RMI Testing Ideas

In order to test RMI we consider the following items:

- Registry
- Socket Factories
- HTTP Tunneling
- Transfer of Remote Objects as parameters
- Distributed Garbage Collection (DGC)
- Class Loader
- Stress tests
- Performance tests

REGISTRY

For implementing the tests on this item we know:

Two registries cannot be exported in the same VM. The Security Manager checks if the current client has privileges to modify the contents of the Registry. All non-local clients should be rejected.

Tests cases proposed:

- Try to export a second Registry in the same VM.

- Try to access from a non-local host.

- Bind a stub in the Registry and bind another stub with the same key name. Also rebind and unbind stubs from a server and lookup these stubs from a client.

SOCKET FACTORIES

The socket factory to be used by an exported object is bounded at exportation time on the server. In order for the client to connect to the server, it must use the correct socket type. When the client deserializes the stub, the stub has a reference to an instance of an implementation of *RMIClientSocketFactory*. When the stub is bound into a naming service, serialization process creates a copy of the correct socket factory. When the client gets the stub from the naming service, it also obtains a copy of the socket factory, and can therefore connect to the server.

Test cases proposed:

- Create different types of Sockets at server side and client side and then test the connection. In all cases, we can test several configurations: set the socket at server side, set the socket at client side, set different sockets and verify that they fail.

HTTP TUNNELING

RMISocketFactory contain a method that creates sockets that transparently provide a firewall tunneling mechanism. The client socket first attempt a direct socket connection. If cannot be contacted with a direct connection, automatically attempts HTTP connection to the server. Server sockets automatically detect if a newly-accepted connection is an HTTP POST request. If that fails, it attempts to send the request to a URL beginning with /cgi-bin/java-rmi.cgi. The interpretation of this URL is that the request will be forwarded to a program which interprets the HTTP request and forwards it, as an HTTP request, to the appropriate port on the server machine.

Test cases proposed:

- Construct different scenarios:

  - A host without a firewall that accepts direct-connections.

  - A host with a firewall installed without a CGI script.

  - A host with a firewall installed with a CGI script.

  And then change the firewall configuration to test different scenarios.

TRANSFER OF REMOTE OBJECTS AS PARAMETERS

DISTRIBUTED GARBAGE COLLECTION (DGC)

CLASS LOADER

For implementing the tests on these items we consider:

(1) Create different types of remote objects based on the concept of Factory Pattern.

(2) Connects a series of hosts (for client/server purposes) and execute the tests on them.

In order to implement that, we based the case on the thesis of Andrei A. Dancus [1]:

Test case:

> In a host x we have an object A and a stub on x_i accessible from x_i's locally, a remote call is sent by x_i to x_{i+1} and creates the copy of A, then the strong reference to A on a host x_i is destroyed. The remote call is received by x_{i+1} and a stub is created therefore a *dirty* call is sent to x. Meanwhile, on host x_i A is identified as garbage and a *clean* message is sent to x. This behavior can propagate for all n-hosts.

We can also test the Class Loader changing the interfaces versions and passing object references that have another implementation. These interfaces can be used through reflection.

STRESS TESTS

PERFORMANCE TESTS

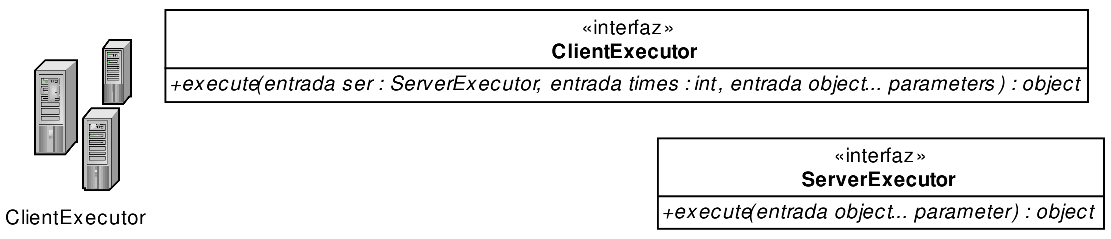
For implementing the stress tests and performance tests, plan to use the tests cases exposed before and overload them. We can add more hosts and execute different methods. We can export and unexport different objects. Finally, all results will be registered and compared with those obtained against Sun´s RMI package.

---

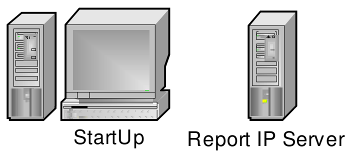
[1] GARBAGE COLLECTION FOR JAVA DISTRIBUTED OBJECTS. Andrei A. Dancus. (http://www.wpi.edu/Pubs/ETD/Available/etd-0502101-140442/unrestricted/dancus.pdf)

Testing structure

In order to test RMI we needed a real net. For implementing this we propose this test scenario.


ClientExecutor
«interfaz»
ClientExecutor
+execute(entrada ser : ServerExecutor, entrada times : int, entrada object... parameters) : object
«interfaz»
ServerExecutor
+execute(entrada object... parameter) : object

These are clients that execute a service several times. The service is a remote object. This object can be exported and executed through a stub. Otherwise if the object is not exported, it is executed directly in the client host. In this scenario all hosts are equals, but the RMISocketFactory configuration can be changed to do tests.



Because they are many and different clients, an additional service was implemented. This service allows to centralize the clients IP's. Thus the client reports his IP to the server and it is possible to ask to the server for reported clients. This server can store an IP address with the information of each client.

However, the clients not execute any activity. For this we construct a Server. It has a progress interface, thus the clients only can be consulted for visible problems.

The clients have a simple code and they need remote access to the server code. For this we use a FTP service and store the access in the correspondent property (*java.rmi.server.codebase*)

Testing methodology

In a first stage, we made tests on Registry, Remote Objects as parameters and Distributed Garbage Collection (DGC) following the testing ideas presented above. The tests on Socket Factories, HTTP Tunneling and Class Loader, will be made in next stages.

Note: The implemented scenarios can be used for implementing the remaining test cases and others that be defined later.

In order to testing *Registry* we have two hosts, a host (called *localhost*) and a host (called *non-localhost*).

local Registry
non-local Registry
localhost
non-localhost

RemoteRegistryTestCase

public void *testBind001( )* ***throws*** *RemoteException, AlreadyBoundException*

Tries to bind a remote object in a *Registry* of a non-local host (*non-local Registry*). This operation throws an exception.

public void *testLookup001( )* ***throws*** *AccessException, RemoteException, NotBoundException*

This test verifies that all elements in a *non-local Registry* are functional.

public void *testLookup002( )* ***throws*** *RemoteException, MalformedURLException, NotBoundException*

This test tries lookup a non-bounded remote object. This operation throws an exception.

public void *testLookup003( )*

This test tries lookup a remote object with a non-existing name. This operation throws an exception

public void *testRebind001( )* ***throws*** *RemoteException, MalformedURLException, NotBoundException*

This test tries to re-bind a remote object with a same name in a *Registry*. This operation throws an exception.

public void *testUnbind( )* ***throws*** *AccessException, RemoteException, NotBoundException*

This test tries to un-bind remote objects from a *non-local Registry*. This operation throws an exception.

public void *testAutoBindAndExportionUsingRemoteRegistry( )* ***throws*** *RemoteException*

This case makes a test on a non-exported object. If this object is deserialized, it is exported and bounded. This object is sent as parameter to a *non-local Registry*. The binding fails. However the object will be bounded because will be deserialized.

One more Registry can be exported in the same VM in the Sun's implementation. However one more Registry cannot be exported in the same VM in this implementation for accomplishment of the design.

- -

In order to test *Remote Objects as parameters, exportation, unexportation* and *remote references* we create an scenario who have several hosts

PortableTestCase

public void *testPortar( )* ***throws*** *RemoteException, NotBoundException*

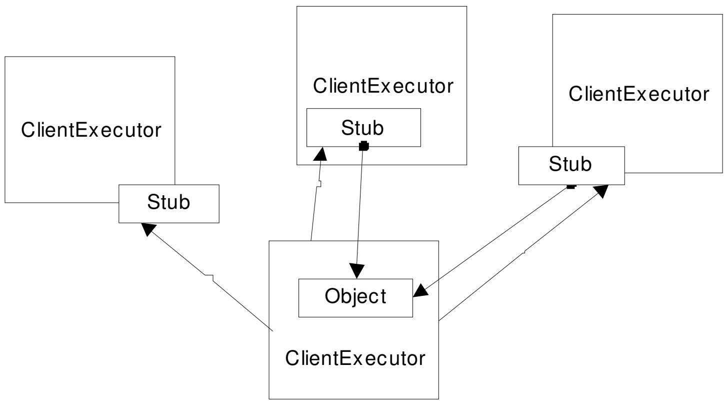
An object is exported to an external host and it is used through his reference. Then, the object is unexported and we can verify that cannot use the reference.

public void *testExportingAndComparing( )* ***throws*** *RemoteException*

The objects are exported in different hosts (one object per each host). In each exportation the destiny is verified.

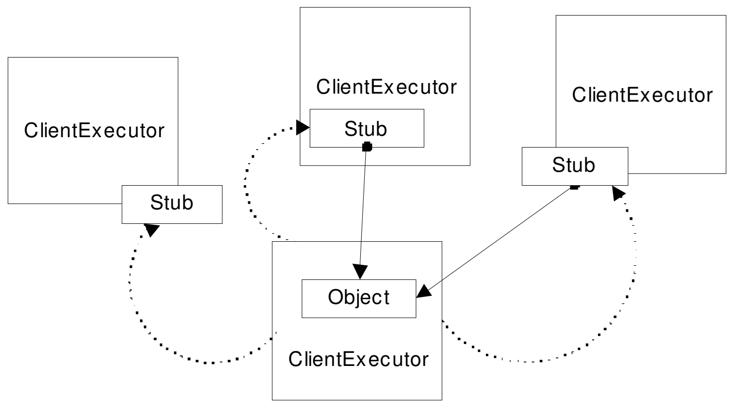
PropagableTestCase

public void *testPropagableOneHost001( )*



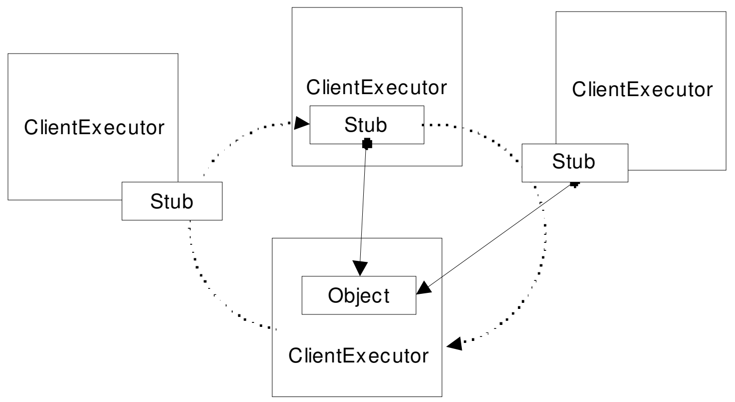
Simply executes an object on a host. This procedure is repeated on all hosts.

public void *testPropagableOneHost002( )*



Executes an object on a unitary list of remote hosts. This procedure is repeated on all reported hosts.

public void *testPropagableAtOnes001( )* ***throws*** *SecurityException, NoSuchMethodException, IllegalArgumentException, IllegalAccessException, InvocationTargetException*



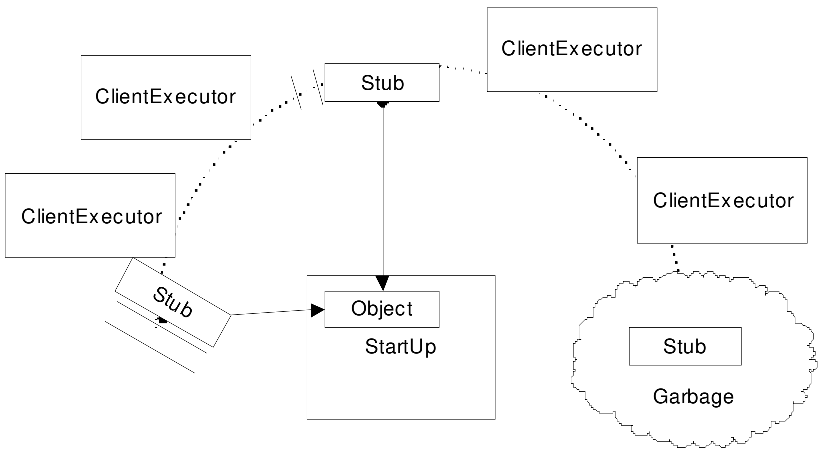
Simply executes an object on a list of remote hosts.

- -

In order to test *Distributed Garbage Collection* (DGC) we build an scenario with several hosts

DGCTestCase

public void *testMovingReference001( )* ***throws*** *RemoteException, NotBoundException*

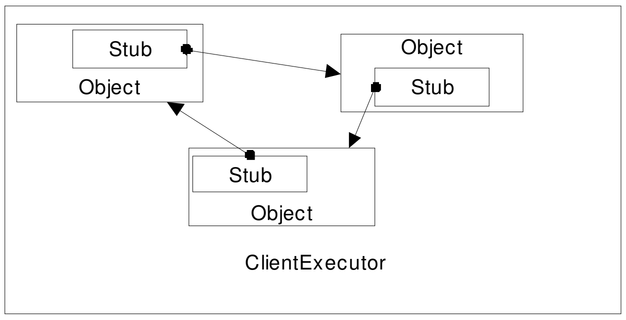


This test copies -through all servers- the reference (stub) of an exported remote object. The life time of these objects is limited, they work and then they are deleted. All local –strong- references to the object are deleted and we expect that the DGC collect the remote object.

public void *testMovingReference002( )* ***throws*** *RemoteException, NotBoundException*

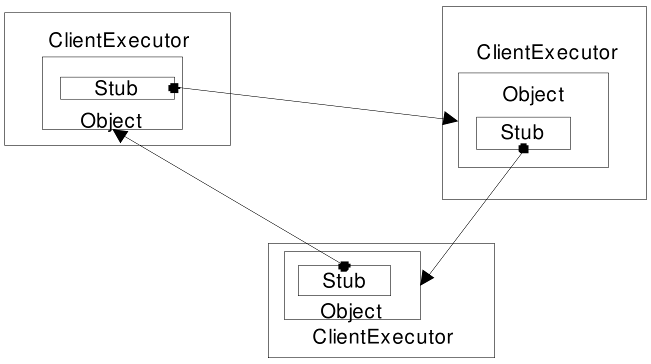
Exports a non exported object that has a reference to an exported object in an external host. If all local strong references are deleted and only the exported object in the external host has a reference to the local object, then the local object is not collected.

public void *testCiclicReference001( )* ***throws*** *RemoteException, NotBoundException*



Makes a cyclic reference using the stub of the remote object. The specified objects are in the same host. The DGC will detect the cyclic reference and collect them.

public void *testCiclicReference002( )* ***throws*** *RemoteException, NotBoundException*



Makes a cyclic reference to the remote objects that they are in different hosts. The DGC not will detect the cyclic reference and not collect them.

Testing Report

The Testing Report is obtained through the standard exit.

See an example of a Testing Report:

```
ReportIPServer Start....

class ar.org.fitc.test.rmi.integration.fase2.test.ReportIPTestCase start

        testMyHostName001

        testGetIt001

        testReport001

class ar.org.fitc.test.rmi.integration.fase2.test.RemoteRegistryTestCase start

        testBind001

        testLookup001

        testLookup002

        testLookup003

        testRebind001

        testUnbind

        testAutoBindAndExportionUsingRemoteRegistry

                Failed with: undeclared checked exception; nested exception is:

        java.io.EOFException
```

First the *reportIP* service is up. The time used for the correct start up and for another consideration of configuration is reported by dots that shows the progress. Then it runs the execution of a testing group. The tests executed are showed and some information of them can be indicated. The failed case is reported, if nothing is reported then the execution of testing was successful.

Initial Parameters

In order to execute the testing is needed to set a *serverhost* and *codebase* in the service-side and the *serverhost* in the client-side.