Object-Based Distributed Systems

IN5020/9020 Autumn 2023
Lecturer: Amir Taherkordi

September 4, 2023

UiO : University of Oslo

Outline

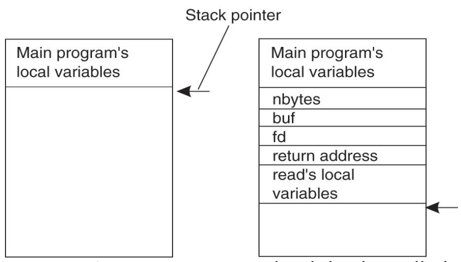
- Local Procedure Call
- Remote Procedure Call (RPC)
- Distributed Objects
- Remote Method Invocation (RMI)
- Object Server
- CORBA
- Java RMI
- Summary

Local Procedure Call

- Many distributed systems:
  - based on explicit **message exchange between processes**
- How is it done in a single machine?

Local Procedure Call, e.g.
```
count = read(fd, buf, nbytes);
```

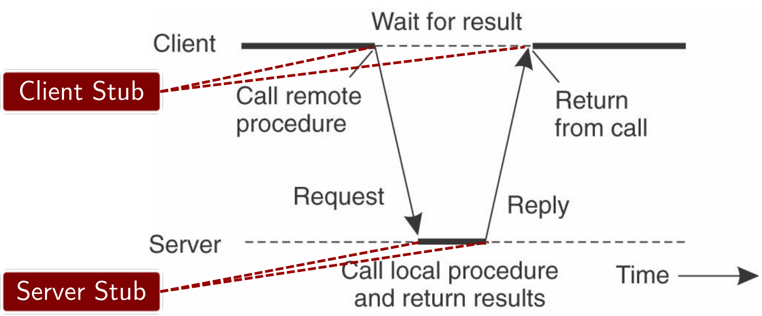
- Parameter passing in a local procedure call
- Parameter passing:
  - call-by-value: `fd` and `nbytes`
  - call-by-reference: `buf`



stack before the call to `read` | stack while the called procedure is active

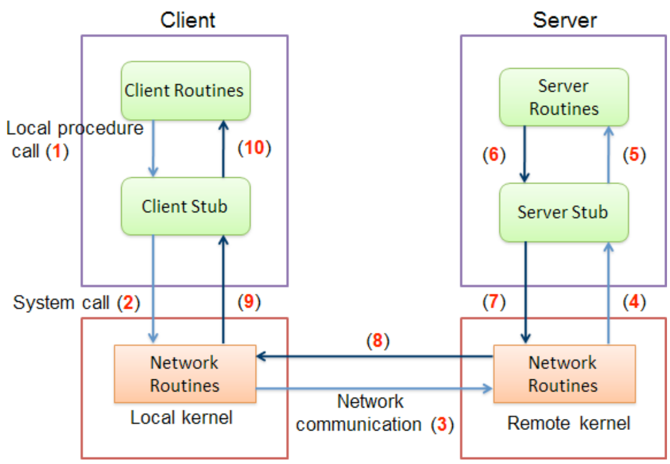
Remote Procedure Call (1)

- Ideally:
  - make a remote call look as a **local** one
  - in other words: achieving **access transparency**
- The basic idea:



Remote Procedure Call (2)

- A RPC occurs in the following 10 steps:



Remote Procedure Call (3)

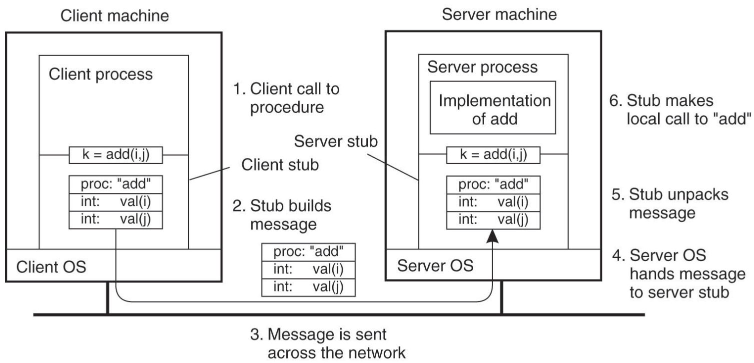
- The net effect of these steps:

> To **convert the local call** by client to a local call to server **without** either client or server **being aware** of the **intermediate steps** or the existence of the **network**

- These steps seem straightforward?
  - how about taking parameters by the client stub, packing them, and sending them to the server stub?
    - passing value parameters
    - passing reference parameters

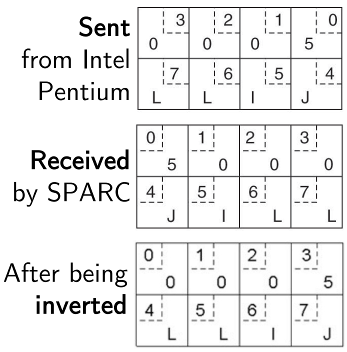
Passing Value Parameters (1)

- Parameter **marshaling**: packing parameters in a message
- **`add(i,j)`** example:



Passing Value Parameters (2)

- This model works as long as:
  - client and server machines are **identical**
  - all parameters and results are **scalar/base types:** int,char,boolean, ...
- Challenge: each machine has its **own representation of data**
  - e.g. IBM mainframe: EBCDIC code, while IBM pc: ASCII
- **Sending[(int)5, "JILL"]**
  - **Byte numbering**; left-to-right or other way?
    - INTEL: sends 5,0,0,0
    - SPARC: interprets as 0,0,0,5
    - Enough to invert the sequence of bytes?
  - **Character array**
    - INTEL: sends J,I,L,L
    - SPARC: interprets as J,I,L,L
    - Can not be inverted.



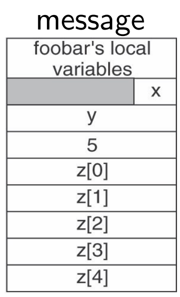
Passing Reference Parameters

- How to pass references (pointers)?
  - pointers are meaningful within the address space of the process
  - not possible to pass only the address of parameter
- One solution (for arrays):
  1. **copy the array** into the message and send to the server
  2. server stub calls the server with a pointer to this array
  3. server makes **changes to the array**
  4. message will be **sent back** to the client stub
  5. client stub copies it back to the client
- How about pointers to arbitrary data structures:
  - e.g. complex graph
  - **solution:** passing pointer to server and generating special code for using pointers, e.g. code to make requests to client to get the data

Stub Generation

- What we understood so far:
  - Client and server must agree on a **protocol**, e.g.
    - agree on the format of messages
    - representation of simple data structure
- A complete example:

```
foobar(char x; float y; intz[5] {…}
```



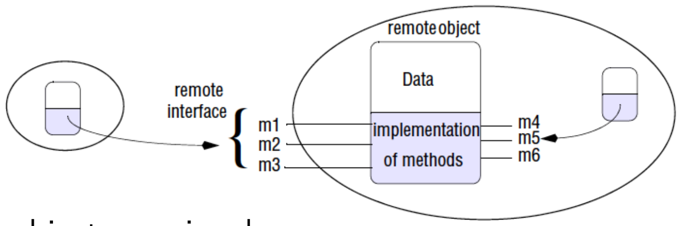
- Next step after defining RPC protocol:
  - implementing client and server stubs
  - stubs for the same protocol but different procedures
    - Differ only in their interface

Outline

Characteristics of Distributed Objects

- Distributed Objects
  - execute in different processes
  - have a **remote interface** for controlling access to its methods and attributes
- Remote Interface
  - **accessed from other objects** in other processes located on the same or other machines
  - declared via an "**Interface Definition Language**" (**IDL**)
- Remote Method Invocation (RMI)
  - "RPC with distributed objects"
  - method call from an object **in one process** to a (remote) object in **another** process

Remote Object



- Local objects can invoke
  - the methods in the remote interface
  - other methods implemented by a remote object
- **Remote Object Reference (ROR):** unique identity of distributed objects
  - other objects invoking methods of a remote object needs access to its ROR
  - RORs are **"first class values"**
    - can occur as arguments and results in RMI
    - can be assigned to variables

Object Type

- Type of an object:

> **Attributes**, **methods** and **exceptions** are properties that objects can **export** to other objects

- The object type is defined by the **interface specification** of the object
- The type is defined once
  - several objects can export the same properties (same type of objects)

Interface Specification

- A remote method is declared by its **signature**
  - a **name**
  - a list of **in** and **out parameters**
  - a **return value** type
  - a list of **exceptions** that the method can raise
- An attribute is declared by
  - a **name**
  - a **value** type
- For example in CORBA:

```
void select (in Date d) raises (AlreadySelected);
```


Outline

- Local Procedure Call
- Remote Procedure Call (RPC)
- Distributed Objects
- **Remote** Method Invocation (RMI)
- Object Server
- CORBA
- Java RMI
- Summary

Remote Method Invocations (1)

- Closely related to **RPC** but extended into the world of distributed objects
- Commonalities
  - both support **programming with interfaces**
  - both typically constructed on top of **request-reply protocols**
  - both offer a similar **level of transparency**
- Differences
  - in RMI: using the full expressive power of object-oriented programming: **use of objects, classes and inheritance**
  - in RMI: all objects have unique RORs
    - object references can also be passed as parameters
    - => **richer parameter-passing semantics** than in RPC

Remote Method Invocations (2)

A client object can request the execution of a **method** of a distributed, remote object

↓

Remote methods are invoked by sending a message (method name + arguments) to the **remote object**

↓

The remote object is identified and **located** using the remote object reference (**ROR**)

↓

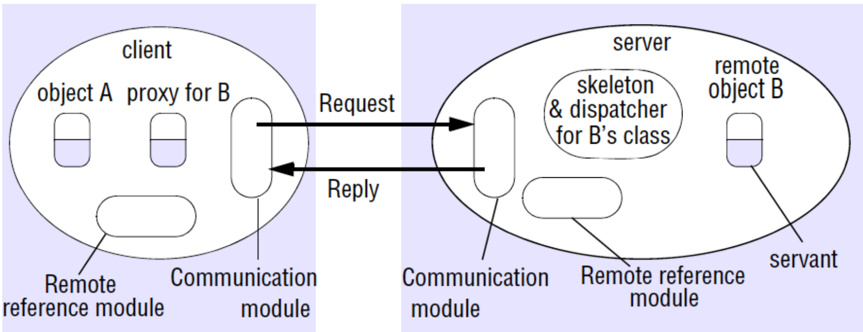
Clients must be able to handle **exceptions** that the method can raise

Implementation of RMI

- Three main tasks:
  - **Interface processing**
    - integration of the RMI mechanism into a programming language
    - basis for realizing **access transparency**
  - **Communication**
    - message exchange (a request-reply protocol)
  - **Object location, binding and activation**
    - locate the server process that hosts the remote object and bind to the server
    - activate an object-implementation
    - basis for realizing **location transparency**

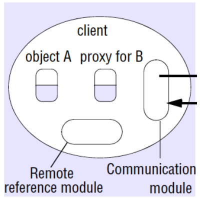
RMI Interface Processing

- Role of proxy and skeleton



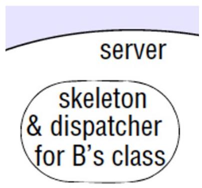
Elements of the RMI Software (1)

- **Client proxy**
  - local "proxy" object for each remote object and holds a ROR ("stand-in" for remote object).
  - the class of the proxy-object has the **same interface** as the class of the remote object
  - can perform **type checking** on arguments
  - performs **marshalling** of requests and **unmarshalling** of responses
  - transmits request-messages to the server and receive response messages.
    - Makes remote **invocation transparent** to client



Elements of the RMI Software (2)

- **Dispatcher**
- A server has one dispatcher for each class representing a remote object:
  - **receives requests messages**
  - uses *method id* in the request message to **select the appropriate method** in the skeleton (provides the methods of the class) and passes on the request message

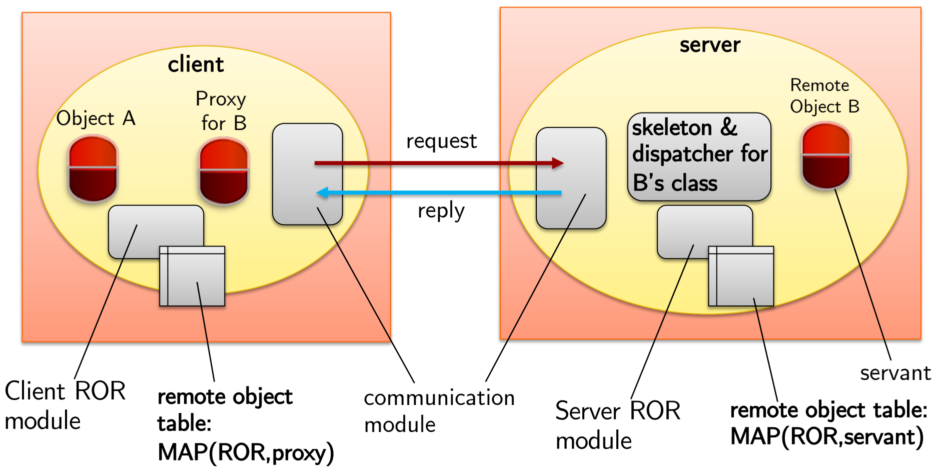


Elements of the RMI Software (3)

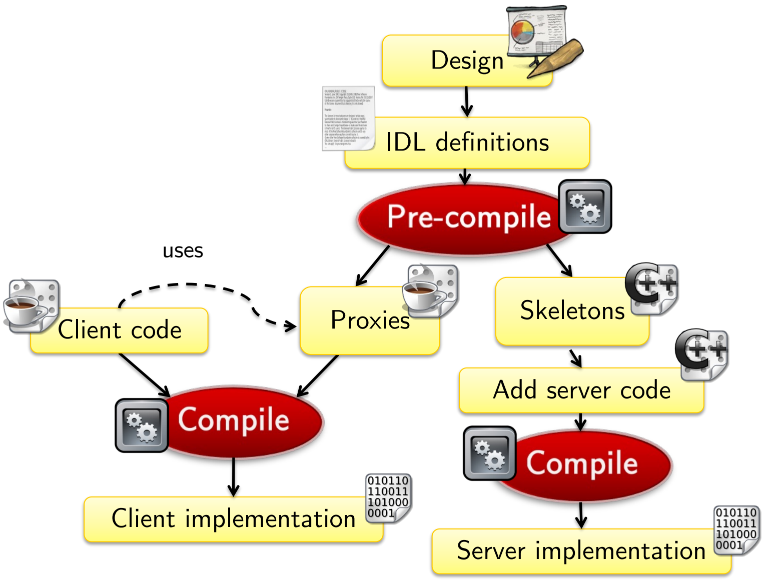
- **Skeleton**
  - one skeleton for each class representing a remote object
  - provides the methods of the remote interface
  - unmarshals the arguments in the request message and invokes the corresponding method in the remote object.
  - waits for the invocation to complete and then
  - marshals the result, together with any exceptions, in a reply message to the sending proxy's method.

Elements of the RMI Software (4)

- Remote object reference module



Generation of Proxies, Dispatchers and Skeletons



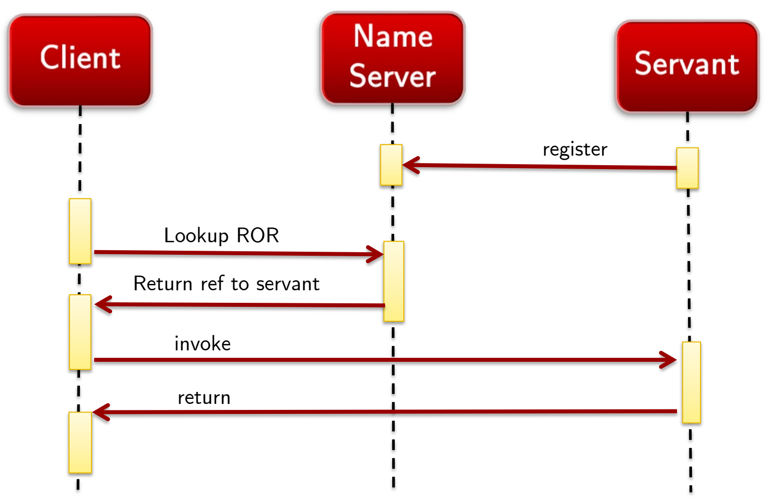
Server and Client Programs

- **Server** program contains
  - the **classes** for the **dispatchers** and **skeletons**
  - the implementation classes of all the **servants**
  - an **initialization section**
    - creates and initializes at least one servant
    - additional servants (objects) may be created in response to client requests
  - register zero or more **servants** with a ***Name server***
  - potentially one or more ***factory methods*** that allow clients to request creation of additional servants (objects)
- **Client** program contains
  - the classes and **proxies** for all the remote objects that it will invoke

RMI Name Resolution, Binding, and Activation

- **Name resolution**
  - mapping a **symbolic object name** to an **ROR**
  - performed by a **name service** (or similar)
- **Binding in RMI**
  - **locating the server** holding a remote object using the ROR of the object, and
  - **placing a proxy** in the client process's address space
- **Activation in RMI**
  - **creating an active object** from a corresponding passive object (e.g., on request).
    - register passive objects that are available for activation
    - activate server processes (and activate remote object within them)

RMI Sequence Diagram



Outline

- Local Procedure Call
- Remote Procedure Call (RPC)
- Distributed Objects
- Remote Method Invocation (RMI)
- Object Server
- CORBA
- Java RMI
- Summary

Object Server

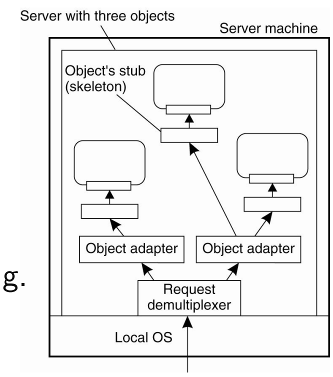
- The server
  - is designed to **host** distributed objects
  - provides the means to **invoke local** objects, based on requests from remote clients
- For object invocation, the object server needs to know
  - which **code** to execute
  - which **data** it should operate
  - whether it should start **a separate thread** to take care of the invocation

Activation Policies

- **Transient objects:** creating object at the first invocation request and destroying it when no clients are bound to it anymore
  - **advantage**: object uses server's resources only it really needs
  - **drawback**: taking time to make an invocation (object needs to be created first)
  - **an alternative policy:** creating all transient objects during server initialization, at the cost of consuming resources even when no client uses the object.
- Data and Code Sharing:
  - sharing **neither code nor data**: e.g. for security reasons
  - Sharing objects' **code**: e.g. a database containing objects that belong to the same class
- Policies with respect to **threading**:
  - **single** thread
  - **several** threads, one for each of its objects: how to assign threads to objects and requests? One thread per object? One per request?

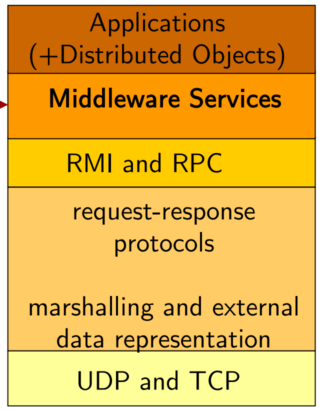
Object Adapter/Wrapper

- A mechanism to **group objects per policy**
- Software implementing a specific activation policy
- Upon receiving invocation request:
  - it is first dispatched to the appropriate object adapter
  - adapter **extracts an object reference** from an invocation request
  - adapter **dispatches** the **request** to the referenced object, but now following a specific activation policy, e.g.
    - single-threaded or
    - multithreaded mode



Outline

- Local Procedure Call
- Remote Procedure Call (RPC)
- Distributed Objects
- Remote Method Invocation (RMI)
- Object Server
- CORBA
- Java RMI
- Summary



Common Object Request Broker Architecture
(CORBA)

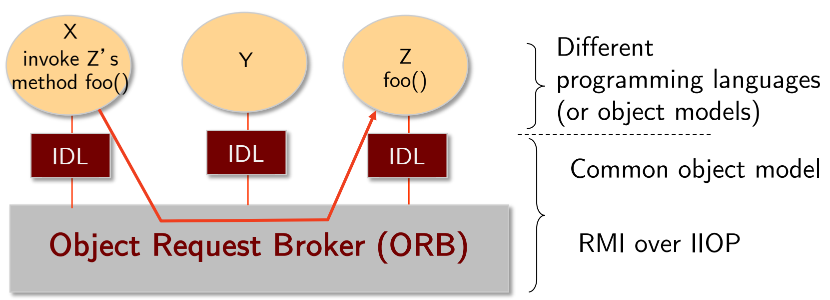
CORBA™

CORBA Middleware (1)

- Offers mechanisms that allow objects to **invoke** remote methods and **receive** responses in a transparent way
  - **location transparency**
  - **access transparency**
- The core of the architecture is the **Object Request Broker** (ORB)
- Specification developed by members of the Object Management Group (www.omg.org)

CORBA Middleware (2)

- Clients may invoke methods of remote objects without worrying about:
  - object location, programming language, operating system platform, communication protocols or hardware.



Supporting Language Heterogeneity

- CORBA allows interacting objects to be implemented in **different** programming **languages**
- **Interoperability** based on a common object model provided by the middleware
- Need for **advanced mappings** (language bindings) between different object implementation languages and the **common object model**

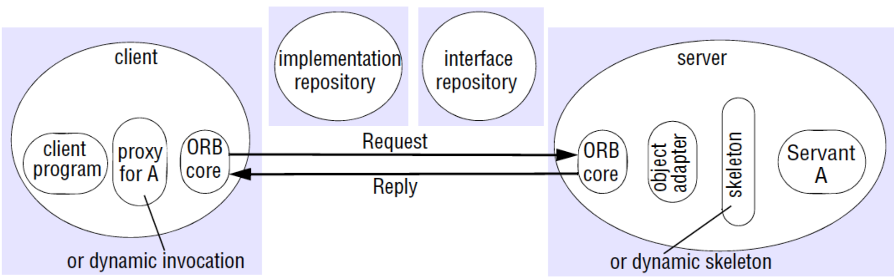
Elements of the Common Object Model

- Metalevel model for the type system of the middleware
- Defines the meaning of e.g.
  - object identity
  - object type (interface)
  - operation (method)
  - attribute
  - method invocation
  - Exception
  - subtyping / inheritance
- Must be general enough to enable mapping to common programming languages
- CORBA **Interface Definition Language** (IDL)

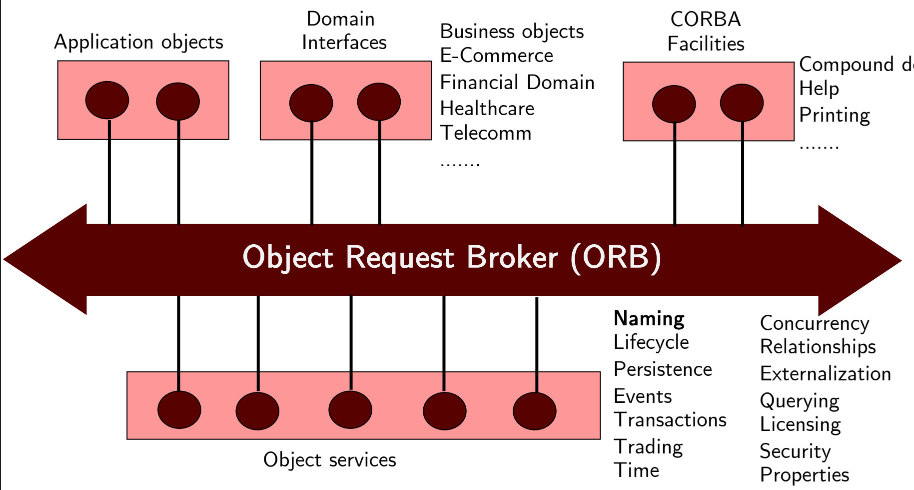
CORBA IDL

- Language for specifying CORBA **object types** (i.e. object interfaces)
- Can express all **concepts** in the CORBA common **object model**
- CORBA IDL is
  - not dependent on a specific programming language
  - syntactically oriented towards C++
  - not computationally complete
- Different bindings to programming languages available

CORBA Architecture



CORBA Services



Description of the services: Coulouris ch. 8, Figure 8.6

Java RMI

Java Remote Method Invocation (RMI)

Java™

Java RMI

- **Remote Method Invocation** (RMI) supports communication between different **Java Virtual Machines** (VM), and possibly over a network
- Provides tight integration with Java
- Minimizes changes in the Java language/VM
- Works for homogeneous environments (Java)
- Clients can be implemented as ***Java applet*** or ***Java application***

Java Object Model

- Interfaces and Remote Objects
- Classes
- Attributes
- Operations/methods
- Exceptions
- Inheritance

Java Interfaces to Remote Objects

- Based on the ordinary Java interface concept
- RMI does **not have** a separate language (IDL) for defining remote interfaces
- Remote objects must implement interfaces that extends the pre-defined interface java.rmi.Remote
- Java RMI provides some convenience classes that implement this interface which other remote implementations can extend, e.g. java.rmi.server.UnicastRemoteObject.

Example

interface name *declares the Team interface as "remote"*

```
interface Team extends Remote {
  String name()throws RemoteException;
  Trainer[] trained_by() throws RemoteException;
  Club club() throws RemoteException;
  Player[] player() throws RemoteException;
  void chooseKeeper(Date d)  throws RemoteException;
  void print() throws RemoteException;
};
```

remote operation

Parameter Passing

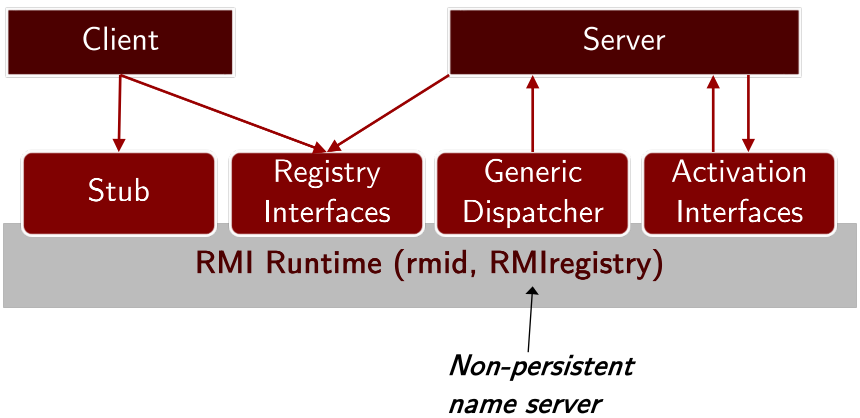
- Atomic types transferred *by value*
- Remote objects transferred *by reference*
- Non-remote objects transferred *by value*

```
class Address {
  public String street;
  public String zipcode;
  public String town;
};

interface Club extends Organisation, Remote {
  public Address addr() throws RemoteException;
  ...
};
```

Returns a copy of the Address-object

Architecture of Java RMI



Summary (1)

- Remote Procedure Calls
- Distributed objects executes in different processes
  - remote interfaces allow an object in one process to invoke methods of objects in other processes located on the same or on other machines
- Object-based distribution middleware
  - middleware that models a distributed application as a collection of interacting distributed objects (e.g. CORBA, Java RMI)

Summary (2)

- Implementation of RMI
  - proxies, skeletons, dispatcher
  - interface processing, binding, location, activation
- Object servers
  - object adapters and activation policies
- Principles of CORBA
  - clients may invoke methods of remote objects without worrying about: object location, programming language, operating system platform, communication protocols or hardware.
- Principles of Java RMI
  - similar to CORBA but limited to a Java environment