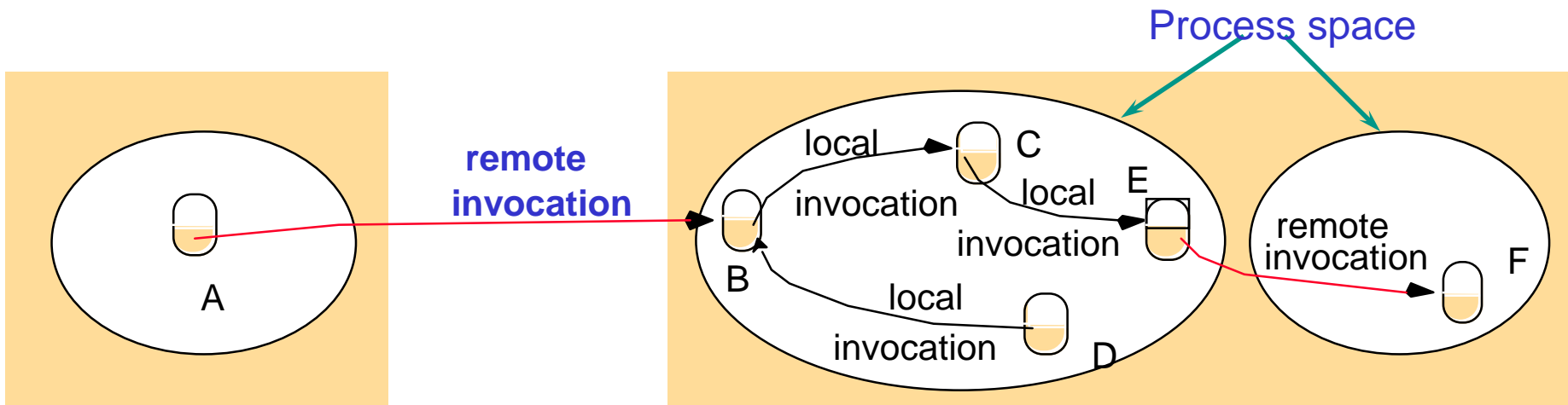# Java RMI

Agenda

- RMI basic
- RMI under the hood
- Dynamic Class Loading
- Security
- Distributed Garbage Collection
- Activation
- Custom RMI sockets
- Development of RMI servers
- RMI/IIOP

RMI Basic

- Java Remote Method (RMI) Invocation allows you to write distributed objects using Java

  – Reference remote Java objects in other JVMs

  – Call methods on the remote Java objects

RMI Basic

- Java RMI is a specification developed by Sun, Oracle, IBM and others
  - java.rmi Package
  - java.rmi.dgc Package
  - java.rmi.registry Package
  - java.rmi.server Package
  - java.rmi.activation Package
- Sun provides an default implementation of Java RMI as a part of the JDK
  - RMI/JRMP
- There are however hooks in the java.rmi interfaces to the JRMP default implementation

RMI Basic
Fundamental design principals

- Distributed Garbage Collection

- Exceptions to report errors

- Java native Interfaces used to expose implementations without exposing the exact source of implementation (No IDL)

- Normal method invocation is used to call methods on remote Java objects

- Dynamic Class Loading to have a mechanism to provide classes, that are not available on the system classpath

RMI Basic
Differences to "normal" Java

- Remote Exception
  - All remote method calls can throw an RemoteException

- Pass-by-value
  - All objects that are not remote objects are copied during a remote method call

- Call overhead
  - Remote method calls has to been accounted for in the design

- Security
  - Because the call is sent over the network

- Not true location transparency as you must at some point know the network name of the machine where the distributed object is living

RMI Basic
Tools provided

- **RMIC**
  - Java RMI Stub compiler

- **RMIRegistry**
  - Java RMI remote object registry

- **RMID**
  - Java RMI activation system daemon

RMI Basic
How to make a distributed Java object

1. To make a distributed object you first make a remote interface specifying which methods to be invoked by remote clients

```
package examples.hello;

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

public interface Hello extends Remote {
    String sayHello() throws RemoteException;
}
```

RMI Basic
How to make a distributed Java object

2. You make an server implemenation of the remote interface defined in 1.

```
package examples.hello;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

public class HelloImpl extends UnicastRemoteObject implements Hello {
  public HelloImpl() throws RemoteException {
  }

  public String sayHello() {
    return "Hello World!";
  }
}
```

! To export the remote object which might fail

RMI Basic
How to make a distributed Java object

- The Java remote object implementation must inherit from *UnicastRemoteObject* which provide the following:

  - export the object in its default constructor, so the remote object implementation can accept incoming remote calls from remote clients

  - the remote object implementation runs all the time

  - uses Java RMIs default sockets-based transport for communication (TCP)

RMI Basic
How to make a distributed Java object

- If you want to inherit from another class that *UnicastRemoteObject* you will have to call *UnicastRemoteObject.exportObject(..)* yourself in the constructor

- Stubs and skeletons are automatically generated by the *rmic* tool provided by JDK

  rmic examples.hello.HelloImpl

RMI Basic
How to make a distributed Java object

3. You implement a server that do the following

- create and install a SecurityManager

- create one or more instances of the remote Java object

- register the objects in the Naming Service so clients can retrieve remote references to the remote object by a lookup

  - The Naming Service is in the Java RMI implementation called RMI Registry '*rmiregistry*'

RMI Basic
How to make a distributed Java object

```java
public class Server {

  public static void main(String args[]) {
    // Create and install a security manager
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }
    try {
     // Create remote Java object
     HelloImpl obj = new HelloImpl();
     // Bind this object instance to the name "HelloServer"
     Naming.rebind("rmi://myhost/HelloServer", obj);
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
}
```

RMI Basic
How to make a distributed Java object

4. Make a Client accessing the remote Java object

```java
public class HelloClient {
  public static void main(String[] args) {
    // "obj" is the identifier that we'll use to refer
    // to the remote object that implements the "Hello" interface
    Hello obj = null;
    try {
      obj = (Hello)Naming.lookup("HelloServer");
      System.out.println(obj.sayHello());
    } catch (Exception e) {
      System.out.println("HelloServer exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

RMI Basic
How to make a distributed Java object

5. Start the Naming Service RmiRegistry

    start rmiregistry

6. Start the rmi server
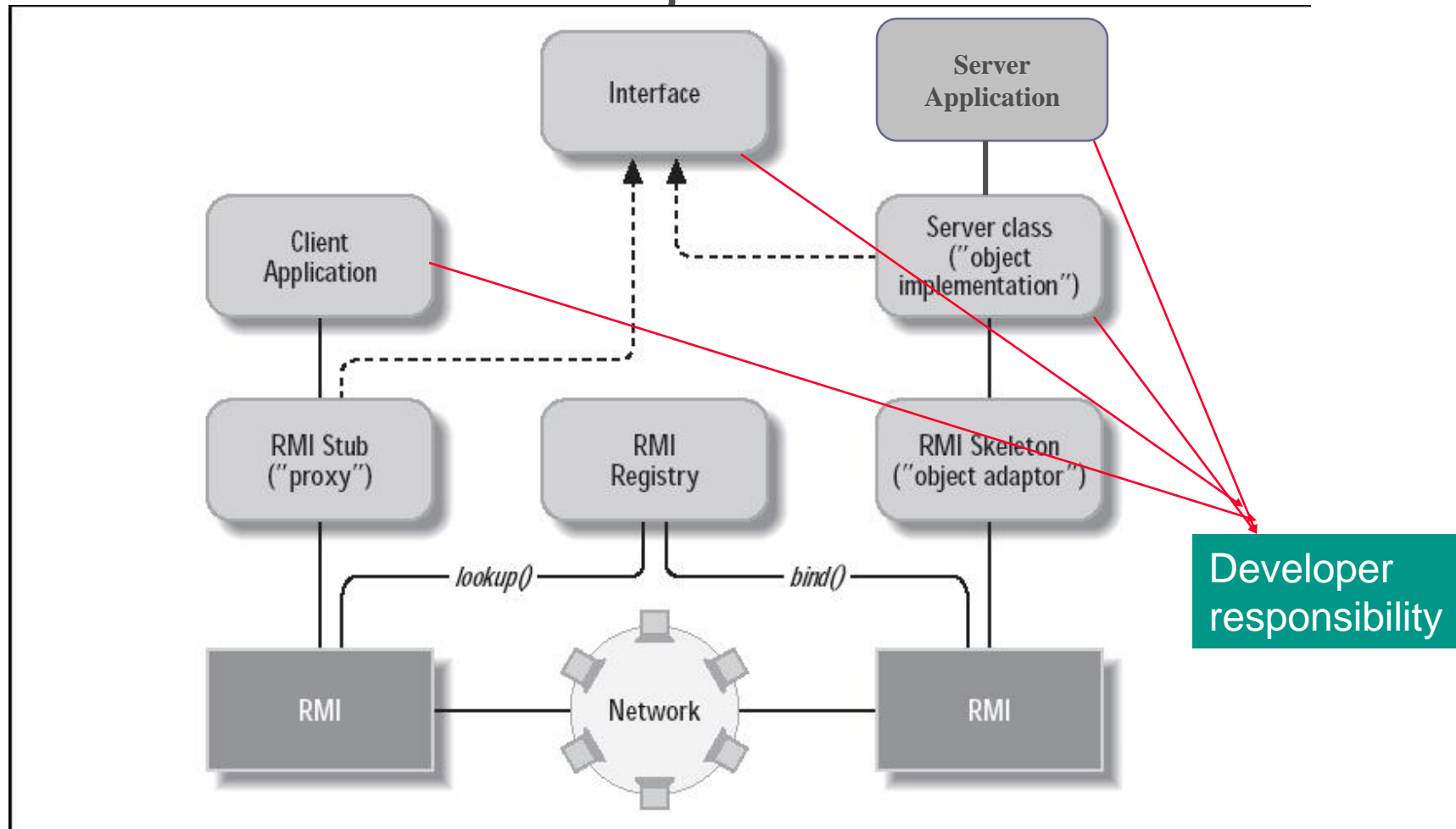
    start java examples.hello.HelloServer

7. Run the client
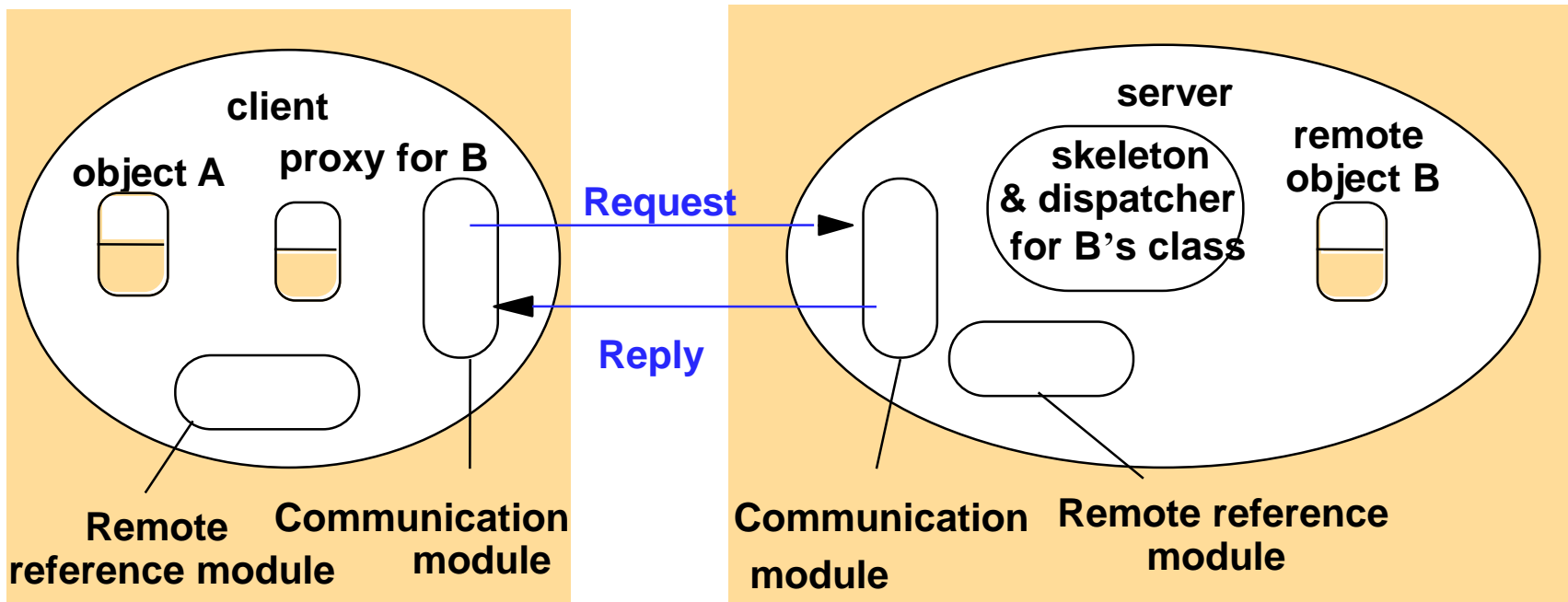
    java examples.helloHelloClient

    ➡ *HelloWorld!*

RMI Basic
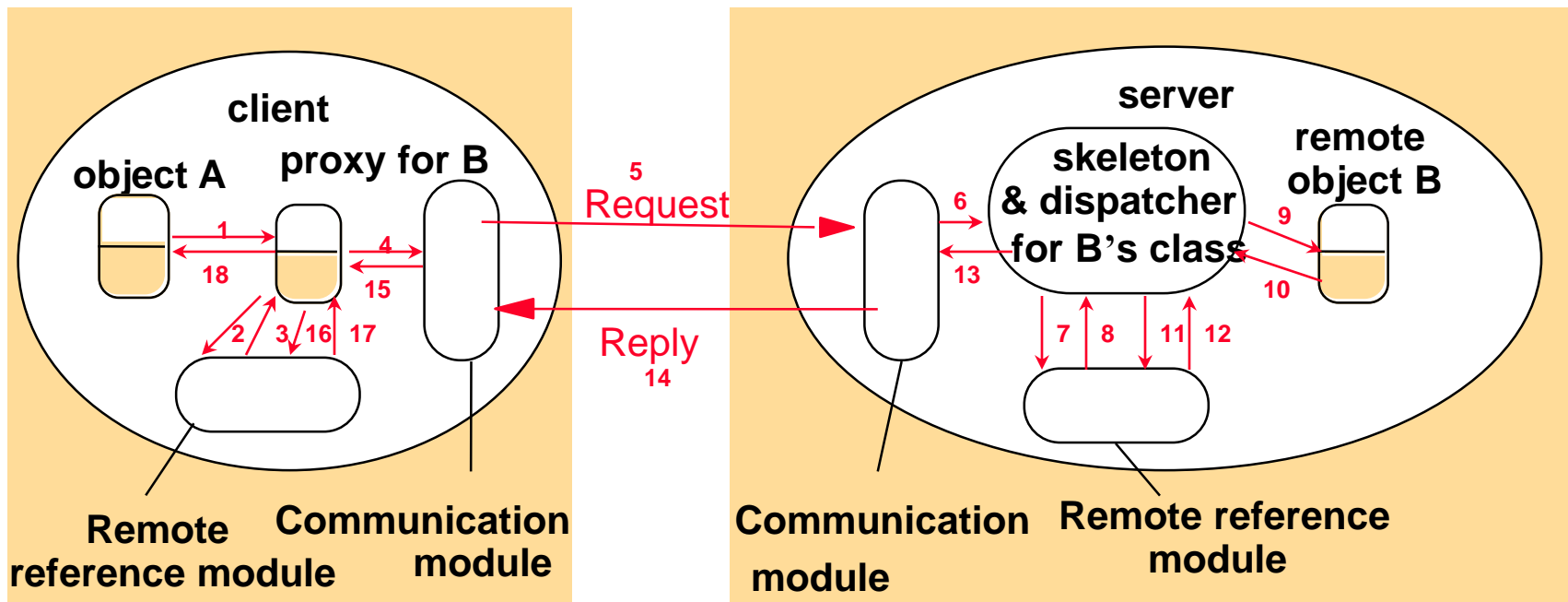## How to make a distributed Java object

RMI Under the hood
Architecture - Generic

Most distributed object systems are build upon the above generic RMI implementation (DCOM/COM, CORBA and also Java RMI)
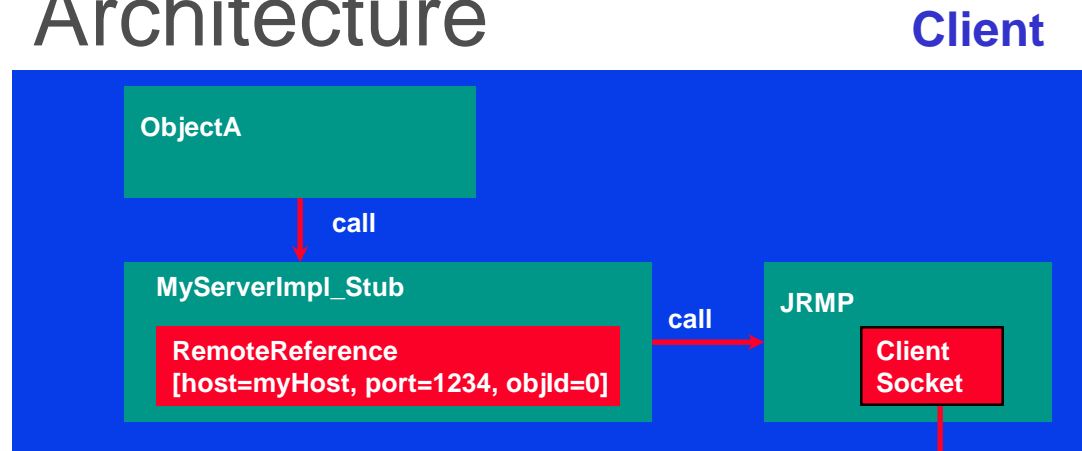
RMI Under the hood
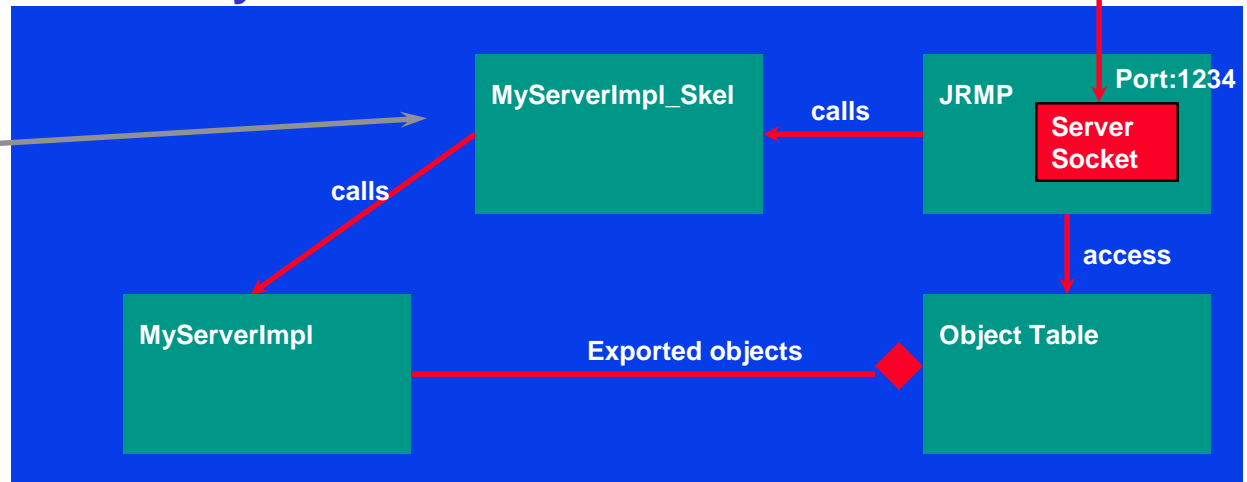## Architecture - Generic

What happens when A calls method on B

RMI Under the hood
Architecture - JRMP high level Architecture

Client

JRMP engine
Stub
Marshalling
Threading/Network C.M
Naming

ObjectA

call

MyServerImpl_Stub

RemoteReference
[host=myHost, port=1234, objId=0]

call

JRMP

Client Socket

Server: myHost

In Java 1.2 and higher skeletons are substituted by Reflection

MyServerImpl_Skel

calls

JRMP

Port:1234

Server Socket

calls

access

MyServerImpl

Exported objects

Object Table

RMI Under the hood
Architecture - JRMP high level Architecture

Client

JRMP engine
Stub
Marshalling
Threading/Network C.M
Naming

ObjectA

call

MyServerImpl_Stub

RemoteReference
[host=myHost, port=1234, objId=0]

call

JRMP

Client Socket

Server: myHost

In Java 1.2 and higher skeletons are substituted by Reflection

MyServerImpl_Skel

calls

JRMP

Port:1234

Server Socket

calls

access

MyServerImpl

Exported objects

Object Table
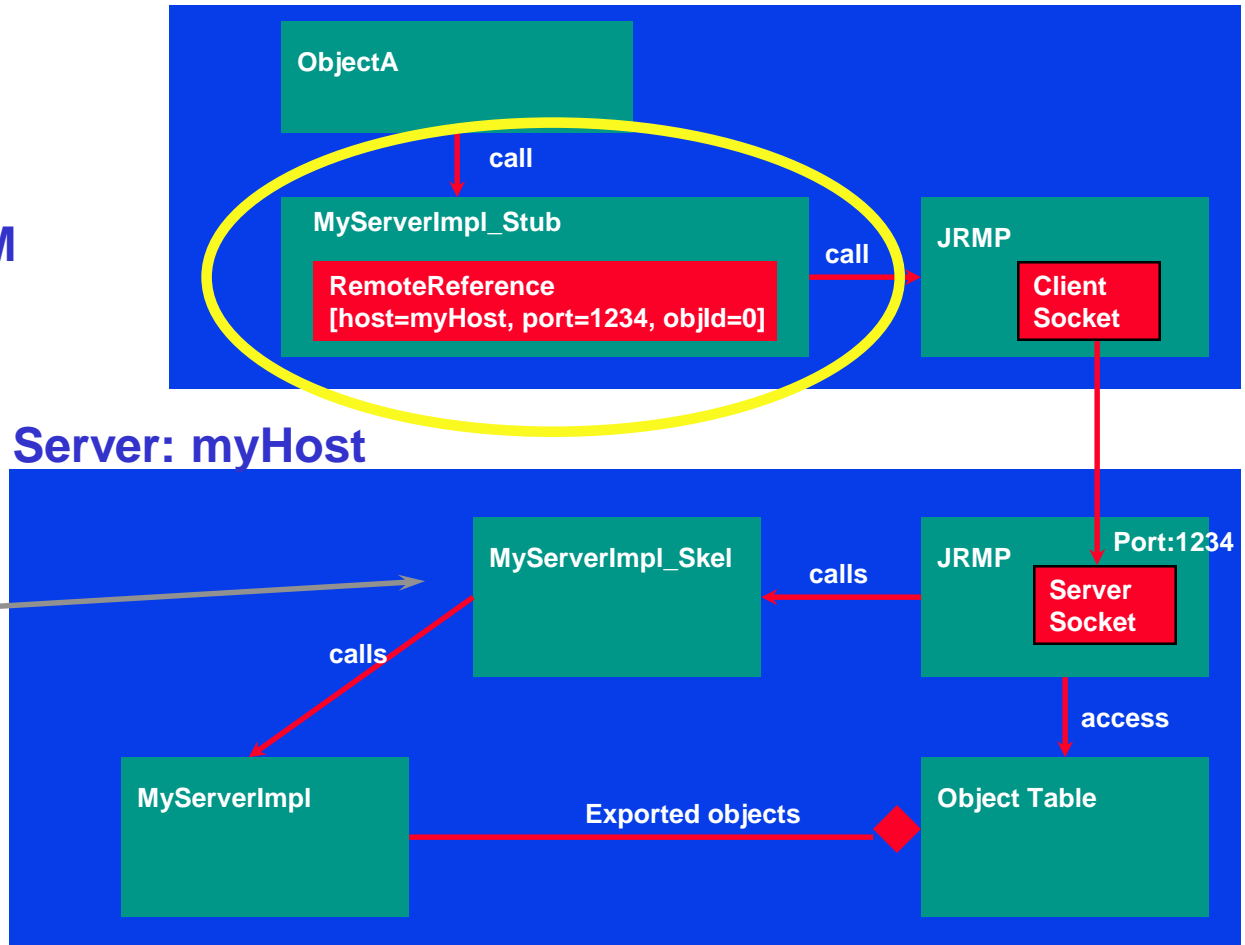
RMI Under the hood
Architecture - Stubs

- Stubs are proxy objects that implements a given set of RMI interfaces, which also are implemented by a remote object
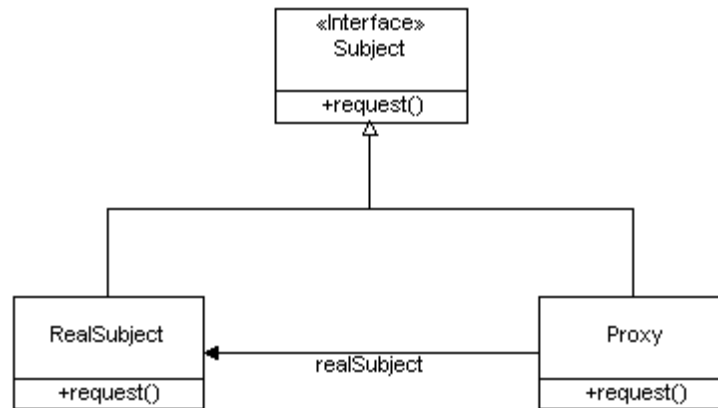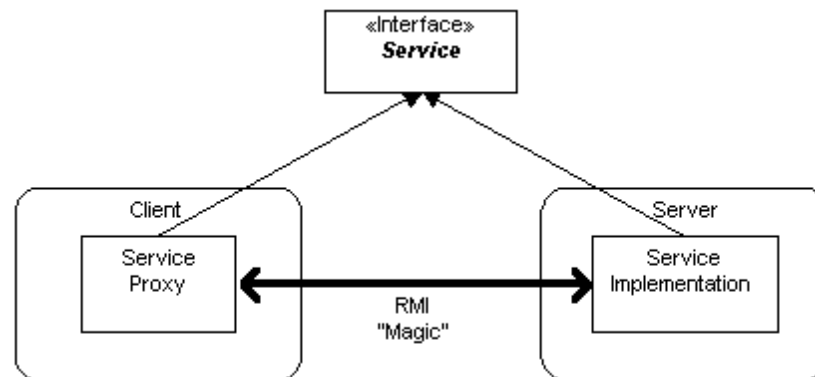

Typically proxies Implement one interface

RMI Under the hood
Architecture - Stubs

- Stubs give clients the "illusion" that the remote objects are in the same process space thus abstracting al the network programming away from the developer (providing location transparency)

RMI Under the hood
Architecture - Stubs

- Stubs contain the remote reference for the remote java object in a form determined by *java.rmi.server.RemoteRef*

  - Host

    - Host name for the RMI server in which the remote object lives

  - Port

    - Port number for which the remote object is exported

  - *java.rmi.server.ObjID*

    - Unique id with repect to a host, used to identify remote object on a host

MyServerImpl_Stub

RemoteReference
[host=myHost, port=1234, objId=0]

RMI Under the hood
Architecture - Stubs

- Stubs are generated automatically from the Java Remote Interface by the '*rmic*' compiler

  - it introspects the class for the remote object to find the methods which should be remotely accessible

  - it generates a class which is a subclass of *java.rmi.server.RemoteStub* and which implements the remote interfaces of the remote java object

    - as it explicit implements the remote interface, you can just cast the RMI stubs to the remote interfaces

      obj = (Hello)Naming.lookup("HelloServer");

RMI Under the hood
Architecture - Stubs

1.2 stub

```
package examples.hello;
public final class HelloImpl_Stub
    extends java.rmi.server.RemoteStub
    implements examples.hello.Hello, java.rmi.Remote
{
    private static final long serialVersionUID = 2;
    private static java.lang.reflect.Method $method_sayHello_0;

    static {
      try {
            $method_sayHello_0 =
examples.hello.Hello.class.getMethod(
                    "sayHello", new java.lang.Class[] {});
      } catch (java.lang.NoSuchMethodException e) {
            throw new java.lang.NoSuchMethodError(
                    "stub class initialization failed");
      }
    }
```

Java Reflection

RMI Under the hood
Architecture - Stubs

1.2 stub

```
// constructors
public HelloImpl_Stub(java.rmi.server.RemoteRef ref) { super(ref); }

// implementation of sayHello()
public java.lang.String sayHello() throws java.rmi.RemoteException {
  try {
    Object $result = ref.invoke(this, $method_sayHello_0,
                 null, 6043973830760146143L);
    return ((java.lang.String) $result);
  } catch (java.lang.RuntimeException e) {
    throw e;
  } catch (java.rmi.RemoteException e) {
    throw e;
  } catch (java.lang.Exception e) {
    throw new java.rmi.UnexpectedException(
           "undeclared checked exception", e);
  }
 }
}
```

Simply delegates the call to the underlying RMI implementation (JRMP runtime)

The JRMP runtime then do marshalling and sends the data to the server
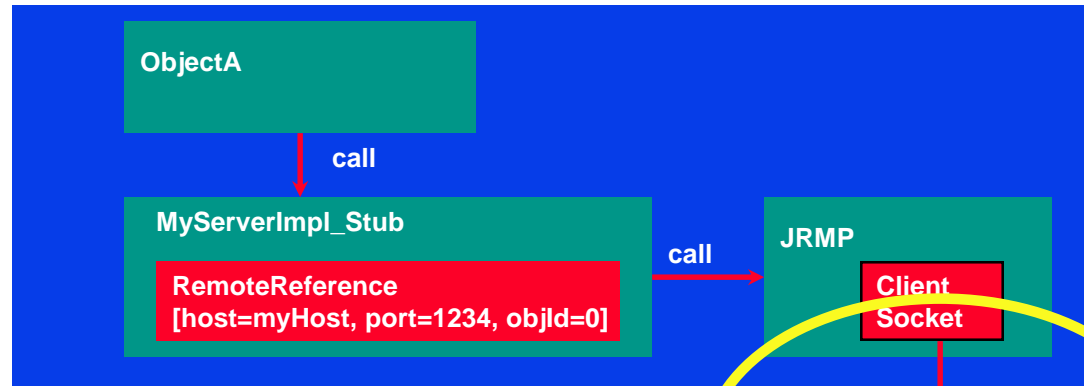
RMI Under the hood
Architecture - Stubs

- There will be one instance of a _Stub class per remote reference in the client code

  – remember the stub contains the RemoteRef which is the handle to the remote object that resides in some RMI server on some host somewhere in the network

RMI Under the hood
Architecture - Marshalling

Client

JRMP engine
Stub
Marshalling
Threading/Network C.M
Naming

ObjectA

↓ call

MyServerImpl_Stub

RemoteReference
[host=myHost, port=1234, objId=0]

→ call

JRMP

Client
Socket

Server: myHost

In Java 1.2 and higher skeletons are substituted by Reflection →

MyServerImpl_Skel

calls ←

JRMP   Port:1234

Server
Socket

calls ↓

MyServerImpl

Exported objects ──◆──

↓ access

Object Table
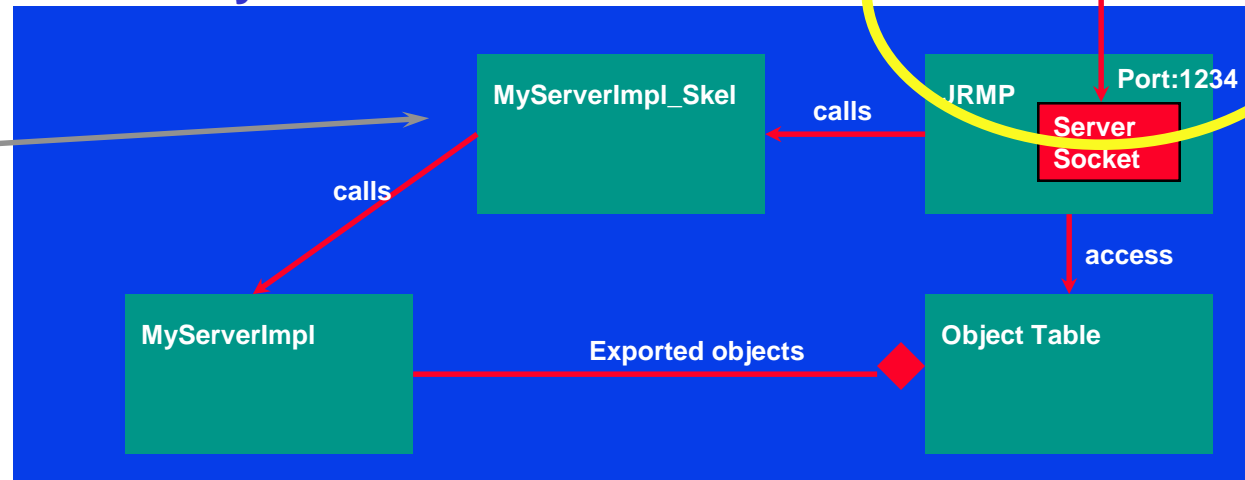
RMI Under the hood
Architecture - Marshalling

- Marshalling is the process that transforms objects into byte streams which can be sent over the network

- Unmarshalling is the reverse process that transforms byte streams into in-memory java objects

- Marshalling is build upon Java Serialization - a specification for how to transform java objects into byte streams and the reverse process

RMI Under the hood
Architecture - Marshalling

| Object | → | Bytes | → | Object |

- Java Serialization doesn't just operate on separate java objects - Java Serialization can take a whole object graph and serialize that into bytes and on the other deserialize back into the java object graph

RMI Under the hood
Architecture - Marshalling

- To make an object serializable, the class has to implement the *java.io.Serializable* interface

  – The actual implementation of the serialization/-deserialization process is in the methods *java.io.ObjectOutputStream* and *java.io.ObjectInputStream*

  – By implementing

    private void writeObject(ObjectOutputStream stream) throws IOException

    You can control the serialization process yourself (reverse: *readObject(….)*)

    It also speed performance up!!!

RMI Under the hood
Architecture - Marshalling

– No *readObject(..)/writeObject(..)* given the default implementation is used

- Introspection of the objects

  - *transient fields* -> nothing

  - *static fields* -> nothing

  - *primitive field* -> serialize the value

  - *another object* -> serialize the object recursively

  - *array* -> serialize all the objects in the array

RMI Under the hood
Architecture - Marshalling

- Marshalling of remote objects are given special treatment in Java RMI

  - if method parameter or the result value of a RMI method invocation is an object that is

    - remote object (implements *java.rmi.Remote*) and

    - exported by the RMI implementation

    RMI replaces automatically the remote java object reference with the stub instance

- JRMP uses a subclass of *ObjectOutputStream* called *sun.rmi.server.MarshalOutputStream*

RMI Under the hood
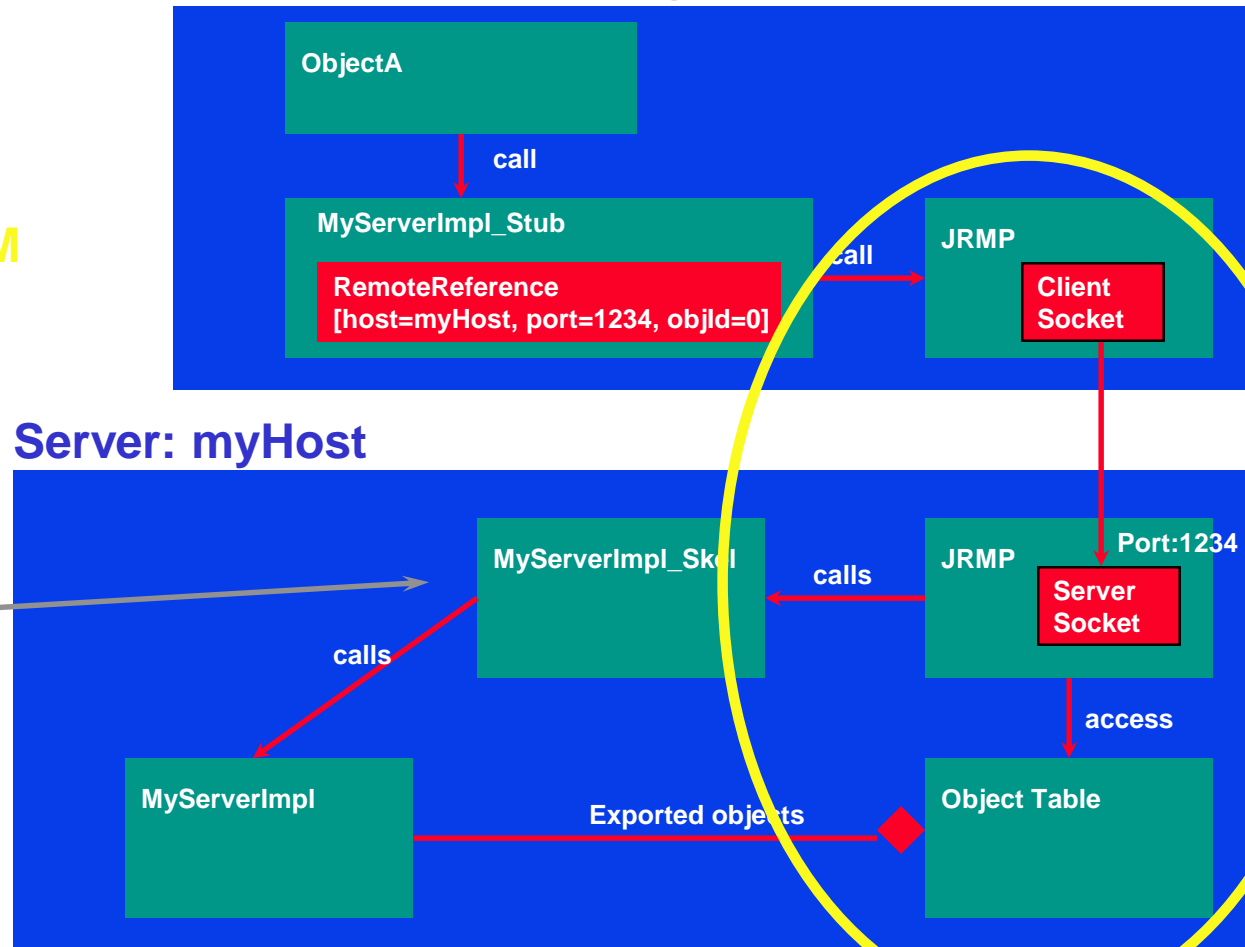## Architecture - Threading/Network connection management

JRMP engine
Stub
Marshalling
Threading/Network C.M
Naming

Client

ObjectA

| call

MyServerImpl_Stub

RemoteReference
[host=myHost, port=1234, objId=0]

call

JRMP

Client Socket

Server: myHost

In Java 1.2 and higher skeletons are substituted by Reflection

MyServerImpl_Skel

calls

JRMP

Port:1234

Server Socket

calls

MyServerImpl

Exported objects

access

Object Table

RMI Under the hood
Architecture - Threading/Network connection management

- Network connection management and threading are essentiel building blocks to achieve a good performance remote object server

- The RMI specification is not very restrictive on these areas

  - there is not guarantee that a remote method invocation always happens in a separate thread

    - Client calls remote object -> separate thread

    - Remote object calls another remote object in the same server -> no guarantee for a separate thread

    (Can skip the marshalling and network layer)

RMI Under the hood
Architecture - Threading/Network connection management

– Remote object can be executed concurrently -> need to be threadsafe

Clients

Server

Remote reference

Remote Object

.
.
.
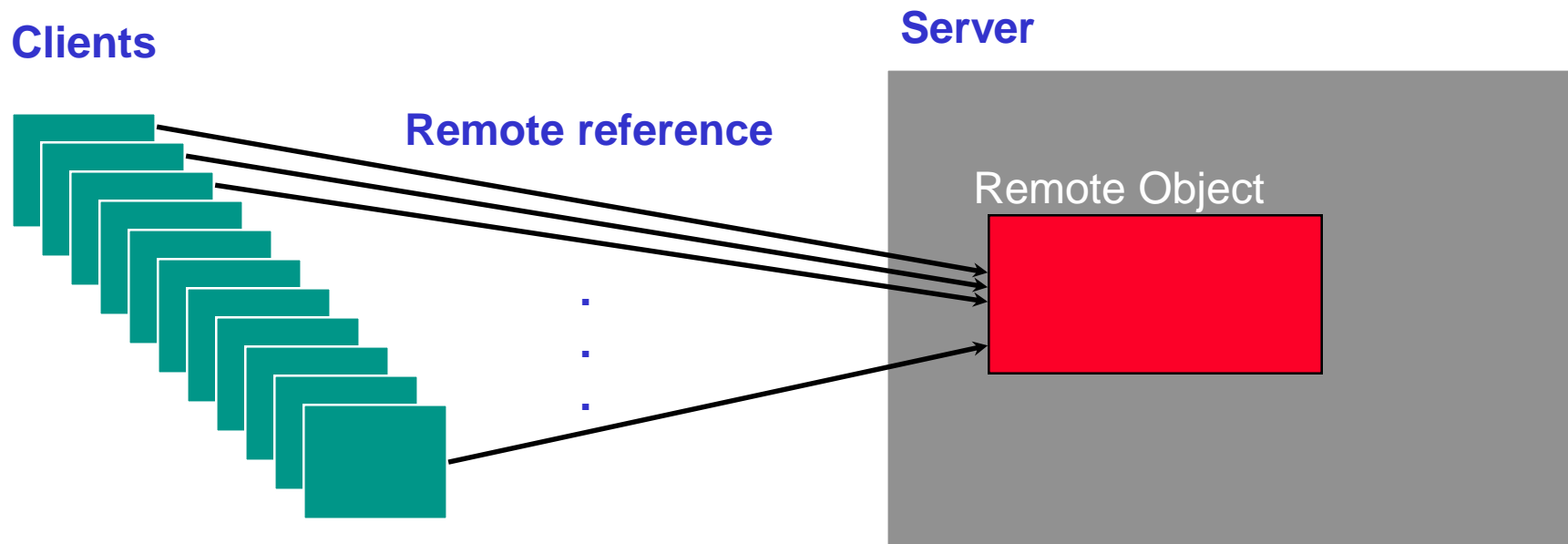
RMI Under the hood
Architecture - Threading/Network connection management

- JRMP engine uses socket factories for aquiring client and server sockets at runtime

  - *java.rmi.server.RMIClientSocketFactory*

  - *java.rmi.server.RMIServerSocketFactory*

- You as a developer has the opportunity to supply custom socket factories when the remote java object is exported

  - we will briefly look at this later

RMI Under the hood
Architecture - Threading/Network connection management

- When the remote java object is exported, it can be done in two ways

  – port specific (*exportObject(Remote obj, int portno)*)

  – anonymously (*exportObject(Remote obj)*)

- For each port on the server - there exists an Object Table/Map holding references to the remote java objects (key is the ObjID)

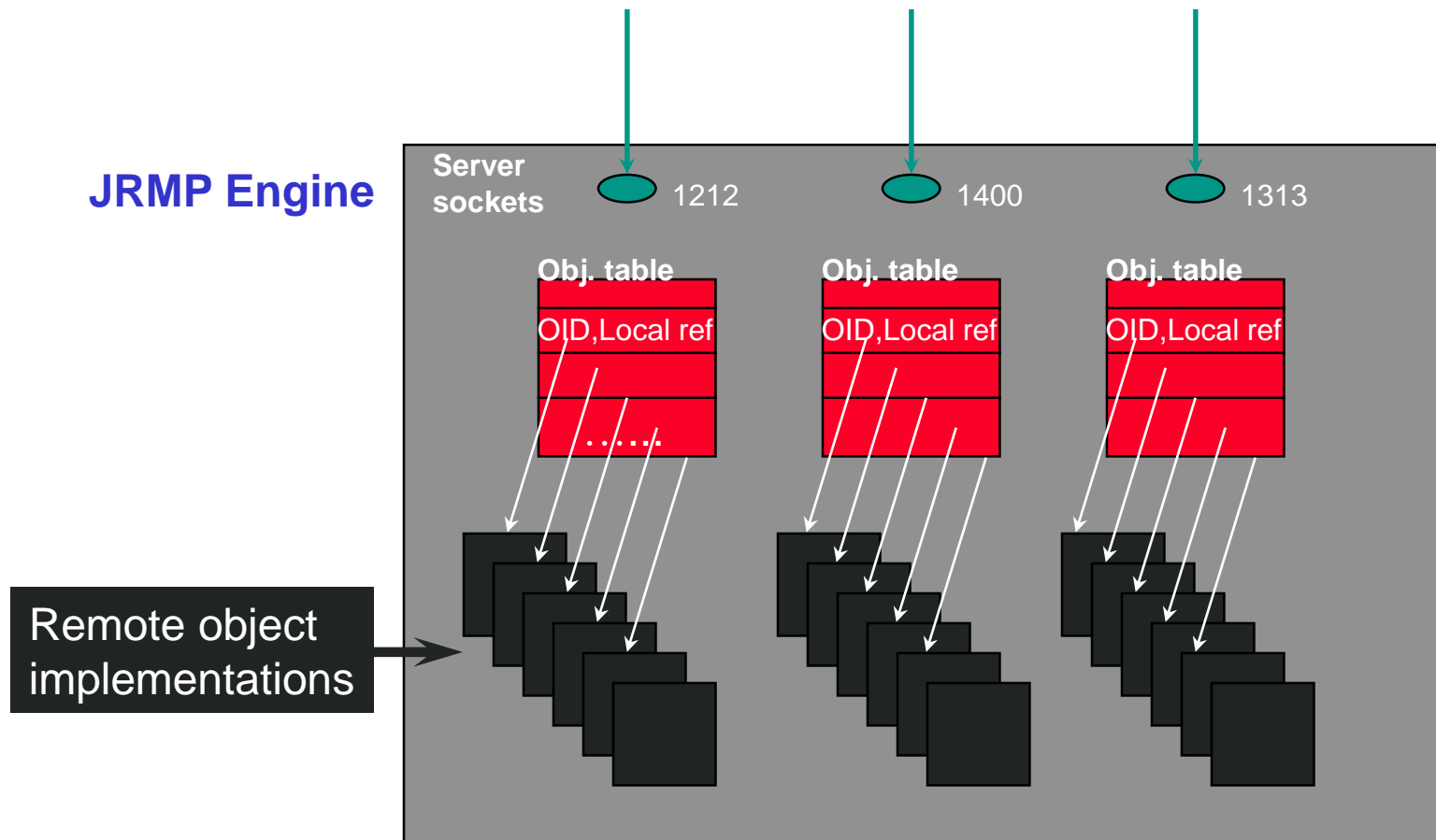- More java remote objects can be exported to the same port

RMI Under the hood
Architecture - Threading/Network connection management

- **What happens when exporting a remote object**

  - The remote object gets an ID

  - The remote object ref is putted into the Object Table (associated to port given in the export statement) with key equal ID and value equal the local object ref

  - A server socket is created by the server SocketFactory which starts listens the actual port for incomming requests

    - The server socket serves all the remote objects exported on that port

  - A associated skeleton is created (used in marshalling)

RMI Under the hood
Architecture - Threading/Network connection management

JRMP Engine

Server sockets 1212   1400   1313

Obj. table   **Obj. table**   **Obj. table**

OID,Local ref   OID,Local ref   OID,Local ref

. . . . .

Remote object implementations

RMI Under the hood
Architecture - Threading/Network connection management
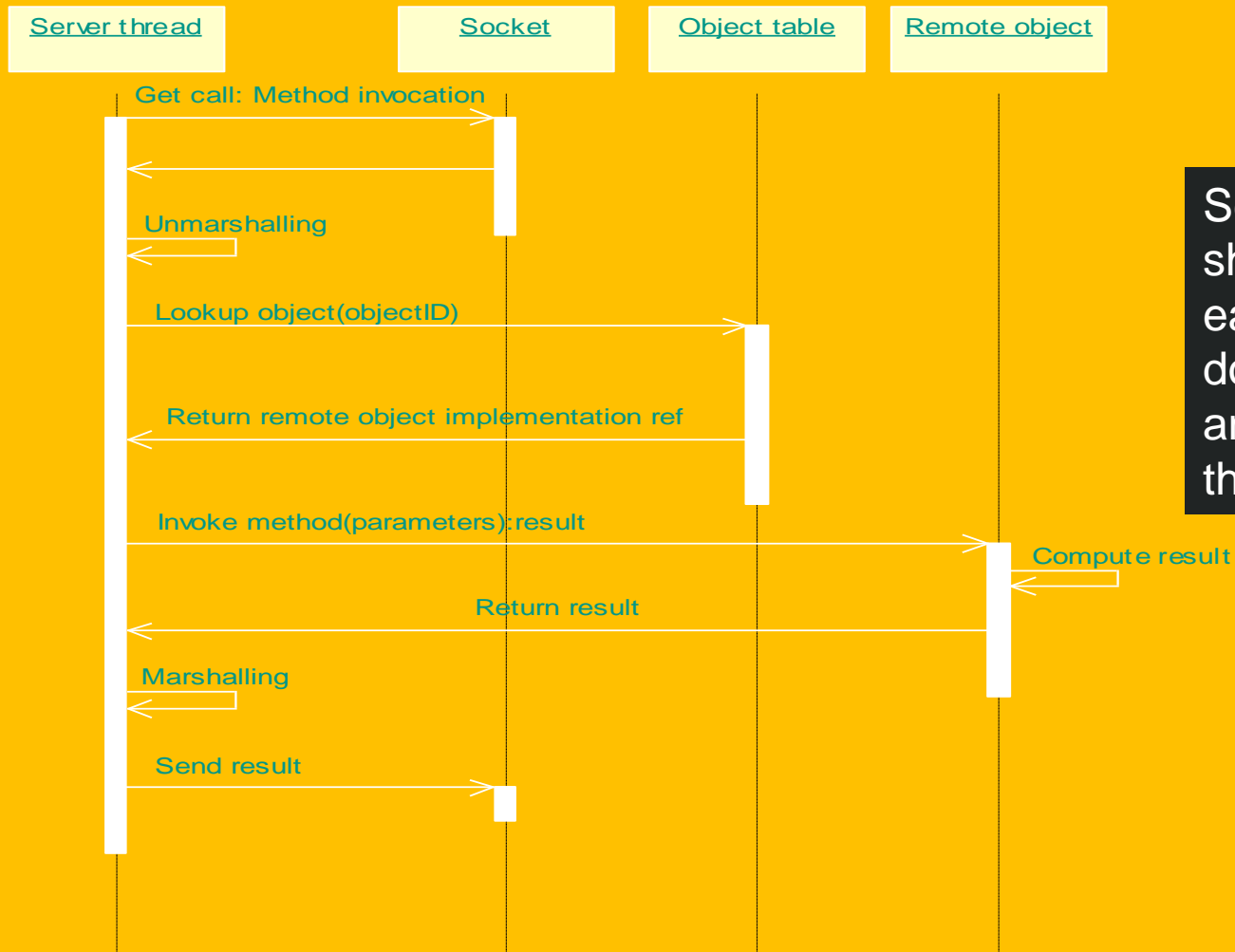
- When a connection from a client is made the following happens:
  - A new thread is spawned to service that connection (is done by handing over the socket to the thread)
  - The thread starts checking for incomming calls sent over the socket connection
  - Once a call is received
    1. The thread looks up the remote object implementations in the Object Table by usage of the OID supplied in the remote method invocation

RMI Under the hood
Architecture - Threading/Network connection management

2 The thread then unmarshals the parameters and the call method to be invoked

3 The thread then invokes the call method with the supplied parameters on the remote object implementation

4 The remote object implementation does its work and returns

5 The thread marshals the result and sends it back to the calling client using the socket connection

6 The thread start listen again for incomming calls on the socket connection

Architecture - Threading/Network connection management

Server thread	Socket	Object table	Remote object

Get call: Method invocation

Unmarshalling

Lookup object(objectID)

Return remote object implementation ref

Invoke method(parameters):result

Compute result

Return result

Marshalling

Send result

Sequence diagram showing the work each server thread does when receiving an incomming call on the socket connection
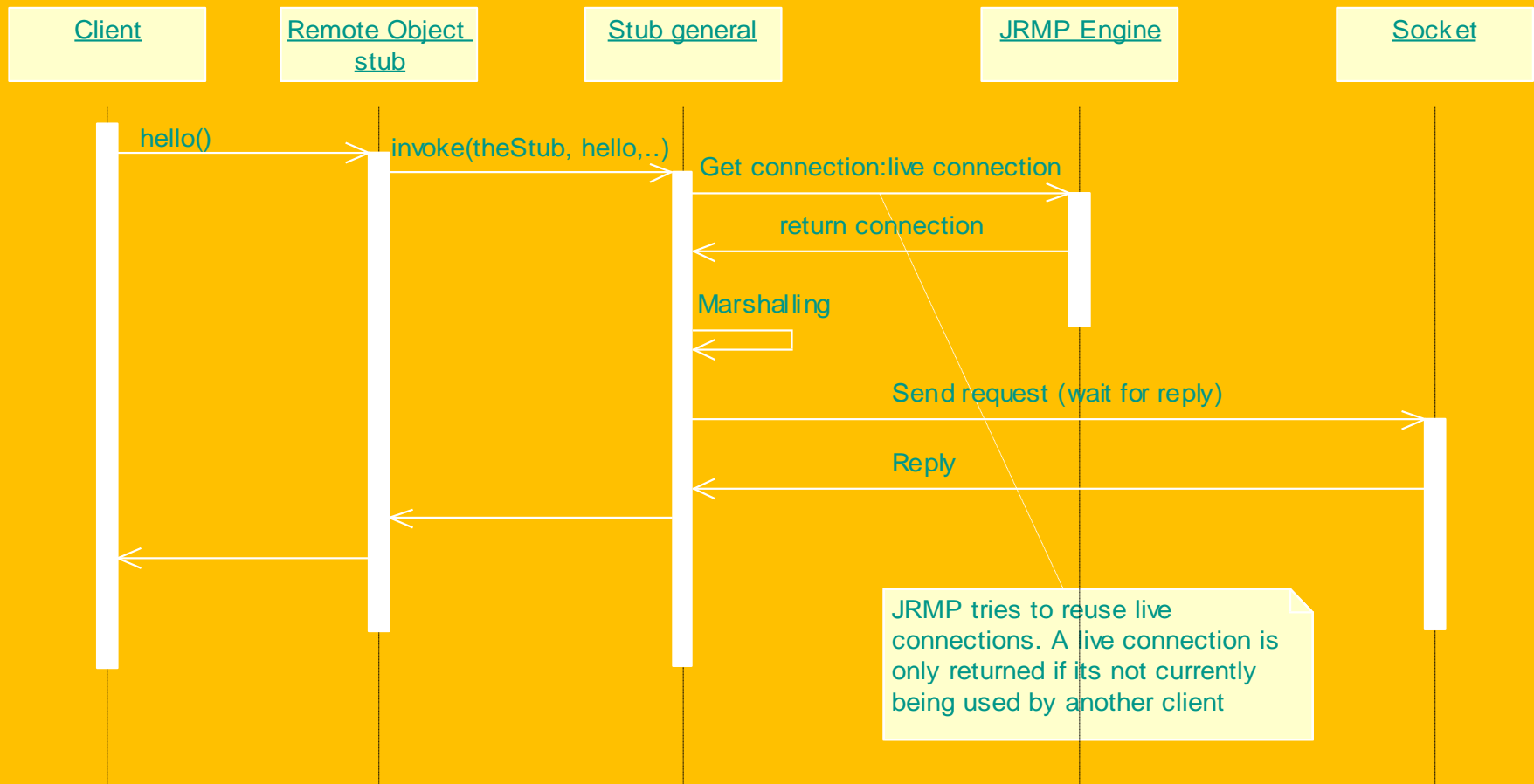
RMI Under the hood
Architecture - Threading/Network connection management

- On the client side the call is made using the thread which makes the call

  – no reason for spawning a new thread

- The RMI specification does not specify how to handle connections, so this is up to the implementation

- JRMP uses a relative simple connection model

  – each client connection must only process one call at a time

  – if two calls are made concurrently, JRMP automatically supply two connections to serve the calls

RMI Under the hood
Architecture - Threading/Network connection management

| Client | Remote Object stub | Stub general | JRMP Engine | Socket |

hello()

invoke(theStub, hello,..)

Get connection:live connection

return connection

Marshalling

Send request (wait for reply)

Reply

> JRMP tries to reuse live connections. A live connection is only returned if its not currently being used by another client

RMI Under the hood
Architecture - Threading/Network connection management

- NOTE: In Java Threads and Sockets are closely related to each other; it was only possible to get data from a socket by using a thread that is dedicated to a paticular socket

    - The reason for this was Java's lacking support for non-blocking IO (introduced in J2SE1.4)

Thats why there is spawned a thread per socket connection in JRMP

RMI Under the hood
Architecture - Naming

- We have now seen how Java RMI works under the hood; what the stub does, what the JRMP engine does and how request flows are processed both client side and server side

- But how do we get remote references to remote java objects living in a RMI server?

- There are more ways

RMI Under the hood
Architecture - Naming

- First: you could have the RMI server serializing the remote reference(stub) to a file which clients could read and deserialize into the stub

Server

Client

S T U B

J R M P

RMI Server process

3

2

1

Disc
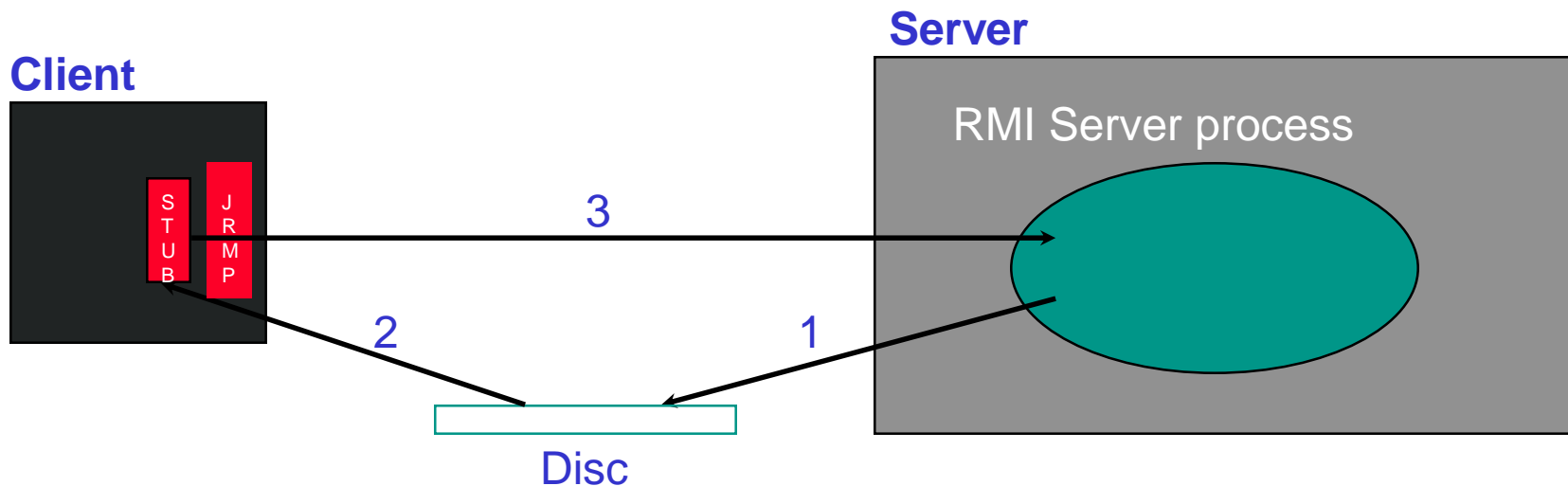
RMI Under the hood
Architecture - Naming

- Second: You could use the Java RMI supplied Naming Service *Rmiregistry* as we used in our previous example

RMI Server process

3. invoke

2. lookup

1. bind

RMIRegistry process

S T U B

J R M P
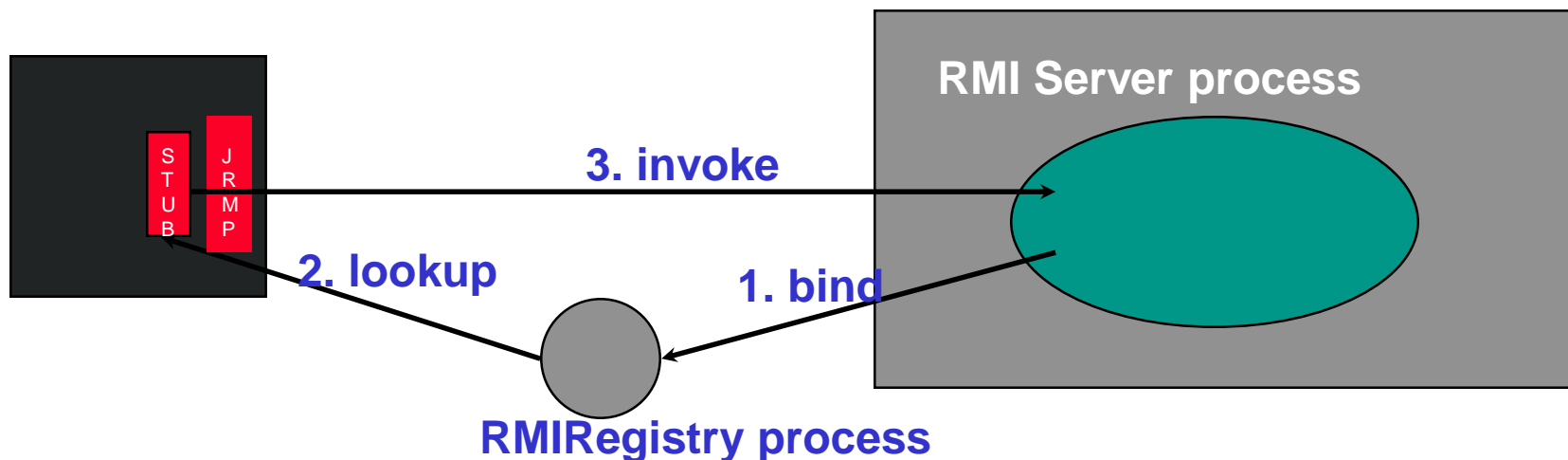
RMI Under the hood
Architecture - Naming

- RMIRegistry is an transient database providing a hierarchical naming service for RMIServers and clients

  - lookup services

  - binding services

- The *Naming* class provides methods to bind/lookup based on URLs

  Naming.bind("myRemoteObject", obj);

  Naming.lookup("Localhost/myRemoteObject")

RMI Under the hood
Architecture - Naming

- There typically exists an 'registry' on each node housing an RMI Server

- To get an remote object reference to a remote 'registry' you can use

  LocateRegistry.getRegistry(String Host, int Port)

- The RMIRegistry itself is a remote object so when invoked (bind/rebind), remote objects are replaced by their stubs

  – so RMIRegistry is a transient database of stubs!!!!

RMI Under the hood
Architecture - Naming

- The RMIRegistry has to be started prior to the RMIServer and the client

- The RMIRegistry is a single point of failure

  - No faul-tolerance service in the current implementation

- If RMIRegistry crashes bindings are lost

  - And the RMI Server has no simple way of rebinding the remote objects

RMI Under the hood
Architecture - Naming

- You could use JNDI to make your clients more consistent/flexible in its lookup of remote references to various things instead of using

  - the Naming class

  - LocateRegistry.getRegistry(..)/Registry

- JNDI is a general Java Naming and Directory Interface which can be used to access a wide range of naming services

- By using JNDI you don't have to change code to change naming service implementation

RMI Under the hood
Architecture - Naming

```java
public class HelloServerJNDI
{
    public static void main(String args[]) {
        // Create and install a security manager
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            HelloImpl obj = new HelloImpl();
            // Bind this object instance to the name "HelloServer"
            Context ctxt = new InitialContext();
            ctxt.rebind("HelloServer", obj);
            System.out.println("HelloServer bound in JNDI Space");
        } catch (Exception e) {
            System.out.println("HelloImpl err: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

The new server code

RMI Under the hood
Architecture - Naming

```
public class HelloClientJNDI {

  public static void main(String[] args) {
    // "obj" is the identifier that we'll use to refer
    // to the remote object that implements the "Hello" interface
    Hello obj = null;
    try {
      Context ctxt = new InitialContext();
      obj = (Hello) ctxt.lookup("HelloServer");
      System.out.println(obj.sayHello());
    } catch (Exception e) {
      System.out.println("HelloServer exception: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

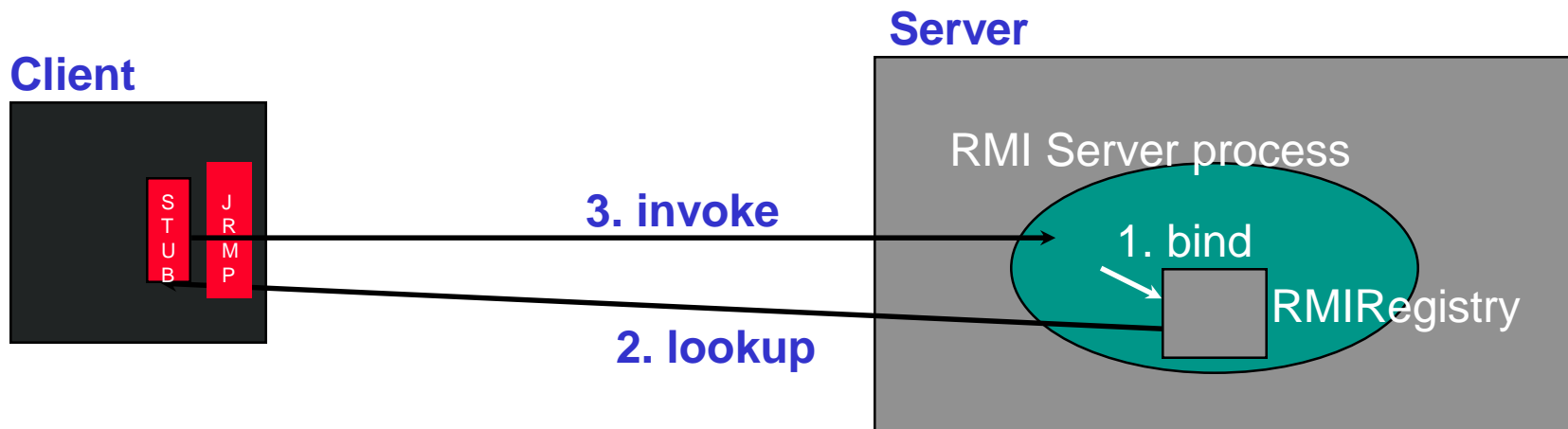The new Client code

Example:
Startjndiserver
startjndiclient

jndi.properties file

```
java.naming.factory.initial=com.sun.jndi.rmi.registry.RegistryContextFactory
java.naming.provider.url=rmi://localhost
```

RMI Under the hood
Architecture - Naming

- Third: You could avoid having the RMIRegistry running in its own process and thus illiminating the single point of error by using the

  LocateRegistry.createRegistry(…)

Server

Client

S
T
U
B

J
R
M
P

3. invoke

RMI Server process

1. bind

RMIRegistry

2. lookup

RMI Under the hood
Architecture - Naming

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

      // Create registry if not allready running
      try {LocateRegistry.createRegistry(1099);}
      catch (RemoteException e) {e.getMessage();return;}
      // Create and install a security manager
      if (System.getSecurityManager() == null)
         System.setSecurityManager(new RMISecurityManager());
      try {
         HelloImpl obj = new HelloImpl();
         // Bind this object instance to the name "HelloServer"
         Context ctxt = new InitialContext();
         ctxt.rebind("HelloServer", obj);
         System.out.println("HelloServer bound in JNDI Space");
      } catch (Exception e) {
         System.out.println("HelloImpl err: " + e.getMessage());
         e.printStackTrace();
      }
   }
}
```

New Server code

Remember to grant permission to open the socket

Example
Startjndiinternalserver
startjndiclient

Dynamic Class Loading
Motivation

- In JDK 1.3, RMI requires that clients use a stub class to connect to a remote VM

- Today, as we have seen, you generate stub classes with the *rmic* command line tool

- However new versions of RMI allow clients to build stubs dynamically at runtime using dynamic proxies

- Until that feature is added, clients need a way to guarantee that stubs are available

  - Installing the stubs on the client? Not the best!

Dynamic Class Loading
Motivation

- Java RMI allows for dynamic class loading which solves the problem of distributing the stubs

Dynamic Class Loading
How to set it up

- Dynamic class loading is enabled by speficing a codebase for the RMIServer

  -Djava.rmi.server.codebase=file:/myhome/server/

- When you specify a codebase, the server annotates all outbound objects with URL location information

  – This happens during marshalling

- Clients can use this location information to download classes when necessary
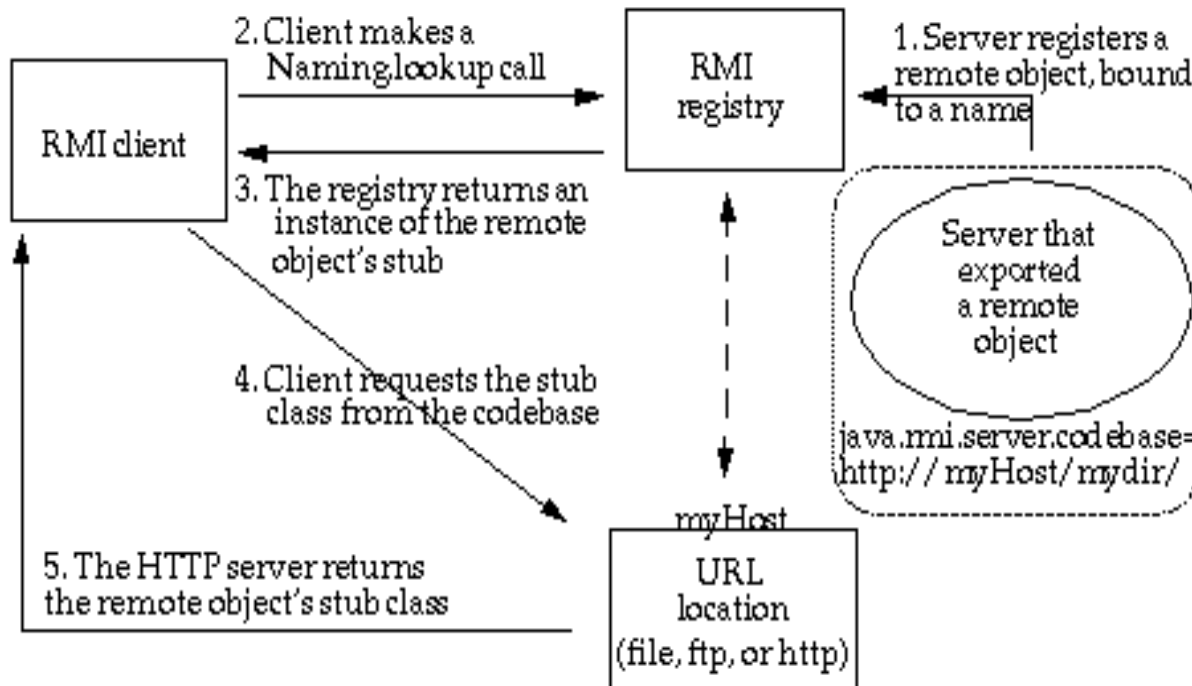
  – Not only stubs but also dependent objects
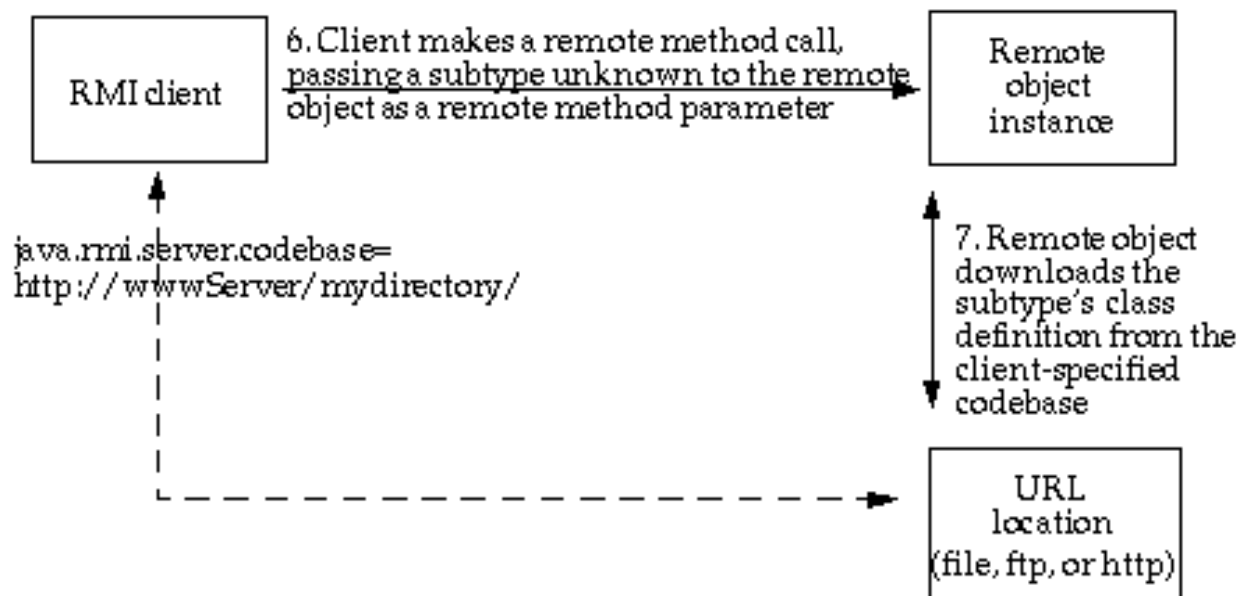
Dynamic Class Loading
How to set it up

Client

Server

Remote Interface

Remote object stub

Foo

Foo

Fetched from the annotated code base information

Remote object stub

Dependent objects

Foo

RMIRegistry

Remote object impl

Foo

In order to create the stub the client needs the stub class
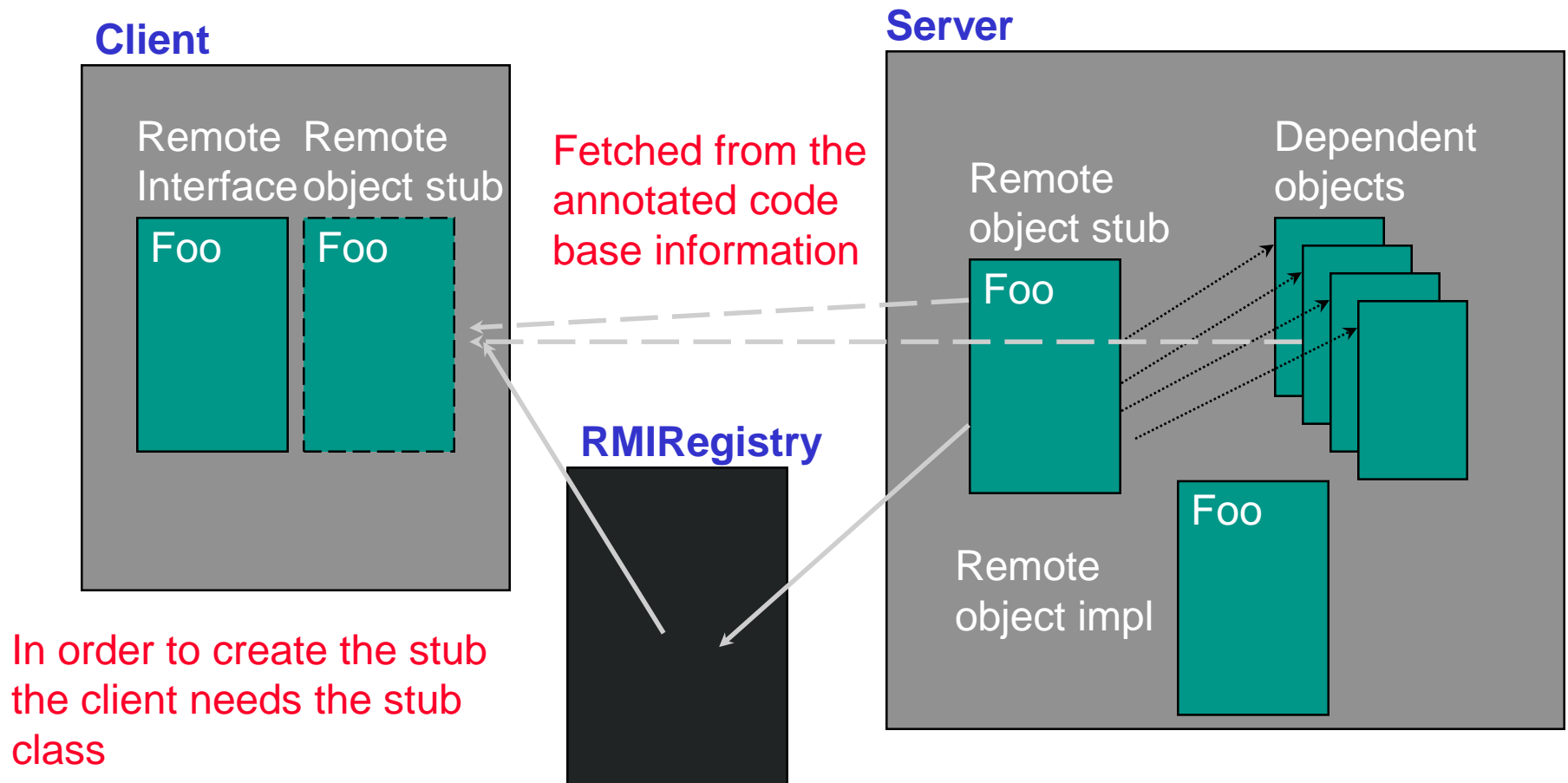
Dynamic Class Loading
Steps in dynamic loading of stubs

Dynamic Class Loading
Steps in dynamic loading of subtypes

Subtypes can be 1. implementation of an Interface
2. subtype of an class

Dynamic Class Loading
How to set it up

- Example

  – Specify a codebase for the HelloServerJNDI

```
java -cp . -Djava.rmi.server.codebase=file:/software/jdevrtm/jdev/mywork/
    presentations/BasicRMIServer/classes/ examples.hello.HelloServerJNDI
```

  No errors anymore as rmiregistry is able to load the stub class automatically (console)

  – Run client

  – Sun has developed a mini web server for dynamic class loading based on HTTP

Start the Application …

- Start the server: in directory containing rmi

  rmi/hello must contain HelloImpl classes:

  Djava.rmi.server.codebase

  rmi.hello.HelloImpl

- Start the client in directory containing rmi

  rmi/hello must contain HelloClient classes:

  Djava.security.policy=policy rmi.hello.HelloClient

JVM1

HelloImpl

Codebase
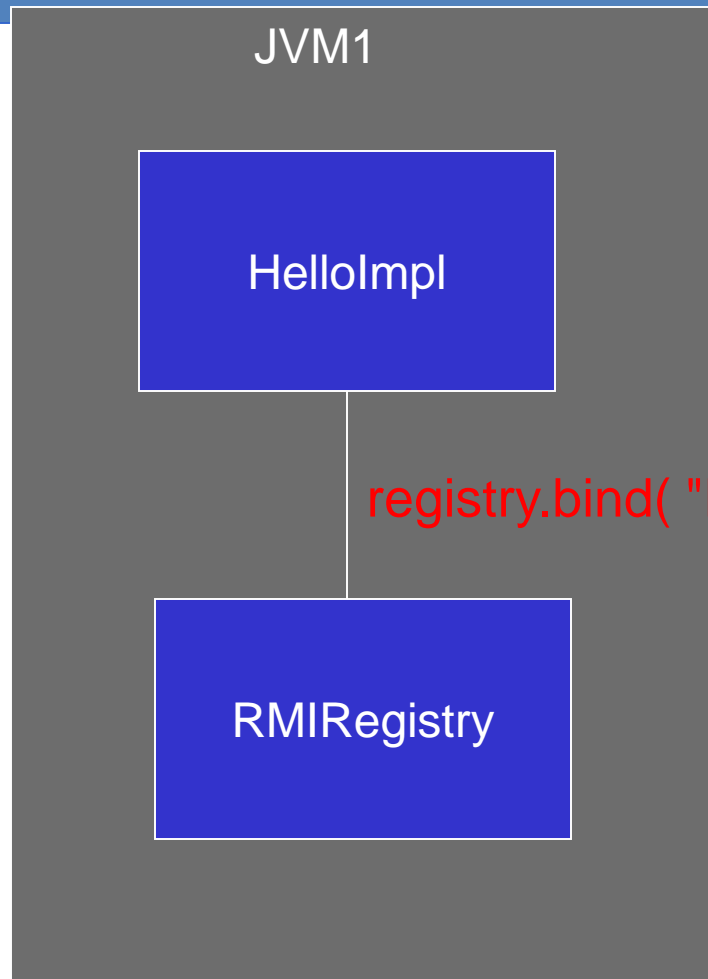
JVM1

HelloImpl

LocateRegistry.createRegistry

RMIRegistry

Codebase

JVM1

HelloImpl

registry.bind("Hello", service)

RMIRegistry
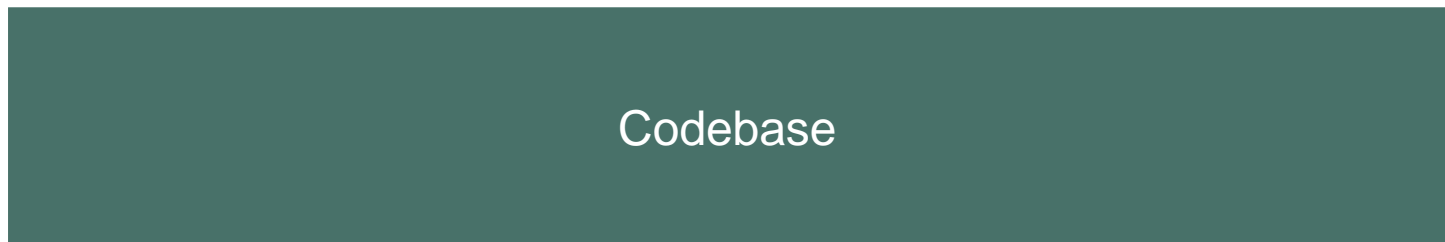
Codebase

JVM1

HelloImpl
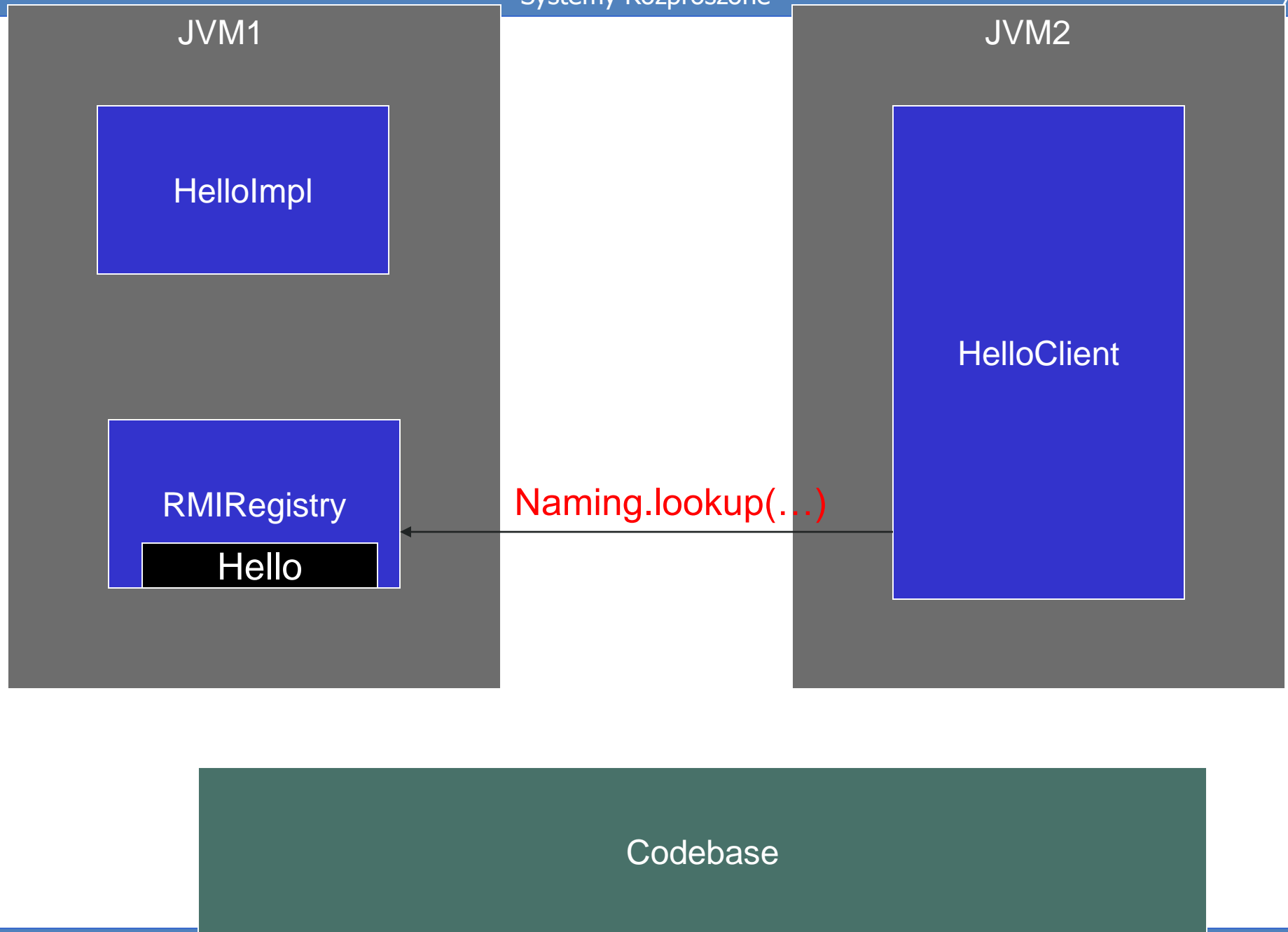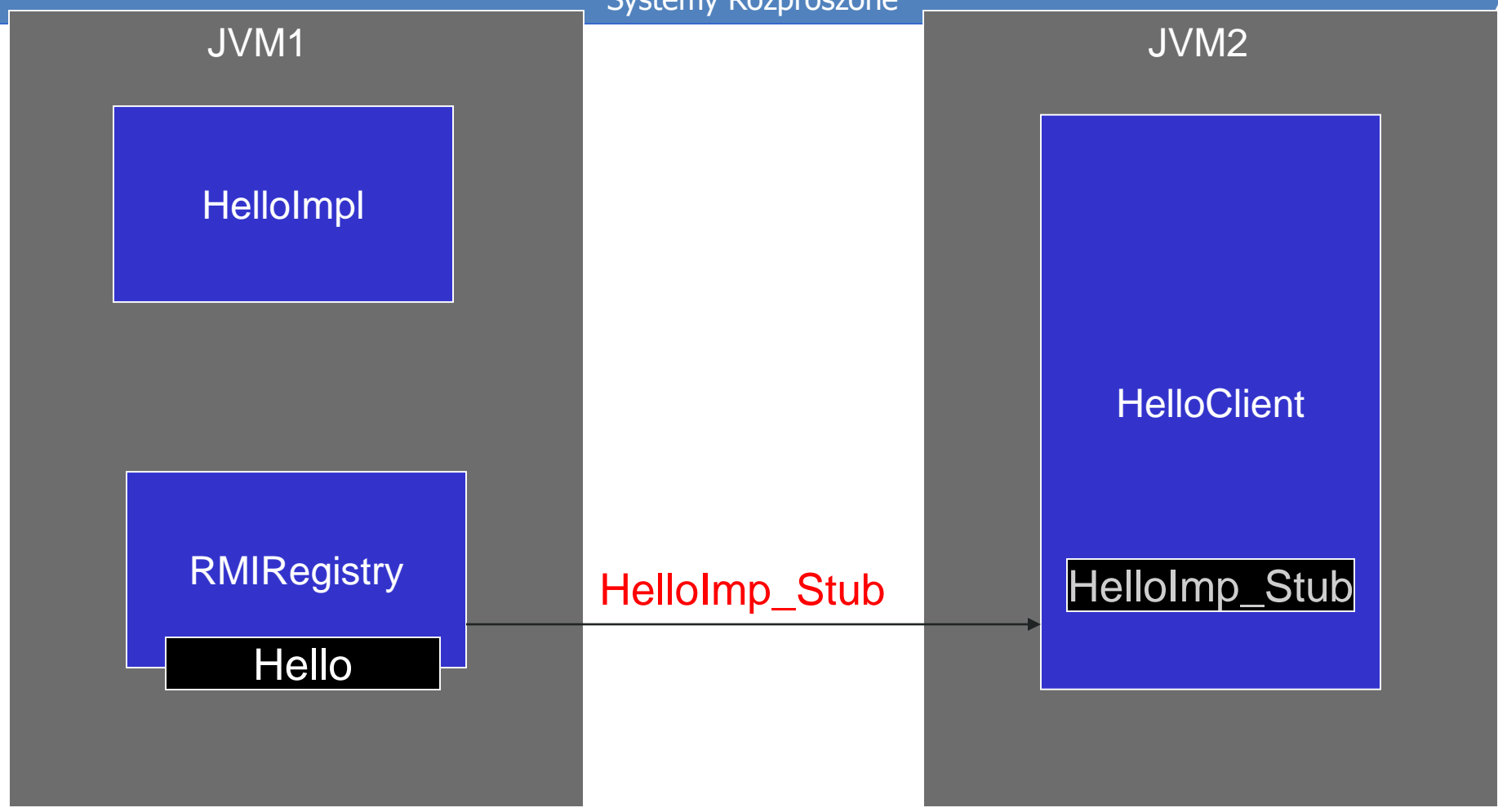
registry.bind("Hello", service)

RMIRegistry
Hello

Annotation points to codebase

Codebase

JVM1

JVM2

HelloImpl

HelloClient

RMIRegistry

Hello

Naming.lookup(…)

Codebase

JVM1

HelloImpl

RMIRegistry

Hello

HelloImp_Stub

JVM2

HelloClient

HelloImp_Stub

Codebase

JVM1

HelloImpl

RMIRegistry

Hello

JVM2

HelloClient

HelloImp_Stub

Classloader gets stub

Codebase

JVM1

HelloImpl

RMIRegistry

Hello

JVM2

HelloClient

service.sayHello

HelloImp_Stub

Codebase

JVM1

HelloImpl

RMIRegistry

Hello
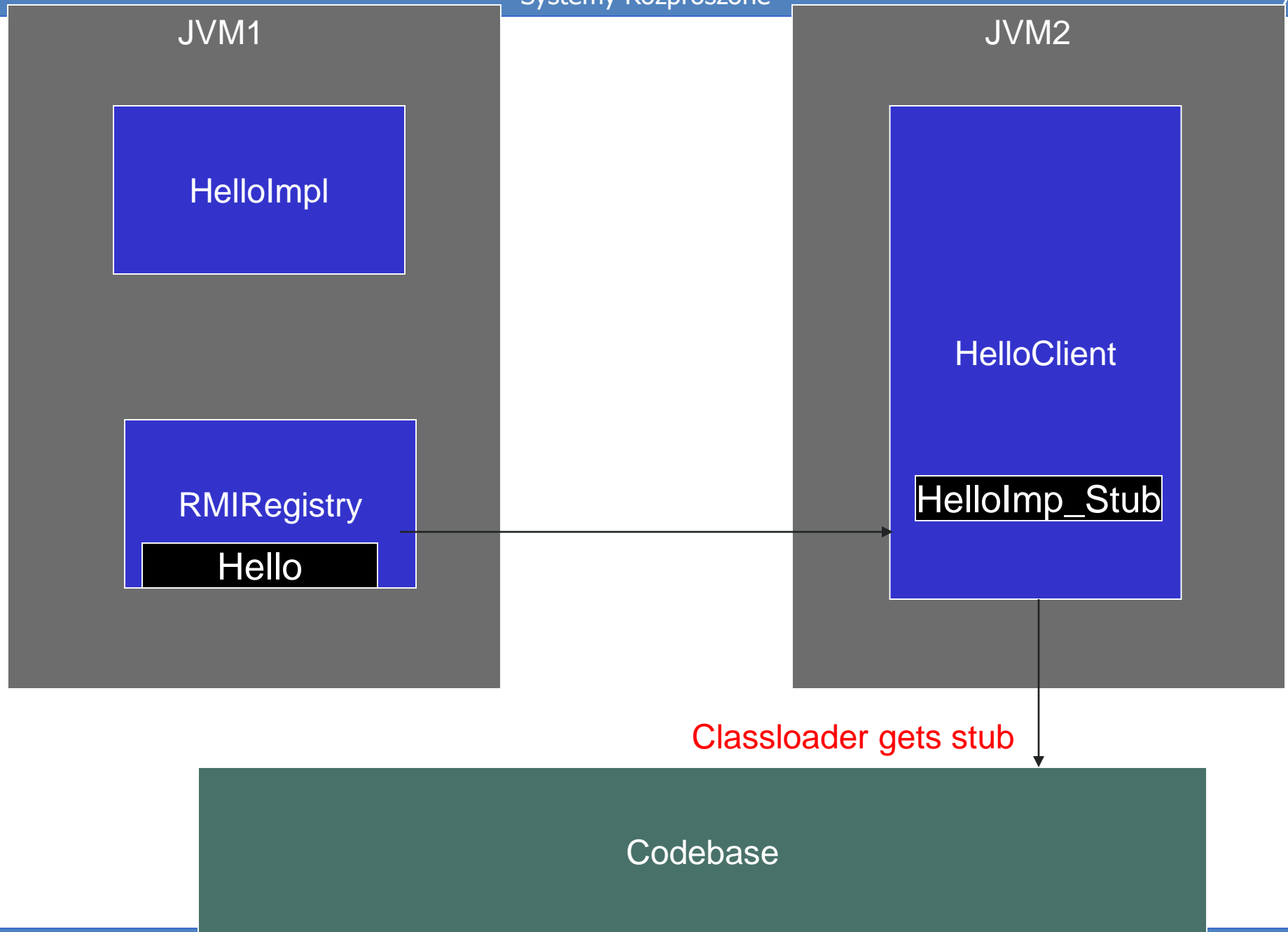
JVM2

HelloClient

service.sayHello

returns "Hello World!"   HelloImp_Stub
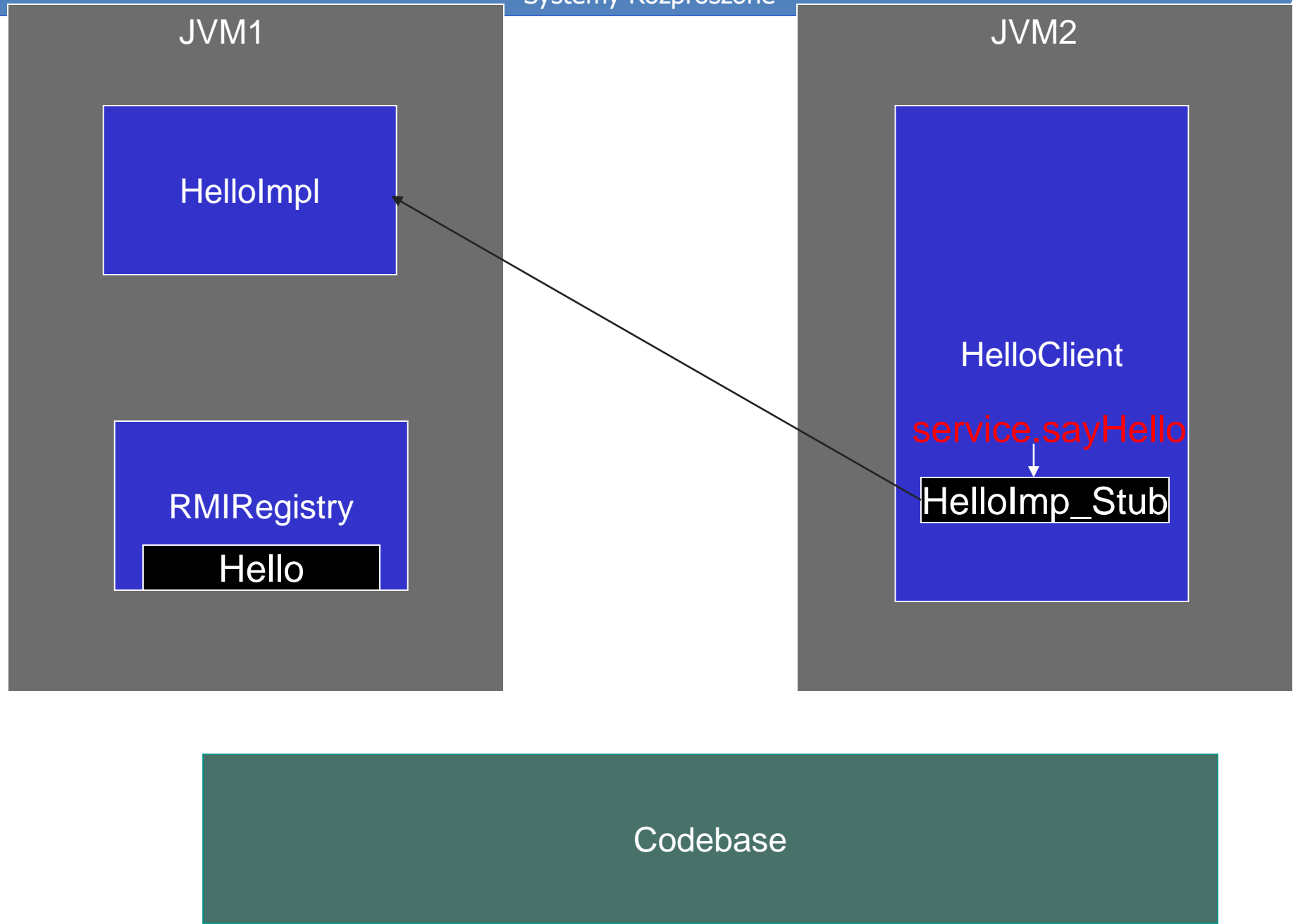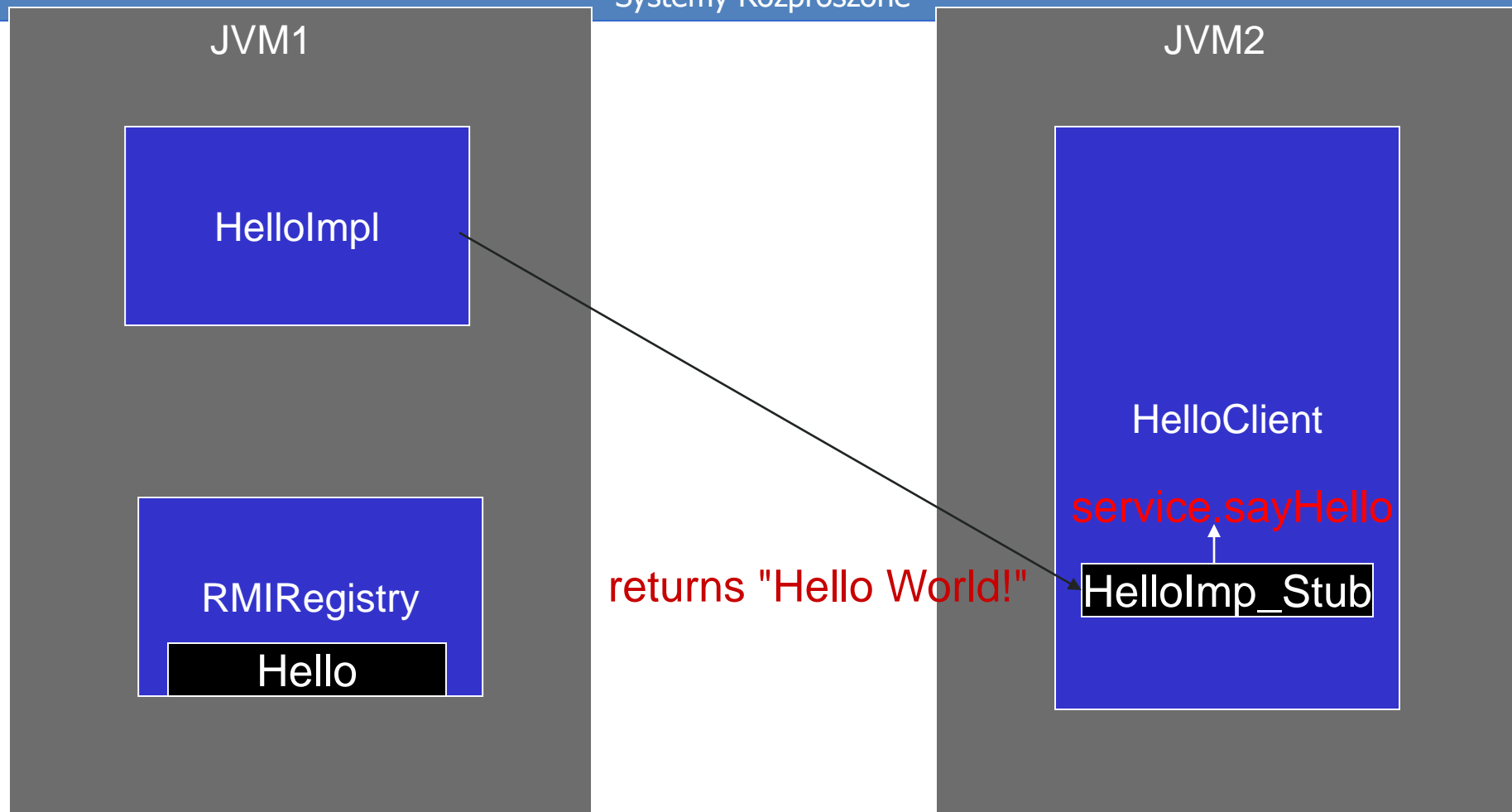
Codebase

Security
Basic

- In a distributed environment there is a strong need for security

    - Both client and server side

- When making RMI clients and RMI servers you have the basic Java 2 security framework to fit your needs

    - Basic Java 2 security based on protection domains, stack inspection and policy files

    - JAAS/JSSE/JCE etc. Need a security manager installed

Security
RMI issues

- When using dynamic class loading, RMI clients are forced to install a Security Manager

    – Either coded: *System.setSecurityManager(..)*

    – Or as property: *-Djava.security.manager*

- The RMI server must also install one if dynamic class loading the other should be allowed otherwise not

Distributed Garbage Collection
Basic

- Java RMI supports Distributed Garbage Collection as Java RMI has the goal to look like "normal" Java programming

  - build up on the JDK garbage collector with distributed enhancements

- RMI uses a reference-counting garbage collection algorithm

- To accomplish reference-counting garbage collection, the RMI runtime keeps track of all live references within each JVM

Distributed Garbage Collection
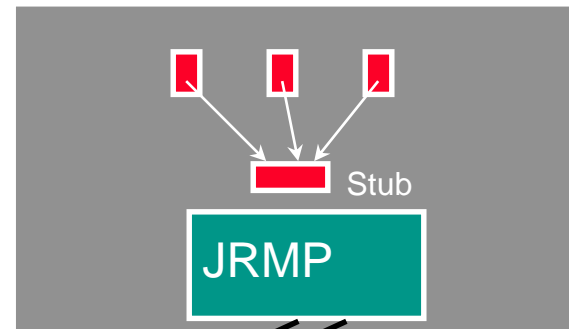## Basic - DGC Algorithm

Client

Client
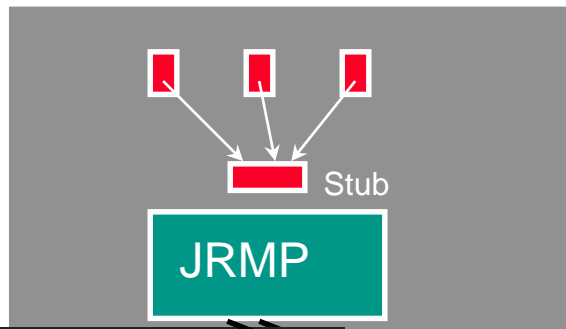
Stub

Stub

JRMP

JRMP

The JRMP keeps track of all local references to a stub

Server

1. First time a stub comes into a JVM a "referenced" message is send back to the server

JRMP

2. When the stub is not referenced locally anymore an "unreferenced" is send back to the server

3. When no more remote client references the remote object, the JRMP holds the remote object through a weak reference ➡ GC when no local refs

Remote Object

Distributed Garbage Collection
Basic - Leases

- The JavaRMI distributed garbage collection also uses *leases*

- Remote references are only leased for a given period of time and if the clients want to be sure that the remote object isn't garbage collected, they have to renew their leases at given intervals -> if not -> "Unreferenced"

  - If a client crash before a remote reference is "unreferenced", then if it was not for leases the remote object would never be claimed

Distributed Garbage Collection
Basic - RMIRegistry

- As the RMIRegistry is an remote object itself which stores bindings internally in a simple hashtable, the registry is a client of the remote object
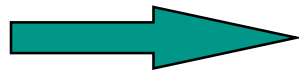
> The remote object is not claimed!

- If you however uses JNDI as the naming service and not upon RMIregistry, the naming service will not cause the DGC algorithm to be used and the remote object will be garbage collected unless you have a local reference for it

Distributed Garbage Collection
Basic - Properties

- You can control the lease renewals through system properties

  - *java.rmi.dgc.leaseValue*

- Also remote objects can be notified when there is no more remote references to it

  - This done by implementing the *java.rmi.Unreferenced* Interface with contains a single method *unreferenced()*

Activation
Motivation

- An exported remote object is alive from it is exported to the RMI server is shutdown/ crashes or the remote object is GC'ed

- For large scale systems this is unrealistic as remote objects take up system resources which always is in need

- Another issue is that even if you restart the server after shutdown or crash, the client stubs doesn't work any longer (new Port/ID)

  - the client has to obtain a new stub from the naming service after the remote object has been rebound
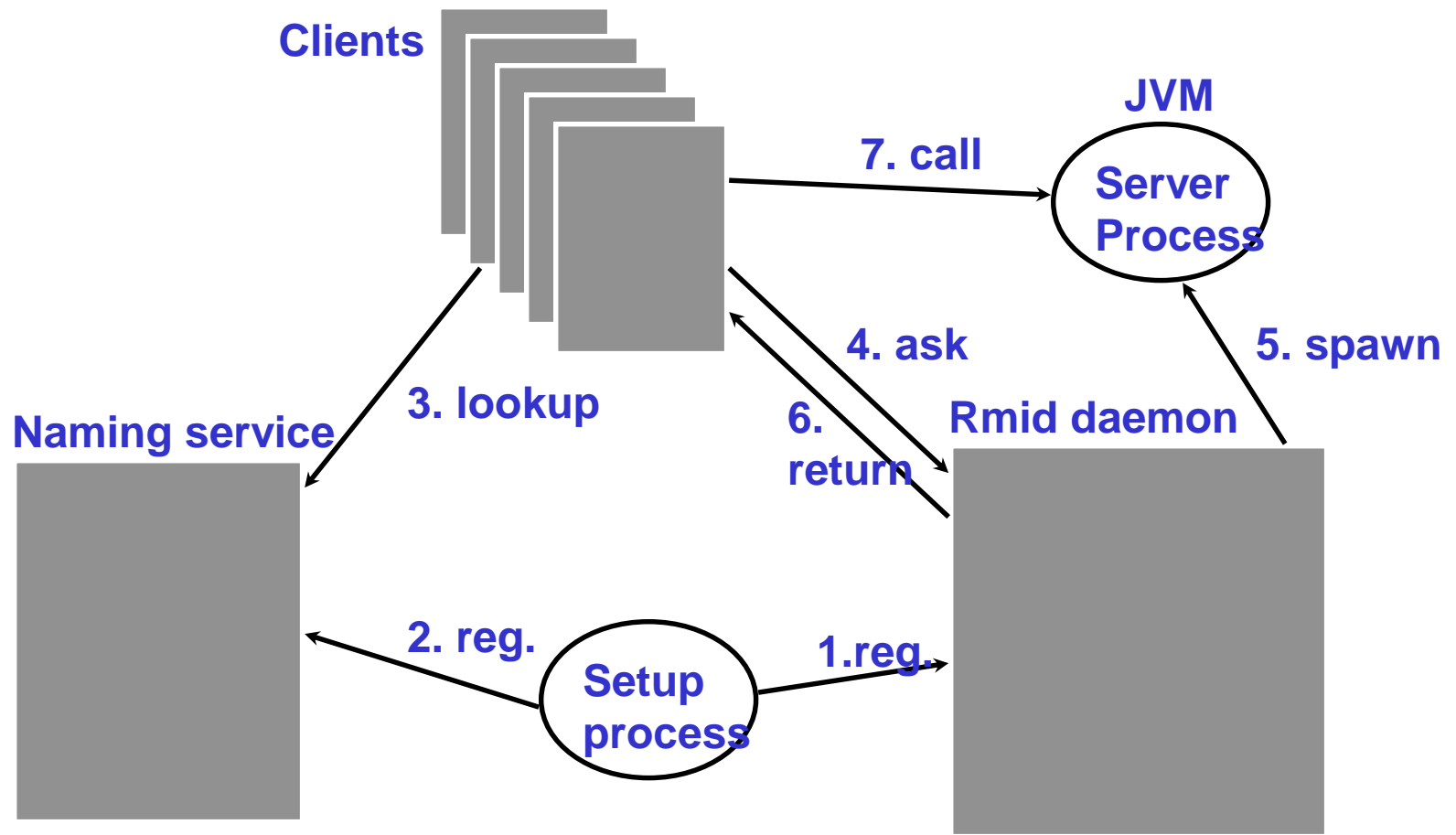
Bad code

Activation
Definition

- Therefore Java RMI provide a facility called *Activation* which provides

  1 .remote objects that can be activated on demand

  2. persistent remote references

- Two main concepts

  - Activation Groups

  - Activatables

Activation
Definition

- Activation Groups are essentially groups of Activatables
  - most one JVM per group in which all activatable remote objects for that group are created and exported
- An Activatable is a remote object that has been exported to RMI runtime through the activation interfaces
- So how does it work? How do we get persistent remote references/activatable objects

Activation
Architecture

Clients

JVM

7. call

Server Process

5. spawn

4. ask

3. lookup

6. return

Rmid daemon

Naming service

2. reg.

Setup process

1.reg.

Activation
Architecture description

RMID − sarted during reboot

RMID
-Manages ActivationGroups
-Manages Activatable Objects
-Create reference to Activatable Objects
-Start new Java VM per ActivationGroups

JVM

Starts

Starts

JVM

ActivationGroup

JVM
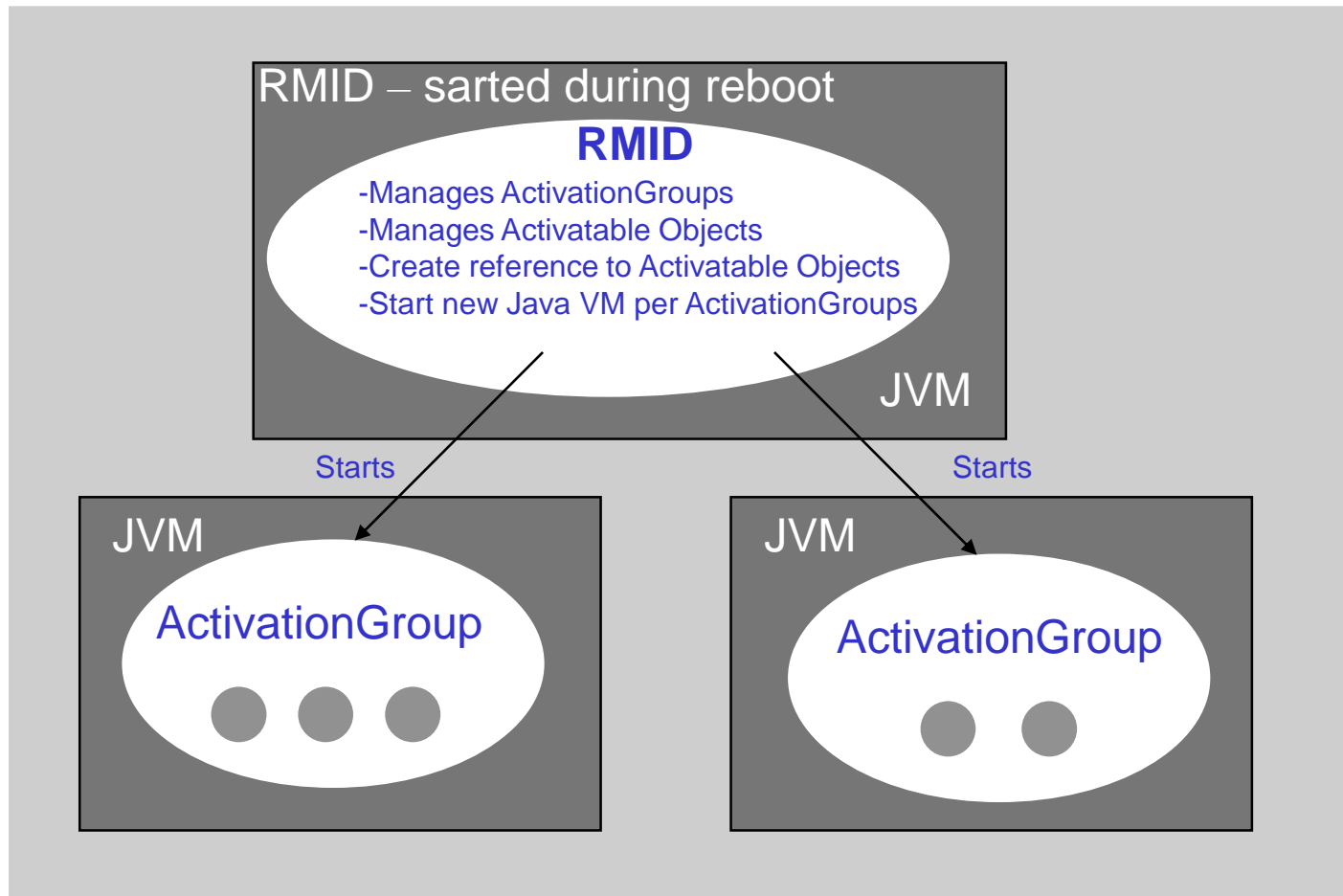
ActivationGroup

ActivationGroup can't be shard between JVM

Activation
Architecture description

1. Activatable objects are described and organized in Activation Groups and registered to Rmid daemon through java.rmi.activation interfaces

   When an activatable object is registered in Rmid daemon, an activatable stub is returned which can be used to register in the naming service

2. The activatable stubs are registered in the naming service

Activation
Architecture description

3. Client make lookups of remote object through the naming service and are giving the activatable stub in return

   – The client doesn't know its a special stub

Activatable stub

[ActivationID=1232323] Activation ID

[host=myHost, port=2233, objId=1] Rmid reference

[host=myHost, port=1234, objId=0] Live reference
empty at start

Activation
Architecture description

- When a client invoke a remote method on one of the activatable remote objects for the first time

4. The live reference is empty

   – Ask the Rmid daemon to provide one for the given ActivationID

   – The Rmid daemon looks in its tables to see whether the remote object is running

5. If no the Rmid daemon starts a new JVM in which the remote object is created (activates it) and returns its live reference

Activation
Architecture description

6. If yes the live reference is returned

7. The live reference is full

- – it just calls the remote method in the normal way

- – If the live reference fails it ask the Rmid daemon to get a new one

  → The Rmid daemon recognizes, that the server has crashed and spawns a new JVM for the activation group with the activatable in it

Development of RMI servers
How to get hold of remote objects

- If you are going to make an RMI implementation for a product catalog - how would you do that?

    - Register fx. each product in RMI registry with a unique name?

        NO!

- You could/would use the Factory pattern known from Gof where you will only register factories for different types of remote objects in the RMI registry

Development of RMI servers
How to get hold of remote objects
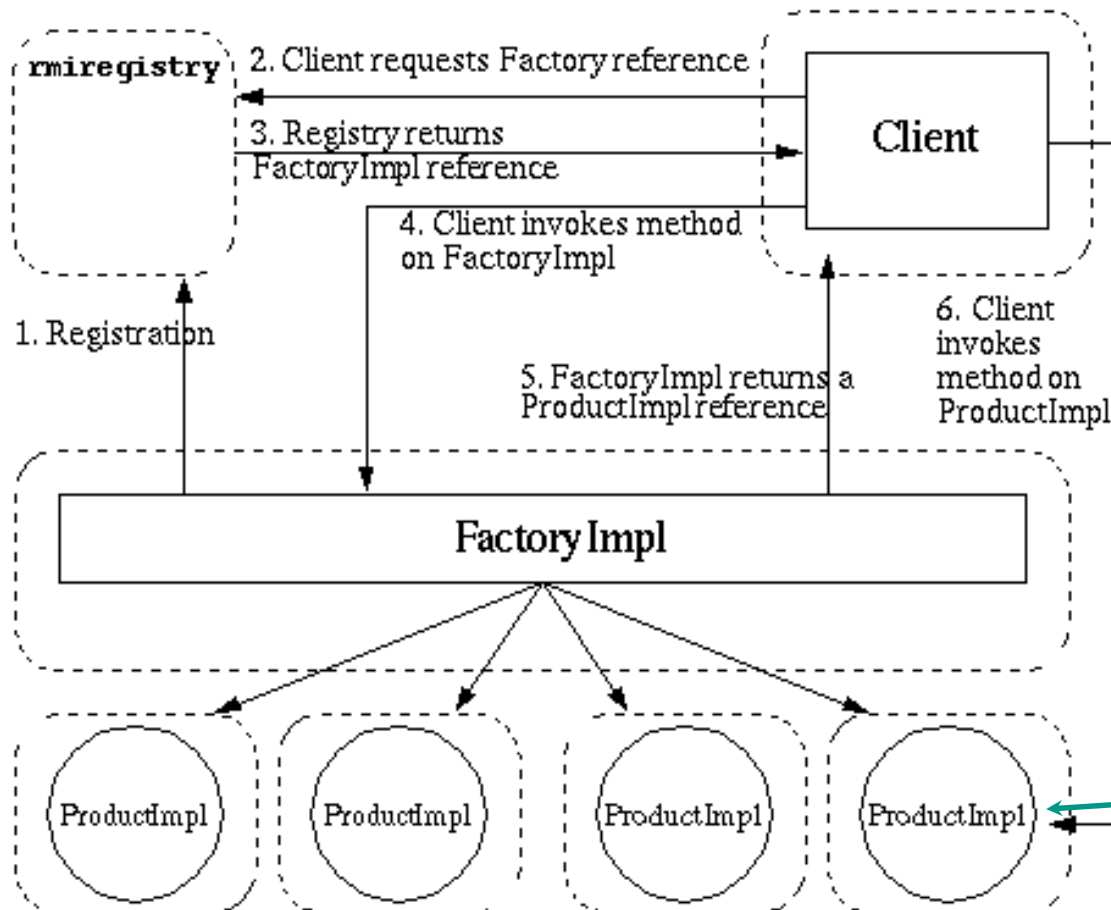
This is how the problem is solved in EJB

EJB provides us an abstraction of the problem in terms ofthe Home interface which essentially is our factory here

Exported remote objects not registered in the RMIRegistry but returned by the factory

Development of RMI servers
Making the distributed object model

- When making the implementation model for a distributed object system the following three aspects are very important if the system is going to be performant and flexible

    - System Partioning -> impact on performance and flexibility

    - Interface design -> impact on performance and reusability of the components of the system

    - Object granularity -> impact on performance and flexibility

Development of RMI servers
Making the distributed object model

- ## Life Cycle and Persistence

  – The mapning of the conceptual entity to the implementation object

  – What processes or process should the implementation live in?

  – Does the conceptual entity exist only when an implementation object is created?

  – Where and how is the entity state stored if it exists beyond the implementation object?

Development of RMI servers
Making the distributed object model

- Java RMI does not offer any services for Persistence management

    - You have to code your persistence of the object implementations either direct through usage of JDBC or by means of a mapping/persistence framework like Toplink

- Java RMI does offer some degree of Life Cycle managment

    - The object implementation is created upon export or first method invocation and destroyed upon server termination or GC

    - Nothing in between – which has impact on scalability

Development of RMI servers
Making the distributed object model

- Java RMI does not offer any support for transaction/concurrency management – you have to code it yourself
  - When more clients share the same remote object concurrency is an issue and has to be taken into account
    - One way of dealing with this is to make sure that each client gets its own copy and then delegate the transaction/locking management to the database
    - Another way but more complicated is to program a kind of locking mechanism at the object level
- Likewise Security and Deployment (replication, load-balancing etc.)

Development of RMI servers
Making the distributed object model

- Many of the patterns we are going to future later on applies to the development of RMI servers as well

  - ValueObjects

  - Session Facade

  - Page by Page Iterator

  - Network latency/concurrency

  - …...

RMI/IIOP

- This is a marriage of the CORBA world and the Java RMI world in that sense they can start to talk to each other

  - RMI remote objects can now be accessed by CORBA clients potentially not written in Java

  - And that means in fact also EJBs which just is a component model build upon Java RMI

- We will look more at that when we look at Java and CORBA