RMI

REMOTE METHOD INVOCATION

INTRODUCTION TO RMI, A JAVA API FOR RPC-STYLE INVOCATION OF REMOTE OBJECT METHODS

Peter R. Egli
INDIGOO.COM

Contents

1. What is RMI?
2. Important RMI Components
3. RMI Layering
4. RMI Stub and Skeleton
5. RMI Registry
6. RMI Java Packages
7. RMI Transport Layer
8. RMI IDL
9. RMI Server Class Hierarchy
10. RMI Garbage Collection
11. RMI Dynamic Class Loading
12. RMI Parameter Passing
13. RMI Callbacks
14. RMI Remote Object Activation

1. What is RMI?

Remoting:

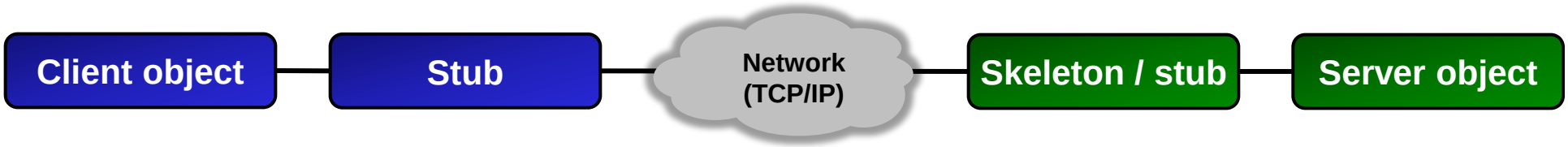
RMI is a lightweight Java technology that provides access to remote methods, similar to RPC, but object-oriented. RMI basically provides remote object access for a client and object registration for servers.

API and transport protocol:

RMI is both a Java API (java.rmi.* package) as well as a transport protocol definition for transporting RMI calls through a network (JRMI, see below).

Java technology:

RMI is a Java technology since it requires that client and server objects run in a JVM. By using IIOP as transport protocol, however, it is possible to connect RMI-clients to non-Java server objects (CORBA, see below).



IIOP: Internet Inter-ORB Protocol

2. Important RMI Components

Client:

The client looks up a remote object and calls methods on the obtained remote object.

Server:

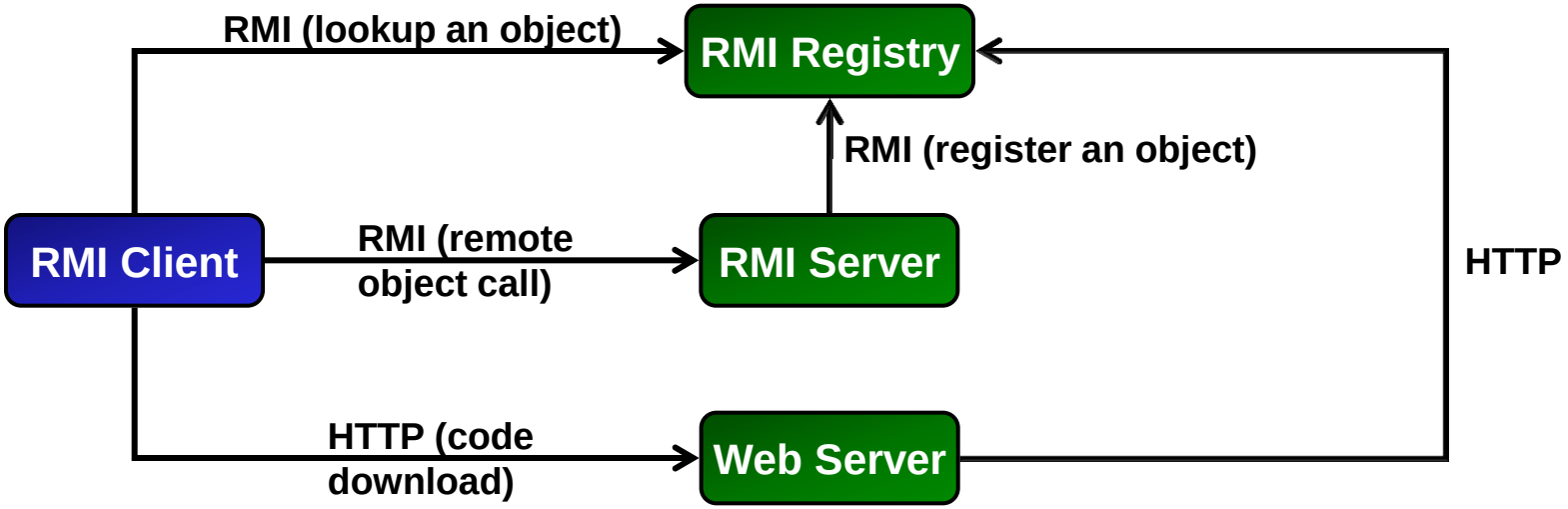
The server registers itself in the RMI registry and accepts method invocations from the client.

RMI Registry:

The registry is a remote object lookup service. The registry may run on the same host as the server or on a different host. The registry can also be a JNDI server.

Web Server:

A plain vanilla HTTP server may hold remote object classes for downloading by the client.



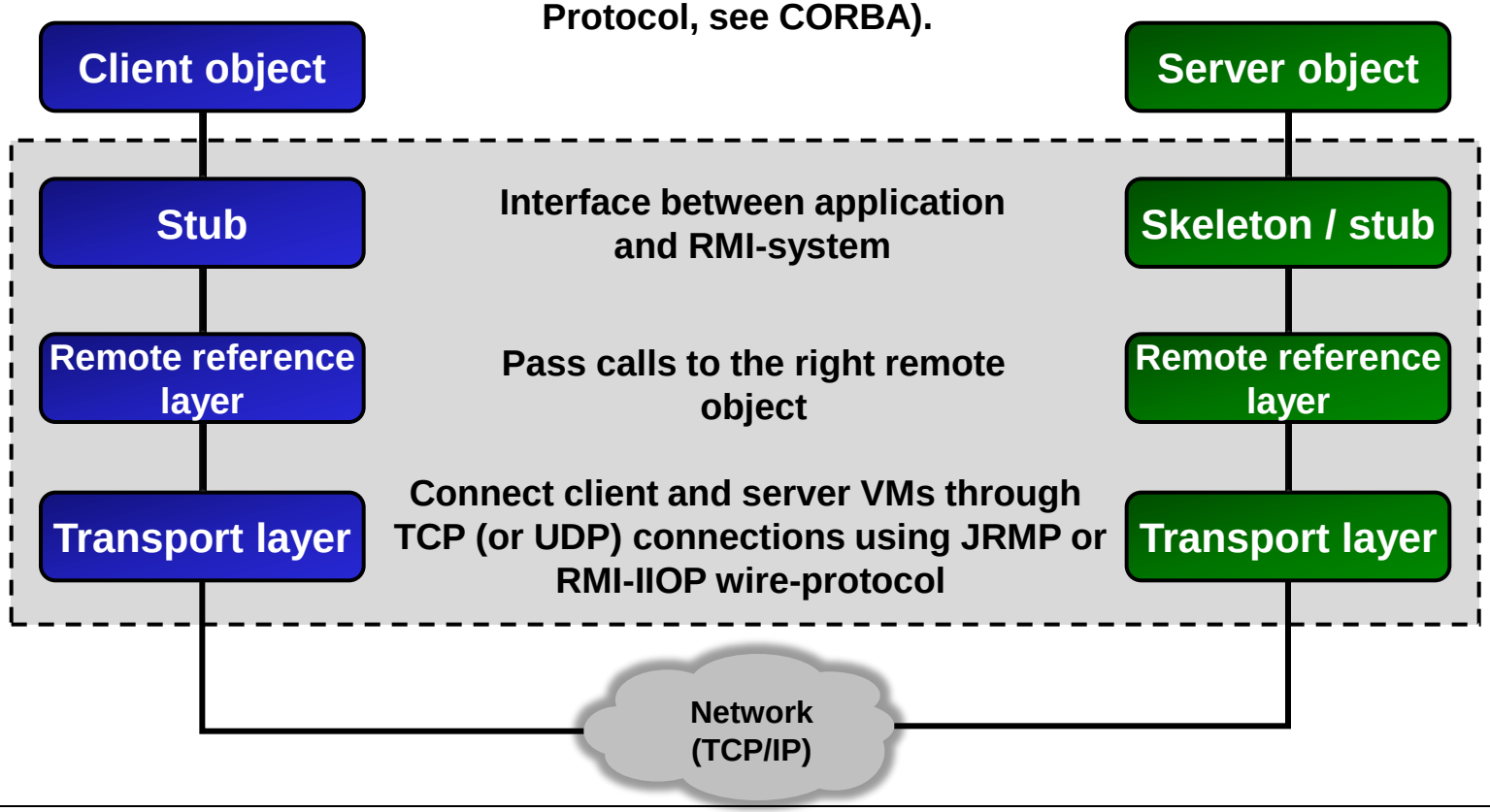
3. RMI Layering

RMI has 3 layers on the client and server side.

Stub layer: Interface between client application (caller) and server object.

Remote reference layer: Connects clients to remote service objects.

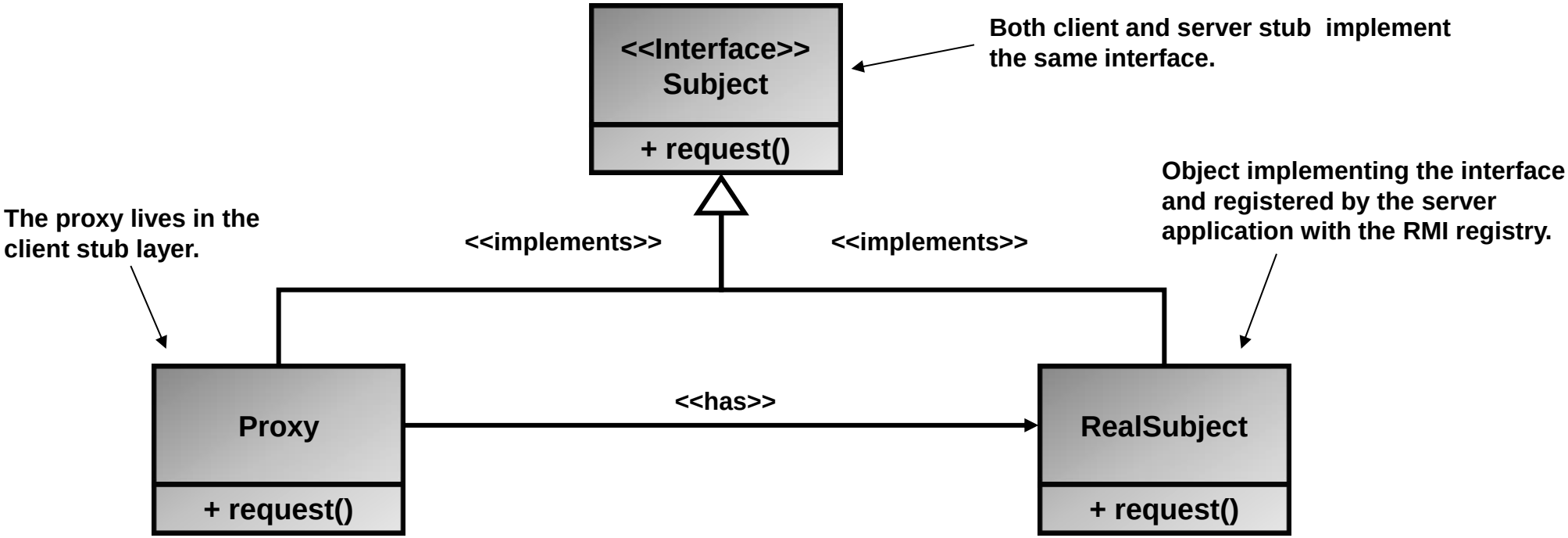
Transport layer: Makes connections between client VM and server VM, formats the data using JRMP (Java Remote Method Protocol) or RMI-IIOP (Internet-Inter-ORB-Protocol, see CORBA).



3 layers of RMI system

4. RMI Stub and Skeleton

RMI uses the proxy design pattern for client and server stub / skeleton:



Client stub: Proxy object on the client for accessing the remote server object. The client stub intercepts the calls of the client and passes it to the remote reference layer.

Server stub/skeleton: The server stub receives calls from the remote reference layer and passes it to the server object implementing the interface (= RealSubject in the picture above).

5. RMI Registry (1/2)

RMI registry function:

The RMI registry is a server registration and client lookup service.
It may run anywhere, i.e. does not need to be co-located with the RMI server object.
Default port for RMI registry: 1099 (may run on another port as well).

Server registration:

The servers register (export) objects that implement the service interface using `bind()` or `rebind()`:

Example:

```
RemServer localObject = new RemServer();
Naming.rebind("MyServ", localObject);
```

Client references:

Clients obtain references to server objects (= proxy objects) through the RMI registry using a URL scheme (with an optional port):

URL: rmi://<host_name> [:<name_service_port>] /<service_name>

Example:

```
RemIf remObject = (RemIf)Naming.lookup("rmi://" + host + "/MyServ");
```

5. RMI Registry (2/2)

RMI registry access to server stub classes:

The RMI registry needs to have access to the server's stub classes. This can be done in 2 ways (strictly either-or, using both will not work!):

1. CLASSPATH:

Add the path to server stub class files to CLASSPATH when starting the RMI registry.

2. Codebase property:

Start the server with the codebase property. This way the server registers the remote object along with the path to the class files.
The codebase property may be a file or HTTP URL.
N.B.: The codebase URL (file or HTTP) must contain a trailing slash / backslash as shown below!

Example Windows file codebase property file path:

```
java -Djava.rmi.server.codebase=file:/c:\RemServer\ -cp . RemServer
```

Example Unix codebase property file path:

```
java -Djava.rmi.server.codebase=file:/usr/RemServer/ -cp . RemServer
```

Example Windows and Unix codebase property HTTP path:

```
java -Djava.rmi.server.codebase=http://www.hsz-t.ch/~pegli/RemServer/ -cp . RemServer
```

6. RMI Java Packages

Online RMI javadoc: https://docs.oracle.com/javase/7/docs/api/

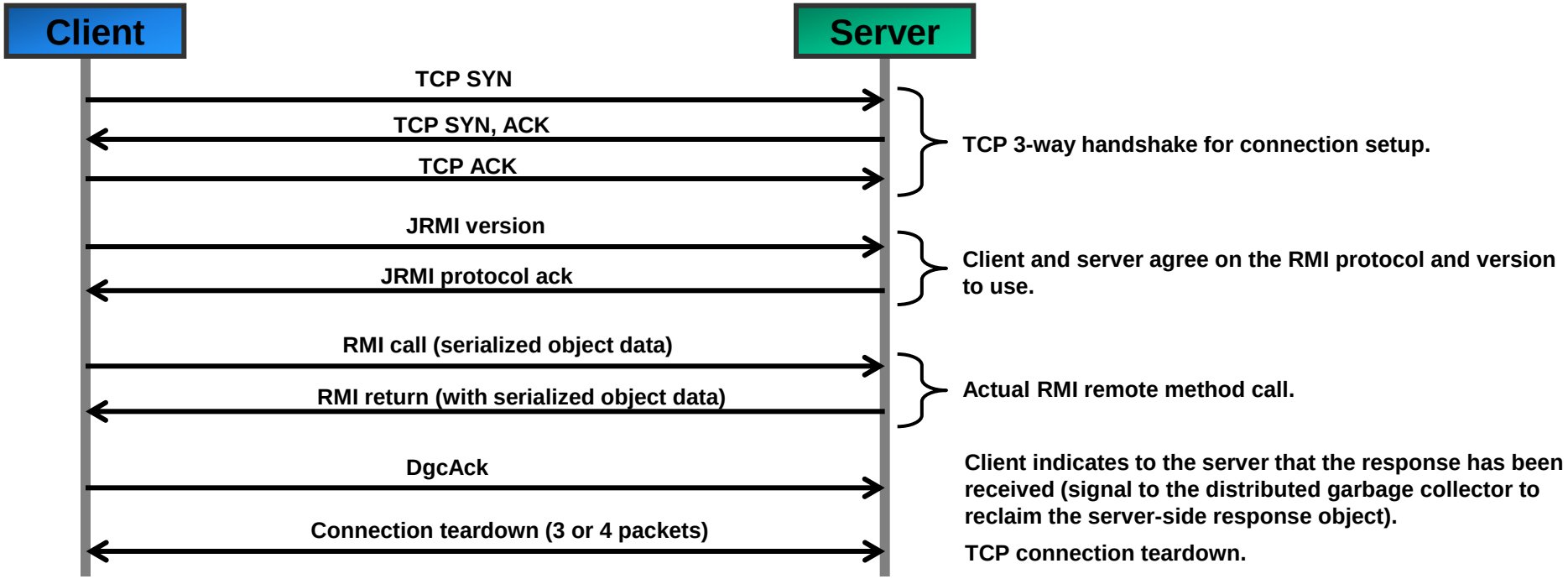
Package	Description
`java.rmi.*`	Core RMI package with classes and interfaces used by both client and server. Contains interface `Remote`, classes `Naming` and `RMISecurityManager` and some basic exception classes.
`java.rmi.activation.*`	Classes and interfaces for dynamic activation of remote objects together with RMI daemon (`rmid`). More information on dynamic invocation see below.
`java.rmi.dgc.*`	Classes and interfaces for distributed garbage collection (DGC).
`java.rmi.registry.*`	`Registry` and `LocateRegistry` classes for directly interacting with a (remote or local) registry. `Registry` class provides `lookup()`, `rebind()`, `list()` and other methods.
`java.rmi.server.*`	Classes for use on the server side like class loader (`RMIClassLoader`) and `UnicastRemoteObject` (base class for remote objects).
`javax.rmi.*`	APIs for RMI-IIOP (interoperability between RMI and CORBA).

7. RMI Transport Layer (1/2)

RMI allows using different wire-protocols for encoding and transporting the object requests.
Wire protocol = protocol for encoding data (objects) on the „wire“ (network).
The RMI transport layer uses stream-based TCP/IP connections only (point-to-point connections, no multicast).

a. JRMP (Java RMI Method Protocol):

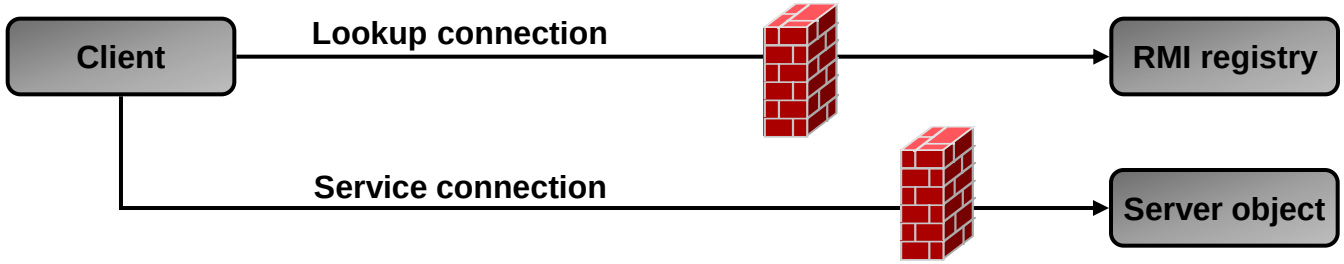
JRMP is the default wire-protocol for RMI.
N.B.: In addition to the connection setup packet exchange, the RMI protocol adds additional overhead. Thus RMI is not suited for use on non-LAN type networks.



7. RMI Transport Layer (2/2)

b. HTTP:

JRMP dynamically opens TCP connections. Firewalls may block such connections. HTTP as wire-protocol is better suited to get through firewalls (tunneling of RMI calls through an HTTP connection).



When the client fails to open a JRMP connection to the server, it automatically falls back to HTTP tunneling by encapsulating the request in an HTTP POST request.

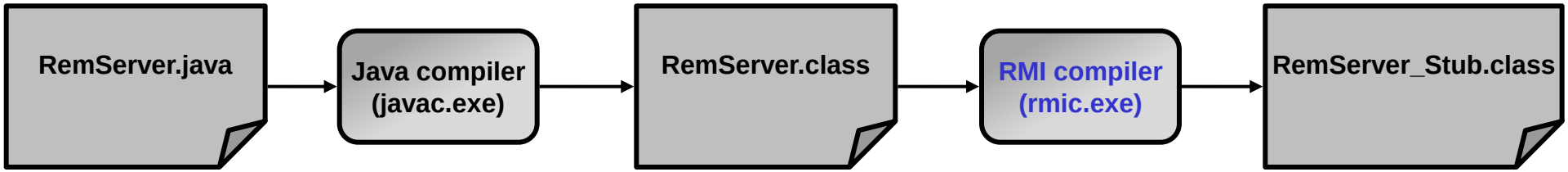
c. RMI-IIIOP (Internet Inter-ORB Protocol):

When using RMI-IIOP, RMI becomes interoperable with CORBA (an RMI client may call a CORBA object).

For further information see https://docs.oracle.com/javase/1.5.0/docs/guide/rmi-iiop/rmi_iiop_pg.html.

8. RMI IDL

RMI does not have a specific syntax for the description of the interface (IDL-file) but rather uses a compiled Java class as IDL (Java is the IDL).



Class file contains the RMI stub

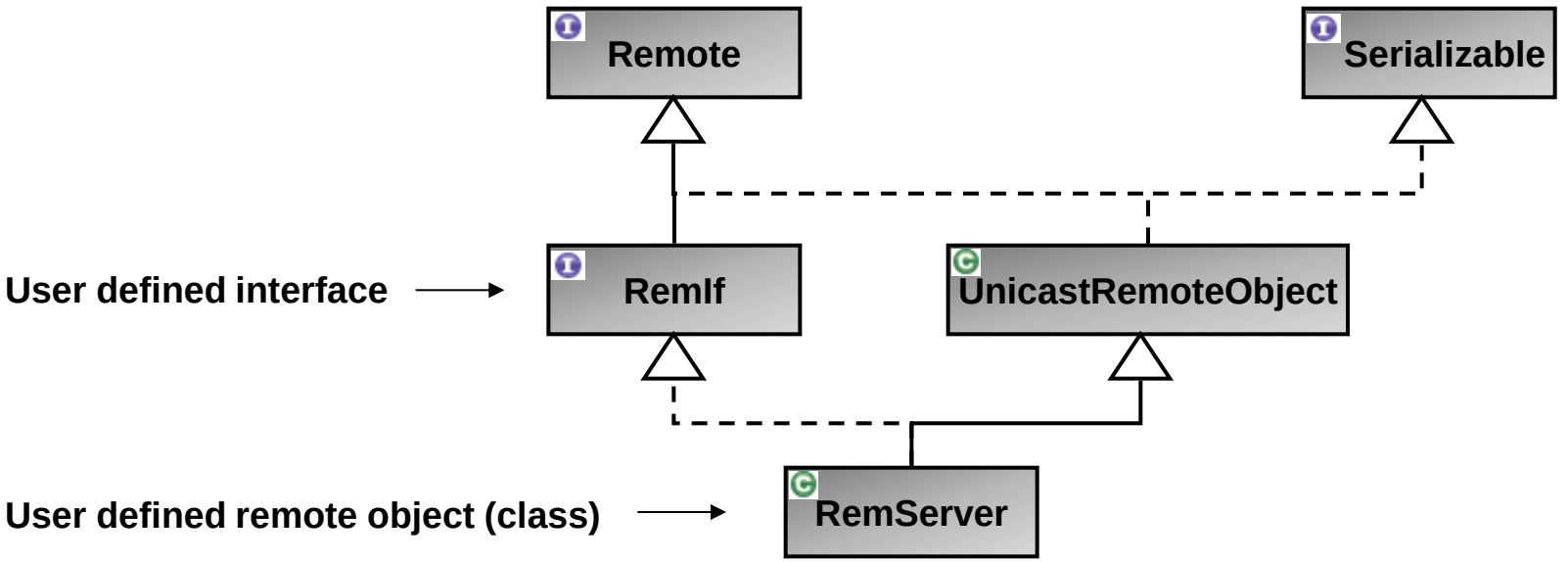
Important rmic options (output of rmic help on command line):

```
-v1.1                   Create stubs/skeletons for 1.1 stub protocol version.
-vcompat                Create stubs/skeletons compatible with both
                        1.1 and 1.2 stub protocol versions.
-v1.2 (default)         Create stubs for 1.2 stub protocol version only
-iiop                   Create stubs for IIOP. When present, <options> also includes:
                              -always               Create stubs even when they appear current
                              -alwaysgenerate       (same as "-always")
                              -nolocalstubs         Do not create stubs optimized for same process
-idl                    Create IDL. When present, <options> also includes:
                              -noValueMethods       Do not generate methods for valuetypes
                              -always               Create IDL even when it appears current
                              -alwaysgenerate       (same as "-always")
-classpath <path>       Specify where to find input class files
-bootclasspath <path>   Override location of bootstrap class files
-extdirs <path>         Override location of installed extensions
-d <directory>          Specify where to place generated class files
-J<runtime flag>        Pass argument to the java interpreter
```

9. RMI Server Class Hierarchy

A remote (server) object must implement the contracted interface (`RemIf`) which in turn must extend the base interface `Remote`.

A remote (server) object must extend the class `UnicastRemoteObject` (adds serializability among other things).

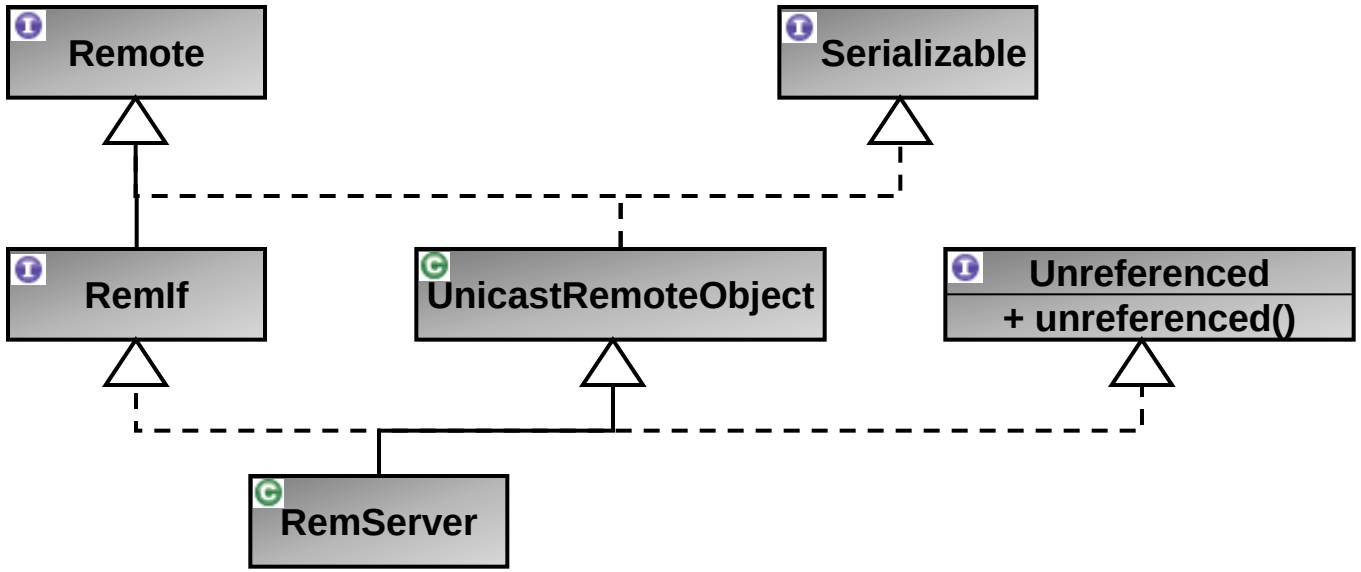


10. RMI Garbage Collection

RMI runs DGC (Distributed Garbage Collection) which frees unused server objects once no client holds a live reference anymore (reference counting algorithm).

Garbage collection is hidden to client and server, very much like local GC is hidden to Java applications.

A server may optionally implement the `Unreferenced` interface to be notified about an impending object removal (a kind of destructor call). RMI calls the method `unreferenced()` before the object is removed.

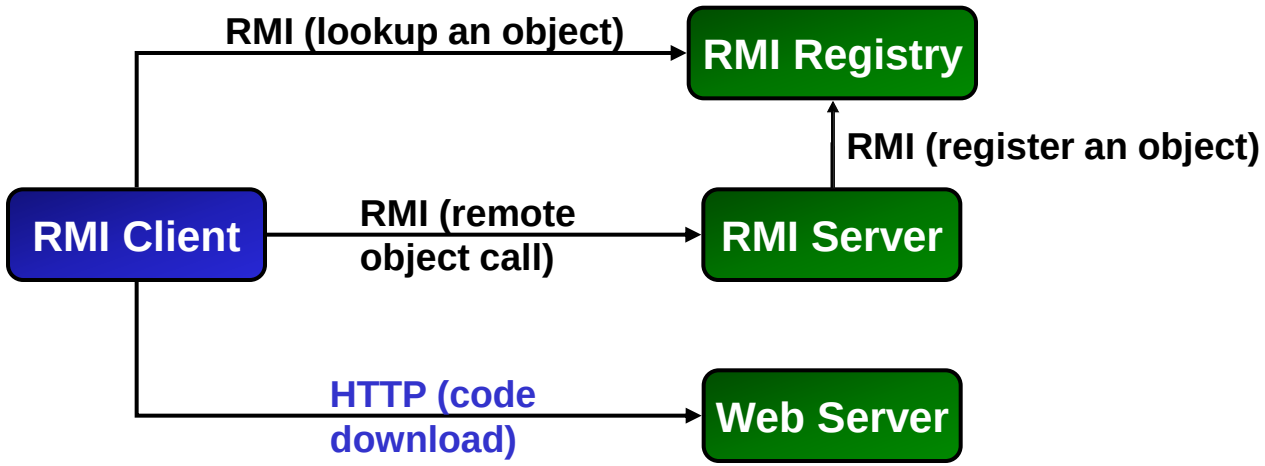


11. RMI Dynamic Class Loading (1/2)

For ease of deployment, stub files can be downloaded on request by the client instead of distributing them manually to each client.

The code (stub class files) can be made available for download on a web server.

N.B.: The interface file (class file containing the compiled remote interface) is still required both on the client and server side (otherwise client and server will not compile).



11. RMI Dynamic Class Loading (2/2)

Both client and server need to install a security manager to be able to load classes remotely:

```
System.setSecurityManager( new RMISecurityManger() );
```

Client and server need a security policy file granting the necessary rights like opening network connections (the following security policy file simply grants everything = no security at all):
mysecurity.policy file:

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

Starting the server with the security policy file (note the trailing backslash in the codebase path):

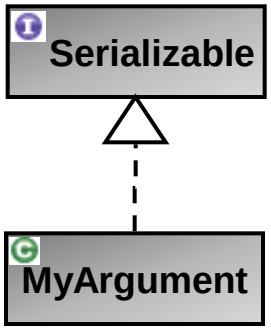
```
java -Djava.rmi.server.codebase="http://myserver/foo/"
  -Djava.security.policy=mysecurity.policy MyServer
```

Starting the client with the security policy file:

```
java -Djava.security.policy=mysecurity.policy MyClient
```

12. RMI Parameter Passing

RMI provides the same object and method call semantics as for local objects.
But: Arguments to method calls must be serializable, i.e. implement the `Serializable` interface.



Contained in java.io.serializable package

Serialization (recursively) „flattens“ objects into a byte-stream, i.e. object attributes that are in turn objects are serialized as well (serialization of a tree of object references).

Types that are not serializable:

- **Any object that does not implement `Serializable`.**
- **Any object that would pose a security risk (e.g. FileInputStream).**
- **Any object whose value depends on VM-specific information (e.g. Thread).**
- **Any object that contains a (non-static, non-transient) unserializable object (recursively).**

13. RMI Callbacks

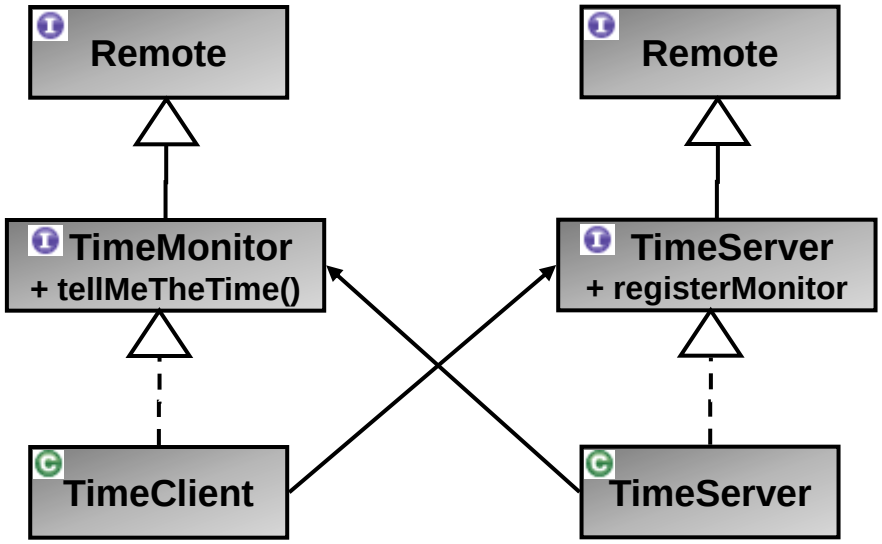
In certain situations it may be desirable to make calls from the server to the client, e.g. for giving progress feedback, warnings or errors etc.

There are 2 ways to make the client callable by the server:

1. **Client is an RMI server as well (extends `UnicastRemoteObject`).**
2. **Client makes itself callable through exportObject.**

Instead of extending UnicastRemoteObject, the client exports itself as an RMI server:

```
UnicastRemoteObject.exportObject( this );
```



In this example, the server provides an interface for a client to register itself for time updates (`registerMonitor()`).

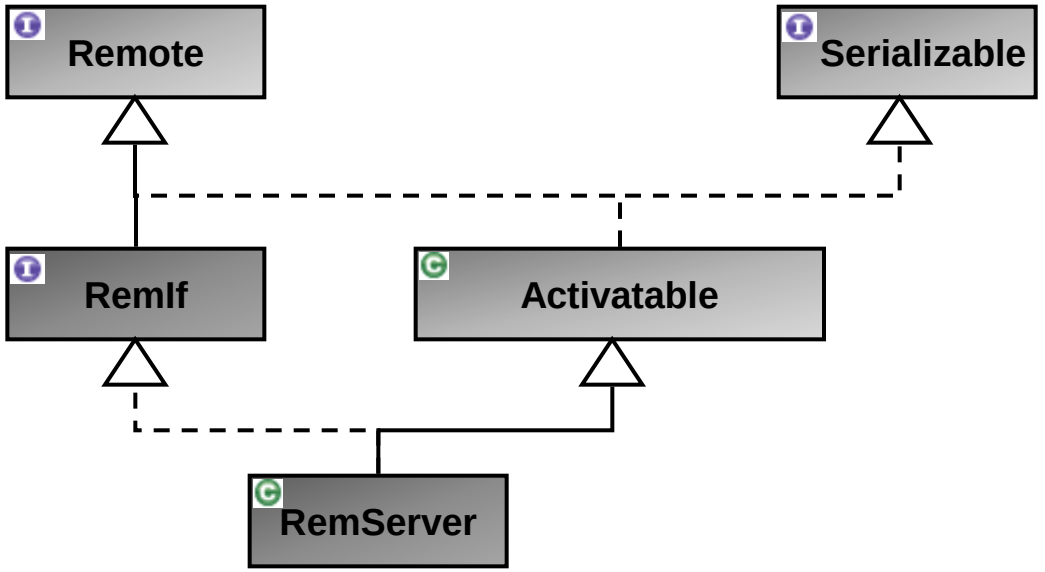
The client needs to export itself through `UnicastRemoteObject.exportObject()`.

More details see
https://docs.oracle.com/cd/E13211_01/wle/rmi/callbak.htm

14. RMI Remote Object Activation

Server objects extending `UnicastRemoteObject` and registered with the RMI registry run permanently.
This may be undesirable when there are a large number of objects (resource usage in terms of memory and TCP connections).

RMI provides an alternative through dynamically activatable objects:



Activatable (dynamically instantiatable objects) are registered with rmid (RMI daemon) instead of the rmiregistry daemon (see `$JAVA_HOME/bin/rmid.exe`).

See also https://docs.oracle.com/javase/7/docs/technotes/guides/rmi/activation/overview.html.