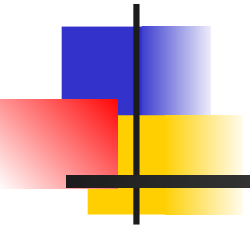
Corso di Sistemi Distribuiti e Cloud Computing

Corso di Laurea Magistrale
in Ingegneria Informatica



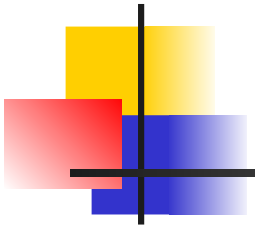
DIMES
Università degli Studi della Calabria

Introduction to Distributed Objects and Remote Method Invocation in Java

Ing. Loris Belcastro

General Informations

- Tutor: Ing. Loris Belcastro
- Office: DIMES, cubo 41C, V piano
- e-mail: lbelcastro@dimes.unical.it
- Office Hour: Thursday, 11:00 – 12:00



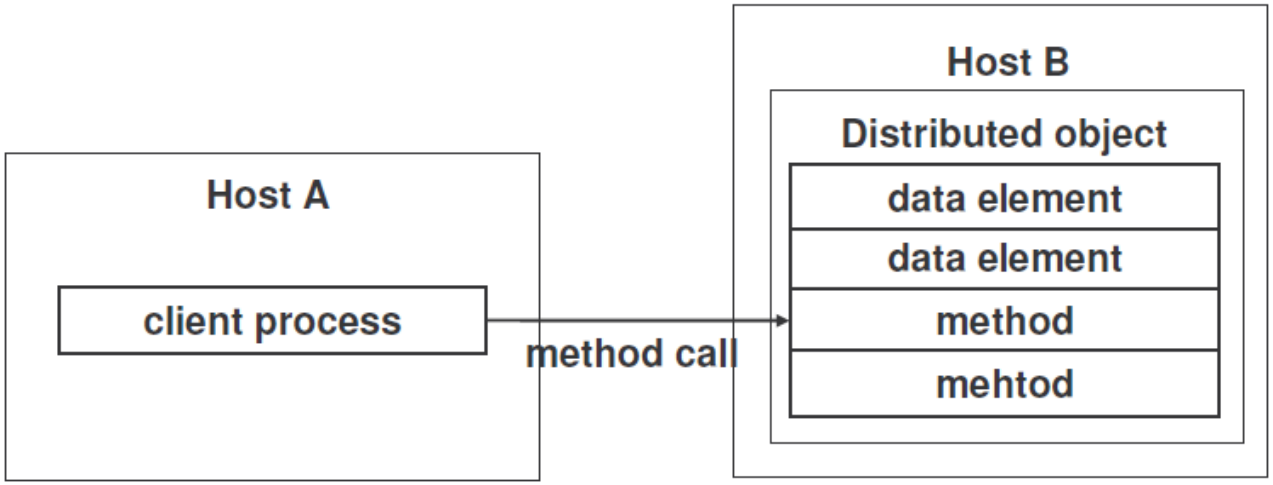
The Distributed Objects paradigm

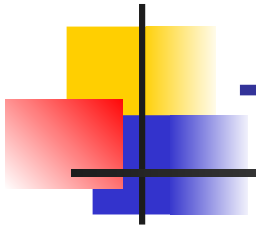
- **Local Objects:** objects whose methods can be invoked by a “local” process. Local means that the process is located on the same machine of the object.
  - Usual *Object-Oriented* Programming paradigm

- **Distributed (Remote) Objects:** objects whose methods can be invoked by a “remote” process. Remote means that the process is running on a remote machine.
  - Of course a (network) connection must exist between who invokes and who provides the object.

The Distributed Objects paradigm

- Resources available in the network are represented as distributd objects:
  - A process can request a service to a resource by invoking a method (or operation) and by passing suitable parameters.
  - The method is executed on the remote host, and the reply is sent to the requesting process (as return value)

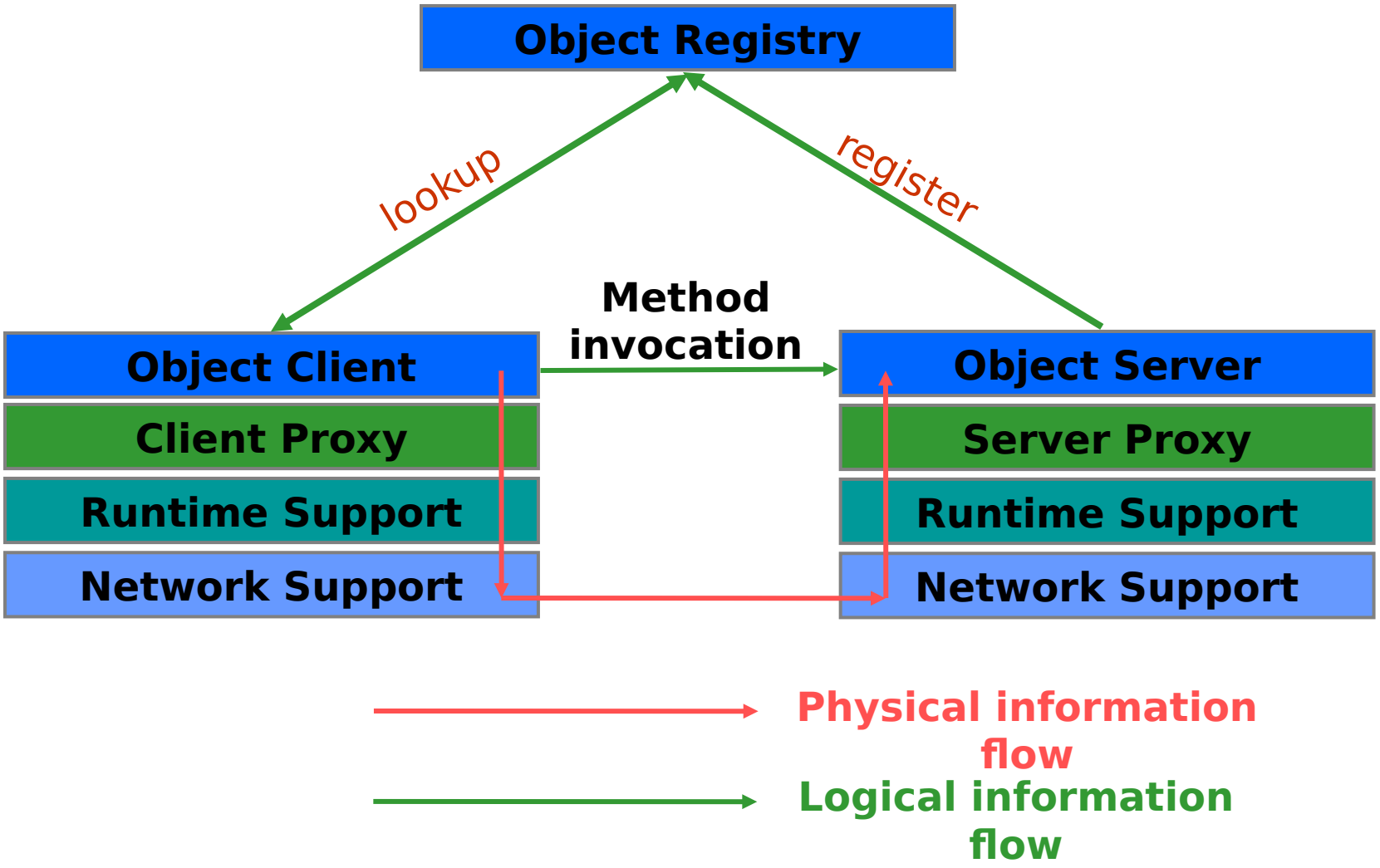




The Distributed Objects paradigm

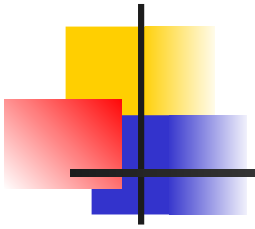
- A process running on the host A invokes a method of a distributed object resideing on the host B
- Such a method invocation generates a computation on the host B. Finally, the return value (if there is) is sent from B to A
- The methods of a distributed object are named remote methods
- The methods of a local object are named local methods

Architecture of a Distributed-Object System



A Distributed-Object System

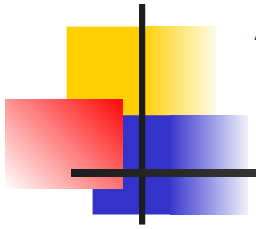
- A Distributed Object is made available by a process called object server.
- An object registry is required to allow the “registration” of the service (register operation).
- The reference to the object provided by the registry allows to locate the object, i.e., to know where the object physically is implemented and running (i.e., which port).
- Therefore a client, for accessing a remote object, contacts (lookup operation) the registry to obtain an object reference that will be used to invoke the methods on this object.
- A reference is a “handle” for an object; it is a representation through which an object can be located in the computer where the object resides



A Distributed-Object System (Client Side)

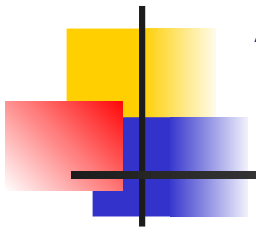
- The client invokes the method as it would be local.
- In practice, the call is handled by a component called client proxy which interacts with the software client offering the runtime support.
- The runtime support takes care of the calls forwarded by the proxy to the remote object. It also handles the **marshalling** of the parameters to be transmitted to the remote object.

A Distributed-Object System (Server Side)



- A similar architecture and a dual process is performed by the server (i.e., who makes available the object)
- It, through the runtime support:
  - handles incoming messages
  - performs the **unmarshalling** of data
  - forwards the request to the server proxy

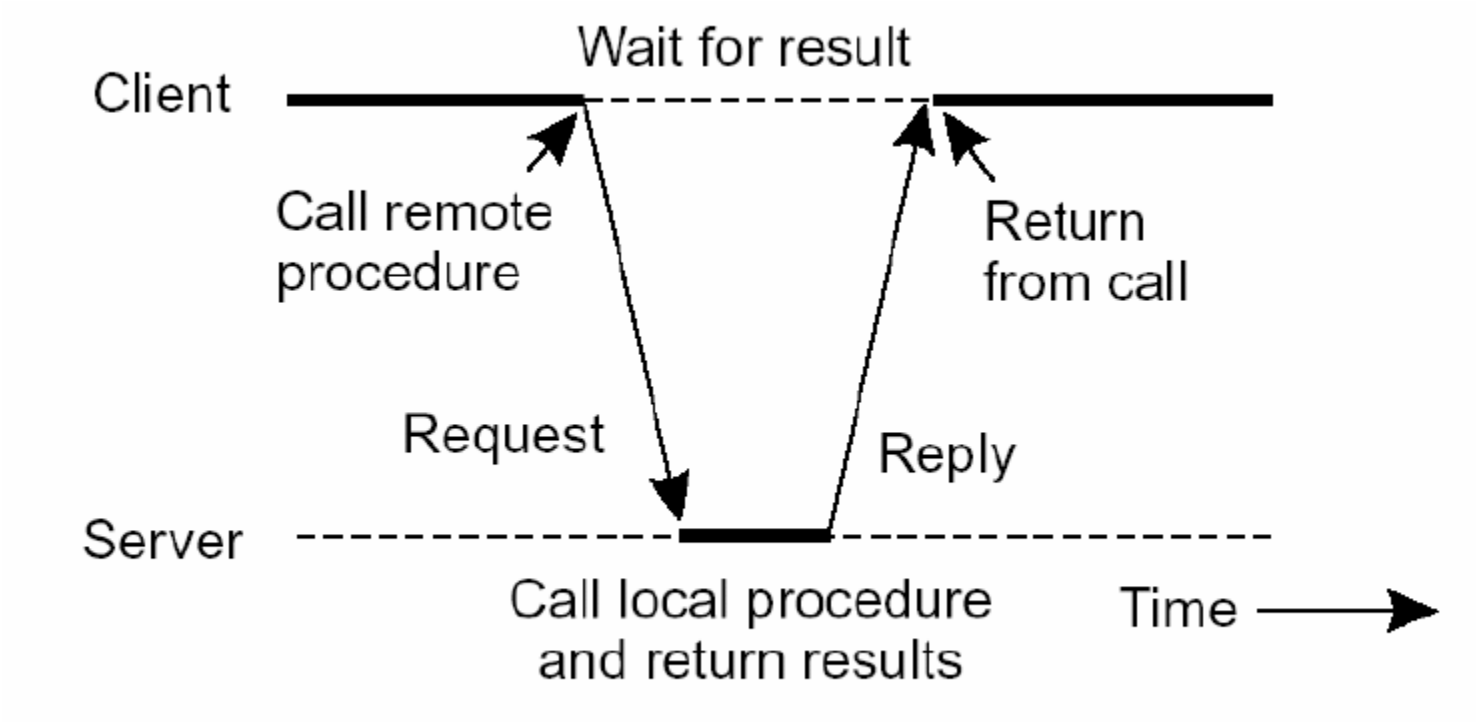
A Distributed-Object System (Server Side)



- The server proxy communicates with the distributed object and invokes the requested method with the unmarshalled parameters
- After the execution of the method, the results of the invocation (marshalling of the return value) are transmitted from the server proxy to the client proxy by the runtime and network supports

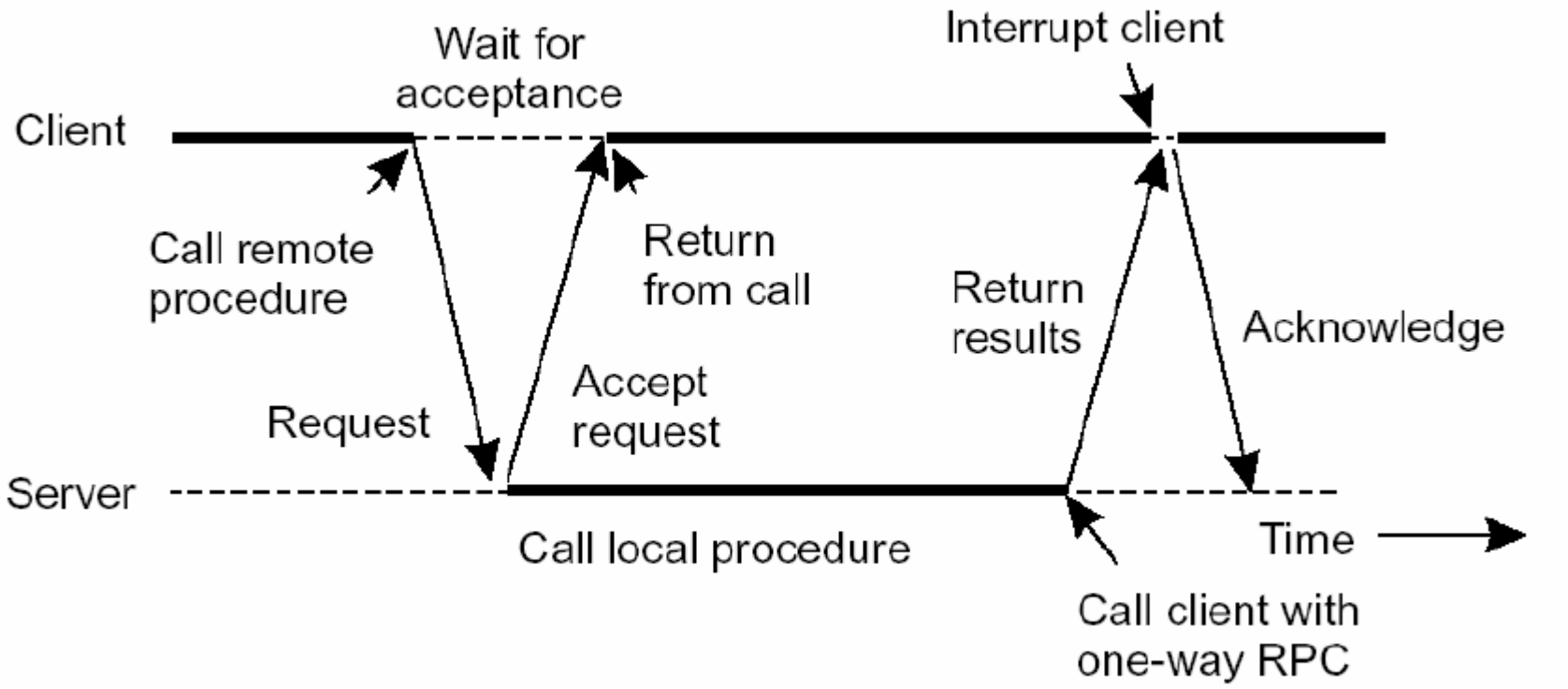
Remote Procedure Call (RPC)

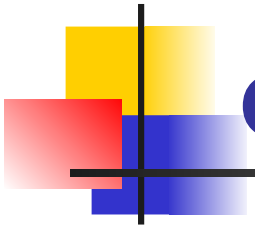
- Synchronous call



Remote Procedure Call (RPC)

- Asynchronous call



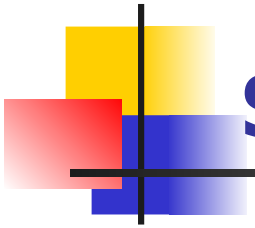


Central Components of Java RMI

- **Remote objects** - these are normal Java objects, but their class extends some RMI library class that incorporates support for remote invocation.

- **Remote references** - object references that effectively refer to remote objects, typically on a different computer.

- **Remote interfaces** - normal Java interfaces, that specify the “API” of a remote object. They should extend the marker interface, **java.rmi.Remote**. The remote interface must be known to both the local and remote code.

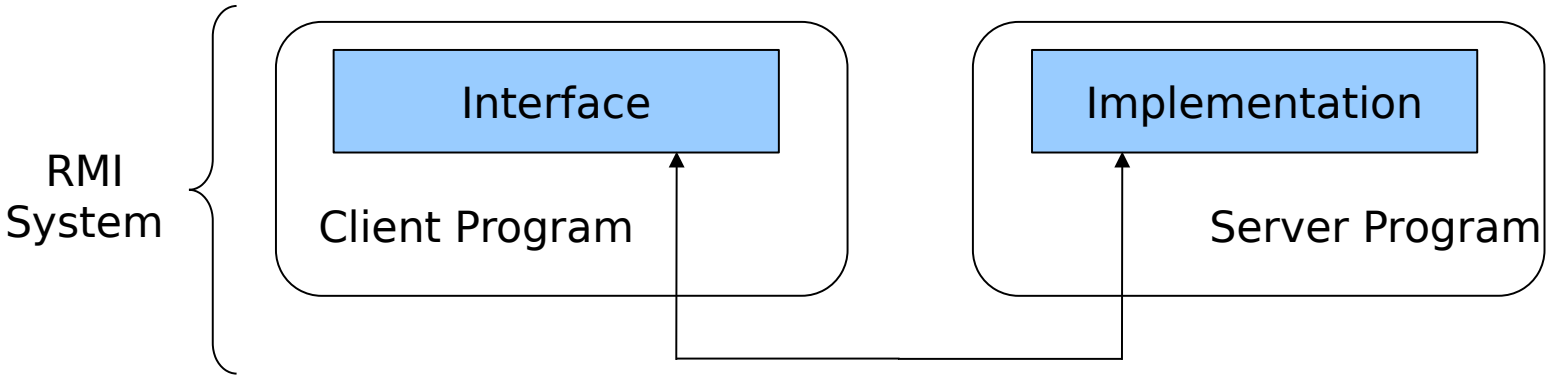


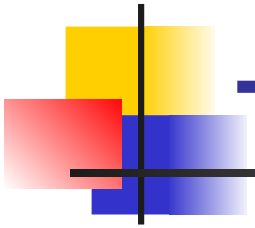
Some Supporting Technologies

- **Registries** - places where the local machine initially looks to find a reference to a remote object.
- **Serialization** - reduction of Java objects to a representation that can be communicated as a byte stream (for arguments and results).
- **Dynamic class loading** - needed in various places. One example is when a remote method returns an object whose class (e.g. **ResImpl**) was not previously known on the calling machine.
- **Security manager** - used to control the behavior of code loaded from a remote host.

Java RMI

- *Java RMI is interface based*
  - a remote object (or distributed service) is specified by its interface
    - *"interfaces define behaviour and classes define implementations"*
    - *in RMI:* ***Remote Interface***

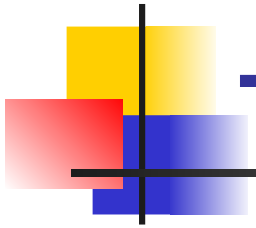




The RMI Registry

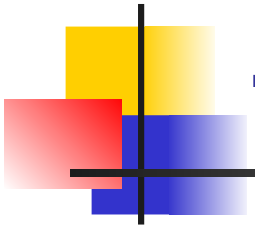
- The RMI Registry is a naming service
  - It is a separately Running service
    - Initiated using Java's "rmiregistry" tool
  - Server programs register remote objects
    - Given the object and a name
  - Client programs lookup object references that match this service name
  - By default, the registry runs on TCP port 1099. To start a registry on a different port: "rmiregistry #port "

- Registry names of the objects have a URL format
  - rmi://<hostname>:<port>/<ServiceName>
  - E.g. rmi://localhost:1099/CalculatorService
  - E.g. rmi://194.80.36.30:1099/ChatService



The RMI Registry

- The string name accepted by the RMI registry has the syntax "**rmi://hostname:port/remoteObjectName**", where:
  - **hostname** and **port** identify the machine and port, respectively, on which the RMI registry is running;
  - **remoteObjectName** is the string name of the remote object.
  - hostname, port and the prefix, "rmi:", are optional.
    - If hostname is not specified, then it defaults to the local host.
    - If port is not specified, then it defaults to 1099.
    - If remoteObjectName is not specified, then the object being named is the RMI registry itself.



The RMI Registry Interface

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name

void bind (String name, Remote obj)

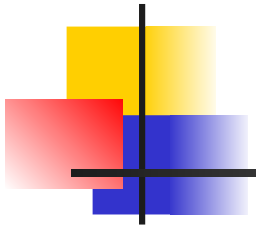
This method can alternatively be used by a server to register a remote object by name, but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

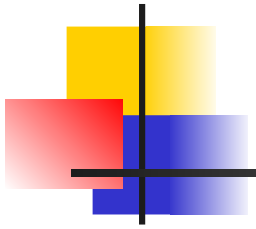
Remote lookup(String name)

This method is used by clients to look up a remote object by name
A remote object reference is returned.



Parameter Passing

- Parameter Passing in Java RMI is different from standard Java
  - *Reminder*: In Java, primitives are passed by value, objects are passed by reference
- In Java RMI
  - Objects and primitives are passed by value
  - Remote objects are passed by reference

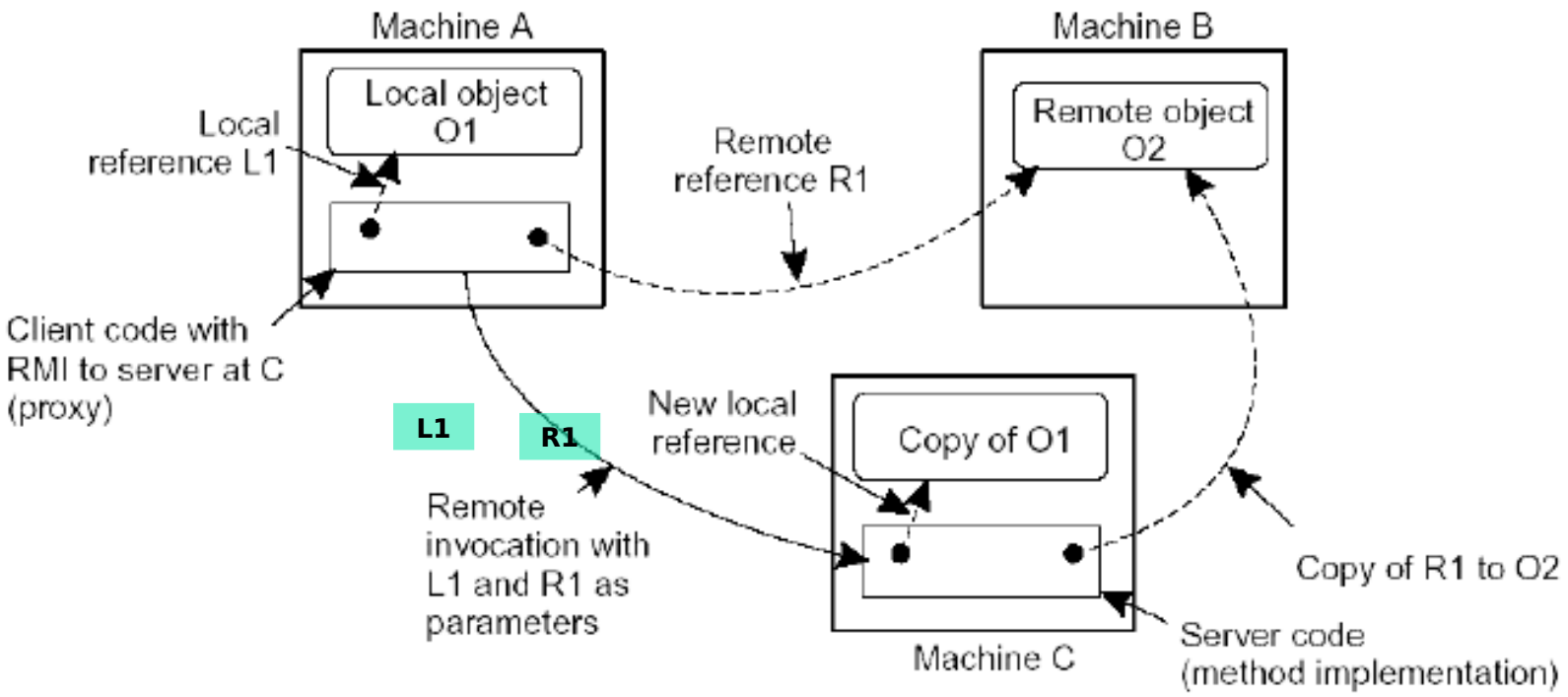


Parameter Passing

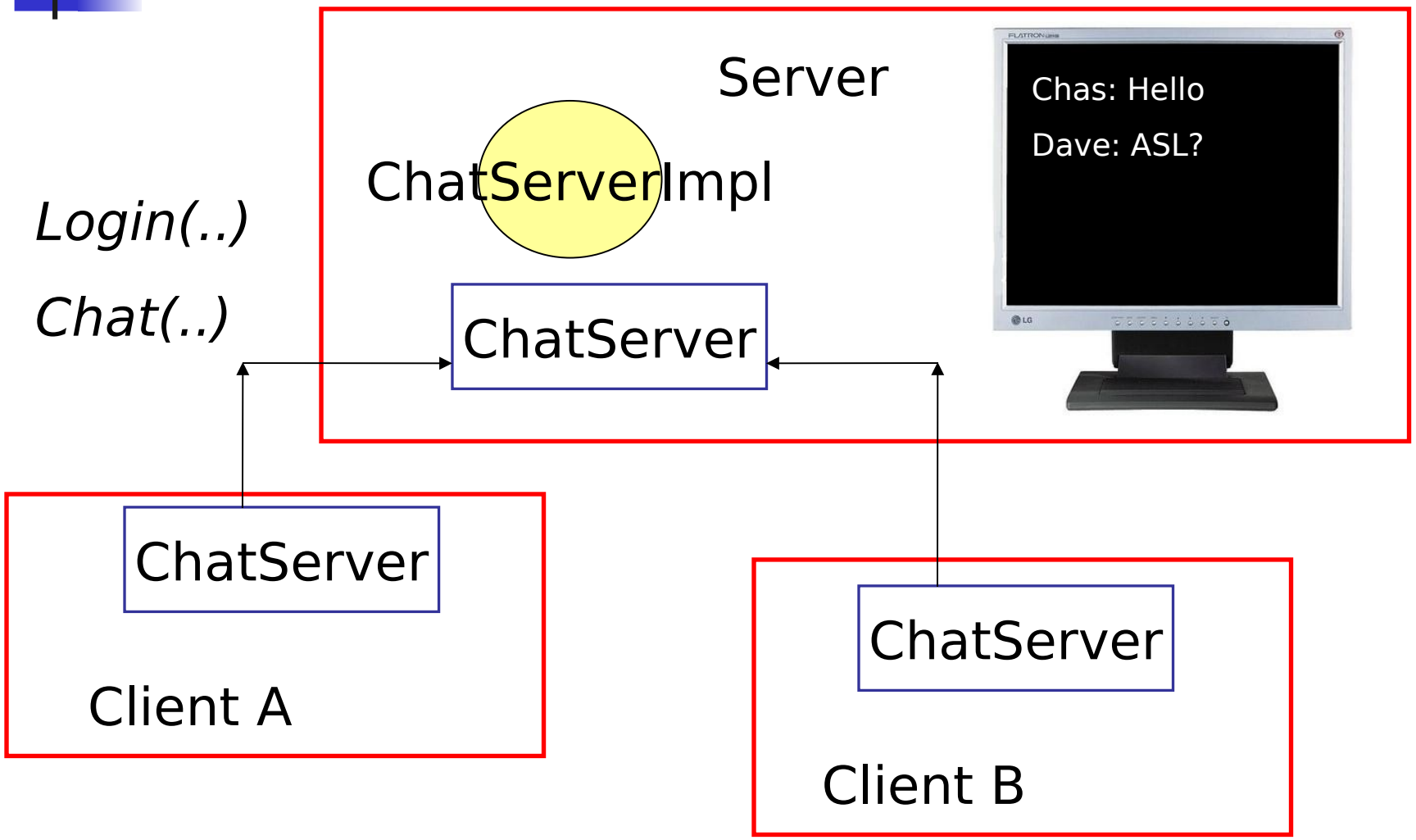
- RMI-Pass by Value
  - All ordinary objects and primitives are serialized and a copy is passed
  - Any changes to the copy **do not** affect the original

- RMI-Pass by Reference
  - Remote Object is the parameter, a stub (reference) is sent
  - the stub is used to modify the object, the original object is modified

Local and Remote References



A Chat Server Example



Building a Java RMI system

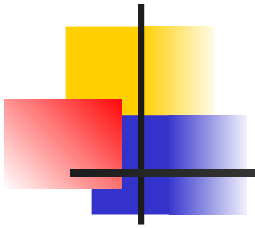
An RMI system must be composed of the following parts:

1. **An interface definition of the remote services;** 
2. **The implementations of the remote services;** 
3. **A server to host the remote services;** 
4. **An RMI Naming service;** 
5. **A client program that uses the remote services.** 

Step 1 - Define the Remote Interface

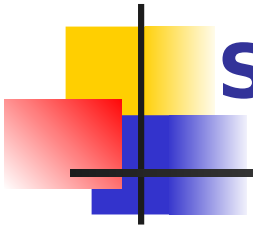
- Declare the methods you'd like to call remotely
  - This interface must extend java.rmi.Remote
  - Each method must declare java.rmi.RemoteException in its throws clause
    - This exception can occur for communication errors (connection failure, marshalling error, etc.)
  - You can have multiple remote interfaces

- Remember about parameter passing
  - Remote objects must be passed as remote interface types
  - Local objects must be serializable



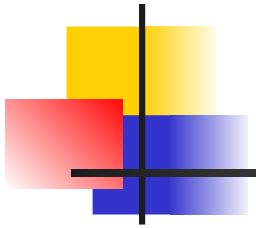
Example ChatServer Interface

```
public interface ChatServer extends java.rmi.Remote {

    public void login(String name, String password)
        throws java.rmi.RemoteException;

    public void logout(String name)
        throws java.rmi.RemoteException;

    public void chat(String name, String message)
        throws java.rmi.RemoteException;

}
```



Step 2 - Implement the remote service

- Your class must implement the Remote interface
- Extend this class with *UnicastRemoteObject*
  - Must provide a constructor that throws a RemoteException.
  - Call *super*() in the constructor
    - This activates code in UnicastRemoteObject that performs the RMI linking and remote object initialization.



Example Remote Object (ChatServiceImpl)

```
public class ChatServerImpl
    extends java.rmi.server.UnicastRemoteObject implements ChatServer {

    public ChatServerImpl() throws java.rmi.RemoteException {
        super();
    }
    public void login(String name, String pass) throws
       java.rmi.RemoteException {
        // Method Implementation
    }
    public void logout(String name) throws java.rmi.RemoteException {
        // Method Implementation
    }
    public void chat(String name, String msg) throws
       java.rmi.RemoteException {
        // Method Implementation
    }
}
```

Step 3 – Create the Server

- The server is a Java application
  - Creates one or more instances of remote objects
  - Binds at least one of the remote objects to a name in the RMI registry
    - Uses the Naming.rebind(“….”) operation

Example Chat Server

```
public class ChatServerAppl {
   public ChatServerAppl() {
     try {
                ChatServer c = new ChatServerImpl();
                Naming.rebind("ChatService", c);
     }
     catch (Exception e) {
       System.out.println("Server Error: " + e);
     }
   }//ChatServerAppl constructor
   public static void main(String args[]) throws java.rmi.RemoteException {
     //Create the new ChatServer
        new ChatServerAppl();
   } //main
} //ChatServerAppl
```

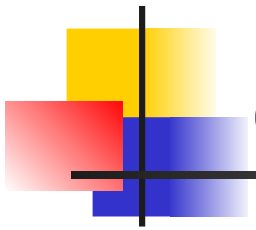
Step 5 – Create the Client

- Get a remote reference by calling Naming.lookup("...")
  - Remember - Lookup by service nam

- Receives a stub object for the requested remote object
  - Loads code for the stub either locally or remotely

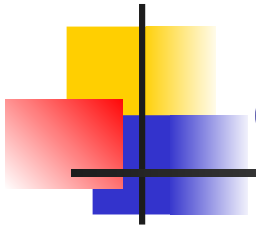
Stub class ← File System **OR** Network

- Invoke methods directly on the reference
  - Suchlike to standard Java objects



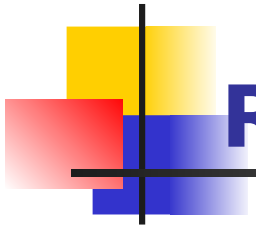
Example Chat Client

```
public class ChatClient {
    public static void main(String[] args) throws Exception {
        try {
          // Get a reference to the remote object through the rmiregistry
              ChatServer c = (ChatServer) Naming.lookup("ChatService");
           // Now use the reference c to call remote methods
              c.login(“Chas”,”*****”);
              c.chat(“Chas”, “Hello”);
              // Catch the exceptions that may occur - rubbish URL, Remote
exception
        } catch (RemoteException re)
{  System.out.println("RemoteException“+re);  }
   }
}
```



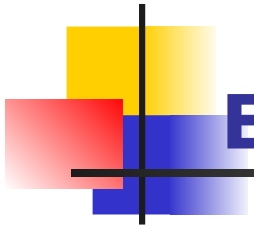
Compile

- Compile interfaces and classes
  - javac ChatServer.java ChatServerImpl.java ChatServerAppl.java
  - javac ChatServer.java ChatClient.java



Run

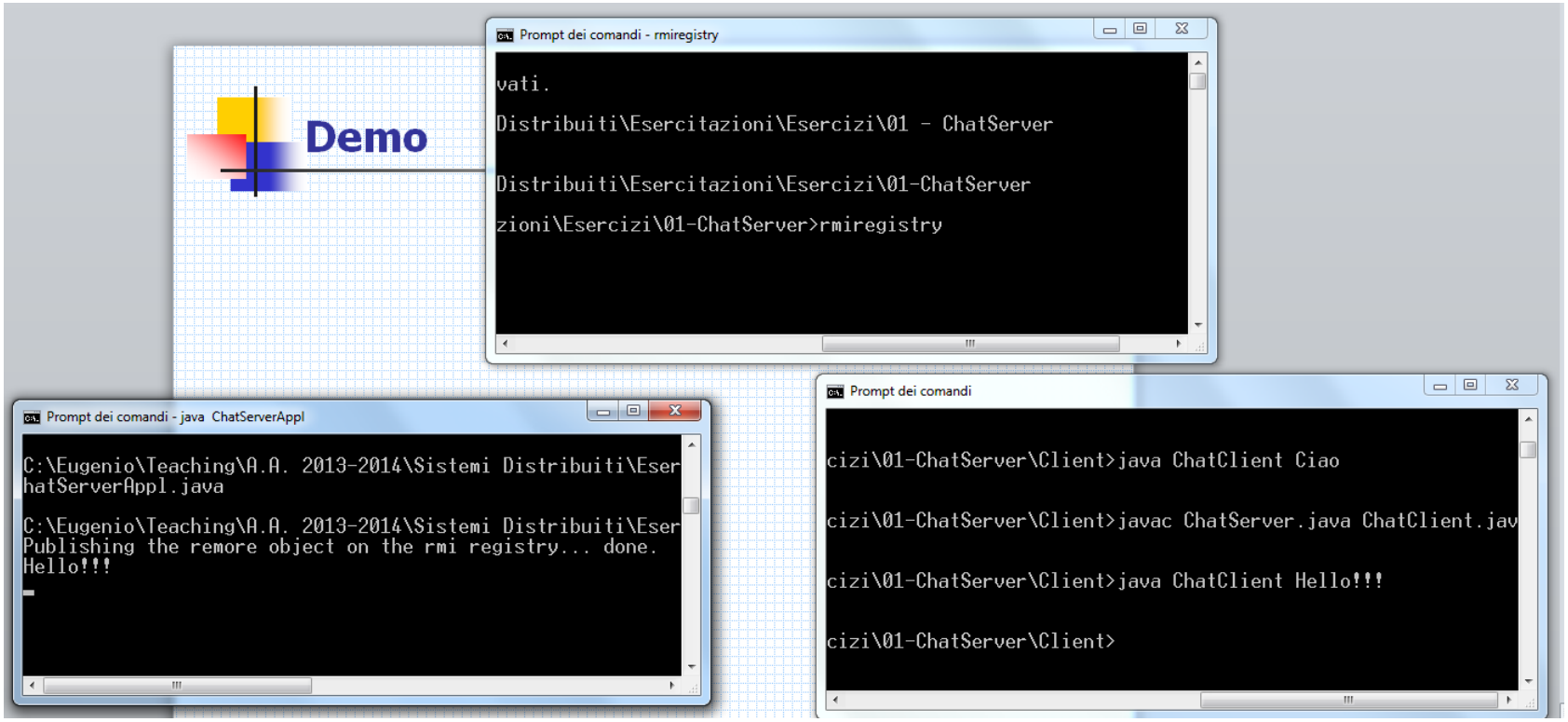
- Run Registry, ChatServerAppl e ChatClient
  - rmiregistry
  - java ChatServerAppl
  - java ChatClient

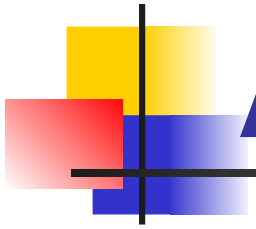


Easy way on localhost: use Eclipse!

- Run Registry as configured External Tools
  - Create a new External Tool configuration
  - Set Arguments as:
    - -J-Djava.rmi.server.codebase=file:///${workspace_loc}/sisdis/bin/
  - Location as: /usr/bin/rmiregistry
- Run ChatServerAppl as Java Application
- Run ChatClient as Java Application

Demo in classroom





Alternative to Naming class

- **Registry class**
- **LocateRegistry class**
- **Example:**

```
 // fire to localhost port 1099
Registry myRegistry = LocateRegistry.getRegistry("127.0.0.1", 1099);

// search for myMessage service
Message impl = (Message) myRegistry.lookup("myMessage");

// call server's method
impl.sayHello("HelloWorld!");
```

Client-side callback

Client-side callback

- In many architectures a server may need to “call” a client
- Examples include progress feedback, time tick notifications, warnings of problems, etc.
- *Client Callback*: a mechanism that allows a client to be NOTIFIED by a server for an event on which the former was waiting (it was registered on this event).

Server notify= Invocation of a remote method on the CLIENT

Client-side callback

- There is nothing really special about this as RMI works equally well between all computers.
- In order to enable such an activity, a RMI client has to act as RMI server (i.e., P2P communication)
- However, it may be impractical for a client to extend the "`java.rmi.server.UnicastRemoteObject`" class
- In this case, a remote object (on the client) can be ready for the "remote" usage by calling the static method

  " `UnicastRemoteObject.exportObject(remote_object,port)` "

Client-side callback
(CLIENT-SIDE steps)

- On the client it is necessary to provide the remote method that the server invokes to perform the callback

- STEPS:
  1. Define the remote interface on the client in which the method invoked by the server to perform the callback is declared.
  2. Implement the remote interface of the client (i.e., the callback method)
  3. Create an instance of the implementation of the remote interface and pass it to the server that will use it to perform the callback .

A running example

- A **TimeServer** provides the date and current time
- The **TimeClient** wants to receive periodic updates (call-backs) with the current date and time (it will act as server in listening for updates).
- On the server
  1. Define the **TimeServer interface** of the server (as usual) and adds the "`registerTimeMonitor`" method (it will be used by the client to send the reference on which the callback has to be done)
  2. Implement the "TimeServerImpl" remote object, create an instance and publish it on the registry (as usual)
- On the Client
  1. Define the **TimeMonitor interface** for the client, including the method ("`time`") used for the call-back (that will be invoked by the server)
  2. Implement the "TimeClient" (implementing the "TimeMonitor" interface), that has to register itself on the server by invoking the "`registerTimeMonitor`" method (passing as parameter a reference to itself)

Client-side callback

```
import java.rmi.*;
import java.util.Date;
public interface TimeMonitor extends java.rmi.Remote {
    public void time(Date d) throws RemoteException;
}
```

it will be invoked by the server for the client callback

```
import java.rmi.*;
public interface TimeServer extends java.rmi.Remote {
    public void registerTimeMonitor(TimeMonitor tm)
                throws RemoteException;
}
```

it will be invoked by the client to send the reference on which the callback has to be done

Client-side callback (TimeTicker running on the SERVER)

```
import java.util.Date;
class TimeTicker extends Thread {
    private TimeMonitor tm;
    TimeTicker( TimeMonitor tm ){
            this.tm = tm;
    }
    public void run()    {
            while(true)
                    try{
                            tm.time(new Date());
                            sleep( 2000 );
                    } catch ( Exception e ) { System.out.println(e); }
    } //run
} //TimeTicker class
```

Client-side callback (SERVER)

```
import java.net.*; import java.io.*; import java.util.Date;
import java.rmi.*; import java.rmi.server.*; import java.rmi.registry.LocateRegistry;

public class TimeServerImpl extends UnicastRemoteObject

    implements TimeServer {


    public void registerTimeMonitor( TimeMonitor tm ) {
            System.out.println( "Client requesting a connection" );
            TimeTicker tt;
            tt = new TimeTicker( tm );
            tt.start();
            System.out.println( "Timer Started" );
    }
    ...
```

Client-side callback (SERVER)

```
private static TimeServerImpl  tsi;

public static void main (String[] args) {
        try {
                tsi = new TimeServerImpl();
                LocateRegistry.createRegistry(1099);
                System.out.println("Registry created");
                Naming.rebind("TimeServer", tsi);
                System.out.println( "Bindings Finished" );
                System.out.println( "Waiting for Client requests" );
        } catch (Exception e) {  System.out.println(e);          }
} //main


}  // class TimeServerImpl
```

Alternative (non usual) way to start the RMI registry:
it creates and exports a Registry instance on the local host that accepts requests on the specified port.

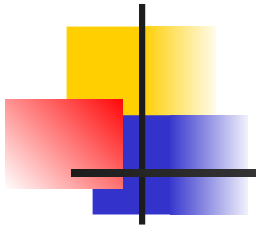
Client-side callback (CLIENT)

```
import java.util.Date;
import java.net.URL;
import java.rmi.*;
import java.rmi.server.*;
public class TimeClient implements TimeMonitor {
    private TimeServer ts;
    public TimeClient() {
            try {
                    System.out.println( "Exporting the Client" );
                    UnicastRemoteObject.exportObject(this,1098);
                    ts = (TimeServer)Naming.lookup(
                            "rmi://localhost:1099/TimeServer");
                    ts.registerTimeMonitor(this);
            } catch (Exception e){ System.out.println(e); }
    }//constructor
    ...
```

It makes the client ready for the "remote" usage. Use a port not used by other services.

Client-side callback (CLIENT)

```
    ...
    public void time( Date d ){
            System.out.println(d);
    }

    public static void main (String[] args) {
            new TimeClient();
    }

} //class TimeClient
```

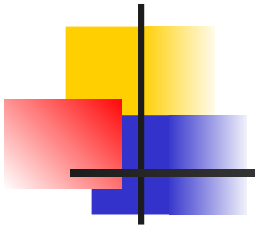


Client-side callback Demo and homework

1. **Demo in classroom (eclipse)**

2. **Homework: Extend to multi-threaded client and parameterize notification period, port number, etc.**

Distributed Garbage Collector



Distributed Garbage Collection (DGC)

- In Java the programmer has not to deal with memory concerns.
- The JVM is endowed with a Garbage Collector (GC) which takes care of freeing memory by deleting objects that are no more used by any running program.
- The design of an efficient GC is difficult, the problem becomes more challenging in a distributed context (i.e., RMI).
- RMI provides a Distributed Garbage Collector (DGC) based on the *reference counting* technique.

DGC: how does it work ?

- In RMI the DGC works by keeping trace of the clients that have an active reference on a remote object running on the server.

- When a **remote object is referenced**, the server marks it as "**dirty**" and increases its reference count by one.

- When a client drops a reference, the DGC decreases the object reference count by one, and marks the object as *clean*. When the reference count reaches zero, the remote object is free of any live client references. It is then placed on the ***weak reference* list** and subject to periodic garbage collection.

DGC: the “Unreferenced” interface

- The mechanisms handling the DGC are hidden within the stub (skeleton) layer.
- Nevertheless, the Distributed Garbage Collection mechanisms can be monitored…
- A remote object can implement the **java.rmi.server.Unreferenced** (unreferenced method) interface, through which can receive notifications if there are no more clients with an active reference.

DGC: the “lease” time

- An active reference of a client on a remote object has, in addition to the reference counting, also a specific *lease time*
- If the client does not “refresh” the connection with the remote object before the lease time expires, the reference is considered not active and the remote object can be destroyed.
- The *lease time* is handled by the **java.rmi.dgc.leaseValue** property.
- The lease time is expressed in milliseconds and the default value is 10 minutes (600K ms).
- Because of these mechanisms, the client should be prepared to deal with “no more existent” remote objects.

DGC: the basic algorithm (1/2)

The basic algorithm is this:

1. A client calls the server and requests a lease for a period of time.
2. The server responds back, granting a lease for a period of time (not necessarily the original amount of time).
3. During this period of time, the distributed reference count includes the client.
4. When the lease expires, if the client hasn't requested an extension, the distributed reference count is automatically decremented.

DGC: the basic algorithm (2/2)

- Clients automatically try to renew leases as long as a stub hasn't been garbage collected.
- The remote object is eventually garbage collected:
  - If the client crashes, then the client application is no longer running, and therefore, the client application certainly isn't attempting to renew leases.
  - If network problems prevent the client application from connecting to the server, the client application won't renew the lease.

DGC: a code example

- Let suppose to have two remote objects:
  1) Hello (used by clients to get remote references to object of type MessageObject)
  2) MessageObject

- The implementations of these objects print out information about when the objects are created, destroyed, not referenced and finalized.

- A remote object can implement the *Unreferenced* interface and its only method, *unreferenced*. This method is called by the DGC when it removes the last remote reference to the object.
  - *MessageObjectImpl* and *HelloImpl* are designed to print a message when this happens.

DGC: a code example

- *MessageObjectImpl* and *HelloImpl* implement the *finalize* and *unreferenced* methods.
  - *finalize* is called by the local GC (class java.lang.Object)
  - *unreferenced* by the DGC

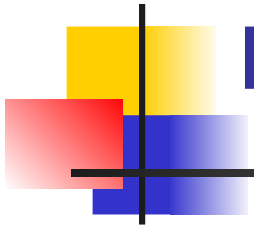
DGC: a code example

- This example can be executed by explicitly setting the heap size and the *leaseValue* for the DGC.

```
java –Xmx512m -Djava.rmi.dgc.leaseValue=10000
  RMIServer
```

where the time for the *leaseValue* is expressed in milliseconds

java.rmi.dgc.leaseValue: The value of this property represents the lease duration (in milliseconds) granted to other VMs that hold remote references to objects which have been exported by this VM. Clients usually renew a lease when it is 50% expired, so a very short value will increase network traffic and risk late renewals in exchange for reduced latency in calls to Unreferenced.unreferenced. The default value of this property is 600000 milliseconds (10 minutes).



DGC Example: Hello Interface

```
import java.rmi.*;

public interface Hello extends java.rmi.Remote

{

    String sayHello() throws RemoteException;

    MessageObject getMessageObject() throws RemoteException;

}
```

DGC Example: HelloImpl

```
import java.rmi.*; import java.rmi.server.*;
public class HelloImpl extends UnicastRemoteObject
                implements Hello, Unreferenced {

    public HelloImpl() throws RemoteException  {
                super();
    }
    public String sayHello() throws RemoteException {
                return "Hello!";
    }
    public MessageObject getMessageObject() throws RemoteException {
                return new MessageObjectImpl();
    }
    ...
```

DGC Example: HelloImpl

```
    ...
    public void unreferenced() {
            System.out.println( "HelloImpl: Unreferenced" );
    }
    public void finalize() throws Throwable {
            super.finalize();
            System.out.println( "HelloImpl: Finalize called" );
    }
} // class HelloImpl
```

DGC Example: MessageObject interface

```
import java.io.*;

import java.rmi.server.*;

public interface MessageObject extends java.rmi.Remote {

    int getNumberFromObject() throws java.rmi.RemoteException;

    int getNumberFromClass() throws
    java.rmi.RemoteException;

}
```

DGC Example: RMIServer

```
import java.net.*;
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.LocateRegistry;
public class RMIServer {
    private static final int PORT = 10007;
    private static final String HOST_NAME = “name";
    private static RMIServer  rmi;

    public static void main ( String[] args )  {
            try {
                    rmi = new RMIServer();
            } catch ...
            ...
```

DGC Example: RMIServer

```
            catch ( java.rmi.UnknownHostException uhe ) {
                        System.out.println( "The host computer name you
                                                have specified, " + HOST_NAME + "
does not
                                                                match your real
computer name." );
            }
            catch ( RemoteException re ) {
                        System.out.println( "Error starting service" );
                        System.out.println( "" + re );
            }
            catch ( MalformedURLException mURLe ) {
                        System.out.println( "Internal error" + mURLe );
            }
            catch ( NotBoundException nbe ) {
                        System.out.println( “Error: an attempt is made to lookup or
                                                unbind in the registry a name that
has no  associated binding. Not Bound" );
                        System.out.println( "" + nbe );
            }
}// main
...
```

DGC Example: RMIServer

```
    ...
    public RMIServer() throws RemoteException,
                              MalformedURLException,
    NotBoundException {
            LocateRegistry.createRegistry( PORT );
            System.out.println( "Registry created on host computer" +

                    HOST_NAME + " on port " + Integer.toString(PORT)
    );
            Hello hello = new HelloImpl();
            System.out.println( "Remote Hello service
                    implementation object created" );
            String urlString = "//" + HOST_NAME + ":" +
                    PORT + "/" + "Hello";
            Naming.rebind( urlString, hello );
            System.out.println( "Bindings Finished, waiting for
                    client requests." );
    }
} // class RMIServer
```

Creates and exports a Registry instance on the local host that accepts requests on the specified port.

DGC Example: RMIClient

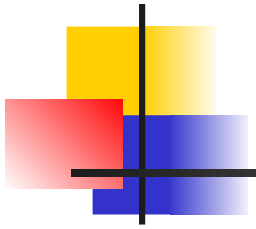
```
import java.util.Date;
import java.net.MalformedURLException;
import java.rmi.*;
public class RMIClient {
    private static final int PORT = 10007;
    private static final String HOST_NAME = "name";
    private static RMIClient  rmi;

    public static void main ( String[] args ) {
            rmi = new RMIClient();
    }
    ...
```

DGC Example: RMIClient

```
...
public RMIClient()  {
        try {
                Hello hello = (Hello)Naming.lookup( "//" +
                        HOST_NAME + ":" + PORT + "/" + "Hello" );
                System.out.println( "HelloService lookup successful" );

                System.out.println( "Message from Server: " +
hello.sayHello() );
                MessageObject mo;
                for ( int i = 0; i< 1000; i++ ) {
                        mo = hello.getMessageObject();
                        System.out.println( "MessageObject: Class
                        Number is #" + mo.getNumberFromClass() +
                                " Object Number is #" +

                                                mo.getNumberFromObject() );
                    mo = null;
                    Thread.sleep(500);
                }//for
                 ...
```

DGC Example: RMIClient

```
            ...
            } catch ( Exception e){ System.out.println(e); }
    }
}  // class RMIClient
```

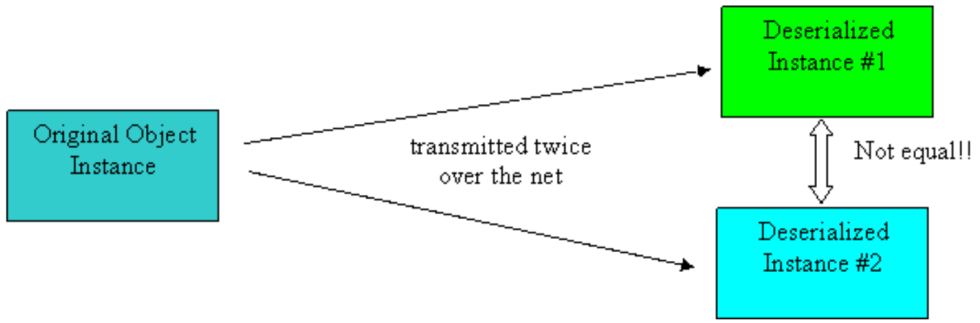


Demo in classroom

Distributed Garbage Collector

RMI pitfall: equals and hashCode

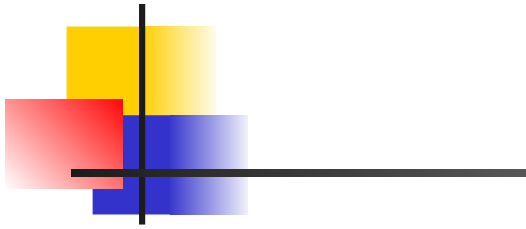
- In a networked application, we are often serializing objects and sending around the globe.
- Sometimes we need to compare those objects against each other….
- But the problem is that the *equals()* method, inherited from Object, by default, uses a direct comparison of the *hashcode* of the object, which in turn is derived from its memory location.
- This means that two deserialized instances of the same original object will return false if compared using equals()!
- If we redefine the methods in the remote object, this will not affect stubs, as they are mechanically generated



RMI pitfall: equals and hashCode

- In order for a remote object to be used as a key in a hash table, the methods equals and hashCode need to be overridden in the remote object implementation.
- These methods are overridden by the class java.rmi.server.RemoteObject:

- The java.rmi.server.RemoteObject class's implementation of the equals method determines whether two object references are equal, not whether the contents of the two objects are equal. This is because determining equality based on content requires a remote method invocation, and the signature of equals does not allow a remote exception to be thrown.
- The java.rmi.server.RemoteObject class's implementation of the hashCode method returns the same value for all remote references that refer to the same underlying remote object (because references to the same object are considered equal).

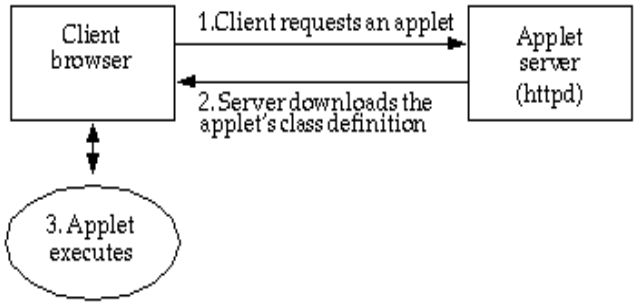


Dynamic class downloading

Dynamic class downloading

- A striking feature of the Java platform is the possibility to dynamically download and run classes from an URL (*Uniform Resource Locator*).

- Therefore, a JVM can execute an application that has not been defined on the system where it is running.
  - Example: executing an applet
    - Once the classes have been downloaded from the server, the client browser can execute the applet by exploiting its local resources.



Dynamic class downloading in RMI

- To run a RMI application, the supporting class files must be placed in locations that can be found by the server and the clients

- For the server, the following classes must be available to its class loader:
  - Remote service interface definitions
  - Remote service implementations
  - Stubs for the implementation classes
  - All other server classes
- For the client, the following classes must be available to its class loader:
  - Remote service interface definitions
  - Stubs for the remote service implementation classes
  - Server classes for objects used by the client (such as return values)
  - All other client classes

- Once you know which files must be on the different nodes, it is a simple task to make sure they are available to each JVM's class loader.

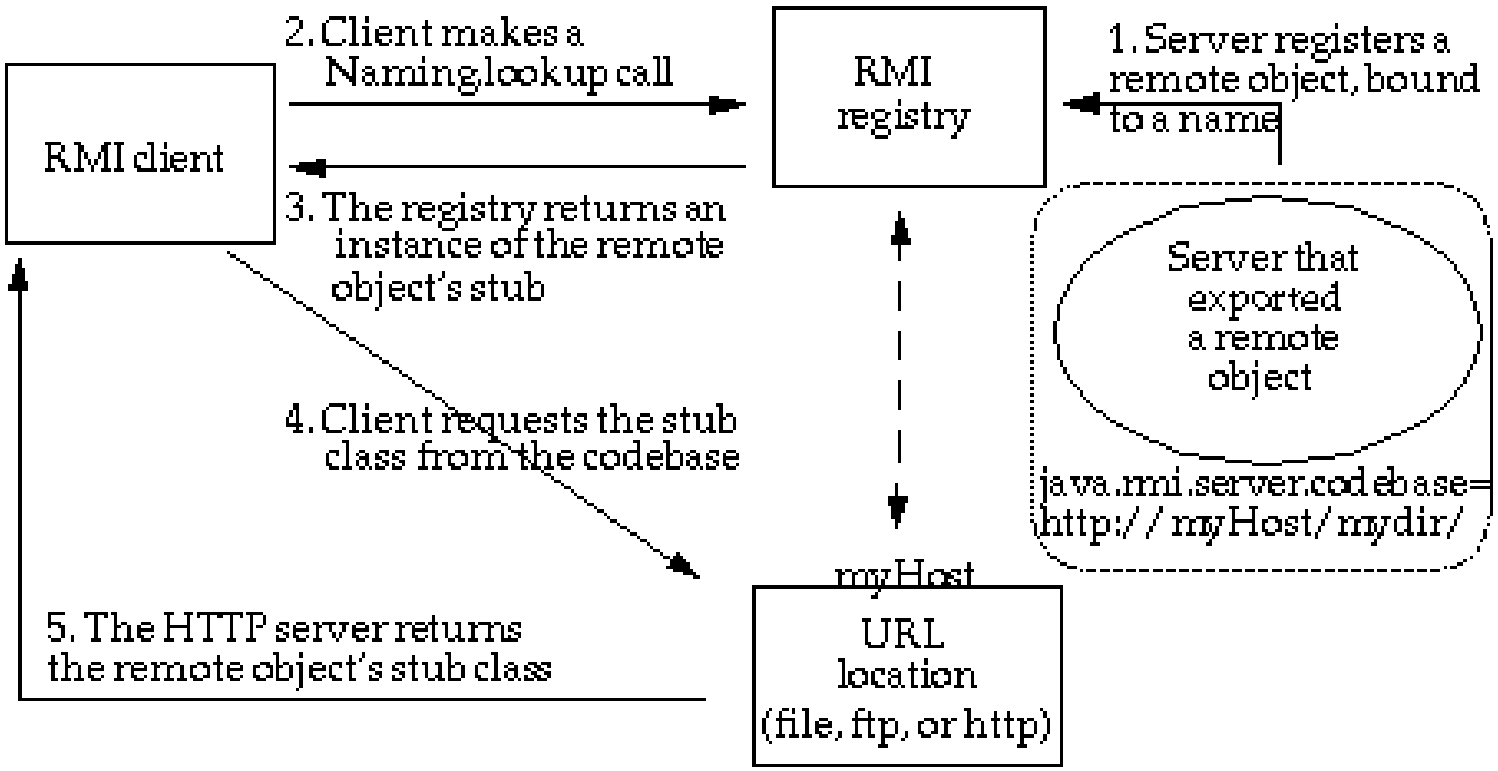
Dynamic class downloading in RMI

- When a Java program uses a *Class Loader*, it needs to know the location/s where finding the classes needed by the program.
  - The Java Class Loader is a part of the JRE that dynamically loads Java classes into the JVM
- Usually, a *Class Loader* is coupled with an HTTP server that makes available the classes.
- A ***codebase*** can be defined in order to specify where classes needed by a JVM are located.

Dynamic class downloading in RMI

- In RMI, the property java.rmi.server.codebase defines one or more URLs from which stubs (and related classes) can be loaded
  - This URL points to a “file:”, “ftp:”, or “http:” location that supplies classes for objects that are sent *from* this JVM.
  - If a program running in a JVM sends an object to another JVM (as the return value from a method), then such a JVM needs to load the class file for that object.
  - When RMI sends the object via serialization of RMI, embeds the URL specified by this parameter into the stream, alongside of the object.
  - RMI does not send class files along with the serialized objects (only the URL).
- If the remote JVM needs to load a class file for an object, it looks for the embedded URL and contacts the server at that location for the file.
- Generally, the classes needed to handle calls to remote methods have to be made available by network resources such as HTTP or FTP servers
  - It is very unusual that classes can be retrieved by a “file:” URL: this would mean that client and server reside on the same host

How *codebase* works in RMI

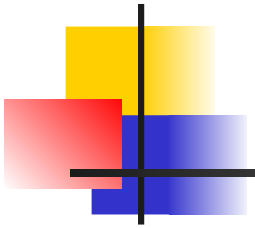


Using *codebase* in RMI

1) The *codebase* of the remote object is specified by the server through the *java.rmi.server.codebase* property

```
java -Djava.rmi.server.codebase=http://webserver/export/
```

2) The RMI server registers an object by the rmiregistry
3) The *codebase* settled on the server JVM is linked to the reference of the remote object in the RMI registry.
4) The client requests a reference to a given remote object. This reference (i.e., an instance of the stub of the remote object) will be exploited by the client to invoke methods on the remote object.
5) RMI registry returns a reference (i.e., a stub instance) to the requester class.



Using *codebase* in RMI

6) If the stub class can be found in the local CLASSPATH of the client, it will be loaded by the client
(REMARK: The local CLASSPATH is the place where the application firstly looks for classes)

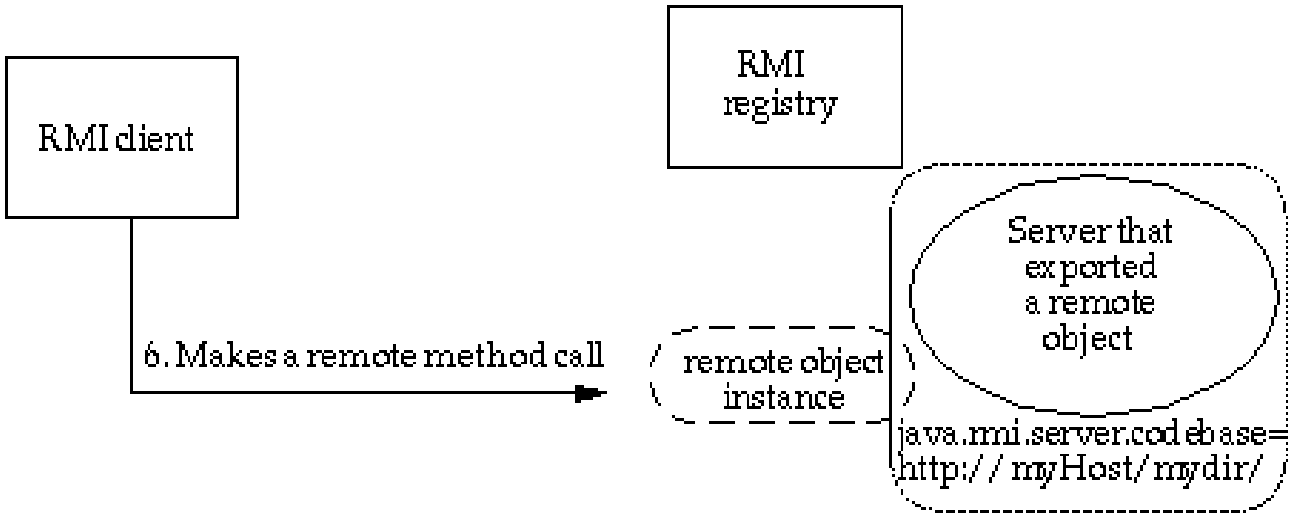
7) If the stub class cannot be found in the local CLASSPATH of the client, then it will attempt to search on the codebase of the remote object.

8) The codebase exploited by the client is the URL that has been annotated to the instance of the stub when the stub class has been loaded from the registry
   - the URL is embedded in the stream (serialization) sent to the client, alongside of the object

9) The definition of the stub class (and other related classes) are loaded by the client.

Using *codebase* in RMI

10) Now, the client has all the information needed to invoke the methods on the remote object.



The difference with an applet is that:

- the applet uses the codebase in order to execute code on the local JVM
- the RMI client uses the codebase of the remote object in order to execute code on a remote JVM

Using the *codebase*

- The value of a codebase can refer to:
  - An URL of a directory in which classes are organized in package
  - The URL of a JAR file
  - A string containing multiple instances of JAR files and/or directory.

- IMPORTANT NOTE:
  - When the codebase refers to an URL of a directory, its value MUST end with a "/”

Using the *codebase*: examples

- Case 1. If the classes to be loaded are located on an HTTP server called “webserver”, in the “export” directory under the “web root”, the codebase has to be settled as follows:

  -Djava.rmi.server.codebase=http://webserver/export/

- Case 2. If the classes are located in a JAR file called "mystuff.jar“, in the “public” directory (under the “web root”), the codebase takes the following value:

  -Djava.rmi.server.codebase=http://webserver/public/mystuff.jar

- Case 3. If the classes to be loaded are located in two JAR files: "myStuff.jar" and "myOtherStuff.jar“, and these files are on two distinct servers called "webserver1“ and "webserver2“ respectively, the codebase takes the following value:

  -Djava.rmi.server.codebase="http://webserver1/myStuff.jar http://webserver1/myOtherStuff.jar" (*separated by space*)

How to use dynamic class loading

- It is necessary to use a Security Manager, otherwise the download of class from remote codebase is disabled for security reasons.

```
java -Djava.security.manager -Djava.security.policy==policy.all App
```

- It is necessary to allow the usage of other codebases.

```
-Djava.rmi.server.useCodebaseOnly=true
```

- Since Java 1.7 java.rmi.server.useCodebaseOnly is false by default
  - If this value is true, automatic loading of classes is prohibited *except* from the local CLASSPATH and from the java.rmi.server.codebase property set on this VM.
  - Use of this property prevents client VMs from dynamically downloading bytecodes from other codebases.

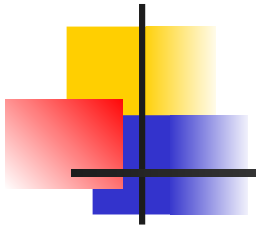
How to use dynamic class loading

- To allow the remote download of their classes, the java application must expose its codebase

```
System.setProperty("java.rmi.server.codebase", "http://code/bin/");
```

```
-Djava.rmi.server.codebase=http://code/bin
```

- The codebase URI is sent with the remote object



Dynamic class downloading

1. **Demo in classroom**