Bern University of Applied Sciences

•

Engineering and Information Technology

Department I

•

Development of Mobile Applications – RMI & Jini

Java RMI

J.-P. Dubois

2007 - 2008

Table of Contents

1 RMI Basics

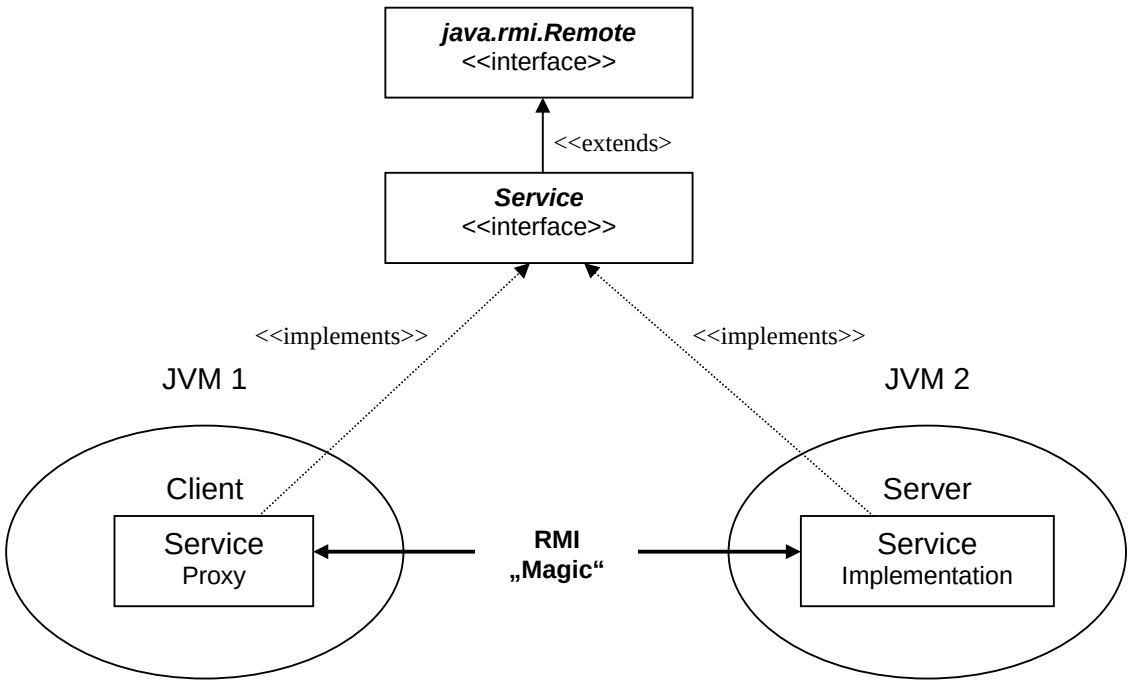
1.1 Goals

RMI applications are often comprised of two separate programs: a server and a client. A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them. Such an application is referred to as a *distributed object application*.

A primary goal for the RMI designers was to allow programmers to develop distributed Java programs with the same syntax and semantics used for non-distributed programs. To do this, they map how Java classes and objects work in a single JVM to a new model where Java classes and objects work in a distributed (i.e. multiple JVMs) computing environment. Although no perfect transparency exists between the two models, the RMI architects created a system, which hides most of the network communication implementation details and extends the safety and robustness of the Java architecture to the distributed computing world.

1.2 Interfaces

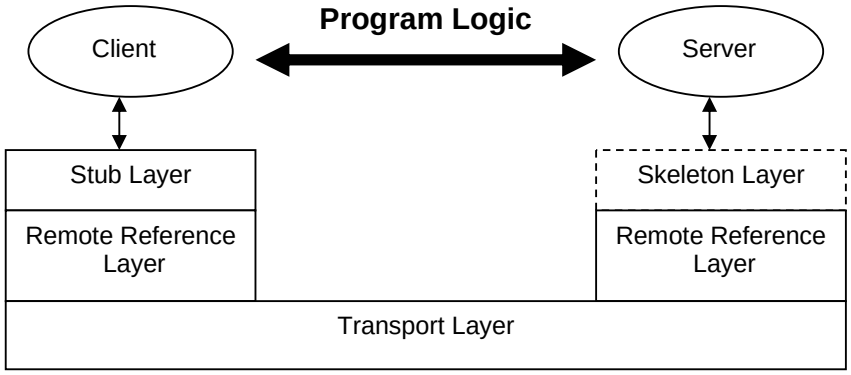
The RMI architecture is primarily based on the use of interfaces. A typical RMI application is composed by an interface which declares some *service* and is needed by at least two classes. One of these classes implements the interface on the server, whereas the other class runs on the client and acts as a proxy for the (remote) service, using the interface to get the type of the service object. The *Service* interface must extends the `java.rmi.Remote` interface.



A client program makes method calls on the proxy object, RMI sends the request to the remote JVM (JVM 2), and forwards it to the implementation. Any return values provided by the implementation are sent back to the proxy and then to the client‘s program.

1.3 Layers

The RMI implementation is essentially build from three abstraction layers:



The *Stub Layer* contains *Stub* objects. A *Stub* is a *Proxy* object, which knows how to forward method invocations to an object instanced on another JVM. To carry out the invocations, the stub needed in the past, on the server side, a corresponding layer, the *Skeleton*, that understood how to communicate with the stub across the RMI link. From the Java 2 implementation of RMI, the new wire protocol has made the skeleton class obsolete: RMI now uses reflection to make the connection to the remote service object.

The *Remote Reference Layer* defines and supports the invocation semantics of the RMI connection. It understands how to interpret and manage references made from clients to service objects across JVM's. For this purpose, it provides a *RemoteRef* object, that represents the link between the client and the remote service implementation object.

The *Transport Layer* is responsible for the network connection between client and server JVM's. This connection is a stream-based connection using the TCP/IP protocol. On top of TCP/IP, RMI originally uses a proprietary protocol called Java Remote Method Protocol (JRMP). The current version of RMI, also supports the Internet Inter-ORB Protocol (IIOP), developped by the Object Management Group, (OMG). This group have defined a vendor-neutral distributed object architecture, the Common Object Request Broker Architecture (CORBA), whose client and servers communicate with each other using IIOP. Hence, RMI-IIOP provides interoperability of RMI objects with CORBA objects, which can be implemented in various languages.

1.4 A Simple Example

1.4.1 A non-distributed Java Program

A simple time application program. The time service is declared in an interface. The client runs on the same JVM as the service.

Service

```
package rmi.time0;

public interface TimeService {
  String getTime();
}
```

Service Implementation

```
package rmi.time0;
```

```
import java.util.*;
public class TimeServiceImpl implements TimeService {

  public TimeServiceImpl() {
    System.out.println("TimeService object created");
  }

  public String getTime() {
    // Get and transmit time
    Date currentTime = Calendar.getInstance().getTime();
    return "Server time: " + currentTime;
  }
}
```

Client

```
package rmi.time0;

public class TimeClient {

  // MAIN
  public static void main(String[] args) {
    TimeService ts = new TimeServiceImpl();
    System.out.println(ts.getTime());
  }
}
```

1.4.2 A distributed Java Program Using RMI

The application is running on two JVM and using RMI.

Service

```
package rmi.time;

import java.rmi.*;
public interface TimeService extends Remote {
  String SERVICE_NAME = "TimeService";
  String getTime() throws RemoteException;
}
```

Service Implementation

```
package rmi.time;

import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class TimeServiceImpl extends UnicastRemoteObject
                             implements TimeService {

  public TimeServiceImpl() throws RemoteException {
    System.out.println("TimeService object created");
  }
  public String getTime(){
```

```
    // Get and transmit time
    Date currentTime = Calendar.getInstance().getTime();
    return "Server time: " + currentTime;
  }
}
```

Server

```
package rmi.time;

import java.rmi.*;
public class TimeServer {

  public static void main(String args[]){
    try {
      // Create and export the TimeService object
      TimeService ts = new TimeServiceImpl();
      // Bind this object's stub instance to the name "TimeService"
      Naming.rebind(TimeService.SERVICE_NAME, ts);
      System.out.println("Time Service bound in registry");
    } catch (java.net.MalformedURLException mue) {
      mue.printStackTrace();
    } catch (RemoteException re) {
      re.printStackTrace();
    }
  }
}
```

Client

```
package rmi.time;

import java.rmi.*;
public class TimeClient {

  // MAIN
  public static void main(String[] args) {
    String host = args[0]; // Assume host name in args[0]
    int port = 1099;
    try {
      TimeService ts =
        (TimeService) Naming.lookup("//" + host + ":" + port + "/" +
                                    TimeService.SERVICE_NAME);
      System.out.println(ts.getTime());
    } catch (java.net.MalformedURLException  mue) {
      mue.printStackTrace();
    } catch (NotBoundException nbe) {
      nbe.printStackTrace();
    } catch (RemoteException re) {
      re.printStackTrace();
    }
  }
}
```

1.5 The RMI Client/Server Solution

1.5.1 The Service Remote Interface

In RMI, the client and server's service can only communicate through a *remote interface*. An interface is **remote** if it extends the interface `java.rmi.Remote`, which is a „tag-interface“ (it contains no methods).

An object of a class that implements a remote interface is called a **remote object**.

Only those methods specified in an remote interface are available remotely, i.e. can be invoked from a non-local JVM..

Each method of a remote interface must list a `java.rmi.RemoteException` in its `throw` clause.

1.5.2 The Service Implementation

The service `TimeServiceImpl.java` must implement the remote interface and typically extends the `java.rmi.server.UnicastRemoteObject` class. This class is a convenience class that mainly includes some contructors and static methods that allow to ***export* a remote object**. Exporting an object makes it available to receive incoming calls from clients running on other JVMs. The export action also builds and returns the object's **stub**, which is a client's local representative or proxy for the remote object.

By extending `java.rmi.server.UnicastRemoteObject`, the `TimeServiceImpl` class is used to create and export a simple remote object that supports TCP unicast (i.e. point-to-point) remote communication and that uses RMI's default socket-based transport for communication.

1.5.3 The Server

To be initially findable by code running in other JVMs, the service implementation must register itself in a remote object registry. For bootstrapping purpose, the RMI system provides such an object registry, called `rmiregistry`, that allows you to *bind* (i.e. to associate) a URL-formatted name to the remote object reference. This name will be used by the client to lookup the remote object. The bind process is controlled by the methods of the class `java.rmi.Naming`.

In our example, the server's main method creates an instance of the remote object and associates it with the service name by using the `rebind` method. The full URL-formatted service name has the form:

```
rmi://hostName:portNo/serviceName
```

- `rmi` : protocol (not needed)
- `hostName` : name or IP address of the server machine. Default is `"localhost"`
- `portNo` : number of the server communication port. Default is "1099"
- `serviceName`: Name of the server service. Mandatory.

In the server, we use all possible defaults:

```
Naming.rebind(SERVICE_NAME, ts);
```

The second argument (`ts`) is the associated remote object.

1.5.4 The Client Implementation

The client gets a reference to the remote object implementation ("TimeService") from the server host's RMI registry with:

```
TimeService ts =
       (TimeService) Naming.lookup("//" + host + ":" + port + "/" +
                                   TimeService.SERVICE_NAME)
```

Like the `Naming.rebind` method, the `Naming.lookup` method takes a URL-formatted string.

1.5.5 Building and Running the RMI Application

Execute the following steps:

1. Compile the java files
2. Start the RMI registry
3. Start the server
4. Start the client(s)

We assume that the current directory is the class root directory.

Step 1: Compile the java files

```
javac –d . *.java
```

Step 2: Start the RMI registry

The RMI registry is a simple server-side name service that allows clients to get a reference to a remote object. Typically, it is used only to locate the first remote object a client needs to communicate with. That object in turn should provide application-specific support for finding other objects.

For security purpose, the RMI registry must be located on the same host as the server. For each service (server) implemented on the host, it contains an associated *stub* object. A *stub* is a simple proxy for a remote object which can be dynamically generated at runtime and forwards RMI calls to the actual remote object implementation. It should be registered in the RMI registry by the `Naming.rebind` method, when the server is started.

You start the RMI registry with the command:

```
rmiregistry [portNo]
```

The default port number is "1099". This command produces no output and should be run in the background (Windows: `start /min rmiregistry`; Unix: `rmiregistry &`). For debugging pupose, you can display a trace of the registry class loading process:

```
rmiregistry –J-Dsun.rmi.loader.logLevel=VERBOSE
```

Step 3: Start the server

To start the server, the service interface (*TimeService*) and implementation (*TimeServiceImpl*) classes must be accessible. When started, the server registers the *stub* object associated to the remote service in the RMI registry.

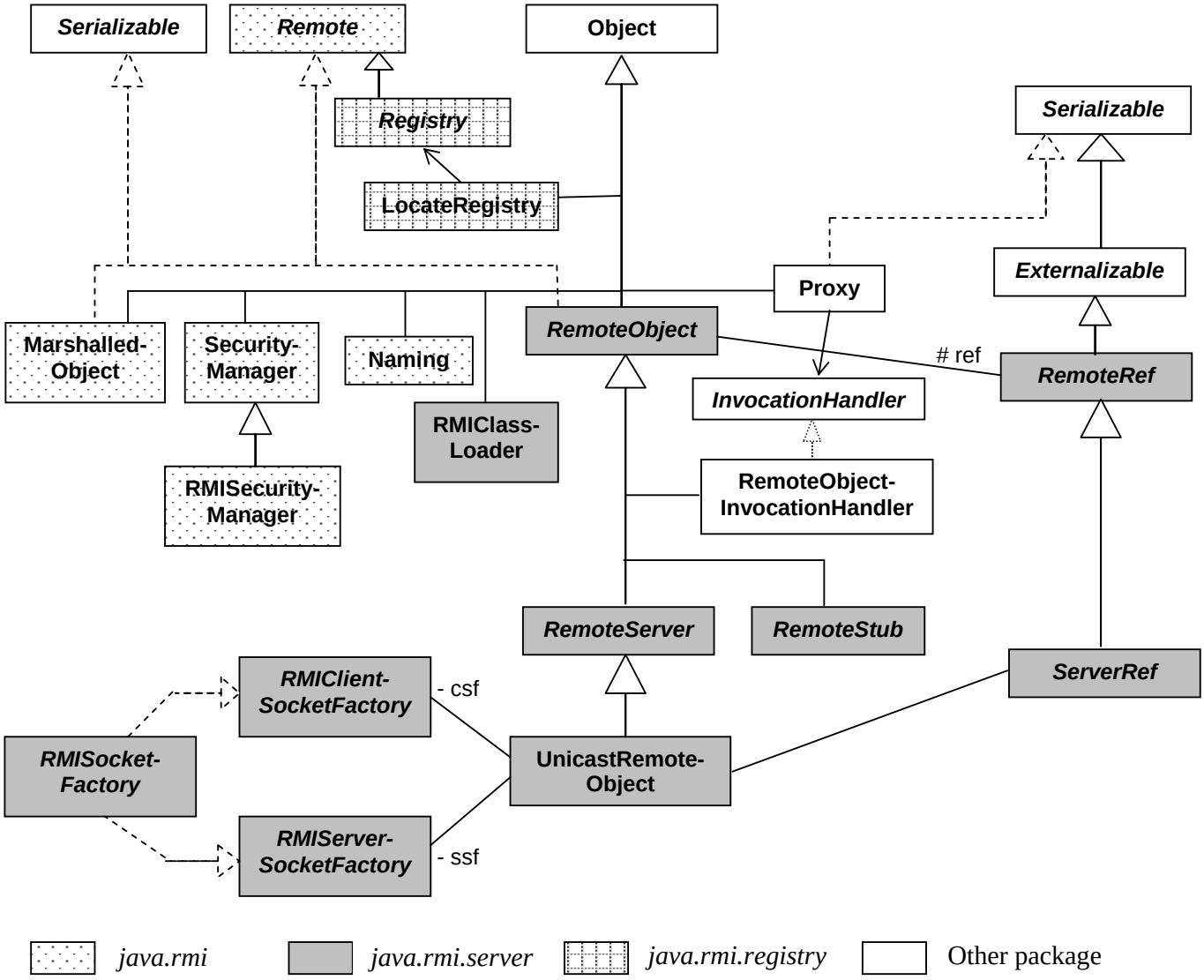
```
java rmi.time.TimeServer
```

Step 4: Start the client(s)

In this simple version, only the interface (*TimeService*) and the client (*TimeClient*) classes must be accessible. When the client is started, the statement `Naming.lookup` contacts the RMI registry and **receive** from it a **copy** of the service stub object. As the stub contains a **remote object reference**, it builds now a client-side proxy that allows the client to access transparently the complete set of remote methods that the remote object implements.

```
java rmi.time.TimeClient
```

1.6 The Basic RMI Classes



1.6.1 Naming

This class provides methods for storing and obtaining references to remote objects in the remote object registry.

```
public final class Naming {

  public static Remote lookup(String url)
    // Returns a stub for the remote object associated
    // with the URL-formatted name

  public static void bind(String url, Remote obj)
    // Binds (associates) the specified name with the
    // remote object, only if the name is not already in use)

  public static void rebind(String url, Remote obj)
    // Binds (associates) the specified name with the remote
    // object, even if the name has already been in use (replace)
```

```
  public static void unbind(String url)
    // Remove the name-object association

  public static String[] list(String url)
    // Returns an array of Strings containing the URL's-
    // formatted names bound in the registry
}
```

Note: Only client that are local to the host on which the registry runs are permitted to execute the operations `bind, rebind` and `unbind`.

1.6.2 RemoteObject

This class mainly implements the `Object` behavior for remote objects. Moreover, it helds a reference to the remote objects' `RemoteRef`.

```
public abstract class RemoteObject
                implements java.rmi.Remote, java.io.Serializable {

  // RemoteRef reference
  protected transient RemoteRef ref;

  // Constructors ...

  // Modelling Object behavior for remote object (see API
  // for exact specification)
  public int hashCode();
  public boolean equals(Object obj);
  public String toString();

  ......
}
```

1.6.3 RemoteServer

This is the common superclass to the server implementation classes `java.rmi.server.UnicastRemoteObject` and `java.rmi.activation.Activatable` (not represented: see chapter *Object Activation*). It allows to log RMI calls to a specified output stream for debugging purpose.

```
public abstract class RemoteServer extends RemoteObject {
  // Constructors ...

  // Log stream
  public static void setLog(java.io.OutputStream out);
  public static java.io.PrintStream getLog();

  ......

}
```

1.6.4 UnicastRemoteObject

This class provides support for creating and exporting remote objects, i.e. make them available for incoming calls from remote JVM's.

```
public class UnicastRemoteObject extends RemoteServer {

  // Constructors
  protected UnicastRemoteObject()
  protected UnicastRemoteObject(int port)
  protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,
                                RMIServerSocketFactory ssf);

  // Exporting methods
  public static RemoteStub exportObject(Remote obj);
  public static Remote exportObject(Remote obj, int port);
  public static Remote exportObject(Remote obj, int port,
                                    RMIClientSocketFactory csf,
                                    RMIServerSocketFactory ssf);
  public static boolean unexportObject(Remote obj);

  // Miscellaneous
  public Object clone();
}
```

The constructors create and export a remote object on an arbitrary or specified port, possibly using sockets created by a *RMISocketFactory*. For that, they simply invoke the corresponding `exportObject` methods, which create a stub object and return a reference to it.

A remote object *must* be exported prior to the first time it is passed in an RMI call. Once exported, the remote object can accept invocations from clients on a remote JVM. It can also be passed as an argument in a RMI called or returned as the result of an RMI call, just as a non-remote object.

Note that `UnicastRemoteObject` implements the `clone()` method. If a subclass implements the `Cloneable` interface, the corresponding remote object will be able to be cloned properly.

However, stubs for remote objects are declared *final* and do not implement the `Cloneable` interface. Therefore, cloning a stub is not possible.

1.6.5 RemoteRef

This interface represents the handle for a remote object. The `RemoteRef` object understands the invocation semantics for remote services.

Each stub contains an instance of `RemoteRef` that contains the concrete representation of a reference. The stubs uses the `invoke() method` in `RemoteRef` to forward the method calls to the remote host.

Moreover, the `RemoteObject` methods `hashCode()`, `equals(Object obj)` and `toString()` forward their calls to the corresponding `RemoteRef` methods `remoteHashCode()`, `remoteEquals(Object obj)` and `remoteToString()`.

1.6.6 ServerRef

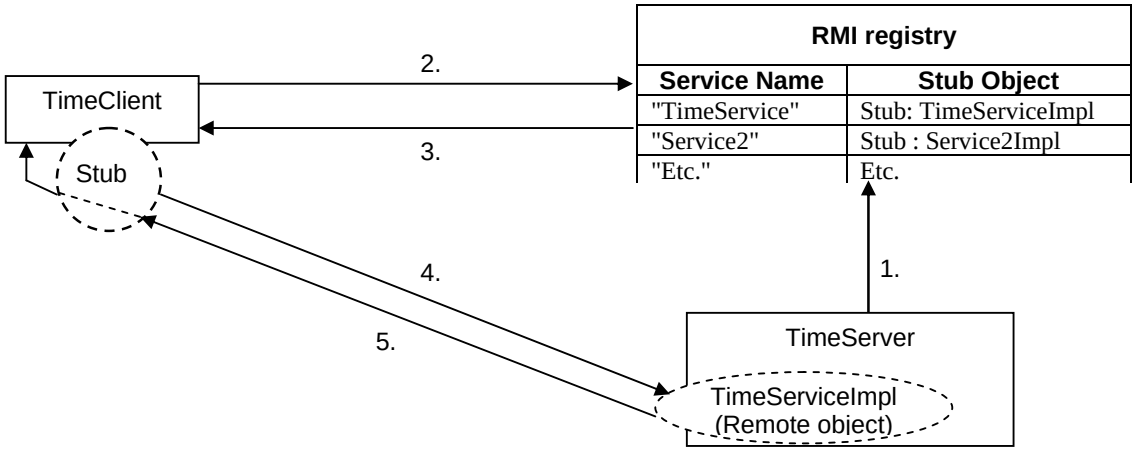
This interface represents the server-side handle for a remote object implementation. Its method

```
RemoteStub exportObject(Remote obj, Object data)
```

finds or creates a stub object for the supplied `Remote` object implementation *obj*. *data* contains information necessary to export the object (e.g. port number).

1.7 Data Flow in a Simple RMI Example

In our simple *TimeService* example, the main data flow can be visualized as follows:



1. The server *TimeServer* registers the stub for the remote object *TimeServiceImpl*, bound to the service name *"TimeService"*, into the RMI registry (**`Naming.rebind(...)`**)
2. The client *TimeClient* makes a lookup call (**`Naming.lookup(...)`**)
3. The RMI registry returns an instance of the remote object's stub to the client (**`TimeService ts = (TimeService)`** `Naming.lookup(...)`)
4. The client makes a remote call (**`ts.getTime()`**)
5. The remote object returns a *String* object to the client (`ts.getTime()` → **`String`**)

1.8 The RMI Registry

The RMI registry is implemented as an RMI server (the corresponding remote interface's name is: `java.rmi.registry.Registry).` The `Naming` class is only a wrapper whose methods redirect to the standard implementation `sun.rmi.registry.RegistryImpl`.

To get an initial reference to the registry, i.e. to get the registry's stub, `Naming` is using the class `java.rmi.registry.LocateRegistry. LocateRegistry` contains two types of static methods:

- "Bootstrap" methods:

```
public static Registry getRegistry(...)
```

- "Creation" methods:

```
public static Registry createRegistry(...)
```

The "bootstrap" methods know how to create a local reference to a remote registry. They parse the URL transmitted by the corresponding `Naming` method and return a stub for the registry running on the given machine and port (without attempting to connect to it!).

The "creation" methods allow to launch a registry from within an application, without using the `rmiregistry` program. These methods must be called directly (not via `Naming`).

With the following code addition, our `TimeServer` would launch a registry, if noone is activ on the given port.

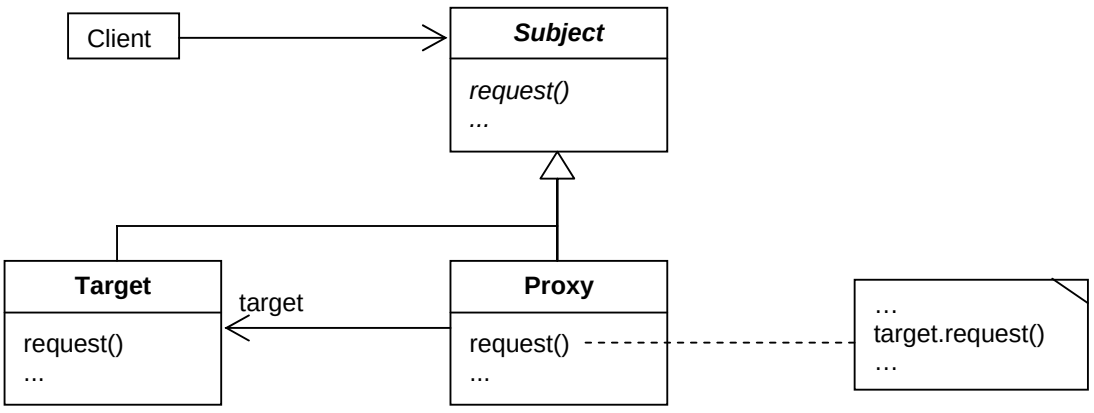
```
package rmi.time;

import java.rmi.*;
import java.rmi.registry.*;

public class TimeServer {

  public static void main(String args[]) {
    final int port = 1099;
    try {
      // Create the TimeService object
      TimeService ts = new TimeServiceImpl();
      // Create and launch a RMI registry if needed
      try {
        LocateRegistry.createRegistry(port);
      } catch (RemoteException re) {
        System.out.println("Registry already activ on port:" + port);
      }
      // Bind this object' stub instance to the name "TimeService"
      Naming.rebind(TimeService.SERVICE_NAME, ts);
      System.out.println("Time Service bound in registry");
    } catch ...
      ......
  }
}
```

The RMI registry is an easy to use, but rather primitive naming service. One of its main drawbacks is its "flat" architecture. Inside it, you cannot organize your servers hierarchically as, for example, in a file system. However, as a RMI registry is a "normal" RMI server, you can sometimes help yourself by storing "registries within registries", building a "registry hierarchy".

1.9 The Stub

A *proxy* (the stub) is an object that supports the interface of another object, its *target* (the remote object), so that the proxy can substitute for the target for all practical purposes. Acting as an intermediary, the proxy delegates some or all of the calls that it receives to its target.

Structure of the Proxy Pattern (Gamma, …)

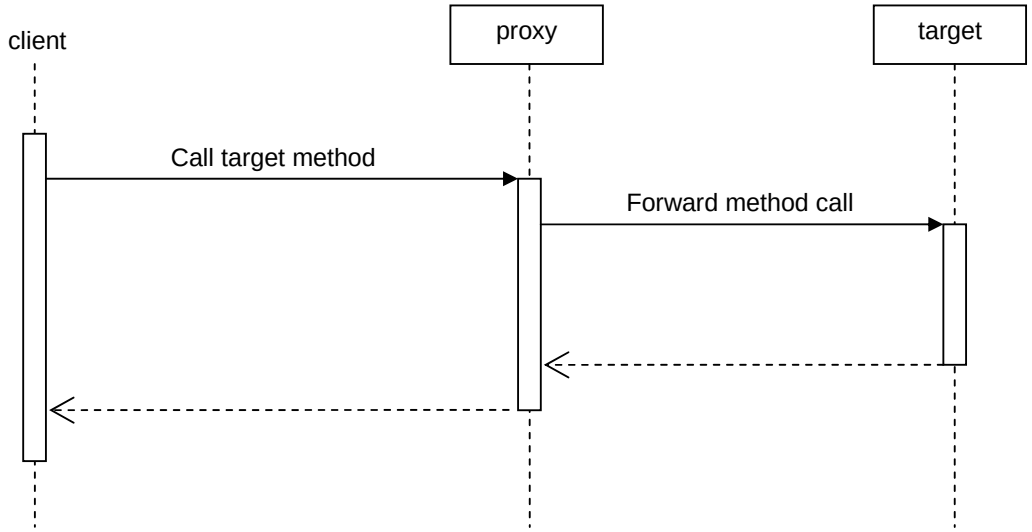


Proxies allow to insert code between the target and the proxy. Therefore, a proxy can hide complex tasks such as the nework communication by RMI from the proxy user, without changing the implementation objects' functionality. When developing Java applications, you can implement *static* or *dynamic* proxies.

1.9.1 Static Proxies

Static proxies are programmed into the system at compile time. Programming static proxies is just like programming any other class, introducing new code into the proxy to implement the new functionality.

Typical use of a proxy:



The following code shows a simple static proxy, which adds an advertisement message to our first (non-distributed, package `rmi.time0`) *TimeService*. (Note: to keep here the analogy with RMI, the proxy constructor simply gets the target implementation as a parameter. Normally, the proxy user (*TimeClient*) should not create the implementation itself, but delegate it to a proxy factory):

Proxy

```
package rmi.time0;


public class TimeAdvProxy implements TimeService {

  private static final String
    ADV = "** Info presented by the CS dept. of the HTI **";

  private TimeService impl;

  public TimeAdvProxy(TimeService impl) {
    this.impl = impl;
  }

  public String getAdvertisement() {
    return ADV;
  }

  public String getTime() {
    // Get time and return it with advertisement
    return ADV + "\n - " + impl.getTime();
```

```
  }
}
```

Proxy Factory

```
package rmi.time0;

public class ProxyFactory {

  public static TimeService getAdvProxy(Object impl) {

    // Normally, should create the service implementation here
    if (impl instanceof TimeService)
      return new TimeAdvProxy((TimeService)impl);
    else
      return null;
  }
}
```

Client

```
package rmi.time0;

public class TimeClient2 {

  // MAIN
  public static void main(String[] args) {
    TimeService ts = new TimeServiceImpl();
    TimeService proxy = ProxyFactory.getAdvProxy(ts);
    if (proxy != null) {
      System.out.println(proxy.getTime());
      if (proxy instanceof TimeAdvProxy)
     System.out.println(((TimeAdvProxy)proxy).getAdvertisement());
    }
  }
}
```

1.9.2 Dynamic Proxies

Dynamic proxies differ from static ones in that they do not exist at compile time. Instead, they are generated at runtime by the JDK, using reflection, and then made available to the user. These *dynamic proxy classes* are classes that implement a list of interfaces specified at runtime such that a method invocation through one of the interfaces on an instance of the class will be encoded and dispatched to another object through a uniform interface. The dynamic creations of the proxy classes are accomplished with static methods of the `java.lang.reflect.Proxy` class:

```
public class Proxy implements java.io.Serializable {

  public static InvocationHandler getInvocationHandler(Object proxy)
    // Returns the invocation handler for the
    // specified proxy instance

  public static Class<?> getProxyClass(ClassLoader loader,
                                       Class<?>... interfaces)
    // Returns the java.lang.Class object for a proxy class
    // given a class loader and an array of interfaces.
```

```
    public static boolean isProxyClass(Class<?> cl)
      // Returns true if and only if the specified class was
      // dynamically generated to be a proxy class using the
      // getProxyClass method or the newProxyInstance method.

    public static Object newProxyInstance(Classloader loader,
                                          Class<?>[] interfaces,
                                          InvocationHandler h)
      // Returns an instance of a proxy class for the
      // specified interfaces that dispatches method
      // invocations to the specified invocation handler.
}
```

Each class constructed by the factory method `getProxyClass` and `newProxyInstance` is a public final subclass of `Proxy` (referred to as *proxy* class). `getProxyClass` retrieves the proxy class specified by the class loader and the array of interfaces. If such a proxy class does not exist, it is dynamically constructed; its name begins with `$Proxy` followed by a number. A proxy instance is assignement compatible with all the interfaces specified in the factory method.

A proxy instance for an object `impl` may be constructed by the following code lines:

```
Class cl = Proxy.getProxyClass(SomeClass.getClassLoader(),
                               New Class[]{SomeInterface.class});
Constructor co = cl.getConstructor(new Class[]{InvocationHandler.class});
Object proxy = co.newInstance(new Object[]{new SomeInvocHandler(impl)});
```

or easier, by using `newProxyInstance`:

```
Object proxy = Proxy.newProxyInstance(SomeClass.getClassLoader(),
                                      new Class[]{SomeInterface.class},
                                      new SomeInvocHandler(impl));
```

`Proxy` allows programmers to delegate task by providing the `InvocationHandler` interface:

```
public interface InvocationHandler {

  public Object invoke(Object proxy, Method method, Object[] args);

}
```

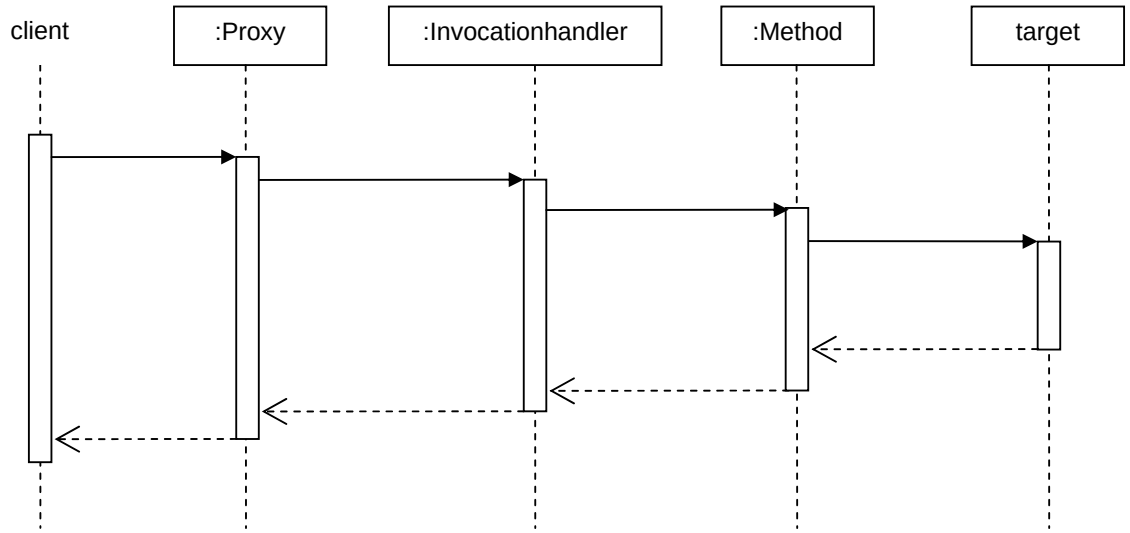
A proxy instance forwards method calls to its invocation handler by calling `invoke`. The arguments of `invoke` are:

- `proxy:` a reference to the proxy instance (itself)
- `method:` a reference to the `Method` object representing the invoked method
- `args:` an array containing the original arguments for the method call

For example, the following implementation of `invoke` passes every call to an `impl` object transparently:

```
public Object invoke(Object proxy, Method, method, Object[] args)
  throws Throwable {

  return method.invoke(impl, args);
}
```

Objects involved in forwarding a method with a Java dynamic proxy:



The following code shows the *TimeService* example above, implemented with a dynamic proxy. Notice that the factory method `getProxy` that instantiates the proxy, can be located in the invocation handler.

Invocation Handler

```
package rmi.time0;

import java.lang.reflect.*;

public class AdvHandler implements InvocationHandler {

  private Object impl;
  private static final String
    ADV = "** Info presented by the CS dept. of the HTI **";

  private AdvHandler(Object impl) {
    this.impl = impl;
  }

  public static Object getProxy(Object impl) {
    Class cl = impl.getClass();
    return Proxy.newProxyInstance(cl.getClassLoader(),
                        cl.getInterfaces(),
                        new AdvHandler(impl));
  }

  public String getAdvertisement() {
    return ADV;
  }

  public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    // Get and transmit time
    return ADV + "\n - " + method.invoke(impl, args);
  }
}
```

Client

```
package rmi.time0;

import java.lang.reflect.*;

public class TimeClient3 {

  // MAIN
  public static void main(String[] args) {
    TimeService ts = new TimeServiceImpl();
    TimeService proxy = (TimeService)AdvHandler.getProxy(ts);
    System.out.println(proxy.getTime());
    InvocationHandler handler = Proxy.getInvocationHandler(proxy);
    if (handler instanceof AdvHandler)
      System.out.println(((AdvHandler)handler).getAdvertisement());
  }
}
```

1.9.3 Dynamic Generation of Stub Classes

When an application exports a remote object, using methods of the `UnicastRemoteObject` class, it will instantiate a static stub proxy if a corresponding stub class has been pregenerated. Otherwise, the remote object's stub will be a `java.lang.reflect.Proxy` instance (whose class is dynamically generated) with a `java.rmi.server.RemoteObjectInvocationHandler` as its invocation handler.

A stub class can be pregenerated by the `rmic` compiler from the service implementation class (`TimeServiceImpl.class`). To create a stub class for (RMI version of) the *TimeService* example, enter:

```
rmic rmi.time.TimeServiceImpl
```

The stub class will be called `TimeServiceImpl_Stub.class`.

As an example, here is the source code of stub generated by `rmic` for the `TimeServiceImpl` class:

```
// Stub class generated by rmic, do not edit.
// Contents subject to change without notice.

package rmi.time;

public final class TimeServiceImpl_Stub extends
java.rmi.server.RemoteStub
                implements rmi.time.TimeService, java.rmi.Remote {
  private static final long serialVersionUID = 2;

  private static java.lang.reflect.Method $method_getTime_0;

  static {
    try {
      $method_getTime_0 =
        rmi.time.TimeService.class.getMethod("getTime",
                                             new java.lang.Class[] {});
    } catch (java.lang.NoSuchMethodException e) {
      throw new java.lang.NoSuchMethodError(
                                  "stub class initialization failed");
    }
```

```
  }

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

  // methods from remote interfaces

  // implementation of getTime()
  public java.lang.String getTime() throws java.rmi.RemoteException {
    try {
      Object $result = ref.invoke(this, $method_getTime_0,
                          null, 1253370244719889472L);
      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);
    }
  }
}
```

An existing application can be deployed to use dynamically generated stub classes unconditionally (i.e. whether or not pregenerated stub classes exist) by setting the system property `java.rmi.server.ignoreStubClasses` to "`true`".

1.10 Implementing a Remote Object Without Extending *UnicastRemoteObject*.

A remote object implementation does not have to extend `UnicastRemoteObject`, but any implementation that does not must export the remote object by calling `UnicastRemoteObject.exportObject` (or `Activatable.exportObject`) itself and supply appropriate implementations of the `java.lang.Object` methods (`equals()`, `hashCode()`, `toString()`).

Service Implementation

```
package rmi.time;

import java.util.*;
public class TimeServiceImpl2 implements TimeService { // No extends

  public TimeServiceImpl2() { // No throws
    System.out.println("TimeService object created");
  }

  public String getTime(){
    // Get and transmit time
    Date currentTime = Calendar.getInstance().getTime();
    return "Server time: " + currentTime;
  }
}
```

Server

```
package rmi.time;

import java.rmi.*;
import java.rmi.server.*;

public class TimeServer2 {

  public static void main(String args[]) {
    try {
      // Create and export the TimeService object
      TimeService ts = new TimeServiceImpl2();
      TimeService stub =
              (TimeService) UnicastRemoteObject.exportObject(ts);
      // Bind the stub to the name "TimeService"
      Naming.rebind(TimeService.SERVICE_NAME, stub);
      System.out.println("Time Service bound in registry");
    } catch (java.net.MalformedURLException mue) {
      mue.printStackTrace();
    } catch (RemoteException re) {
      re.printStackTrace();
    }
  }
}
```

The static method **`UnicastRemoteObject.exportObject(Remote)`** is declared to return an object of type `RemoteStub` and therefore **cannot be used** to export a remote object **to use a dynamically generated stub** class for its stub. An instance of a dynamically generated stub class is an object of type `Proxy`, which is not assignable to `RemoteStub`.

1.11 Passing Parameters in RMI

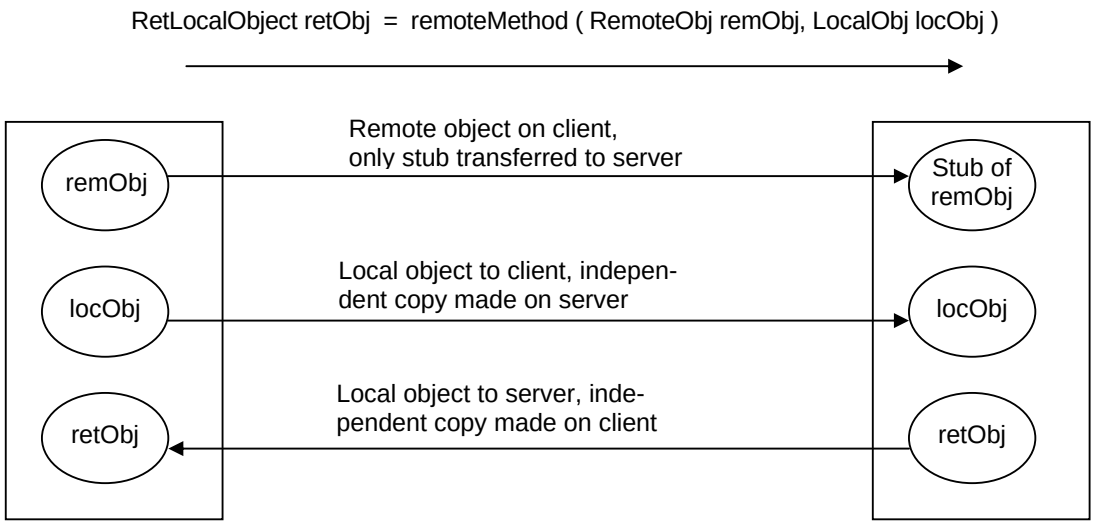
In RMI, you can move forward (from client to server) and backward (from server to client) the following entity types:

- Values of primitive data type
- Remote objects
- Serializable objects

These items are passed as arguments or return value in methods of a remote interface according to the following rules:

- All primitive data type variables are passed by value.
- Remote objects are essentially passed by reference. The remote reference passed is the object stub. In a RMI call, the replacement of a (exported) remote object by its stub is carried out by the RMI system itself.
- Local objects are passed by value, using the serialization technique.

These rules are true regardless of the direction of the call (forward or backward).



Example:

The following program creates a *remote Circle* object, which is determined by a ***remote Point*** (the center) and a radius.

Circle - A Remote interface

```
package rmi.circle;

import java.rmi.*;

public interface Circle extends Remote {
  String SERVICE_NAME = "CircleService";
  Point getCenter() throws RemoteException;
  double getRadius() throws RemoteException;
  void setCircle(Point c, double r) throws RemoteException;
}
```

CircleImpl Class - The Implementation of the Remote Circle

```
package rmi.circle;

import java.rmi.*;
import java.rmi.server.*;

public class CircleImpl extends UnicastRemoteObject
                        implements Circle {

  private Point center;
  private double radius;

  public CircleImpl(int x, int y, double r) throws RemoteException {
    setCircle(new PointImpl(x, y), r);
  }

  public void setCircle(Point c, double r) throws RemoteException {
```

```
    center = c;
    radius = r;
    System.out.println("Circle defined - Center: " +
                       center.getCoord() + "   Radius: " + radius);
  }

  public Point getCenter() { return center; }

  public double getRadius() { return radius; }
}
```

Point - A Remote interface

```
package rmi.circle;

import java.rmi.*;

public interface Point extends Remote {
  public void move(int x, int y) throws RemoteException;
  public String getCoord() throws RemoteException;
}
```

PointImpl Class - The Implementation of the Remote Point

```
package rmi.circle;

import java.rmi.*;
import java.rmi.server.*;

public class PointImpl extends UnicastRemoteObject
                       implements Point {

  private int x, y;

  public PointImpl(int x, int y) throws RemoteException {
    this.x = x; this.y = y;
  }

  public void move(int x, int y) {
    this.x += x;
    this.y += y;
    System.out.println("Point has been moved to: " + getCoord());
  }

  public String getCoord() {
    return "[" + x + "," + y + "]";
  }
}
```

CircleServer Class - The Remote Circle Server

```
package rmi.circle;

import java.rmi.*;

public class CircleServer {
```

```
  public static void main(String args[]) throws Exception {
    Circle circle = new CircleImpl(10, 20, 30);
    Naming.rebind(Circle.SERVICE_NAME, circle);
    System.out.println("Circle bound in registry");
  }
}
```

CircleClient Class

```
package rmi.circle;

import java.rmi.*;

public class CircleClient {

  // MAIN
  public static void main(String[] args) throws Exception {
    String host = args[0]; // Assume host name in args[0]
    int port = 1099;
    Circle circle =
      (Circle) Naming.lookup("//" + host + ":" + port + "/" +
                             Circle.SERVICE_NAME);
    System.out.println(circle);
    double r = circle.getRadius() * 2;
    Point p = circle.getCenter();
    System.out.println(p);
    p.move(30, 50);
    System.out.println("Circle - Center: " + p.getCoord() +
                  "   Radius: " + r);
    circle.setCircle(p, r);
  }
}
```

This program produces the following output:

Server Side

```
F:\rmi\circle>java rmi.circle.CircleServer
Circle defined - Center: [10,20]   Radius: 30.0
Circle bound in registry
Point has been moved to: [40,70]
Circle defined - Center: [40,70]   Radius: 60.0
```

Client Side

```
F:\rmi\circle>java rmi.circle.CircleClient
Proxy[Circle,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[147.87.7.160:1038](remote),objID:[12e78c:ea4e471d10:-8000, 0]]]]]
Proxy[Point, RemoteObjectInvocationHandler[UnicastRef: [liveRef:
[endpoint:[147.87.7.160:1038](remote),objID:[12e78c:ea4e471d10:-8000, 1]]]]
Circle - Center: [40,70]   Radius: 60.0
F:\rmi\circle>
```

Circle Interface, CircleServer Class, CircleClient Class

```
package rmi.circle2;

. . . dito corresponding classes . . .
```

CircleImpl Class

```
package rmi.circle2;

. . . dito . . .

  public CircleImpl(int x, int y, double r) throws RemoteException {
    setCircle(new Point(x, y), r);
  }

. . . dito . . .
```

Point Class - The Implementation of the Local Point

```
package rmi.circle2;

public class Point implements java.io.Serializable {

  private int x, y;

  public Point(int x, int y) {
    this.x = x; this.y = y;
  }

  public void move(int x, int y) {
    this.x += x;
    this.y += y;
    System.out.println("Point has been moved to: " + getCoord());
  }

  public String getCoord() {
    return "[" + x + "," + y + "]";
  }
}
```

The program output is now:

Server Side

```
F:\rmi\circle2>java rmi.circle2.CircleServer
Circle defined - Center: [10,20]   Radius: 30.0
Circle bound in registry
Circle defined - Center: [40,70]   Radius: 60.0
```

Client Side

```
F:\rmi\circle2>java rmi.circle2.CircleClient
Proxy[Circle,RemoteObjectInvocationHandler[UnicastRef [liveRef:
[endpoint:[147.87.7.160:1053](remote),objID:[239137:ea4e5abf82:-8000, 0]]]]]
rmi.circle2.Point@58957f
Point has been moved to: [40,70]
Circle - Center: [40,70]   Radius: 60.0
F:\rmi\circle2>
```

1.12 Which Objects are Equals?

As remote objects are passed by reference from one JVM to another, you may have to define their remote semantic for the `equals()`, `hashCode()` and `toString()` methods. The standard RMI remote semantic for these methods is implemented in the class `RemoteObject`, which is a superclass of `UnicastRemoteObject`. According to this semantic, all objects which act as proxies for the same underlying remote objects (i.e. the *stubs* of a remote object) are considered to be **equal**. Hence:

- When comparing two of these stubs, the corresponding `equals()` method must return *true*
- The corresponding `hashcode()` method must return the same value for any of these stubs.

RMI do not require a fixed remote semantic, when comparing a stub with its underlying remote object. Currently, a remote object which extends `UnicastRemoteObject` implements the following semantic:

- A remote object is **equal** to its corresponding stub instance, when the stub class is from type `RemoteStub` (i.e. stub class has been pregenerated)
- A remote object is **not equal** to to its corresponding stub instance, when the stub class is from type `Proxy` (i.e. stub class has been dynamically generated).

When the remote object is not a subclass of `UnicastRemoteObject`, it could be necessary to override its `equals()`and `hashCode()` methods to implement the same behavior. A typical implementation of could be the following (updated from "Java RMI", by W.Grosso):

```
import java.rmi.*;
import java.rmi.server.*;

public class RemoteServiceImpl implements RemoteService {
  ...
  ...
  ...
  public boolean equals(Object obj) {
    // Three cases. Either it's us, or it's our remote stub
    // (with pregenerated stub class), or it's not equal.
    // "our stub" can arise, for example, if one of our
    // methods took an instance of RemoteServiceImpl().
    // A client could then pass in, as an argument, our stub.
    if (obj instanceof RemoteServiceImpl)
      // obj is a remote object
      return obj == this;
    if (obj instanceof RemoteStub) {
      // obj is a remote stub
      try {
        Remote stub = RemoteObject.toStub(this);
        return stub.equals(obj);
      } catch (NoSuchObjectException nsoe) {
```

```
          System.out.println("Stub does not exist");
        }
      }
      return false;
    }

    public int hashCode() {
      try {
        Remote stub = RemoteObject.toStub(this);
         return stub.hashCode();
      } catch (NoSuchObjectException nsoe) {}
      return super.hashCode();
    }
  }
```

2 Using Custom Sockets

RMI, through the definition of custom socket factories, contains a flexible way of letting you use whatever sockets you fell like using to handle the raw bytes that need to get send over the wire. For example, if your server sends or receives sensitive data, you might want a socket that encrypts the data; if very large amount of data is transmitted, you might want a socket that compresses the data, etc.

Let's assume that an appropriate custom socket and server socket class have already been written and are ready to be used. You have to do two things: create custom socket factories for server and client sockets and implement them in the server.

2.1 Custom Socket Factories

Your socket factories must implement the RMI socket factories interfaces:

class ServerSocketFactory

```
package rmi.socketFac;

import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class ServerSocketFactory implements RMIServerSocketFactory {

  private int hash = "ServerSocketFactory".hashCode();

  public ServerSocket createServerSocket(int port)
                                          throws IOException {
    return new CustomServerSocket(port);
  }

  public boolean equals(Object obj) {
    if (obj == null) return false;
    return getClass() == obj.getClass());
  }

  public int hashCode() {
    return hash;
  }
}
```

This class creates instances of `CustomServerSocket` and overrides `equals` and `hashCode`. Because most servers in any given application use the same types of sockets, RMI allow them to share sockets. In order to handle this, RMI maintains a mapping from socket factories to open sockets. When a server needs a socket, RMI uses a map, which is keyed on the associated socket factory to see if there are any available sockets. If yes, it re-uses one of them; if not it creates a new one. In order for RMI to do this effectively, `equals` and `hashCode` must be overridden (source: "Java RMI", by W. Grosso, O'Reilly).

class ClientSocketFactory

```
package rmi.socketFac;

import java.io.*;
import java.net.*;
```

```
import java.rmi.server.*;

public class ClientSocketFactory implements RMIClientSocketFactory,
                                             Serializable {

  private int hash = "ClientSocketFactory".hashCode();

  public Socket createSocket(String host, int port)
                                                 throws IOException {
    return new CustomSocket(host, port);
  }

  public boolean equals(Object obj) { ... }

  public int hashCode() { ... }

}
```

The type of socket that a server uses is entirely a server-side property. In order for the client to connect to the server, it must be using the correct socket type, i.e. it must deserialize an instance of the appropriate socket factory. This happens automatically when the client deserializes the stub, which has a reference to the implemented `RMIClientSocketFactory`. When the client obtains the stub, it also obtains a copy of the socket factory. Therefore, the client socket factory must be *serializable*.

2.2 Incorporating a Custom Socket into an Application

To pass on references of the custom socket factories to the RMI runtime, you have to call the following constructor of the `UnicastRemoteObject` class (or the corresponding `exportObject` method):

```
protected UnicastRemoteObject(int port, RMIClientSocketFactory csf,
                              RMIServerSocketFactory ssf);
```

For example:

```
package rmi.socketFac;

import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class TimeServiceImpl extends UnicastRemoteObject
                             implements TimeService {

  public TimeServiceImpl(RMIClientSocketFactory csf,
                         RMIServerSocketFactory ssf)
                                        throws RemoteException {
    super(0, csf, ssf);
    System.out.println("TimeService object created");
  }
  ...
}
```

The RMI socket factories are typically instantiated in the server:

```
package rmi.socketFac;
```

```
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;

public class TimeServer {

  public static void main(String args[]) throws Exception {
    final int port = 1099;
    // Create the socket factories
    RMIClientSocketFactory csf = new ClientSocketFactory();
    RMIServerSocketFactory ssf = new ServerSocketFactory();
    // Create TimeService object
    TimeService ts = new TimeServiceImpl(csf, ssf);
    // If needed, create a RMI registry, using the same factories
    try {
      LocateRegistry.createRegistry(port, csf, ssf);
    } catch (RemoteException ex) {
      System.out.println("Registry already activ on port: " + port);
    }
    // Bind the stub's object instance to the name "TimeService"
    Naming.rebind(TimeService.SERVICE_NAME, ts);
    System.out.println("Time Service bound in registry");
  }
}
```

Note that RMI comes with an abstract class, `RMISocketFactory` (package: `java.rmi.server`), which implements both `RMIServerSocketFactory` and `RMIClientSocketFactory`. An `RMISocketFactory` instance is used by the RMI runtime in order to obtain client and server sockets for RMI calls. An application server may use the `setSocketFactory` method to request that the RMI runtime use its socket factory instance instead of the default implementation. For example:

For example, in the server code:

```
package rmi.socketFac;

import java.rmi.*;

public class TimeServer {

  public static void main(String args[]) throws Exception {
    ...
    // Set the socket factory
    RMISocketFactory.setSocketFactory(new CustomSocketFactory());
    // Create the TimeService object
    TimeService ts = new TimeServiceImpl();
    // If needed, create a RMI registry, using the same factories
    try {
      LocateRegistry.createRegistry(port);
    } catch (RemoteException ex) {
    ...
  }
}
```

… and in the implementation of the socket factory:

```
package rmi.socketFac;
```

```
import java.io.*;
import java.net.*;
import java.rmi.server.*;

public class CustomSocketFactory extends RMISocketFactory {

  private int hash = "TracingSocketFactory".hashCode();

  public ServerSocket createServerSocket(int port)
                                              throws IOException {
    return new CustomServerSocket(port);
  }

  public Socket createSocket(String host, int port)
                                              throws IOException {
    return new CustomSocket(host, port);
  }

  public boolean equals(Object obj) { ... }

  public int hashCode() { ... }
}
```

2.3 Parameter Affecting the Use of Sockets

The following parameters directly affect RMI's use of sockets and configure RMI's use of TCP/IP. These parameters are specified in ms.

(Source: "Java RMI", by W. Grosso, O'Reilly)

sun.rmi.transport.connectionTimeout

Specifies how long a socket will remain dormant before RMI closes it. As long as a socket is "dormant", it can be reuse by RMI. Default: 15'000.

sun.rmi.transport.tcp.readTimeout

Specifies the socket timeout. It controls how long RMI will wait while trying to read a byte before determining that the socket is no longer working. At timeout, RMI generates a `RemoteException` on the client side. Default: 7'200'000.

sun.rmi.transport.proxy.connectTimeout

Determines how long RMI will wait while attempting to establish a socket connection between two JVMs. At timeout, RMI generates a `RemoteException` on the client side. Default: 15'000.

Example:

Setting *connectionTimeout* on client side:

```
java -Dsun.rmi.transport.connectionTimeout=5000 rmi.socketFac.TimeClient
```

3 Object Serialization

In order to be copied (passing by *value*) from one JVM to another, the objects are *serialized*. Object serialization is the name of the process, that saves an object's state in a sequence of bytes onto a stream („serialization") so that the object can be reconstitued from this stream at a later time („deserialization").

3.1 The Serialization Process

Objects are serialized by object output streams and deserialized by object input streams (package *java.io*):

```
public class ObjectOutputStream extends OutputStream
  implements ObjectOutput, ObjectStreamConstants

public class ObjectInputStream extends InputStream
  implements ObjectInput, ObjectStreamConstants
```

Objects possess state. This state is stored in the values of the nonstatic, nontransient fields of an object's class. For example, in the `Point` class we saw before, the object state, i.e. the current object information, is stored in the `x` and `y` fields. The methods only affect the behavior of the object, i.e. the actions it can perform, but do not change what an object is. So if you save the state of an object, you can restore the complete object at a later time, getting this state and the object class file.

Serialization (into a file)

```
Point p = new Point(10, 15);
FileOutputStream fos = new FileOutputStream("Point.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(p);
oos.close();
```

Deserialization

```
FileInputStream fis = new FileInputStream("Point.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
Point p = (Point) ois.readObject();
ois.close();
```

If the object is an *aggregate* (e.g. the object references other objects, as in tables or linked lists), the serialization process saves (and restore) the states of all aggregated objects. Hence, serialization and deserialization is a (rather slow) mean to create a deep copy of an object.

Java does not allow instances of arbitrary classes to be serialized. You can only serialize instances of classes that implement the `java.io.Serializable` (tag-) interface. Note that this rule is also valid for inner classes (it is not enough that the outer class be serializable).

3.2 Non-Serializable Objects

Just because a class *may* be serialized does not mean that it *can* be serialized. You can encounter the following problems:

1. The object is an aggregate, that references non-serializable objects. If any of the objects in the aggregate is not serializable, then the aggregate won't be, either.

   Possible solution: Make non-serializable fields `transient`.

2. A non-serializable superclass has no no-arg constructor. When an object is deserialized, the no-arg

constructor of the closest superclass that does not implement `Serializable` is invoked to establish the state of the object's non-serializable superclass. If that class does not have a no-arg constructor, then the object cannot be deserialized.

Possible solution: Define a no-arg constructor.

3. Class throws `NotSerializableException`. Doing this, it is possible, e.g. for security reason, to prevent the serialisation of a subclass, even if one of its superclasses implements `Serializable`.

3.3 Versioning

When an object is serialized, only the state of the object and the name of the object's class are stored: the byte codes for the object's class are not. Therefore, there is no guarantee that the serialized object will be deserialized using the same class. In a distributed system, the involved classes could very well have changed because they have another version no (jdk1.1, jdk1.2, …). Not all changes, however, prevent deserialization. The following is a list of compatible changes (source: "Java I/O", by Elliotte R. Harold, O'Reilly and the "Java Object Serialization Specification"):

- Most changes to constructors and methods, whether instance or static.
- All changes to static and transient fields. (Serialization ignores them)
- Adding an instance field (in the class being reconstitued from the serialized stream). As the new instance field is not in the stream, it will be initialized to the default value for its type.
- Adding or removing an interface (except `Serializable`)
- Adding or removing inner classes, provided no (nontransient) instance field has the type of the inner class
- Changing the access modifiers of a field. (Serialization does not respect access protection).
- Changing a field from static to nonstatic or transient to nontransient. (Same as adding a field).

The following changes are incompatible and prevent deserialization of serialized objects:

- Changing the name of a class
- Removing an instance field (from the class being reconstitued from the serialized stream).
- Changing the declared type or the name of an instance field
- Changing a field from nonstatic to static or nontransient to transient. (Same as removing a field).
- Moving classes up or down the hierarchy
- Changing the `writeObject()` or `readObject()` method in an incompatible fashion
- Changing a class from `Serializable` to `Externalizable` or vice-versa.

In order for serialization to gracefully detect when a versioning problem has occurred, each serializable class have a *stream unique identifier* SUID. This is a `long` calculated by a special hash function from the following information:

- The class name and modifiers
- The names of any interfaces the class implements
- The descriptions of all nonprivate methods and constructors
- The description of all fields except `private static` and `private transient` fields

The SUID is stored in the class in a static field called `serialVersionUID`, which is used to detect when a class changes. Compatible changes should not change the SUID, whereas incompatible changes will. Comparing the SUID of the serialized class stream with the SUID of the class being reconstitued allows the serialization process to decide if both classes are compatible or not.

However, the default behavior of the SUID calculation is extremely sensitive ("better safe than sorry" strategy) to tell when a class has changed. To allow more flexibility, the programmer may overrides the default calculation by specifying its own SUID in the corresponding class field:

```
private static final long serialVersionUID = 1234;  // User SUID
```

In this case, of course, the programmer is completely responsible to change the SUID, when a (really) imcompatible change has been made to the class.

The jdk provides a utility to know if a class is serializable and to get the SUID. This is the SUID of our `Point` class:

```
F:\rmi\circle2>serialver rmi.circle2.Point
rmi.circle2.Point:    static final long serialVersionUID = -3735228367479308200L;
```

(The –show option starts a simple GUI version of serialver).

3.4 Object References Integrity

As long as objects are serialized to a single stream, the original references between objects are not lost. Serialization does not modify the objects graph (i.e. the existing references between objects) and avoid accidental duplication of objects.

Serializable Class Point

```
package rmi.serial;

import java.io.*;
public class Point implements Serializable {

  private int x, y;

  public Point(int x, int y) {
    this.x = x; this.y = y;
  }

  public static void main(String args[])
       throws IOException, ClassNotFoundException {
    Point p1 = new Point(10, 15);
    Point p2 = new Point(25, 30);
    Point p3 = p1;
    System.out.println("Points - p1:" + p1 + "  p2:" +
                       p2 + "  p3:" + p3);
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    // Serialize three points
    oos.writeObject(p1); // First time
    oos.writeObject(p2);
    oos.writeObject(p3);
    oos.writeObject(p2); // Second time (in another sequence)
    oos.writeObject(p3);
    oos.writeObject(p1);
    // Deserialize the points
```

```
    ObjectInputStream ois = new ObjectInputStream(
            new ByteArrayInputStream(baos.toByteArray()));
    p1 = (Point) ois.readObject();
    p2 = (Point) ois.readObject();
    p3 = (Point) ois.readObject();
    System.out.println("Points1- p1:" + p1 + "  p2:" +
                       p2 + "  p3:" + p3);
    p2 = (Point) ois.readObject();
    p3 = (Point) ois.readObject();
    p1 = (Point) ois.readObject();
    System.out.println("Points2- p1:" + p1 + "  p2:" +
                       p2 + "  p3:" + p3);
  }
}
```

Output:

```
Points - p1:rmi.serial.Point@3fbdb0  p2:rmi.serial.Point@3e86d0  p3:rmi.serial.Point@3fbdb0
Points1- p1:rmi.serial.Point@2dd2dd  p2:rmi.serial.Point@6ee36c  p3:rmi.serial.Point@2dd2dd
Points2- p1:rmi.serial.Point@2dd2dd  p2:rmi.serial.Point@6ee36c  p3:rmi.serial.Point@2dd2dd
```

After deserialization, the objects have not been duplified and `p1` and `p3` still reference the same object.

Note that you have to decide if p1, p2 and p3 after deserialization should be considered the "same" objects as the correponding p1, p2 and p3 before being serialized. If yes, you will have to reimplement the `equals()` (and `hashCode()`) methods.

You can customize this behavior by using the method:

```
public void reset() throws IOException
```

from the class `ObjectOutputStream`.

Reset will disregards the state of any object already written to the stream. The state is reset to be the same as a if the serialization would happen into a new `ObjectOutputStream`. After "reset", the objects previously written to the stream will no more be referred as already being in the stream. Hence, they will eventually be written to the stream again. The current point in the stream is marked as "reset" so the corresponding `ObjectInputStream` will be reset at the same point.

Serializable Class Point Using *reset()*

```
package rmi.serial;

import java.io.*;
public class Point implements Serializable {

  ... dito ...

  public static void main(String args[])
       throws IOException, ClassNotFoundException {

    ... dito ...

    oos.writeObject(p2); // Second time (in another sequence)
    oos.writeObject(p3);
    oos.writeObject(p1);
    oos.reset();
    oos.writeObject(p3); // Third time (in another sequence)
```

```
        oos.writeObject(p2);
        oos.writeObject(p1);

        // Deserialize the points
        ... dito ...

        p2 = (Point) ois.readObject();
        p3 = (Point) ois.readObject();
        p1 = (Point) ois.readObject();
        System.out.println("Points2- p1:" + p1 + "  p2:" +
                           p2 + "  p3:" + p3);
        p3 = (Point) ois.readObject();
        p2 = (Point) ois.readObject();
        p1 = (Point) ois.readObject();
        System.out.println("Points3 - p1:" + p1 + "  p2:" +
                           p2 + "  p3:" + p3);
    }
}
```

Output:

```
Points - p1:rmi.serial.Point@3fbdb0  p2:rmi.serial.Point@3e86d0  p3:rmi.serial.Point@3fbdb0
Points1- p1:rmi.serial.Point@2dd2dd  p2:rmi.serial.Point@6ee36c  p3:rmi.serial.Point@2dd2dd
Points2- p1:rmi.serial.Point@2dd2dd  p2:rmi.serial.Point@6ee36c  p3:rmi.serial.Point@2dd2dd
Points3- p1:rmi.serial.Point@14df86  p2:rmi.serial.Point@5efa1a  p3:rmi.serial.Point@14df86
```

3.5 Customizing Serialization

The simplest way to customize serialization is to declare certain fields *transient*. These fields will not be handled by the serialization process, which means that you exclude certain information from serialization.

For more control over the details of your class's serialization, you can provide custom `readObject()` and `writeObject()` methods.

3.5.1 The `readObject()` and `writeObject()` Methods

The code that serializes objects is built into the JVM and is not part of the `ObjectInputStream` and `ObjectOutputStream` classes. This allows the private data of an object to be read and written.When an object is passed to a `writeObject()` method, the data in the object are written onto the underlying output stream in a specified format. Data is written starting with the highest serializable superclass of the object and continuing down through the hierarchy. However, before the data of each class is written, the JVM checks to see if the class in question has a method with this signature:

```
private void writeObject(ObjectOutputStream out) throws IOException
```

In the same way, before the data of each class is read, the JVM checks to see if the class in question has a method with this signature:

```
private void readObject(ObjectInputStream in)
                            throws IOException, ClassNotFoundException
```

If the appropriate method is present, it is used to serialize/deserialise the fields of this class.

Most of the time, you don't want to change the whole format of an object that is serialized, but only to handle some special cases, perhaps something that is not normally serialized, like a static field. In this

case, you can use `defaultWriteObject()` and `defaultReadObject()` methods. To "handle" a non-serializable object as a `Socket`, you would write (example from "Java I/O", by E. R. Harold, O'Reilly) :

```
public class AClass implements Serializable {
  private transient Socket aSocket;
  // . . . . .
  // Several dozen other fields and methods
  // . . . .

  private void writeObject(ObjectOutputStream out)
                                              throws IOException {
    out.defaultWriteObject();
    out.writeObject(aSocket.getInetAddress());
    out.writeInt(aSocket.getPort());
  }

  private void readObject(ObjectInputStream in)
                     throws IOException, ClassNotFoundException {
    in.defaultReadObject();
    InetAddress ia = (InetAddress) in.readObject();
    int port = in.readInt();
    aSocket = new Socket(ia, port);
  }
}
```

Note that if you are going to use the default mechanism to write the serializable part of your object, you must call `defaultWriteObject()` as the first operation in `writeObject()` and `defaultReadObject()` as the first operation in `readObject()`.

When customized, the serialized form of a class should be correctly documented. Javadoc supports three tags: `@serial`, `@serialField` and `@serialData` for this purpose. Please, check the *Java Object Serialization Specification* document to see how to use these tags.

It may be necessary to customize the serialization process when an object's physical representation differs substantially from its logical data. Logically speaking, the following class represents a sequence of strings. Physically, however, it represents the sequence as a doubly linked list (example from "Effective Java", by J. Bloch, Addison Wesley, slightly corrected).

```
import java.io.*;

public class StringList implements Serializable {
  private int size;
  private Entry head;

  private static class Entry implements Serializable {
    String data;
    Entry  next;
    Entry  previous;
  }

  // Appends the specified string to the list
  public void add(String s) {
    Entry e = new Entry();
    e.data = s;
    e.next = head;
    if (head != null)
```

```
      head.previous = e;
    head = e;
    size++;
  }

  ... // Remainder omitted

}
```

If the default serialization process is applied to this `StringList` class, the serialized object graph will mirror every entry in the linked list and all the links between the entries, in both directions. It will unnecesseraly consume excessive space (entries and links are mere implementation details), consume excessive time (the serialization logic must go through an expensive graph traversal) and, worse, will cause stack overflow errors, due to recursive graph traversal, even for moderatly sized object graphs. Moreover, the `StringList.Entry` class becomes part of the public API.

It would be reasonable to customize the serialization process of `StringList` as follows:

```
import java.io.*;

public class StringList implements Serializable {
  private transient int size;
  private transient Entry head;

  private static class Entry { // No longer Serializable!
    String data;
    Entry next;
    Entry previous;
  }

  // Appends the specified string to the list
  public void add(String s) { ... }

  // Emits the size of the list, followed by all of
  // its elements, in the proper sequence.
  private void writeObject(ObjectOutputStream out)
                                          throws IOException {
    out.writeInt(size);
    Entry e = head;
    if (e != null) {
      // Get the first string added to the list
      while (e.next != null) e = e.next;
      // Write the strings in the list in the proper sequence
      while (e != null) {
        out.writeObject(e.data);
        e = e.previous;
      }
    }
  }

  private void readObject(ObjectInputStream in)
                        throws IOException, ClassNotFoundException {
    int size = in.readInt();
    // Read in all elements and insert them in list
    for (int i = 0; i < size; i++)
      add((String)in.readObject());
  }
```

It could also be necessary to customize the standard serialization process to prevent *incorrect* behavior of an application. Consider, for example, the case of a hash table: which bucket an entry is placed in is a function of the hash code of the key, which is *not*, in general, guaranteed to be the same between two different implementations of a JVM. Therefore, accepting the default serialization could constitute a serious bug.

3.5.2 The `readResolve()` Method

The `readResolve` method is invoked on a newly created object after it is deserialized, but before it is returned to the caller. The object reference returned by this method is then returned in lieu of the newly created object. By implementing the `readResolve` method, a class can directly control the types and instances of its own instances being deserialized.

A `readResolve` method should be provided for all *instance-controlled* classes, i.e., for all classes that strictly control instance creation to maintain some invariant. The following class `Universe` should be instanced only once (singleton):

```
import java.io.*;

public class Universe implements Serializable {

  private static final Universe INSTANCE = new Universe();

  private Universe() {}

  public static Universe getInstance() {
    return INSTANCE;
  }
```

However, you only have to serialize and then deserialize the `Universe` instance to get a second instance of this class! To prevent this, add the following `readResolve()` method to the `Universe` class:

```
  private Object readResolve() throws ObjectStreamException {
    return INSTANCE;
  }
```

Note that `readResolve` does not have to be private. Any access modifier is allowed and will define the accessibility of the method.

3.5.3 The `Externalizable` Interface

If your customization requires you to manipulate values stored for the superclass of an object as well as for the object's class, you should implement the `java.io.Externalizable` interface (which extends the `Serializable` interface). This interface declares the two methods:

```
public void writeExternal(ObjectOutput out) throws IOException
public void readExternal(ObjectInput in)
                    throws IOException, ClassNotFoundException
```

If you declare an object `Externalizable`, the serialization process only rely on the implementation of these methods to execute correctly. These methods are completely responsible for saving the object's state, including the state stored in its superclasses. Since some of the superclass's state may be stored in private or package fields that are not accessible to the externalizable object, saving and restoring can be sometimes tricky. Moreover, no version (`serialVersionUID` field) check is made for externalizable objects.

Example (from "Java I/O", by E. R. Harold, O'Reilly) :

A serializable vector "no matter" what it contains. In this example, if a vector element does not implement `Serializable`, then `writeExternal()` writes `null` in its place

```
package rmi.serial;

import java.io.*;
import java.net.*;
import java.util.*;
public class SerializableVector extends Vector<Object>
                                implements Externalizable {

  public SerializableVector() {
    System.out.println("Serializable Vector created");
  }

  public void writeExternal(ObjectOutput out) throws IOException {
    out.writeInt(capacityIncrement);
    out.writeInt(elementCount);
    for (int i = 0; i < elementCount; i++) {
      if (elementData[i] instanceof Serializable) {
        out.writeObject(elementData[i]);
      } else {
        out.writeObject(null);
      }
    }
  }

  public void readExternal(ObjectInput in) throws IOException,
                                                  ClassNotFoundException {
    capacityIncrement = in.readInt();
    elementCount = in.readInt();
    elementData = new Object[elementCount];
    for (int i = 0; i < elementCount; i++) {
      elementData[i] = in.readObject();
    }
  }
```

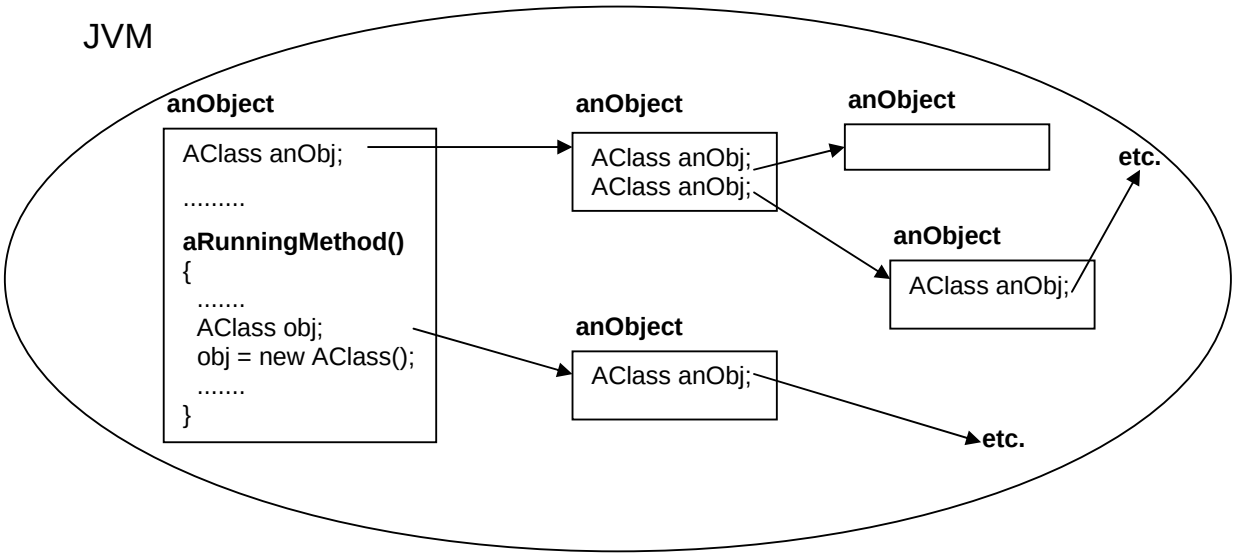
Of course, this is not a perfect general solution. The vector may contain an object that implements `Serializable` but is not serializable: e.g. a hash table that contains a socket.

When an externalizable object is recovered, its *public* default constructor is called. This is different from recovering a serializable object, in which the object is constructed entirely from its stored bits, with no constructor calls. With an externalizable object, all the normal default construction behavior occurs, and *then* `readExternal()` is called.

As for serializable classes, a `readResolve` method can be defined in an externalizable class.

4 Distributed Garbage Collection

4.1 Ordinary Garbage Collection



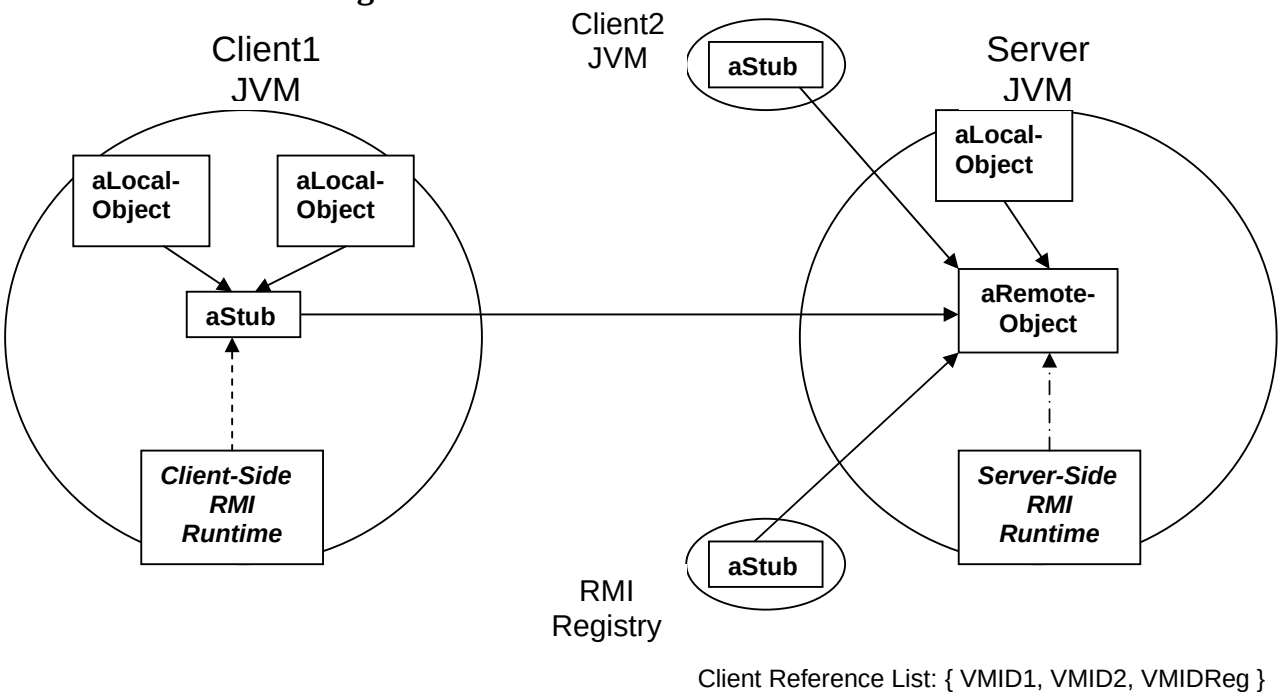
Reachable objects in a JVM:

- Each active thread running in a method, in an instance of an unknown object makes this object reachable ("running object").
- Every object directly referenced by the method-level variables and the instance's fields of each running object is reachable.
- Every object referenced by a reachable object is also reachable.

A garbage collection algorithm mainly does the following:

- It maintains a possibility to find out the set of all currently allocated objects
- It occasionally computes the set of objects that are reachable from the active threads.
- It reclaims (erases) objects that are no more reachable

4.2 Network Garbage



RMI uses a reference-counting garbage collection algorithm and keeps therefore track of all live reference within each JVM. When a live reference enters a JVM, its reference count is incremented and as live references are found to be unreferenced, the count is decremented.

On the client, the stub is forced by RMI to send two additional messages to the server: one message ("referenced") is sent when the stub is instantiated, to let the server know that there is a new active client in the system; another message ("unreferenced") is sent when the stub is freed, to let the server know when the client is finished using the server.

On the server, the RMI runtime simply keeps track of the active clients in a *Client Reference List.* When a remote object is not referenced by any client, it is removed from the RMI distributed garbage collection scheme, thus makes it possible to be reclaimed by the local garbage collector.

The RMI distributed garbage collection algorithm interacts with the local JVM's garbage collector by holding either normal or weak references to objects. A *weak reference* (see `WeakReference` in package `java.lang.ref`) is a reference that does not prevent an object from being garbage collected. That is, if the object referred to has no other references than weak references, an attempt to get the actual reference from weak references will return `null`; otherwise, the valid reference to the object will be returned. The RMI client-side runtime retains weak references to stubs, whereas the RMI server-side runtime refers to a remote object using a weak reference, when this object is not referenced by any client. This allows the client's and server's local garbage collectors to discard the objects, if no other local references exist.

4.3 Leasing

To overcome the problems appearing when a client crashes, or the network is down (i.e. when the messages from client to server can be lost), the client must **lease** the server resources:

1. A client calls the server and requests for a resource (a service) a lease for a period of time.
2. The server grants the lease (not necessarily the required amount of time)

3. During this period of time, the client reference list includes the client (provided the client has not freed the resource before).
4. If the client has not required an extension, the lease expires at the end of the time period and the client is considered as being no more activ. If the resource is not referenced by any other client, it will be garbage collected.

In RMI, the clients automatically try to renew leases as long as a stub has not been garbage collected.

4.4 The Actual Garbage Collector

The `DGC` interface (in package `java.rmi.dgc`) defines the following two methods:

```
public Lease dirty(ObjID[] ids, long seqNum, Lease lease)
public void clean(ObjID[] ids, long seqNum, VMID vmid, boolean strong)
```

Both methods are automatically called by the RMI client's runtime. A `dirty` call is made when a remote reference is instantiated in a client. A corresponding `clean` call is made when no more references to the remote reference exists in the client, thus the client no longer needs a reference to the server.

The method `dirty` requests leases for the remote object references associated with the object identifiers `ids`. The *lease* contains a client's unique virtual machine identifier (VMID) and a requested lease period. If the lease is granted, the garbage collector adds the client's VMID to the object's reference list and the attributed lease value is returned to the client. The same `dirty` method is used to renew a lease.

For other details, please see the API's.

4.5 The `Unreferenced` Interface

This interface contains only the following method:

```
public void unreferenced()
```

This method is called on a remote object that implements `the Unreferenced` interface when its client reference list becomes empty, i.e. when the remote object is not referenced by any client.

However, as long as a local reference to the remote object exists, it can be passed in remote calls or returned to clients. This will add the corresponding client's VMID in the reference list again. Therefore, it is possible for `unreferenced() to` be called more than once.

4.6 Parameter Affecting the Distributed Garbage Collector

(Source: "Java RMI", by W. Grosso, O'Reilly)

java.rmi.dgc.leaseValue (server side)

Set a standard duration for leases granted by a particular server. Specified in [ms]. Default: 600'000.

sun.rmi.dgc.client.gcInterval (client side)

Specifies how often RMI checks to see whether a stub is no longer referenced by the rest of the client application (in order for the client to send `a clean()` message. Specified in [ms]. Default: 60'000.

sun.rmi.dgc.server.gcInterval (server side)

Similar to *sun.rmi.dgc.client.gcInterval*. Controls the server side's refresh rate for distributed

garbage collection, i.e. how often the server examines the consequence of the expired leases and `clean()` messages and attempts to determine whether `unreferenced()` should be called. Specified in [ms]. Default: 60'000.

sun.rmi.dgc.server.checkInterval (server side)

Specifies how often RMI checks for expired leases. Specified in [ms]. Default: 300'000.

sun.rmi.dgc.server.cleanInterval (client side)

Specifies how long the client waits before trying to call `clean()` again, after a previous `clean()` operation failed (e.g. because the network is down). Specified in [ms]. Default: 180'000.

Examples.

Setting *gcInterval* on client side:

```
java -Dsun.rmi.dgc.client.gcInterval=10000 rmi.circle.CircleClient
```

Setting *leaseValue* and *checkInterval* on server side (enter command on one line):

```
java -Djava.rmi.dgc.leaseValue=30000
     -Dsun.rmi.dgc.checkInterval=10000 rmi.circle.CircleServer
```

4.7 RMI Standard Log Facility

You can log the RMI activity on the transport layer by turning on the RMI log facility. E.g.:

```
java -Djava.rmi.server.logCalls=true rmi.circle.CircleServer
```

The logging destination uses per `System.err` per default. You can redirect it by using the `setLog(OutputStream out)` static method of the class `RemoteServer`.

5 Dynamic Class Loading

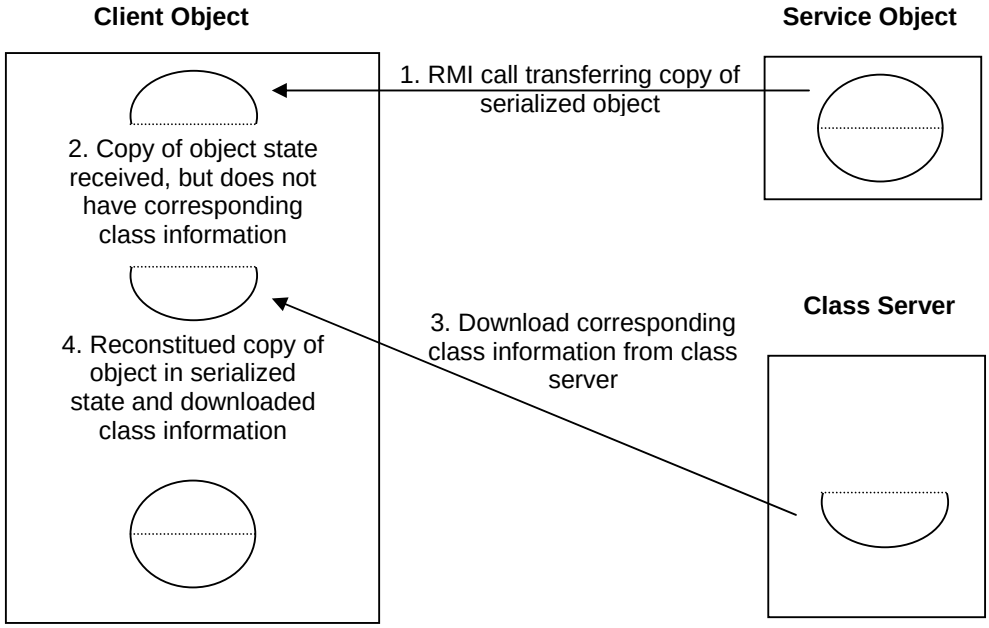
For this chapter, please, look at the SUN tutorial:

http://java.sun.com/javase/6/docs/technotes/guides/rmi/codebase.html

5.1 Definition

A JVM running from a web browser can download the bytecodes for subclasses of `java.applet.Applet` and any other classes needed by that applet. Java RMI takes advantage of this capability by allowing a RMI client or server to work with classes that have never been installed on disk, i.e. with classes that the client/server does not know about ahead of time.

Suppose that a RMI client is calling a remote method provided by a service object, and suppose that, during the course of this calling process, a serialized object is passed from the service to the client object. On the client side, this object must be reinstantiated and, for that, needs the corresponding class(es). It is not mandatory, however, for these classes to be both (redondanly) installed on the client and on the server: they can be dynamically loaded across the network when needed.



When a serialized object is transmitted, only the object's state information is actually passed, the class information is not. When needed, the class is dynamically loaded via a remote class server. It means that the client object (in this case) has no idea what it will be working with ahead of time: hence, RMI is sometimes said to allow *behavioral transfer*. Of course, the client object must have a possibility to know, where the class information has been stored: for this, RMI uses the *codebase* concept during the *object marshalling*.

5.2 Object Marshalling

Marshalling refers to the action of serializing objects onto a serial medium, involving the serialization of both state and type (class) information. Reconstituating the object from the serial medium is called *unmarshalling*. To ship objects around, RMI stores during the marshalling phase, all the necessary information into a *container* object, which is an instance of the `MarshalledObject` class. Such a `MarshalledObject` instance contains:

1. A codebase, that is a URL (or a list thereof), which specifies a global location for a Java class file
2. A stream of data bytes that represents the serialized object to transfer

The codebase URL of the `MarshalledObject` contains the location in the network where the class file is to be loaded from: hence, it is also called *class annotation.* At end of transfer, when the unmarshalling code finds this codebase URL, it is able to obtain the class code over the network before deserializing the object data and reinstantiating the object.

A RMI application may have to load class files dynamically for three categories of objects:

- Objects that implement an interface that is declared as parameter or return type of a remote method
- Objects of a subclass of a class that is declared as parameter or return type of a remote method
- Stubs with pregenerated class (and any other classes they need)

The marshalling layer receives the class annotation from the `java.rmi.server.codebase` system property, that can be set via the `-D` switch when starting the JVM. However, for any stub object that is serviced by the RMI registry (or any object that the stub relies on), the marshalling layer will create a class annotation if and only if the RMI registry cannot find the class(es) in its `CLASSPATH`. If the RMI registry can find a stub class (or any relying class), it will not remember that the class(es) can be downloaded from your server's codebase and therefore will not convey to clients the corresponding class annotation (the annotation will be *null*). Therefore:

> **When starting the RMI registry, make sure the `CLASSPATH` does not include any (pregenerated) stub class, or other classes that the stub depends on, that have to be dynamically and remotely loaded.**

5.3 Delivering Dynamically Loaded Classes

When the client has read the codebase URL (class annotation) that is inside the marshalled object, the class will be downloaded by using the `RMIClassLoader`. This class loader supports the *file*, the *http* and the *ftp* protocols.

The *file* protocol is of limited utility since it assumes that both the server and the client have the same mapping to a shared file system. While the third alternative is also interesting, it is the *http* protocol that is most frequently used. Therefore, in order to allow a client to download the class from an `http://...` URL, you must have an HTTP server serving the class requests over the network.

Given this, the `RMIClassLoader` takes a codebase URL of the form *http://hostName:port/path/* and a classname and does the following:

1. It creates a path from the classname by interpreting each package as a directory.
2. It prepends the *path* from the URL, to the path it just created in order to form a request path.
3. It then issues an HTTP GET request to a web server running on *port* of *hostName*, in the form:

```
GET request-path HTTP/1.1
```

Example:

- Classname: `rmi.time.TimeServiceImpl`
- Codebase URL: `http://DOJ.cablecom.ch:8080/server/`

  → Request-path:

  `http://DOJ.cablecom.ch:8080/server/rmi/time/TimeServiceImpl.class`

You also can give a *jar* archive file where the class can be found:

Example:

- Codebase URL: `http://DOJ.cablecom.ch:8080/server/Time.jar`
- → Request-path: `http://DOJ.cablecom.ch:8080/server/Time.jar`

(Note: If the codebase URL does not end with a `/`, it is assumed to be complete and to describe the location of a *jar* file).

In both cases, the root of the request-path is the one of the web server.

The class downloading process requires that an appropriate *Security Manager* be defined in the application. We use here the default `RMISecurityManager`. From Java 1.2 on, the security manager requires a `Policy` object that relies on a *policy file* that determines the rights granted to the application. We use here the following user policy file that grants *unlimited access* to the application.

```
grant {
  permission java.security.AllPermission;
};
```

The user policy file has the default location:

`<user.home>\.java.policy` (windows)

`<user.home>/.java.policy` (solaris)

However, this default can be overriden with the `-Djava.security.policy=<policy_file>` switch when starting the JVM.

5.4 A RMI Example with Dynamic Class Loading

We modify our distributed *TimeService* example (see chap. RMI Basics) so that the time value is returned to the client within a *TimeHolder* object, instead as a string. Moreover, the *TimeHolder* class implements the *Time* interface:

Time Interface (new)

```
package rmi.dynTime;

import java.io.*;

public interface Time extends Serializable {
  String getValue();
}
```

Time Implementation (new)

```
package rmi.dynTime;

import java.util.*;

public class TimeHolder implements Time {
  private Date currentTime;

  public TimeHolder(Date currentTime) {
```

```
    this.currentTime = currentTime;
  }

  public String getValue() {
    return "Server time: " + currentTime;
  }
}
```

Service

```
package rmi.dynTime;

import java.rmi.*;

public interface TimeService extends Remote {
  String SERVICE_NAME = "TimeService";
  Time getTime() throws RemoteException;
}
```

Service Implementation

```
package rmi.dynTime;

import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class TimeServiceImpl extends UnicastRemoteObject
                             implements TimeService {

  public TimeServiceImpl() throws RemoteException {
    System.out.println("TimeService object created");
  }

  public Time getTime() {
    // Get time and return time holder object
    Date currentTime = Calendar.getInstance().getTime();
    System.out.println("Current Server time: " + currentTime);
    return new TimeHolder(currentTime);
  }
}
```

Server

No change (except package).

Client

```
package rmi.dynTime;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

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

```
      ... dito ...

      // Set security manager to allow dynamic class loading
      System.setSecurityManager(new RMISecurityManager());

      try {
        TimeService ts =
          (TimeService) Naming.lookup("//" + host + ":" + port + "/" +
                                      TimeService.SERVICE_NAME);
        Time t = ts.getTime(); // Get the Time object
        System.out.println(t.getValue()); // Output time value
      } catch ... dito ...
    }
  }
```

To get the time, the client must download the `TimeHolder` class, in order for the *Time* object to be instantiated on the client side. This requires a *Security Manager* to be set. We use the default *RMISecurityManager*.

You can check the codebase of a downloaded class with the *RMIClassLoader*'s static method *getClassAnnotation(Class cl)*:

```
String annotation =
        java.rmi.server.RMIClassLoader.getClassAnnotation(t.getClass());
System.out.println("Class annotation of TimeHolder object:" + annotation);
```

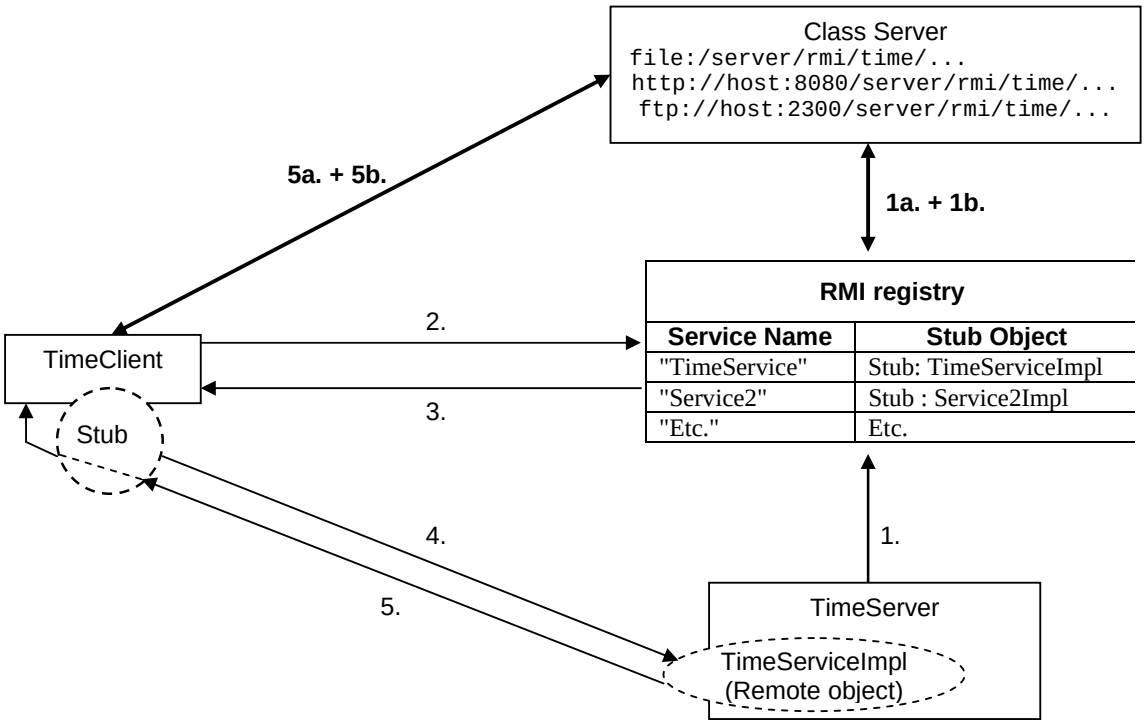
5.4.1 Data Flow

The application classes are deployed between the hosts, so that each one only contain the classes that are necessary for the compilation:

ServerHost	ClientHost
`Time`	`Time`
`TimeHolder`	`TimeService`
`TimeService`	`TimeClient`
`TimeServiceImpl`	
`TimeServer`	

We assume here that no pregenerated stub class exists and that the *rmiregistry* has no classes relying to the stub in its classpath.

(Compare this figure with the corresponding one in chapter "RMI Basics")



1. The remote object's stub is registered into the RMI registry

1a. The registry requests the classes relying to the stub object (*TimeService* and *Time*) from the codebase

1b. The corresponding server (http:, ftp:) or the file system returns the requested classes

2. The client makes a lookup call
3. The RMI registry returns an instance of the remote object's stub to the client
4. The client makes a remote call
5. The remote object returns a *Time* object to the client

5a. The client requests the *TimeHolder* class from the codebase

5b. The corresponding server (http:, ftp:) or the file system returns the requested class

5.4.2 Running the Application

We assume:

1. On the *ServerHost*, the class files are stored in: `F:\classes\server\rmi\dynTime\`
2. On the *ClientHost*, the class files are stored in: `F:\classes\client\rmi\dynTime\`
3. The root directory for the HTTP class server is: `F:\classes`
4. The security policy file (on the *ClientHost*) is: `F:\security\java.policy`

Running the *TimeServer* on the *ServerHost*

1. Start the *RMI registry* (outside CLASSPATH):

```
rmiregistry
```

2. Start a HTTP class server (e.g. on port 8080, given as command line parameter).

```
java HttpClassServer 8080
```

3. Start the *TimeServer*: The codebase annotation will allow the client-side RMI classloader to request the stub class from the class server:

```
java –Djava.rmi.server.codebase=http://ServerHost:8080/server/ rmi.dynTime.TimeServer
```

Running the *TimeClient* on the *ClientHost*

Start the *TimeClient*: The security policy file is checked by the security manager before requesting the class to be downloaded:

```
java –Djava.security.policy=F:\security\java.policy rmi.dynTime.TimeClient ServerHost
```

5.4.3 Miscellaneous

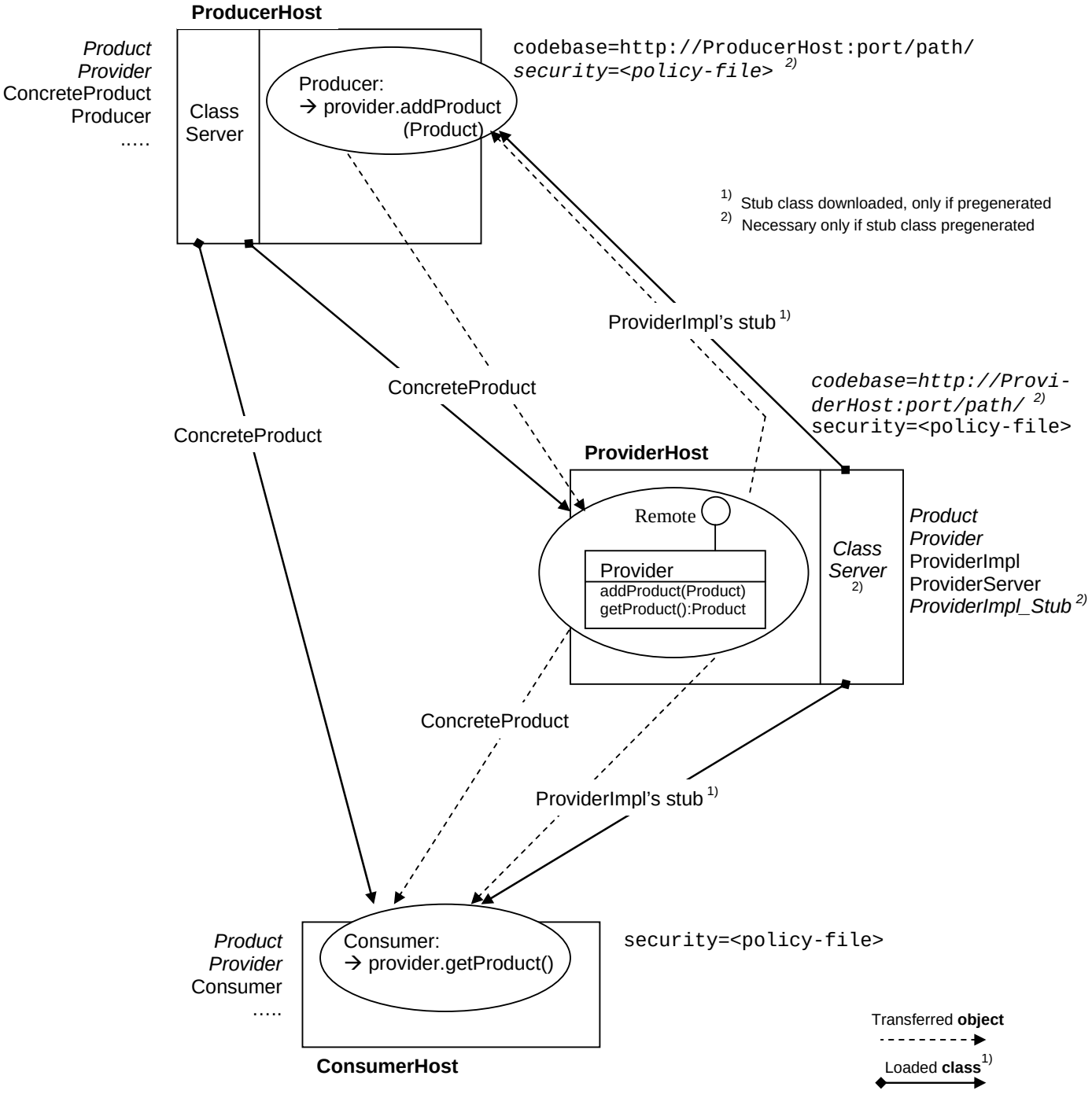
Codebase Annotation

You are not restricted to define a single URL as codebase annotation. RMI supports a space-separated sequence of URLs, which will be sucessively send to the corresponding class server(s) until the class is found.

Downloading and Uploading

Dynamic classloading may be needed by a client ("download") or a server ("upload"). To RMI, this makes no difference. Wherever you need to dynamically load a class, you have to set a security manager, with a policy file and you have to bring a class annotation within the marshalled object, in order for RMI to know, where the class is to load from. If a server or a client both needs to get and deliver classes, you have to instantiate a security manager and to enter both the `java.rmi.server.codebase` and the `java.security.policy` options, when you run the application.

Deployment Example:



Disabling Dynamic Classloading

To prevent RMI to activate dynamic classloading (e.g. for security purpose), you have two ways:

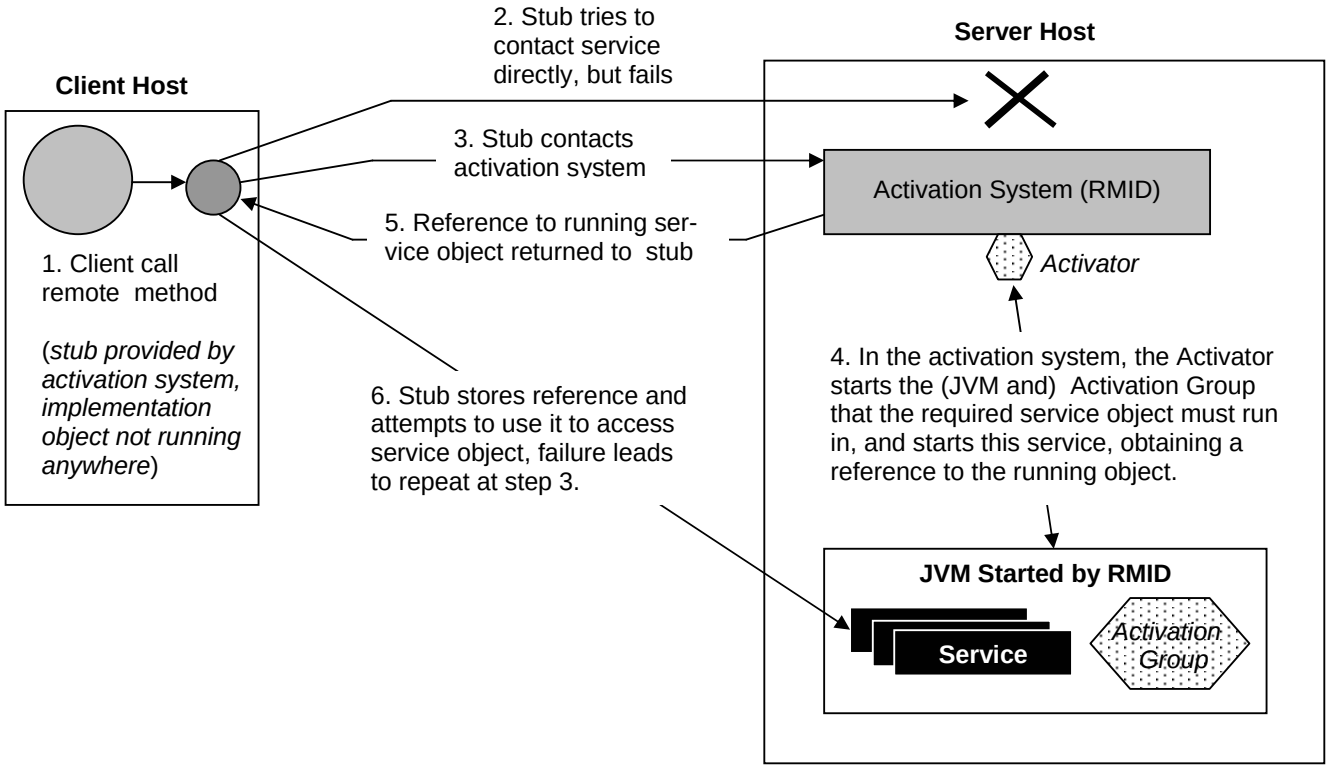
- Do not install a security manager
- Set the system property `java.rmi.server.useCodebaseOnly` to *true*. In this case, RMI will only load classes from the local file system, regardless of whether the class has a codebase annotation.

6 Remote Object Activation

6.1 Overview

Object activation is a mechanism for providing persistent references to objects and managing the execution of object implementation. Activatable objects make it possible to create object references to remote objects that appear to be „immortal“. RMI accomplishes this by implementing an „Activation System“, which supports:

- A way to re-instantiate a server-side implementation object if the system resets or crashes
- A way to maintain the state of the server-side implementation in a persistent way that can survive software failures and system reboots.



The activation system is a RMI support service called RMI Daemon (RMID). RMID must be restarted when the system is reset or rebooted. It will then re-create all instances of JVMs and server objects that had registered with it as `Activatable`, and do so inside one or more JVM as determined by the grouping of `Activatable` objects into `ActivationGroup`s.

6.2 The Activation Protocol

On the client side, the activation works by using “smarter” stubs, which do not only know about the remote object, but also have enough information to contact an instance of an activation daemon somewhere and, if needed, engage the reactivation of the remote object.

The reactivation of objects is defined by the *activation protocol*, which involves several entities: the *activator*, an *activation group* and the remote object being activated. It corresponds to the steps 3 to 5 of the above figure and is as follows:

1. The stub contacts the activation system by calling the `activate` method of the RMID **activator** instance, which will supervise the activation process and finally return a stub (in "marshalled form") of the (reactivated) remote object.

   ```
   public MarshalledObject activate(ActivationID id, ...)
   ```

   `activate` receives mainly the object's activation ID, which is a unique identifier for the object to be activated. This activation ID has been obtained previously when the object was registered with the activation system (see the API documentation for the `Activator` interface and the `ActivationID` class).

2. The *activator* looks up the **activation descriptor** corresponding to the objects activation ID.

   An activation descriptor contains the information necessary to activate an object, i.e. mainly:

   - The object's class name
   - A codebase URL path from where the object's code can be loaded
   - A `MarshalledObject` containing object-specific initialization data needed for the activation
   - The object's *activation group ID*, which identifies an activation group uniquely within the activation system.

3. If necessary, the *activator* initiates the execution of activation groups.

   An **activation group** (one per JVM) is associated with a related set of "activatable" objects and is responsible for creating new instances of these objects in its group. All objects belonging to the same activation group will run in the same JVM and will share system properties and environment variables (e.g. the security file).

   If an activation group in which an object must be activated is not already executing, the *activator* will start a child JVM for this activation group. It will use for that the information contained in an **activation group descriptor**, which describes the child JVM and contains the information necessary to (re-)create the group and starts its associated objects. This information includes the group's class name, codebase path and initialization data, so as the JVM system properties and command environment (only non default information must be specified).

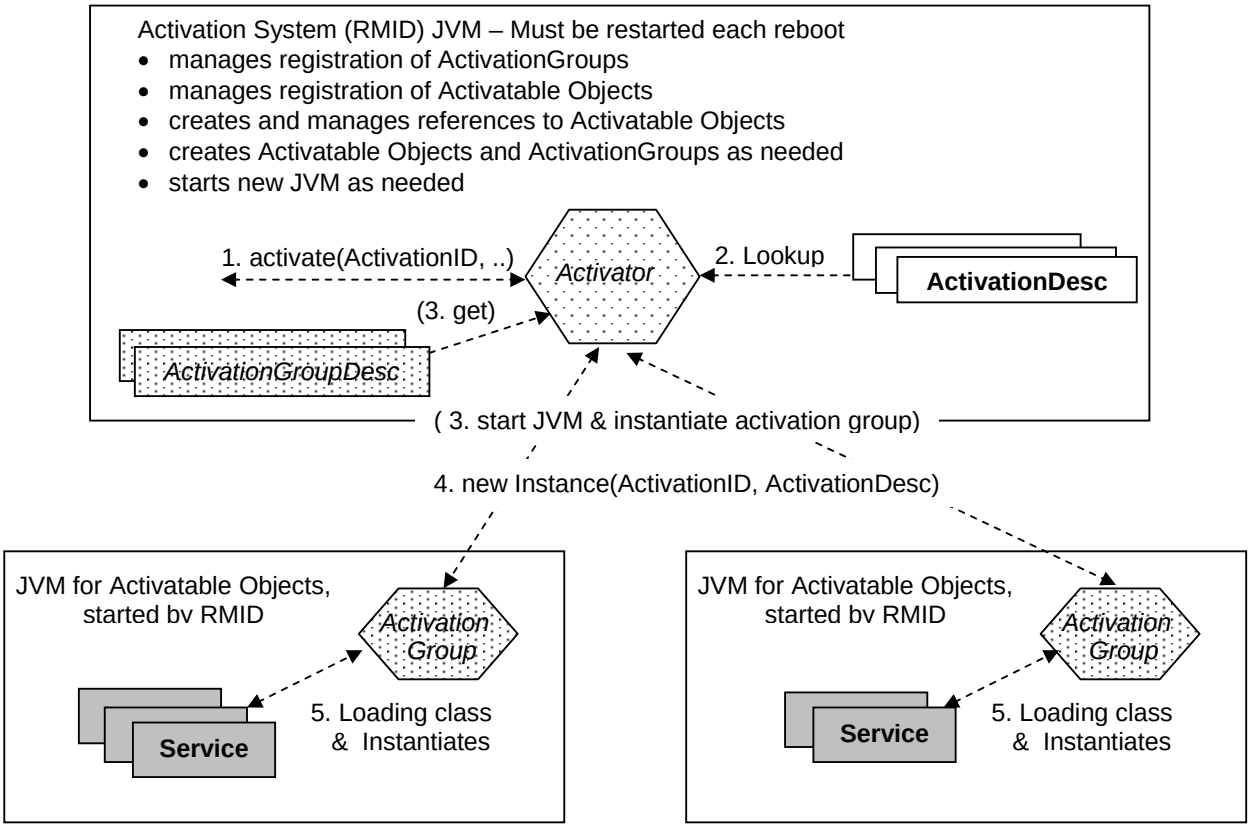
4. If (or once) the activation group is executing, the *activator* calls the activations group's new`Instance` method, passing to it the object's activation ID and activation descriptor, in order to activate the corresponding object.

5. The object will be activated. The activation's steps are:

   - determining the class for the object using the object descriptor's `getClassName` method,
   - loading the class from the URL path obtained from the descriptor (`getLocation` method),
   - creating an instance of the class by invoking a special constructor of the object's class that takes two arguments: the object's `ActivationID` and a *MarshalledObject* containing the object's initialization data, and
   - returning a serialized version of the remote object just created to the *activator* (a `MarshalledObject` instance).

6. The *activator* returns an active remote object's reference to the stub, which then forwards method invocations via this active reference directly to the remote object.

All classes involved in the activation process belong to the package `java.rmi.activation`.



6.3 Implementing an Activatable Object

To implement a remote object that requires persistent access over time and that can be activated by the system (i.e. an "activatable object"), the developers have to do two things: adapt the service code to make it activatable and produces a “launch code”, also called *object’s setup*, to define the object’s properties to the activation system.

As example, we use again our simple Time Service (see chap. „RMI Basics“):

We assume (on *MyHost)*:

1. The service and setup class files are stored in: `F:\classes\server\rmi\actTime\`
2. The client class files are stored in: `F:\classes\client\rmi\actTime\`
3. The security files are: `F:\security\java.policy`
   `F:\security\rmid.policy`
4. The codebase of the service class files is specified with a `file:/` URL (and not a `http://` one: we focus here on object activation, not on dynamic class loading).

6.3.1 Making a Service into an Activatable Object

During the activation process, an activatable object is (re)instantiated by the `newInstance` method of the object's activation group. This method invokes an object constructor with the following signature:

```
MyActivatableService(ActivationID id, MarshalledObject data)
```

Therefore, any activatable object must contain such an "activation" two-arguments constructor, in order for the Activation Framework to work. This constructor takes two arguments:

- The `ActivationID` is a globally unique identifier that contains the information needed by RMID for activating the object, i.e. a remote reference to the activator and a unique identifier for the object. The `ActivationID` is obtained when the object is registered with the activation system.
- The `MarshalledObject` allows to pass data to activatable services. This data is created by the object setup code, serialized, and stored into a `MarshalledObject` instance. The `MarshalledObject` is then passed into the Activation Framework. which stores it and passes to the service's "activation" constructor when the service is activated. The service is responsible for extracting the data from the `MarshalledObject`, if appropriate, using the method `get()`.

The main function of the "activation" constructor is to export the object to the RMI run-time system. The easiest way to do this is to make the object extend from the `Activatable` (abstract) class, which extends `RemoteServer` and can be used in a way that's exactly parallel to `UnicastRemoteObject`. The object's "activation" constructor only needs to call the correponding constructor of the `Activatable` class:

```
protected Activatable(ActivationID id, int port)
                                        throws java.rmi.RemoteException
```

which will export the activatable object on the specified *port* (anonymous, if *port* = 0) by invoking the corresponding `exportObject` method.

Apart from that, the service code remains unchanged.

The Activatable TimeService

```
package rmi.actTime;

import java.rmi.activation.*;
import java.rmi.*;
import java.util.*;

public class TimeServiceImpl extends Activatable
                            implements TimeService {

  // "Activation" constructor.
  public TimeServiceImpl(ActivationID id, MarshalledObject data)
                                          throws RemoteException {

    // Export the object an anonymous (0) port
    super(id, 0);
  }

  public String getTime(){
    // Get and transmit time
    Date currentTime = Calendar.getInstance().getTime();
    return "Server time: " + currentTime;
  }
}
```

6.3.2 Object's Setup Code

The job of the *Setup* code is to specify to the activation system all the information necessary to launch an activatable service, without necesseraly create an instance of the service. Typically, the *Setup* code

specifies the activation information, passes it to RMID, registers a remote reference for the service in the RMI registry and exits.

To tell RMID how to launch (a) service(s), we have to:

1. Describe a JVM, where the service(s) will run, i.e. creating an instance of `ActivationGroup`.
2. Describe the service object(s), i.e. creating an instance of `ActivationDesc`.

Part 1: Create an instance of `ActivationGroup`

We have to:

1. Create an activation group descriptor, i.e. an instance of `ActivationGroupDesc`, in order to provide all the information that RMID will require to (activate) and contact the appropriate JVM for the activatable service.

   This descriptor must contain the JVM properties and a possible command environment, that controls the exact command and options used when the JVM is started by RMID.
2. Register the activation group specified by the group descriptor with the activation system in order to obtain the activation group ID.

Part 2: Create an instance of `ActivationDesc`

The job of the activation descriptor is to provide all the information that RMID requires to create an instance of the service implementation. We have to:

1. Specify the service's class name and codebase URL.
2. Possibly specify `MarshalledObject` containing data for the service object.
3. Create the activation descriptor, passing the previous information to it.
4. Register the activation descriptor with the activation system, getting the `Remote` stub for the activatable object as return value.

A Setup Code for the TimeService

```
package rmi.actTime;

import java.rmi.*;
import java.rmi.activation.*;
import java.util.*;

public class Setup {

  public static void main(String[] args) throws ActivationException,
                   RemoteException, java.net.MalformedURLException {

    // Part 1: Create an ActivationGroup instance.
    // ------
    // As single property, we specify a security policy (1.)
    Properties props = new Properties();
    props.setProperty("java.security.policy",
                      "F:/security/java.policy");

    // No special command environment to start the JVM: use
    // RMID's default (1.)
    ActivationGroupDesc.CommandEnvironment cmd = null;
```

```
      // Create the activation group descriptor (1.)
      ActivationGroupDesc agDesc = new ActivationGroupDesc(props, cmd);

      // Register the activation group with the activation system (2.)
      ActivationGroupID agId =
        ActivationGroup.getSystem().registerGroup(agDesc);

      // Part 2: Create service ActivationDesc instance
      // ------
      // Set the name and location Strings defining the class URL (1.)
      String className = "rmi.actTime.TimeServiceImpl";
      String classLocation = "file:///F:/classes/server/";

      // No data is passed to the service (2.)
      MarshalledObject data = null;

      // Create the service activation descriptor (3.)
      ActivationDesc aDesc =
        new ActivationDesc(agId, className, classLocation, data);

      // Register the activation descriptor with the
      // activation system (4.)
      TimeService ts = (TimeService)Activatable.register(aDesc);

      // Bind the stub to a name in the RMI registry.
      Naming.rebind(TimeService.SERVICE_NAME, ts);

      // Quit the setup application.
      System.exit(0);
    }
  }
```

6.3.3 Compile and Run the Activatable Service

Do the follwing steps:

- Compile the classes `TimeService, TimeServiceImpl and Setup` as usual.
- Start the *RMI registry*
- Start the activation daemon, `rmid`

  ```
  rmid –J-Djava.security.policy=F:\security\rmid.policy
  ```

  where `rmid.policy` is the name of the security policyfile for `rmid.` This policy file is used to verify whether or not the information in each activation group descriptor is allowed to be used to launch a JVM for an activation group.
- Run the setup program. The setup program normally binds the service stub into the RMI registry. Therefore, the registry must be able to find the interfaces (`TimeService`) and classes, which are necessary to instantiate the stub. These can be either on the registry's `CLASSPATH` (in which case the stub class annotation will be `null`), or their location must be specified in a system codebase property. For example:

```
java -Djava.rmi.server.codebase=file:///F:/classes/server/ rmi.actTime.Setup
```

- Run the client

Since an `Activatable` object is a purely server-side implementation detail, there is nothing special to be specified within the client. For it, the activation is totally transparent.

Client Code

```
package rmi.actTime;

import java.rmi.*;

public class TimeClient {

  public static void main(String[] args) throws Exception {
    String host = args[0]; // Assume host name in args[0]
    int port = 1099;
    // Set a security manager (to load the stub class)
    System.setSecurityManager(new RMISecurityManager());
    TimeService ts =
      (TimeService) Naming.lookup("//" + host + ":" + port + "/" +
                                  TimeService.SERVICE_NAME);
    System.out.println(ts.getTime());
  }
}
```

Run the client with (on one line):

```
java -Djava.security.policy= F:\security\java.policy
     rmi.actTime.TimeClient MyHost
```

6.3.4 The RMID Registry

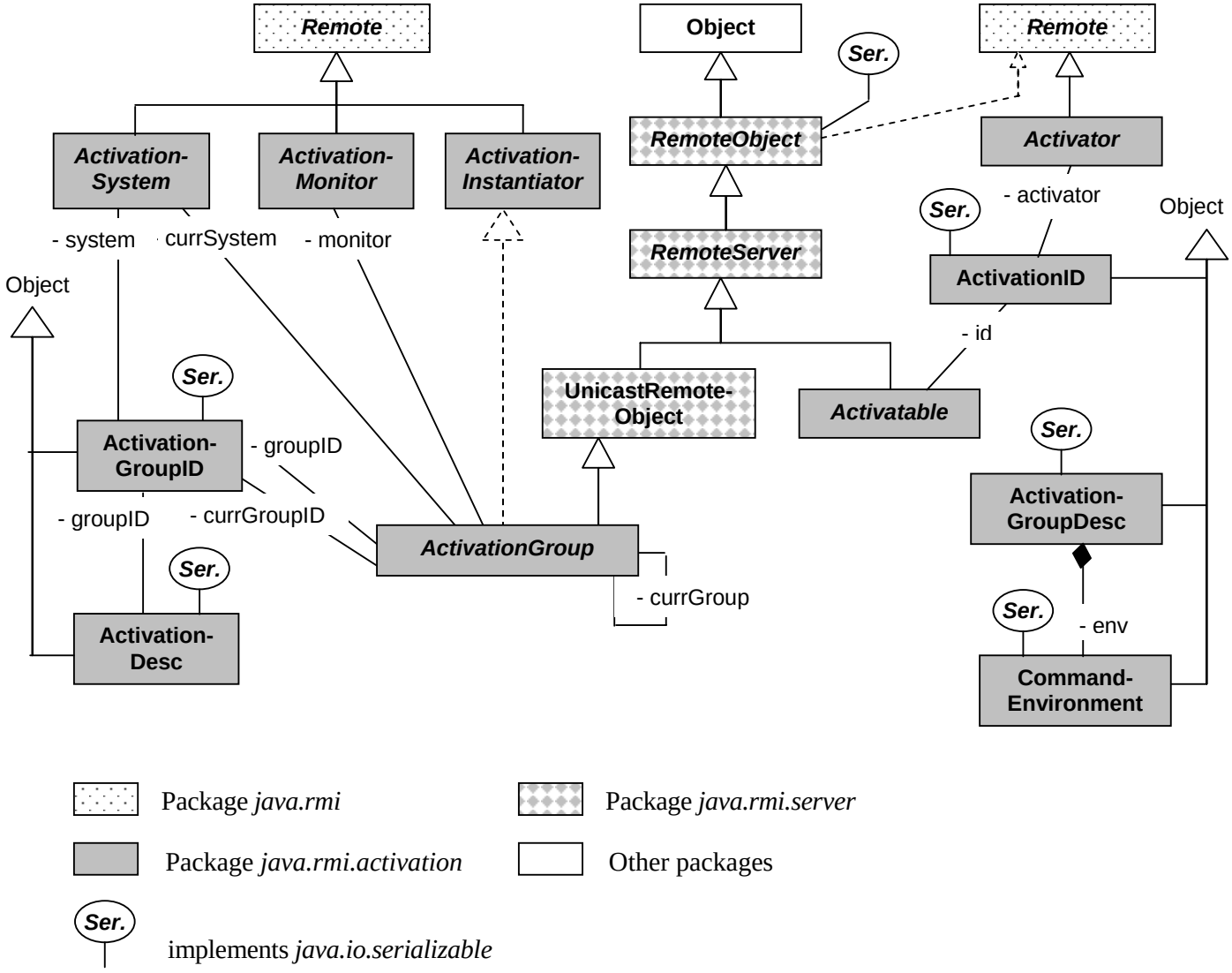
The activation daemon has a registry to enable clients to connect with it. It is created by default on port 1098, using `LocateRegistry.createRegistry()`, when you run RMID. The default port can be changed (see chap. “RMID command line arguments”).

This registry only contains a reference (stub) to the `ActivationSystem` remote object, giving you the possibility to communicate directly with the activation system:

```
ActivationSystem system = (ActivationSystem)Naming.lookup(″//:″ +
                    rmidPort + ″/java.rmi.activation.ActivationSystem″);
```

The RMID registry is a fully functional registry, which you can use instead of the *RMI registry* to store any remote service references.

6.4 RMI Activation Classes



6.5 Details and Activation Customization

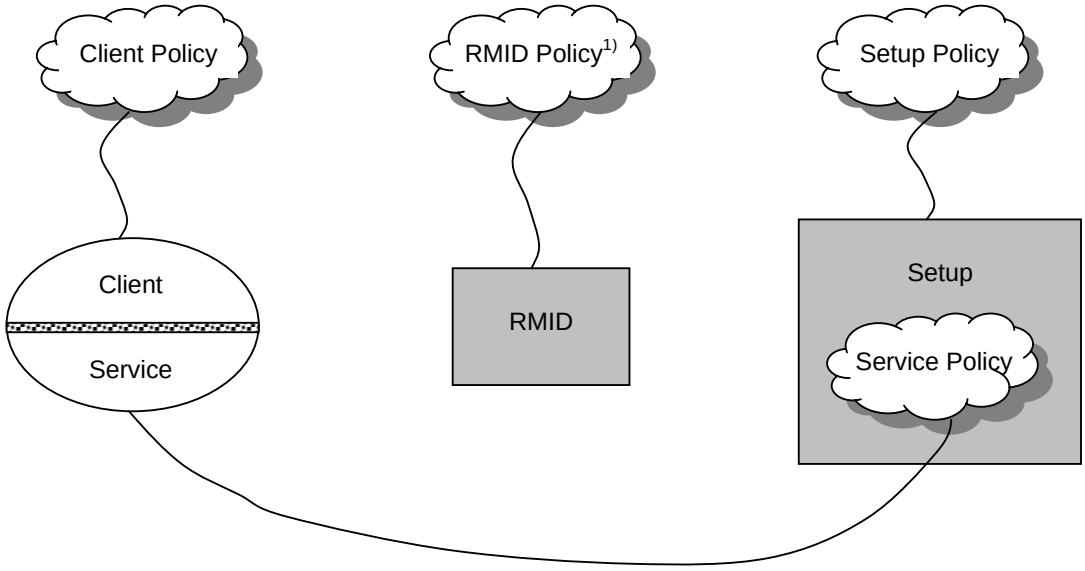
6.5.1 Codebase and Object Behavior

Since dynamic stub downloading and behavior transfer both depend on the `java.rmi.codebase` system property of the service JVM, you must be careful to understand the different JVMs that are involved in the activation process.

- The JVM in which the setup program runs is only alive during the lifetime of the setup program. Setting any environment or system properties here will not affect the service object's behavior.
- The JVM that hosts the activation group where a particular service object will be activated is the one whose system properties and environment will affect the service object's behavior. To control this, you have to configure these parameters when creating the activation group.

6.5.2 Policy Files

A standard application using the RMI Activation Framework may need four policy files:



1) In the Sun's implementation of RMID, by default you need to specify a security policy file, so that `rmid` can verify whether or not the information in each `ActivationGroupDesc` is allowed to be used to launch a JVM for an activation group.

6.5.3 Activating an Object That Does Not Extend `Activatable`

You can build activatable objects that do not extend `Activatable`. For this, update the service implementation and use the static `exportObject` method of the `Activatable` class in the object's constructor:

An Activatable TimeService

```
public class TimeServiceImpl implements TimeService { // No extend

  public TimeServiceImpl(ActivationID id, MarshalledObject data)
                                          throws RemoteException {

    // Export the object an anonymous (0) port
    Activatable.exportObject(this, id, 0);
    System.out.println("TimeService object activated");
  }
```

6.5.4 Service Activation at RMID Restart

Standard RMI systems use *lazy activation*, which means that a service activation is deferred until a client makes a first invocation of the service's method. However, it is possible to force a service to be restarted automatically when the activator is restarted. For this, you have to create the service's activation descriptor by using the following constructor and setting the *restart* parameter to *true*:

```
public ActivationDesc(String className, String location,
```

```
                                java.rmi.MarshalledObject data, boolean restart)
```

6.5.5 Activating a Service During Initial Construction

The *Setup* code allows to specify and register service's information without creating a service's instance. However, if a service's instance must already be activated during initial construction, you can instantiate it directly (e.g. in a *TimeServiceImpl* class) by calling the following constructor of the `Activatable` class (commented code excerpt):

***Activatable* Constructor**

```
protected Activatable(String codebase, MarshalledObject data,
                      boolean restart, in port)
                      throws ActivationException, RemoteException {
  super();
  exportObject(this, codebase, data, restart, port);
}
```

Method *exportObject*

```
public static ActivationID exportObject(Remote obj, String codebase,
                   MarshalledObject data, boolean restart, int port)
                        throws ActivationException, RemoteException {
  // Create an activation descriptor for the activatable object
  ActivationDesc desc = new ActivationDesc(obj.getClass().getName(),
                                           codebase, data,restart);

  // Register the object with the activation system to get an id.
  ActivationID id = ActivationGroup.getSystem().registerObject(desc);


  // Export the object to the RMI runtime system
  Remote stub = exportObject(obj, id, port);

  // Inform the activation group monitor, that the object is active
  ActivationGroup.currentGroup().activeObject(id, obj);

  return id;
}
```

Note that if your class does not extend `Activatable` you can invoke `exportObject` directly.

To activate an object during initial construction, you have to create and set the corresponding activation group for the current JVM, in order for the group to obtain a group monitor. (In the "standard" activation protocol, this action is carried out by the `Activator` when it initiates the re-creation of an activation group in order to carry out incoming `activate` requests). For that, you have to call in the setup code the `ActivationGroup.createGroup` method, after registering the activation group with the activation system.

Setup Code to Activate a Service at Initial Construction (excerpt)

```
  // Part 1: Create an activation group
  // ------
  props = ...;
  cmd = ...;
  // Create the activation group descriptor
```

```
  ActivationGroupDesc agDesc = new ActivationGroupDesc(props, cmd);
  // Register the activation group with the activation system
  ActivationGroupID agId =
    ActivationGroup.getSystem().registerGroup(agDesc);

  // Create and set the group for the current JVM
  ActivationGroup.createGroup(agId, agDesc, 0);

  // Part 2: Create and activate the service
  // ------
  String classLocation = ...;
  MarshalledObject data = ...;

  // Instantiate the (time) service
  TimeService ts = new TimeServiceImpl(classLocation, data);

  // Bind the stub to a name in the RMI registry
  .....
  (No exit)
```

`createGroup` sets the activation group as *current (default) group.* Instances of `ActivationDesc` will be automatically associated with this group (note that the `Activatable.exportObject` method gets no group id to be transmitted to the `ActivationDesc` constructor).

To create and set the activation group with *createGroup*, you need to specify a security policy file (default security manager is `RMISecurityManager`):

```
java  -Djava.rmi.server.codebase=file:///F:/classes/server/
      -Djava.security.policy=F:/security/java.policy rmi.actTime.Setup
```

Service Constructors:

```
public class TimeServiceImpl extends Activatable
                             implements TimeService {

  // Constructor called at initial construction
  public TimeServiceImpl(String codebase, MarshalledObject data)
                       throws ActivationException, RemoteException {

    boolean restart = ...;
    super(codebase, data, restart, 0);
  }

  // "Activation" constructor
  public TimeServiceImpl(ActivationID id, MarshalledObject data)
                                            throws RemoteException {
    super(id, 0);
  }
  ...
```

6.6 Shutting Down an Activatable Service

To shut down an activatable service, you have to do two things:

1. Unexport the service. It will tell RMI runtime that the service is no longer accepting remote

method invocation. The RMI runtime will discard all references to the service, so that it will be eligible for garbage collection. Use for this the following method of the `Activatable` class:

```
public static boolean unexportObject(Remote obj, boolean force)
                                        throws NoSuchObjectException
```

This method makes the remote object, *obj*, unavailable for incoming calls, as long as it has not been re-exported with one of the `exportObject` method. If *force* is true, the object is unexported, even if there are pending calls to the remote object or the remote object still has calls in progress.

2. Tell RMID, that the service is no longer active. This lets RMID reset its records so that, the next time a client tries to access the service, the service will be launched again. Use for this the following method of the `Activatable` class:

```
public static boolean inactive(ActivationID id)
   throws UnknownObjectException, ActivationException, RemoteException
```

In addition, you can revoke the previous service registration to completely suppress a service, so it won't be reactivated. Use for this the following method of the `Activatable` class:

```
public static void unregister(ActivationID id)
   throws UnknownObjectException, ActivationException, RemoteException
```

A typical code sequence that shuts down a service could be:

```
if (Activatable.unexport(service, true) {
  Activatable.inactive(service.getID());
}
```

or (f the service should not be later activated):

```
if (Activatable.unexport(service, true) {
  ActivationID serviceId = service.getID();
  Activatable.inactive(serviceId);
  Activatable.unregister(serviceId);
}
```

6.7 The Activatable Class

The `java.rmi.activation.Activatable` class is the main API that developers need to use to implement and manage activatables objects. Here, the main methods of this class with a short comment (see also the API documentation):

```
protected Activatable(String codebase, java.rmi.MarshalledObject data,
                      boolean restart, in port)
                 throws ActivationException, java.rmi.RemoteException
```

This constructor is used by a subclass to register an object with the activation system and export it during "initial" construction. It first creates an activation descriptor (`activationDesc`) for the object and registers this descriptor with the default `ActivationSystem`. Then it exports the activatable object to the RMI runtime and reports the object as an active object to the local `ActivationGroup`.

The object's class code comes from *codebase* and initialization data is *data*. If *restart* is true, the object will be restarted, when the activator is restarted, else the object will be restarted when it is first invoked after a crash.

```
protected Activatable(ActivationID id, int port)
```

```
                    throws java.rmi.RemoteException
```

"Activation constructor". Must be present in order for the Activation framework to work. When an activation group tries to launch a service, it calls this two-argument constructor, which exports the activatable object to the RMI runtime.

```
public static Remote register(ActivationDesc desc)
                    throws UnknownGroupException, ActivationException,
                    java.rmi.RemoteException
```

Register an activatable object with the activation system without having to first create the object. This method returns the `Remote` stub, so that the object can be instantiated and activated at a later time when it is reclaimed for the first time. The object descriptor registration can be revoked by using `unregister.`

```
public static boolean inactive(ActivationID id)
                    throws UnknownObjectException, ActivationException,
                    java.rmi.RemoteException
```

Informs the JVM's `ActivationGroup` that the object with corresponding *id* is inactive. If the object is currently known to be activ and there are no pending or exceuting calls, it will be unexported from the RMI runtime.

```
public static void unregister(ActivationID id)
                    throws UnknownObjectException, ActivationException,
                    java.rmi.RemoteException
```

Revokes a previous registration for the object 's activation descriptor associated eith *id*.

```
public static ActivationID exportObject(Remote obj, String codebase,
                                        java.rmiMarshalledObject data,
                                        boolean restart, in port)
                    throws ActivationException, java.rmi.RemoteException
```

Registers the object's activation descriptor, constructed from the supplied *codebase* and *data*, with the activation system and export the remote object to the RMI runtime. This method is called by the "initial" constructor (the first one in this list). If appropriate, it should be invoked by an activatable object that does not extend the `Activatable` class.

```
public static Remote exportObject(Remote obj, ActivationID id,
                                  int port) throws java.rmi.RemoteException
```

Exports the remote object with identifier *id* to the RMI runtime. This method is called by the "activation" constructor (the second one in this list). During activation, it should be invoked by an activatable object that does not extend the `Activatable` class.

```
public static boolean unexportObject(Remote obj, boolean force)
                                  throws NoSuchObjectException
```

Unexports the remote object *obj*, making it unavailable for incoming calls. If *force* is true, the object is unexported, even if there are pending calls to the remote object or the remote object still has calls in progress.

(Note: Both constructors and both *exportObject* methods in this list also exist in a version that includes `RMIClientSocketFactory` and `RMIServerSocketFactory` arguments).

6.8 RMID Command-Line Arguments

(Source: "Java RMI", by W. Grosso, O'Reilly). For more information, see the JDK Standard

Documentation.

-port

RMID has a registry to listen for clients activation requests. It uses by default the port 1098. You can specify another port by setting the –port option (see also the property `java.rmi.activation.port`):

```
rmid –port 8000
```

-log

Allows to specify the directory, where `rmid` writes its log file

```
rmid –log <log_directory>
```

-stop

Allows to shut down gracefully the activation daemon (for a given port)

```
rmid –port 8000 –stop
```

-C<*argument*>

Passes the <*argument*> value along to all the child JVM's started by the activation daemon. For example:

```
rmid –C-Djava.rmi.server.logCalls=true
```

activates the standard RMI log facility in all child JVM's.

-J<*argument*>

Passes the <*argument*> to the JVM running RMID.

-J-Dsun.rmi.activation.execPolicy=<*policy*>
(only in Sun's implementation of RMID)

Specifies the policy that RMID employs to check commands and command-line options used to launch the JVM in which an activation group runs. Possible values of <*policy*> are:

- **default (or if property is unspecified)**

  Allows RMID to execute commands with specific command-line options only if RMID has been granted permission to execute them in the *security policy file* that RMID uses.

- ***<policyClassName>***

  Name of a class whose `checkExecCommand` method is executed in order to check commands to be executed by RMID (allows customized checks).

- **none**

  No check. RMID will not perform any validation of commands when launching activation groups.

6.9 Activation Parameters

(Source: "Java RMI", by W. Grosso, O'Reilly)

The most used parameters are:

java.rmi.activation.port

Port on which `rmid` will listen for incoming remote calls. The default value is 1098. This value is used by both `rmid` (set in the `–port` command-line argument) and by the setup code that registers instances of `ActivationDesc` with `rmid`. If you decide to use a different port for your setup code, then you have to set both parameters to the same value.

For the setup code, you can use a comand-line argument:

```
java –Djava.rmi.activation.port=8000 ...  rmi.actTime.Setup
```

or specify a system property in the code:

```
System.setProperty("java.rmi.activation.port", 8000);
```

sun.rmi.activation.snapshotInterval

Controls how often the activation daemon writes to its log file. The default is 200 operations. An operation is something that changes the number of registered or the number of active services.

sun.rmi.rmid.maxstartgroup

Limits the number of JVMs that `rmid` may be restarting concurrently (to smooth system performance by JVM spwaning). Default is 3.

sun.rmi.server.activation.debugExec

If set to *true*, the activation system will print out the command line used to create every JVM's.