caller suspended till a return from a function
- Callee capable of returning a value to caller

Difference in semantics between local and remote procedure calls

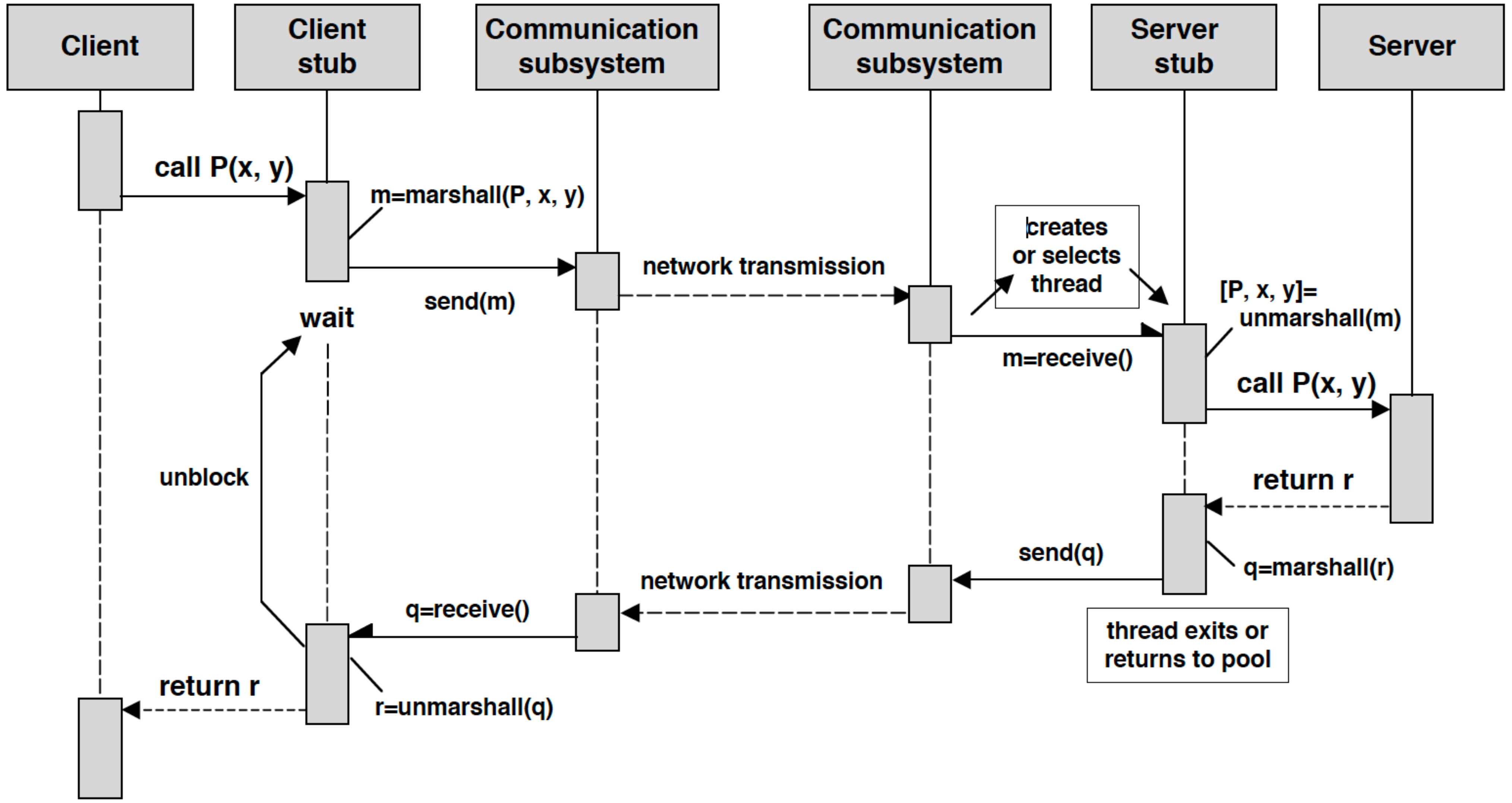
- No call by reference and no pointer-involved arguments
- Error handling required for communication (Ex. RemoteException in Java)
- Performance much slower than local calls

RPC and RMI similarities

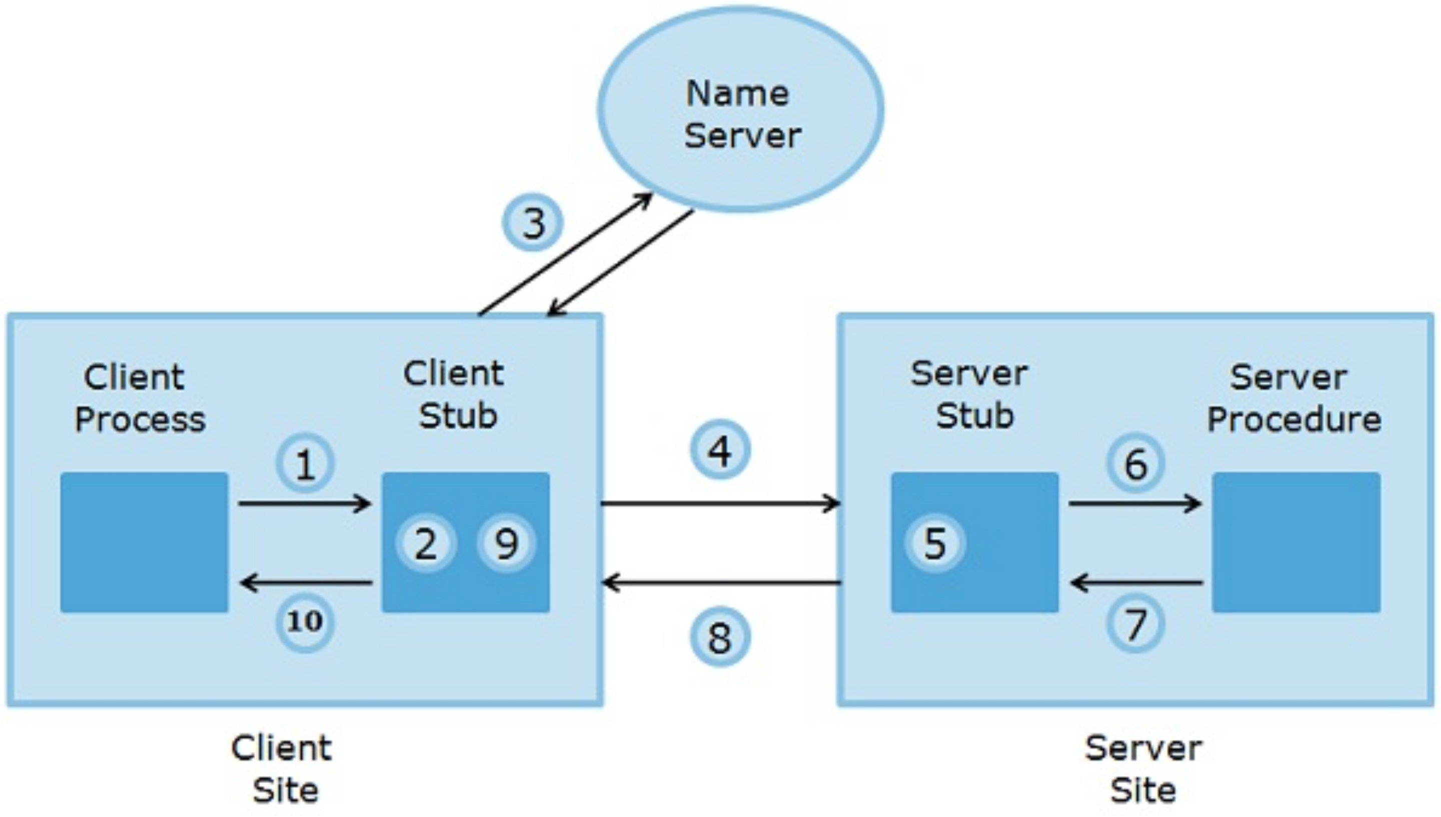



Machine A
Caller
Stub
Machine B
Called Object
Skeleton
Network

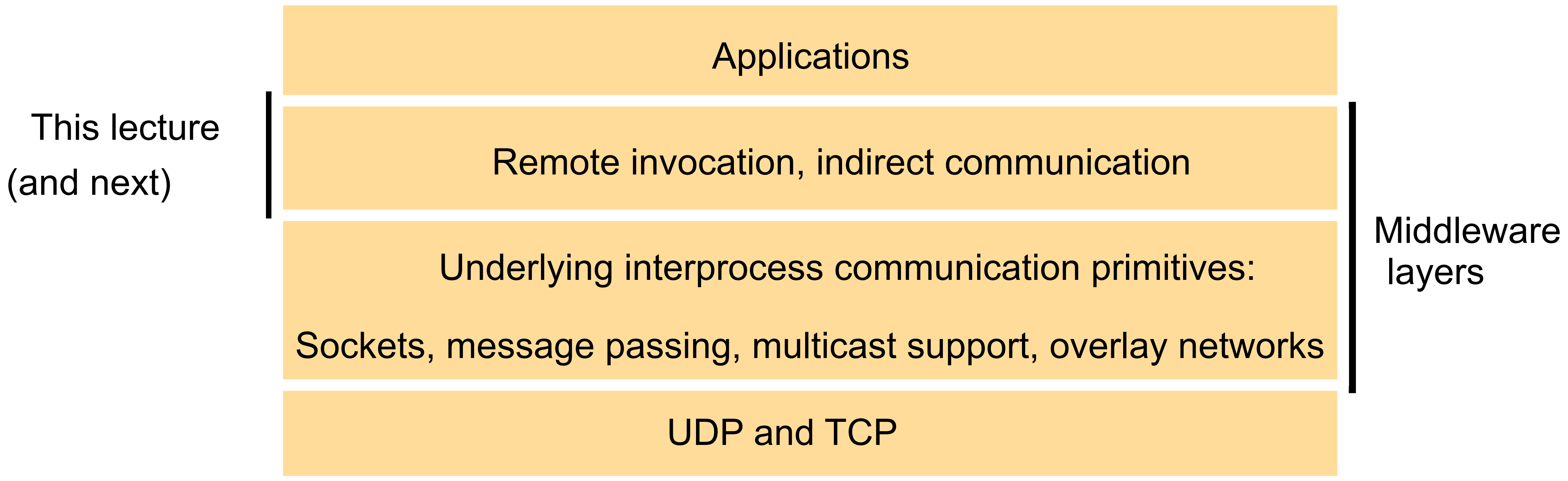
RPC: overall flow of control



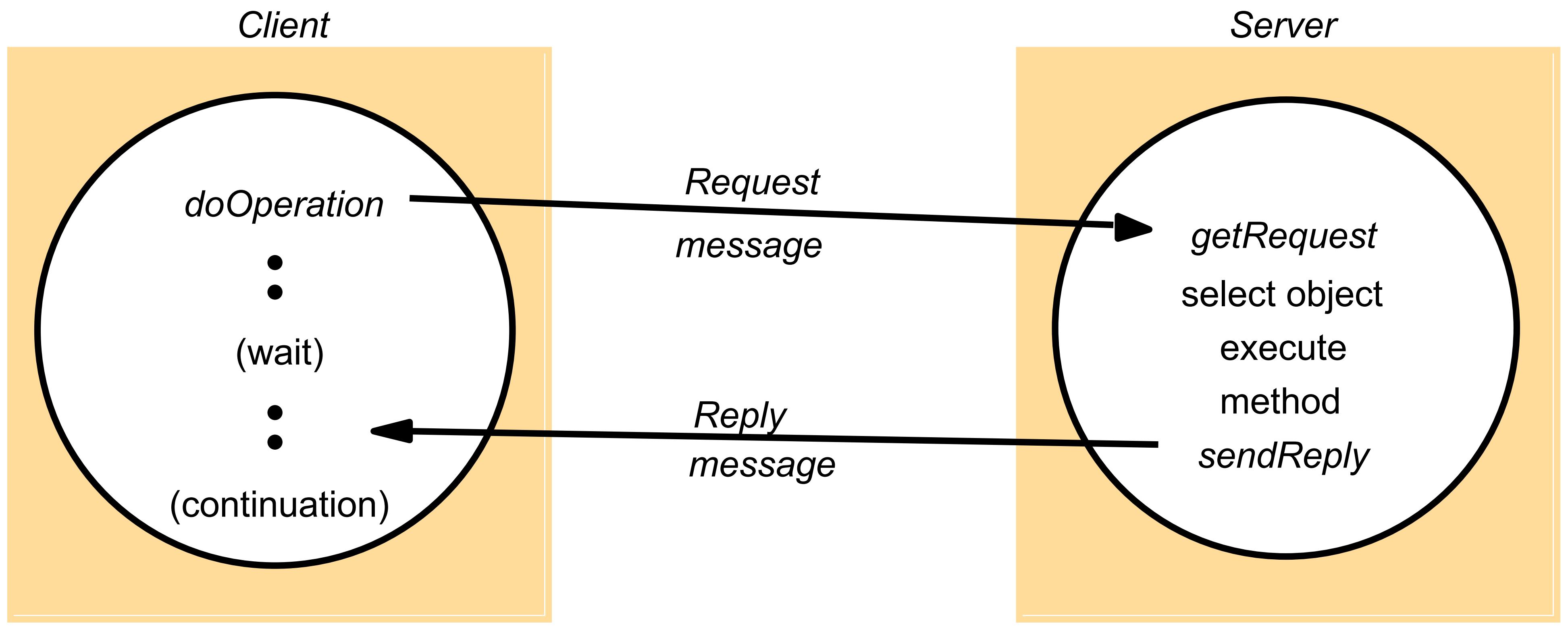
RPC


Name Server
3
Client Process
Client Stub
1
2
9
10
4
8
Server Stub
5
Server Procedure
6
7
Client Site
Server Site

Middleware layers


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

Request-reply communication



Operations of the request-reply protocol

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.

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*array of bytes*

RPC exchange protocols

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

The protocols produce differing behaviours in presence of communication failures and are used for implementing various types of request behaviour:

- the request (R) protocol;
- the request-reply (RR) protocol;
- the request-reply-acknowledge reply (RRA) protocol.

HTTP as a RR protocol

HTTP is implemented over TCP

In the original version of the protocol, each client-server interaction consisted of the following steps:

1. The client requests and the server accepts a connection at the default server port or at a port specified in the URL.
2. The client sends a request message to the server.
3. The server sends a reply message to the client.
4. The connection is closed

HTTP methods: GET

- Each client request specifies the name of a **method** to be applied to a resource at the server and the URL of that resource.
- The reply reports on the status of the request. Requests and replies may also contain resource data, the contents of a form or the output of a program resource run on the web server.

GET: Requests the resource whose URL is given as its argument. If the URL refers to data, then the web server replies by returning the data identified by that URL. If the URL refers to a program, then the web server runs the program and returns its output to the client.

With GET, all the information for the request is provided in the URL

HTTP methods

HEAD: This request is identical to GET, but it does not return any data. However, it does return all the information about the data, such as the time of last modification, its type or its size

POST: Specifies the URL of a resource (for example a program) that can deal with the data supplied in the body of the request. The processing carried out on the data depends on the function of the program specified in the URL. This method is used when the action may change data on the server.

PUT: Requests that the data supplied in the request is stored with the given URL as its identifier, either as a modification of an existing resource or as a new resource.

DELETE: The server deletes the resource identified by the given URL. Servers may not always allow this operation, in which case the reply indicates failure. OPTIONS: The server supplies the client with a list of methods it allows to be applied to the given URL (for example GET, HEAD, PUT) and its special requirements.

TRACE: The server sends back the request message. Used for diagnostic purposes.

HTTP methods

- The operations PUT and DELETE are idempotent, but POST is not necessarily so because it can change the state of a resource.
- The others are safe operations in that they do not change anything

HTTP request message (method GET)

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

HTTP *Reply* message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Design issues for RPC

Three issues are important :

1. the style of programming promoted by RPC –that is, programming with interfaces;
2. the call semantics associated with RPC;
3. the issue of transparency and how it relates to remote procedure calls.

Interface description language

- An interface description language (IDL), is a language used to describe a software component's application programming interface (API).
- IDLs describe an interface in a language-independent way, enabling communication between software components that do not share one language.
- For example, between those written in C++ and those written in Java.

https://docs.oracle.com/javase/7/docs/technotes/guides/idl/jidlExample.html

Interface based programming

- The use of interfaces to allow teams to collaborate raises the question of how interface changes happen in interface-based programming. The problem is that if an interface is changed, e.g. by adding a new method, old code written to implement the interface will no longer compile – and in the case of dynamically-loaded or linked plugins, will either fail to load or link, or crash at runtime.
- There are two basic approaches for dealing with this problem:
  1. a new interface may be developed with additional functionality, which might inherit from the old interface
  2. a software versioning policy may be imposed to interface implementors, to allow forward-incompatible, or even backward-incompatible, changes in future "major" versions of the platform
- Both of these approaches have been used in the Java platform.

Interface definition language (IDL)

- An RPC mechanism can be integrated with a programming language if it includes a notation for defining interfaces, allowing input and output parameters to be mapped onto the language’s normal use of parameters.
- This approach is useful when all the parts of a distributed application can be written in the same language. It is also convenient because it allows the programmer to use a single language for local and remote invocation.
- However, many existing useful services are written in different languages. It would be beneficial to allow programs written in a variety of languages, to access them remotely.
- Interface definition languages (IDLs) are designed to allow procedures implemented in different languages to invoke one another: an IDL provides a notation for defining interfaces in which each of the parameters of an operation may be described as for input or output in addition to having its type specified

IDL

- An *interface description* (or *definition*) *language* (IDL), is a specification language used to describe a software component's application programming interface (API).
- IDLs describe an interface in a language-independent way, enabling communication between software components written in different languages.

IDL: examples

- AIDL: Java-based, for Android; supports local and remote procedure calls, can be accessed from native applications by calling through Java Native Interface (JNI)
- Apache Thrift: from Apache, originally developed by Facebook
- Data Distribution Service: In DDS IDL was identical to OMG IDL until version 3.5 when it branched in order to evolve independently of the OMG specification
- JSON Web-Service Protocol (JSON-WSP)
- Microsoft Interface Definition Language (MIDL) extension of OMG IDL to add support for Component Object Model (COM) and Distributed Component Object Model (DCOM)
- OMG IDL: implemented in CORBA for DCE/RPC services, also selected by the W3C for exposing the DOM of XML, HTML, and CSS documents
- OpenAPI Specification: a standard for REST interfaces.
- Protocol Buffers: Google's IDL
- RESTful Service Description Language (RSDL)
- Universal Network Objects: OpenOffice.org's component model
- Web Application Description Language (WADL)
- Web IDL: can be used to describe interfaces intended for implementing in web browsers
- Web Services Description Language (WSDL)

A simple 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();
};
```

- The interface named PersonList specifies the methods available for RMI in a remote object that implements that interface.
- For example, the method addPerson specifies its argument as in , meaning that it is an input argument, and
- the method getPerson that retrieves an instance of Person by name specifies its second argument as out , meaning that it is an output

RPC call semantics

doOperation can be implemented in different ways to provide different delivery guarantees. The main choices are:

- Retry request message: Controls whether to retransmit the request message until either a reply is received or the server is assumed to have failed.
- Duplicate filtering: Controls when retransmissions are used and whether to filter out duplicate requests at the server.
- Retransmission of results: Controls whether to keep a history of result messages to enable lost results to be retransmitted without re-executing the operations at the server.

Combinations of these choices lead to a variety of possible semantics for the reliability of remote invocations as seen by the invoker

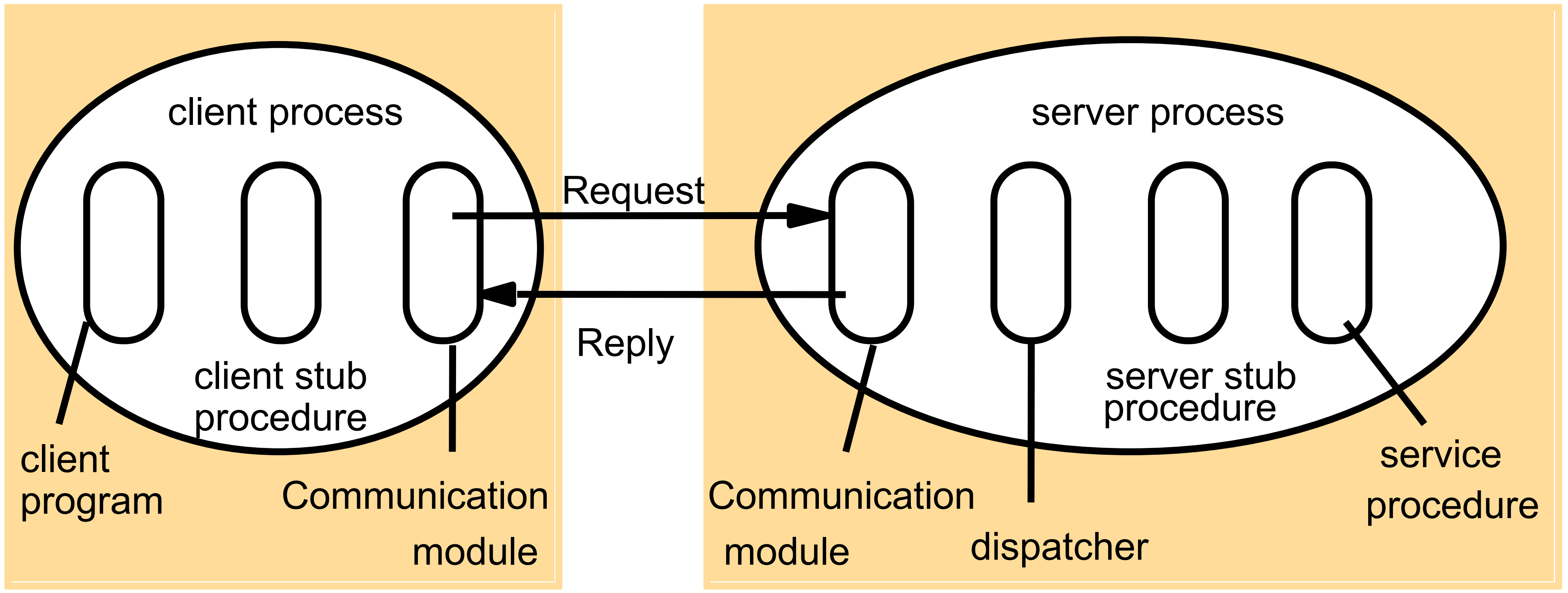
Call semantics in languages

Fault tolerance measures				
Retransmit request message	Duplicate filtering	Re-execute procedure or Retransmit reply	Definition of Invocation Semantics	
No	Not applicable	Not applicable	**MayBe**	CORBA (optionally)
Yes	No	Re-execute procedure (*)	**At-least once**	Sun RPC
Yes	Yes	Retransmit reply	**At-most once** (**)	**Java RMI**, CORBA

(*) unacceptable if the procedures on server are not idempotent.

(**) it is the semantic of Java RMI.

Role of client and server stub procedures in RPC



SUN XDR

- The Sun XDR (External Data Representation) language, originally designed for specifying the serialization of external data representations, was extended to interface definition language
- It may be used to define a service interface for Sun RPC by specifying a set of procedure definitions together with supporting type definitions.
- The notation is rather primitive in comparison with that used by CORBA IDL

Files interface in Sun XDR

```
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;        1
      Data READ(readargs)=2;          2
   }=2;
} = 9999;
```

From RPC to RMI

Remote method invocation (RMI) is closely related to RPC but extended into the world of distributed objects.

In RMI, a calling object can invoke a method in a potentially remote object.

As with RPC, the underlying details are generally hidden from the user.

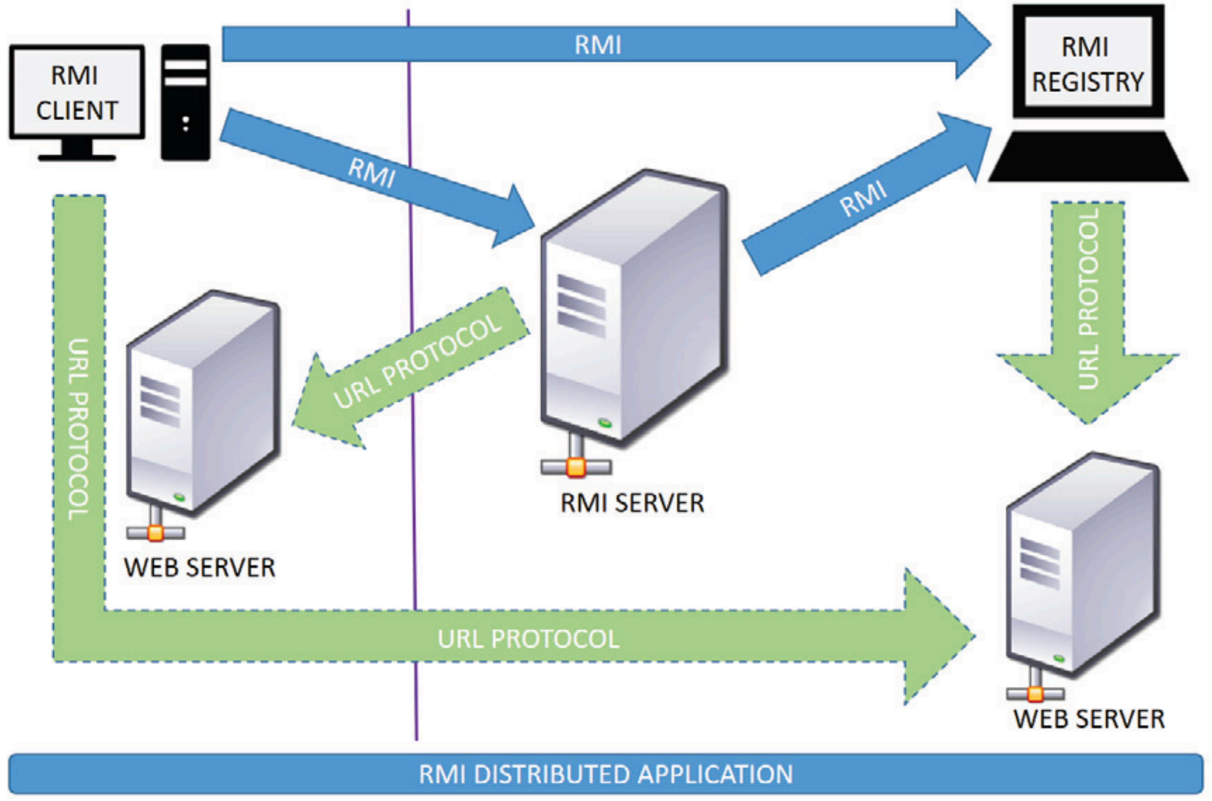
The commonalities between RMI and RPC are:

- They both support programming with interfaces, with the resultant benefits that stem from «design by contracts»
- They are both typically constructed on top of request-reply protocols and can offer a range of call semantics such as at-least-once and at-most-once.
- They both offer a similar level of transparency – that is, local and remote calls employ the same syntax but remote interfaces typically expose the distributed nature of the underlying call, for example by supporting remote exceptions

RMI architecture

RMI allows an object A in one Java Virtual Machine (JVM) to interact with an object B in another JVM by invoking the methods in that remote object B

This communication architecture makes a distributed application look like a group of objects communicating across a remote connection



Object identity

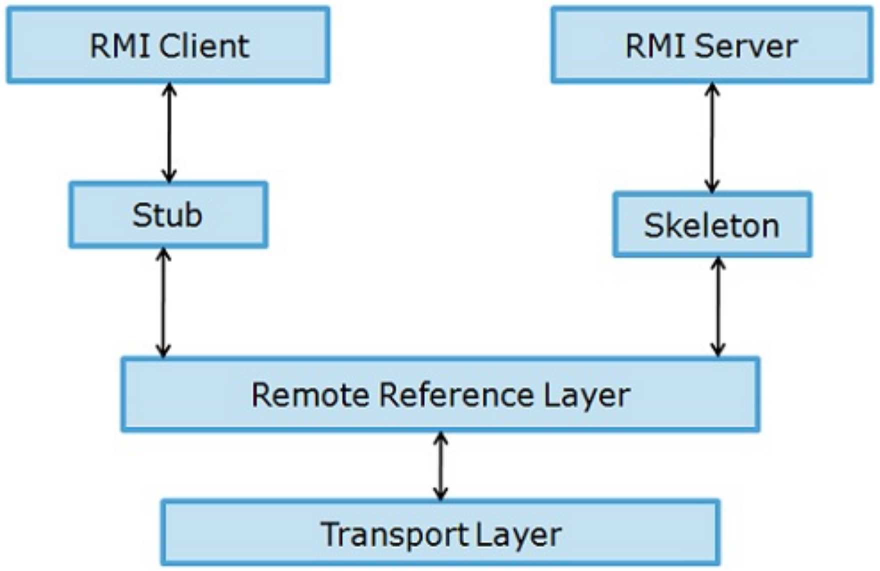
- Identity allows the construction of an ideal world, the ontology or conceptual model, that is used as basis of object-oriented thinking.
- The conceptual model describes the client side view to a domain, terminology or an API.
- This world contains point-like objects as instances, properties of the objects, and links between those objects.
- The objects in the world can be grouped to form classes.
- The properties of the objects can be grouped to form roles.
- The links can be grouped to form associations.
- All locations in the world together with the links between the locations form the structure of the world.
- These groups are types of the corresponding instances of the world.

RMI vs RPC

The following differences lead to added expressiveness when it comes to the programming of complex distributed applications and services.

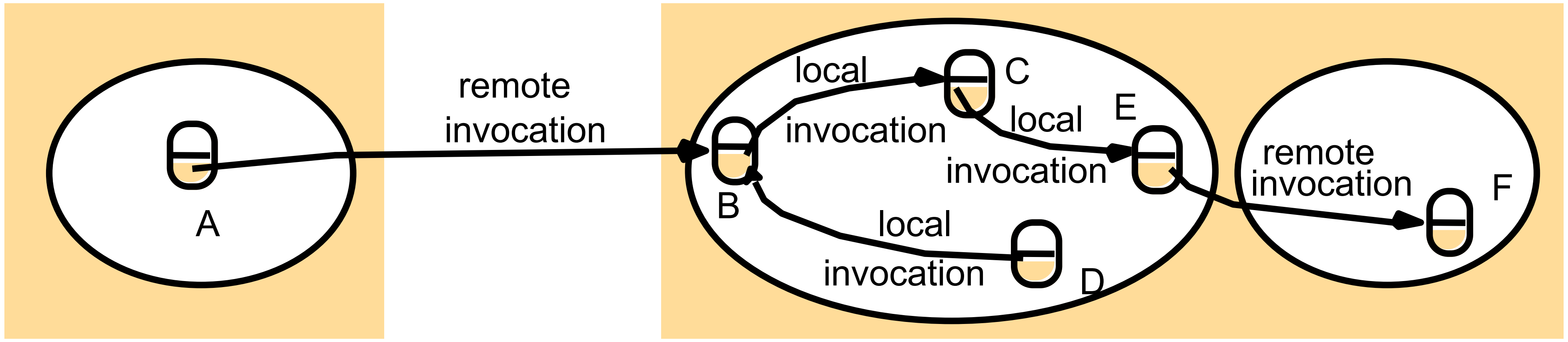
- The programmer can use the full expressive power of object-oriented programming in the development of distributed systems software, including objects, classes and inheritance, and can also employ related object-oriented design methodologies and associated tools.
- Building on the concept of object identity in object-oriented systems, all objects in an RMI-based system have unique object references (whether they are local or remote), such object references can also be passed as parameters, thus offering significantly richer parameter-passing semantics than in RPC.
- The issue of 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

RMI



- In Java, the parameters are passed to methods and returned in the form of reference. This could be troublesome for RMI service since not all objects are possibly remote methods. So, it must determine which could be passed as reference and which could not.
- Java uses **serialisation** where the objects are passed as value.
- The remote object is localised by pass by value.
- It can also pass an object by reference through passing a remote reference to the object along with the URL of the stub class. Pass by reference restricts a stub for the remote object.

Remote and local method invocations



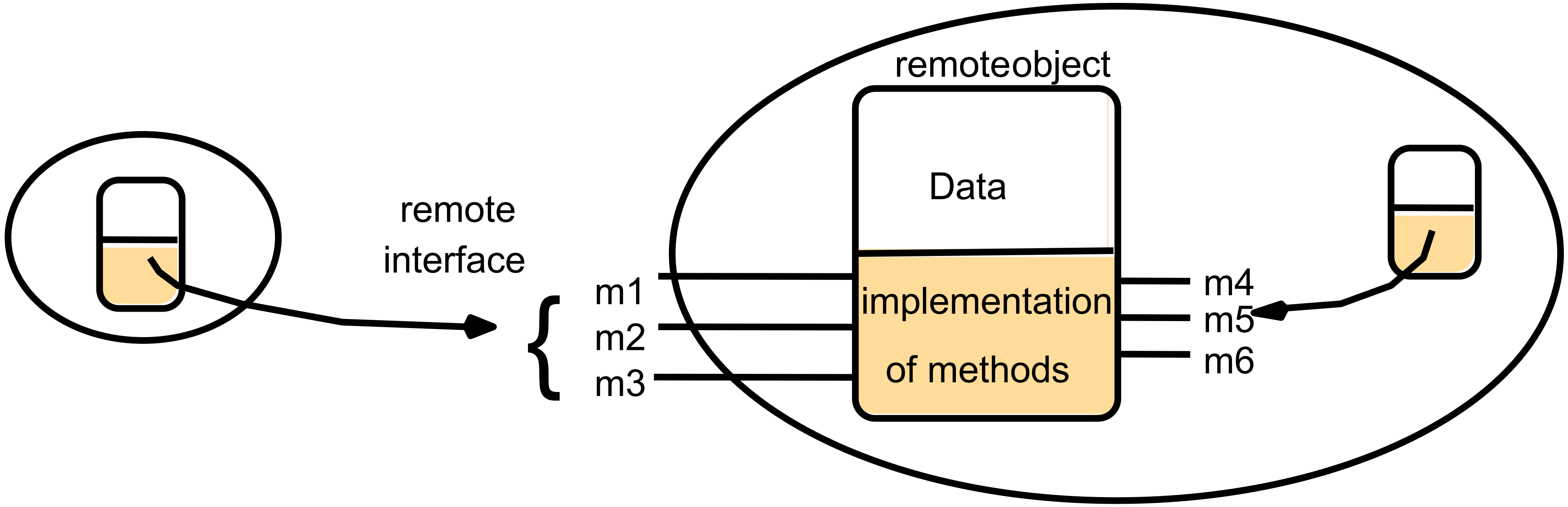
Each process contains a collection of objects, some of which can receive both local and remote invocations, whereas the other objects can receive only local invocations

The distributed object model

The following two fundamental concepts are at the heart of the distributed object model:

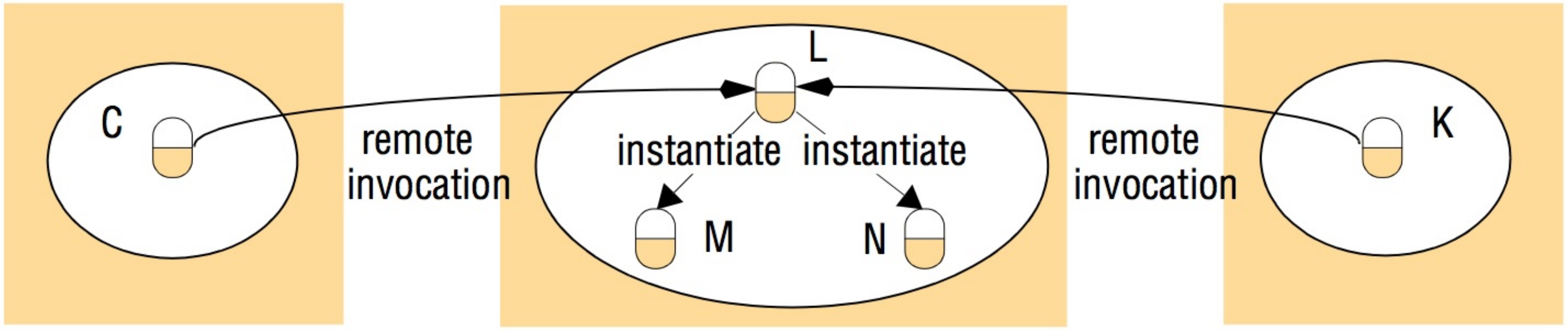
- Remote object references : Other objects can invoke the methods of a remote object if they have access to its remote object reference
- For example, a remote object reference for B in Figure 5.12 must be available to A.
- Remote interfaces : Every remote object has a remote interface that specifies which of its methods can be invoked remotely.
- For example, the objects B and F in Figure 5.12 must have remote interfaces.

A remote object and its remote interface



The class of a remote object implements the methods of its remote interface, for example as public instance methods in Java. Objects in other processes can invoke only the methods that belong to its remote interface, as shown here

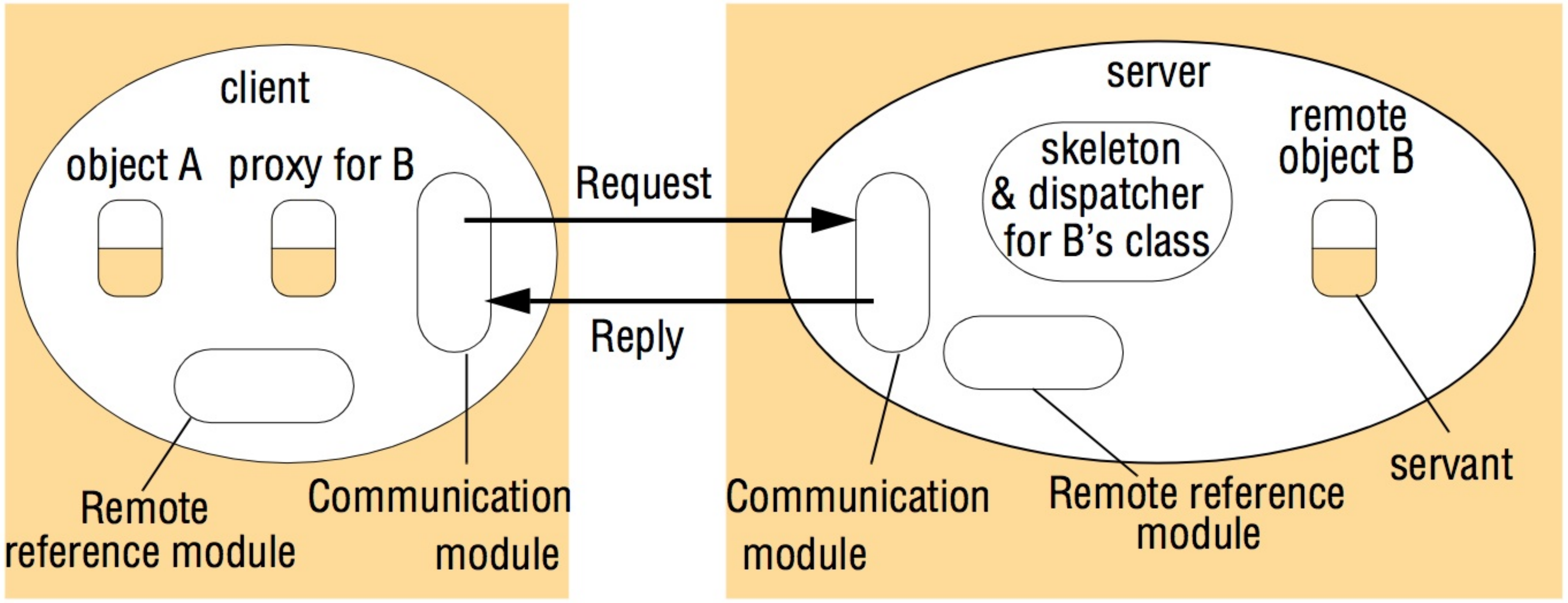
Instantiation of remote objects



Distributed applications may provide remote objects with methods for instantiating objects that can be accessed by RMI, thus effectively providing the effect of remote instantiation of objects.

For example, if the object L contains a method for creating remote objects, then the remote invocations from C and K could lead to the instantiation of the objects M and N respectively

The role of proxy and skeleton in remote method invocation



RMI architecture

a layer of software between the application-level objects and the communication and remote reference modules. The roles of the middleware objects are:

- Proxy: make remote method invocation transparent to clients by behaving like a local object to the invoker; but instead of executing an invocation, it forwards it in a message to a remote object
- Dispatcher: A server has one dispatcher and one skeleton for each class representing a remote object. In our example, the server has a dispatcher and a skeleton for the class of remote object B. The dispatcher and the proxy use the same allocation of operationIds to the methods of the remote interface.
- Skeleton: The class of a remote object has a skeleton, which implements the methods in the remote interface. They are implemented quite differently from the methods in the servant that incarnates a remote object. A skeleton method unmarshals the arguments in the request message and invokes the corresponding method in the servant. It waits for the invocation to complete and then marshals the result, together with any exceptions, in a reply message to the sending proxy's method.

Java RMI

- Java RMI extends the Java object model to provide support for distributed objects in the Java language.
- In particular, it allows objects to invoke methods on remote objects using the same syntax as for local invocations; in addition, type checking applies equally to remote invocations as to local ones.
- However, an object making a remote invocation is aware that its target is remote because it must handle RemoteExceptions; and the implementor of a remote object is aware that it is remote because it must implement the Remote interface.
- Although the distributed object model is integrated into Java in a natural way, the semantics of parameter passing differ because the invoker and target are remote from one another.
- In Java RMI, the parameters of a method are assumed to be input parameters and the result of a method is a single output parameter

Java RMI and distributed objects

RMI applications often include two separate programs, a server and a client.

- A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects.
- A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth: such an application is sometimes referred to as a *distributed object application*

Java Remote interfaces *Shape* and *ShapeList*

```
import java.rmi.*;
import java.util.Vector;
public interface Shape extends Remote {
    int getVersion() throws RemoteException;
    GraphicalObject  getAllState() throws RemoteException;            1
}
public interface ShapeList extends Remote {
    Shape newShape(GraphicalObject g) throws RemoteException; 2
    Vector allShapes() throws RemoteException;
    int getVersion() throws RemoteException;
}
```

Moving code for supporting remote invocations

- Java is designed to allow classes to be downloaded from one virtual machine to another.
- This is particularly relevant to distributed objects that communicate by means of remote invocation.
- non-remote objects are passed by value and remote objects are passed by reference as arguments and results of RMIs.
- If the recipient does not already possess the class of an object passed by value, its code is downloaded automatically.
- Similarly, if the recipient of a remote object reference does not already possess the class for a proxy, its code is downloaded automatically

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.

Java class *ShapeListServer* with *main* method

```
import java.rmi.*;
public class ShapeListServer{
      public static void main(String args[]){
            System.setSecurityManager(new RMISecurityManager());
             try{
                  ShapeList aShapeList = new ShapeListServant();                              1
                        Naming.rebind("Shape List", aShapeList );                             2
                  System.out.println("ShapeList server ready");
                  }catch(Exception e) {
                  System.out.println("ShapeList server main " + e.getMessage());}
      }
}
```

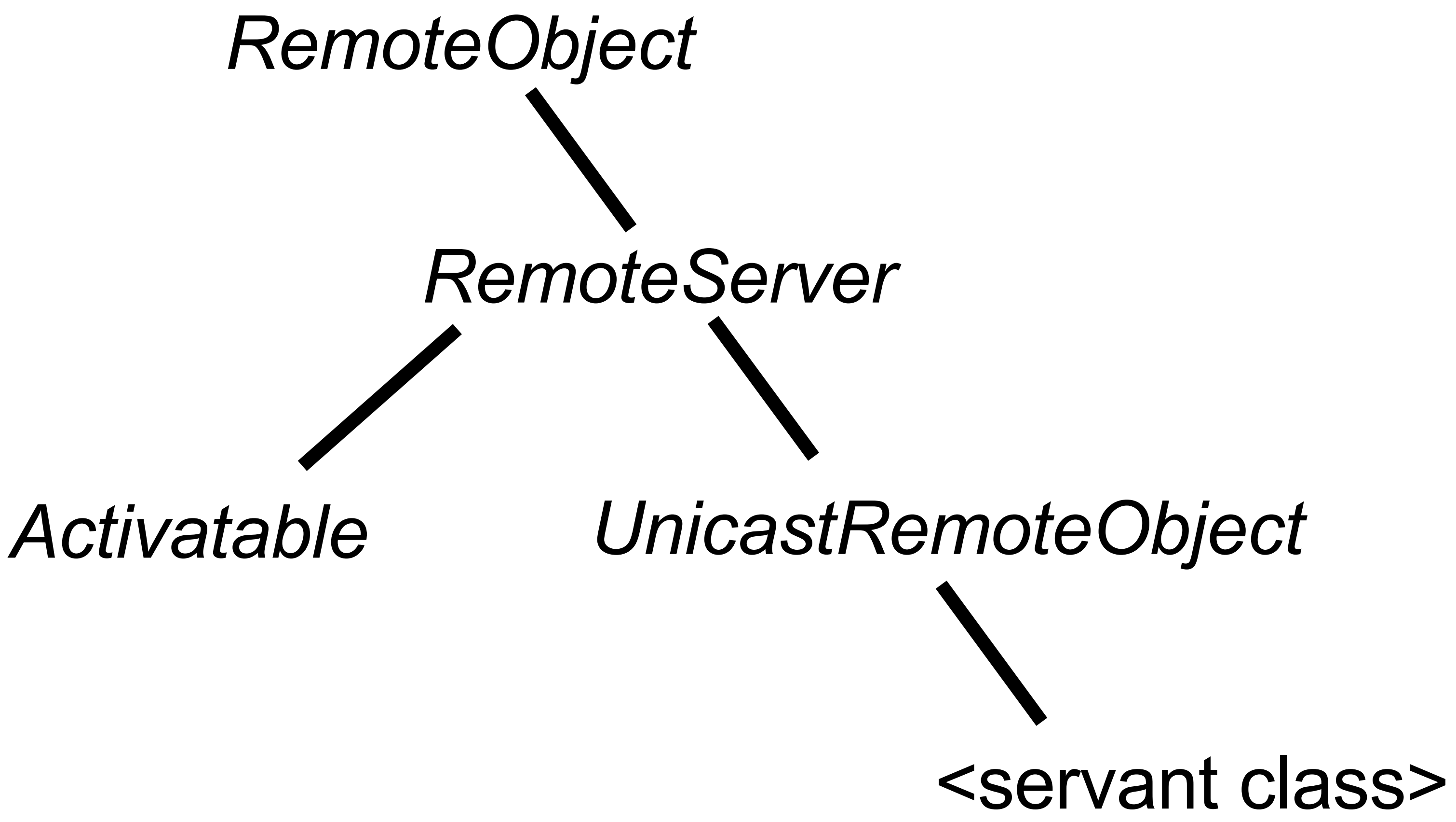
Java class *ShapeListServant* implements interface *ShapeList*

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Vector;
public class ShapeListServant extends UnicastRemoteObject implements ShapeList {
     private Vector theList;          // contains the list of Shapes
     private int version;
    public ShapeListServant()throws RemoteException{...}
    public Shape newShape(GraphicalObject g) throws RemoteException {        1
        version++;
            Shape s = new ShapeServant( g, version);                          2
            theList.addElement(s);
            return s;
    }
    public  Vector allShapes()throws RemoteException{...}
    public int getVersion() throws RemoteException { ... }
}
```

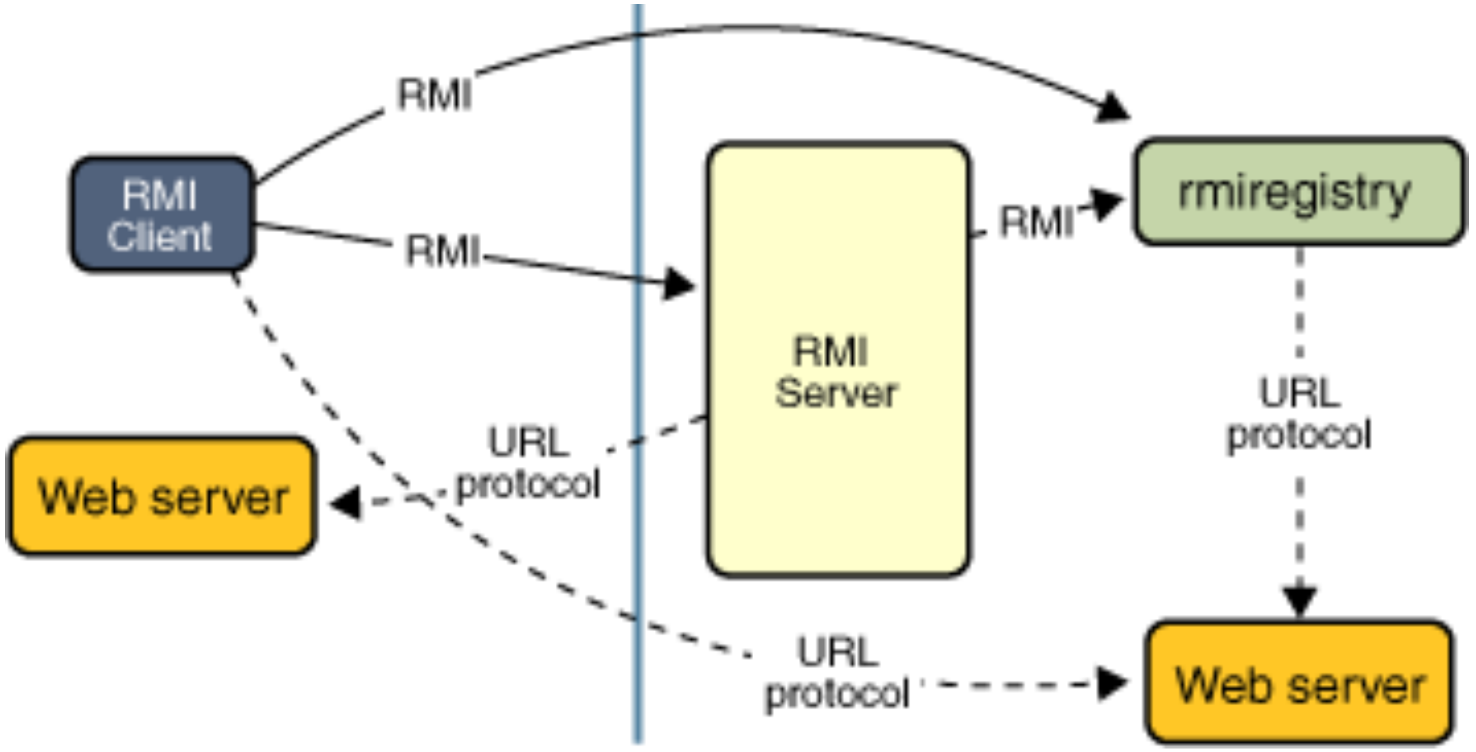
Java client of *ShapeList*

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class ShapeListClient{
   public static void main(String args[]){
     System.setSecurityManager(new RMISecurityManager());
     ShapeList aShapeList = null;
     try{
          aShapeList  = (ShapeList) Naming.lookup("//bruno.ShapeList") ;          1
          Vector sList = aShapeList.allShapes();                                  2
     } catch(RemoteException e) {System.out.println(e.getMessage());
     }catch(Exception e) {System.out.println("Client: " + e.getMessage());}
   }
}
```

Classes supporting Java RMI



RMI example



The illustration depicts an RMI distributed application that uses the RMI registry to obtain a reference to a remote object.

The server calls the registry to associate (or bind) a name with a remote object.

The client looks up the remote object by its name in the server's registry and then invokes a method on it.

The illustration also shows that the RMI system uses an existing web server to load class definitions, from server to client and from client to server, for objects when needed

Conclusions

- We discussed three paradigms for distributed programming – request-reply protocols, remote procedure calls and remote method invocation.
- All of these paradigms provide mechanisms for distributed independent entities (processes, objects, components or services) to communicate directly with one another

Conclusions

- The RPC approach was a breakthrough in distributed systems, providing higher-level support for programmers by extending the concept of a procedure call to operate in a networked environment: this provides important levels of transparency in distributed systems.
- due to their different failure and performance characteristics and to the possibility of concurrent access to servers, it is not necessarily a good idea to make RPC appear to be exactly the same as local calls.
- RPC provide a range of invocation semantics, from *maybe* invocations through to *at-most-once* semantics.

Conclusions: Java RMI

The Java distributed object model differs from the Java object model:

- Clients of remote objects interact with remote interfaces, never with the implementation classes of those interfaces.
- Nonremote arguments to, and results from, a remote method invocation are passed by copy rather than by reference; this is because references to objects are only useful within a single virtual machine.
- A remote object is passed by reference, not by copying the actual remote implementation.
- The semantics of some of the methods defined by class Object are specialized for remote objects.
- Since the failure modes of invoking remote objects are inherently more complicated than the failure modes of invoking local objects, clients must deal with additional exceptions that can occur during a remote method invocation.

Exercise

An Election interface provides two remote methods:

vote : with two parameters through which the client supplies the name of a candidate (a string) and the ‘voter’s number’ (an integer used to ensure each user votes once only). The voter’s numbers are allocated sparsely from the range of integers to make them hard to guess.

result : with two parameters through which the server supplies the client with the name of a candidate and the number of votes for that candidate.

Which of the parameters of these two procedures are input and which are output parameters?

Exercise

- Discuss the invocation semantics that can be achieved when the request-reply protocol is implemented over a TCP/IP connection, which guarantees that data is delivered in the order sent, without loss or duplication.
- Take into account all of the conditions causing a connection to be broken.

Answer

A process is informed that a connection is broken:

- when one of the processes exits or closes the connection.
- when the network is congested or fails altogether

Therefore a client process cannot distinguish between network failure and failure of the server

- Provided that the connection continues to exist, no messages are lost, therefore, every request will receive a corresponding reply, in which case the client knows that the method was executed exactly once.
- However, if the server process crashes, the client will be informed that the connection is broken and the client will know that the method was executed either once (if the server crashed after executing it) or not at all (if the server crashed before executing it).
- But, if the network fails the client will also be informed that the connection is broken. This may have happened either during the transmission of the request message or during the transmission of the reply message. As before the method was executed either once or not at all. Therefore we have at-most-once call semantics.