

Using The Visibroker™ Native Messaging Service

An Approach to Insulating CORBA-based Applications from the Effects of Frequent Network Connectivity Interruptions

A Borland Technical Paper

By Brenton Camac

March, 2003

Borland®

Contents

Section I - The Problem	1
Requirements Summary	2
Additional Constraints.....	2
Section II - The Proposed Solution	3
Visibroker Native Messaging	3
Client-side Object Wrappers	4
A description of the Sequence of Interactions	4
Section III - An illustrative implementation.....	8
Executing the test in an unreliable network environment	10
Extending the test to use Native Messaging	10
Executing the test in an unreliable communications environment with Native Messaging	12
Appendix A - Code Listing: test.client.ServerINMObjectWrapper	13
Appendix B - Code Listing: com.borland.s1.nm.NativeMessagingUtils	16
Appendix C - Code Listing: com.borland.s1.nm.Init	18

Section I - The Problem

The problem addressed here is when a CORBA client invokes a remote CORBA object's operation and loses network connectivity before the object has replied with a response.

Such network connectivity interruptions can arise, for instance, when the remote operation takes a long time to complete and the communication is routed through a proxy which terminates inactive sessions.

In such scenarios, the default behaviour on the server side is to discard the response (since it has lost contact with the client), and on the client-side for the ORB to raise a `org.omg.CORBA.COMM_FAILURE` indicating a problem with the network communications.

This behavior often causes the application to move into an indeterminate state. The client does not know the new state of the server-side; whether the operation completed successfully or not. Additionally, the client may have been expecting information to be returned from the operation which has now been discarded.

Recovering from such situations is often more involved than simply resubmitting the operation again. If, for instance, the operation is not idempotent, then resubmitting the request can result in duplicated work if the first request was successful. Also, if the network interruption occurred because a session timeout was reached, then simply resubmitting the operation will likely just lead to the same scenario.

This document proposes a solution whereby such network connectivity interruptions are accommodated by the CORBA infrastructure thus insulating the application from these undesirable effects.

The particular requirements of the solution being sought are specified below.

Requirements Summary

The solution must satisfy the following requirements:

1. Allow invocation responses from Visibroker-based servers to be retained for subsequent delivery to Visibroker-based clients when communication between said server and client has been terminated by a 3rd party.
2. Allow Visibroker-based clients to receive an expected invocation response from a Visibroker-based server once communication is re-established after having been previously terminated by a 3rd party.
3. Prevent Visibroker-based clients from re-submitting invocation requests on a Visibroker-based server when communication with said server is re-established after having been terminated by a 3rd party.

Additional Constraints

The solution must also comply with the following constraints of the existing application environment:

1. The client side platform is based on the Visibroker version 3.x ORB
2. The server side platform is based on the Visibroker version 3.x ORB
3. The solution should minimize any changes to the existing client-side application code
4. The solution should minimize any changes to the existing server-side application code

Section II - The Proposed Solution

Based on the above requirements and constraints, the proposed solution uses the Native Messaging capabilities of Visibroker™ to satisfy the overall requirements and client-side Object Wrappers to satisfy the constraints.

Visibroker Native Messaging

The Visibroker Native Messaging service allows clients to perform asynchronous, non-blocking method invocations in CORBA and J2EE environments. Asynchronous non-blocking communication explicitly separates the normally-combined actions of sending a request to a target object and receiving a reply from that target. To achieve this, a message broker is used; in the Visibroker implementation this is the Request Agent.

The Request Agent creates and manages delegate objects (Request Proxies) which receive invocations intended for the target object. When a Request Proxy receives such an invocation, it replies immediately (usually with an exception indicating that no response to the invocation is available yet). The proxy will then make the same invocation on the target object and wait for its response. Thus the interaction between the proxy and the target object is an ordinary synchronous CORBA call. The caller repeats the same invocation on the proxy as a way to test for the presence of a response from the target. If a response is available, then it is returned otherwise a specific CORBA exception is raised indicating no response is available yet. In this way, the caller interacts with the proxy in an asynchronous non-blocking manner; none of its calls will block for any significant amount of time.

Several variations to the above mode of operation are possible with the Native Messaging service. It is possible, for instance to have the proxy callback the caller when the response becomes available. It is also possible, for instance, to have the caller poll a group proxies with just one invocation rather than poll them individually. Neither of these facilities are used in the solution being proposed here, although their use in this situation is conceivable.

In this solution, the Request Proxy is used to temporarily store responses from the server while the client's network connection is inoperable. A more detailed explanation of the

sequence of interactions amongst these components is described below after Client-side Object Wrappers have been introduced.

Client-side Object Wrappers

Object Wrappers or "Smart Stubs" allow user-supplied code to be executed whenever an application invokes a method on a bound object. Object wrappers are used in the solution on the client side to host the additional functionality necessary to interact with the Native Messaging service (i.e. contacting the Request Agent and polling the Request Proxy). Object wrappers were chosen because they provide a means for introducing this additional functionality in the client without changing its existing code-base - one of the problem's constraints. Furthermore, typed object wrappers are used because they allow the original CORBA invocation to be redirected to another target (i.e. to the Request Proxy instead of the original target object).

A description of the Sequence of Interactions

The Native Messaging service and Typed Object Wrapper facilities of Visibroker are combined to provide the solution sought. The arrangement and interaction of these two services in achieving the proposed solution is described below.

The following sequence diagram shows how a CORBA invocation from the client is now re-routed through the Request Agent, and how the Object Wrappers installed in the client manage the polling activity to obtain the response when it becomes available.

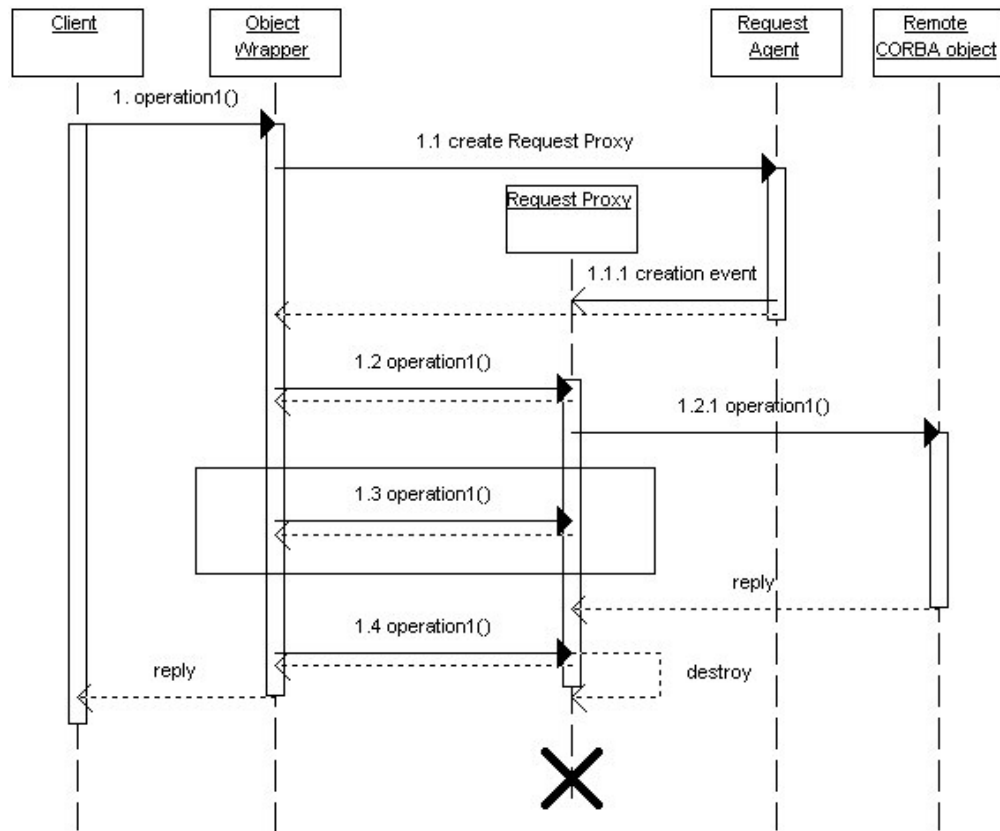


Figure 1. Sequence diagram of Client employing proposed solution

For a single CORBA invocation, the following steps are taken.

- Step 1 A Client invokes an operation of a remote CORBA object. The code to perform this would already be present in the existing client. However, because a Typed Object Wrapper has been installed for this object type, the invocation is delegated to the registered object wrapper. (Note. If no Typed Object Wrapper was installed for this object type or if the particular operation was no overridden in the wrapper then the invocation would have proceeded directly on the target object. Such an arrangement maybe satisfactory for certain types of objects and/or operations; i.e. ones which are not expected to take a long-time to complete and for which the

overhead of asynchronous messaging can't be justified. The decision of whether to use the Native Messaging approach or not should be evaluated on a case-by-case basis weighing the benefits and costs involved.)

- Step 1.1 The typed object wrapper co-located with the client first requests that the remote Request Agent create a new Request Proxy object. This is necessary because in the current version of the Visibroker Native Messaging Service, a Request Proxy can only be used once - for one invocation. Therefore a new proxy must be created for each invocation. Later versions of the service will allow the request proxy to be reused for subsequent invocations.
- Step 1.1.1 The Request Agent creates a new request proxy instance. This instance is co-located with the Request Agent. The proxy will be used to store the response from the Server until the caller returns to collect it.
- Step 1.2 The object wrapper delegates the call it received to the Request Proxy. Although this is a synchronous CORBA call, the Request Proxy returns a response immediately (i.e. not blocking the caller) with an exception indicating that no response is yet available.
- Step 1.2.1 The Request Proxy delegates this call to the target Server (object) and blocks waiting for a response from it. Because the Request Proxy is making a synchronous call to the target object, the network connection between the two should be reliable; not suffering from interruptions like those that may be present between the client and Request Proxy.
- Step 1.3 While the Request Proxy is blocked waiting for the target object invocation to return, any invocations it receives on the same operation will result in an exception (`org.omg.CORBA.NO_RESPONSE` with minor code `REPLY_NOT_AVAILABLE`) being thrown back to the caller indicating that no response is currently available. The responses to these invocations are always immediate; the caller is not blocked for any significant amount of time. Since the caller repeats this polling activity which doesn't block, it doesn't require the

network connection to be continuously present. If there is no connection present when the object wrapper attempts to poll the Request Proxy, it will receive a CORBA.COMM_FAILURE. The example object wrapper supplied in the sample solution will respond to such a condition by continuing to wait for another opportunity to poll the Request Proxy.

Step 1.4 When the Request Proxy has received a response from the target object, the next polling call from the Object Wrapper will be given the response just returned to the Request Proxy. Note that this could be an exception raised by the remote object. The Object Wrapper will then propagate this response to the caller. The Native Messaging service can be configured to automatically delete Request Proxys once the response has been returned. This is the approach taken in the supplied sample code.

The overhead associated with using the Native Messaging service compared to contacting the Server directly can be seen in the above sequence diagram. Specifically:

1. The creation of a Request Proxy for each invocation. (Although this won't be the case in future versions of the service), and
2. Extra remote calls made by the Client (Object Wrapper actually) as it polls the Request Proxy for a response.

Because of this overhead, it is recommended that this approach be applied only to those operations which would benefit from it. Candidate operations include those which take longer than usual to complete, and those for which recovery is difficult when network connectivity is disrupted (e.g. non-idempotent operations).

Section III - An illustrative implementation

To reproduce the problem and to demonstrate the proposed solution the following CORBA server and associated artifacts have been developed.

```
module CORBADefs {  
  
    interface Server1Interface {  
  
        exception ApplicationException { string s; };  
  
        // This operation waits for replyDelay amount of milliseconds before replying  
        // with param2. If param2 is zero, then it will raise an application defined exception  
        // after the specified delay period.  
        float operation1(in long replyDelaymsec, in float param2)  
            raises (ApplicationException);  
    };  
};
```

Figure 2. IDL of the Server (Server1)

This server provides a single operation (`operation1`) which takes two parameters. The first parameter specifies how long (in milliseconds) the operation invocation is to wait (block) in the server before responding with a response. The response given depends on the second parameter. If the second argument is a non-zero number, then the response will be the value of the second parameter, otherwise the response will be an application defined exception. This operation is implemented in the class `test.server.Server1Impl`.

This operation is used to simulate arbitrarily long running server-side operations and to test the possible types of responses that can be returned by a CORBA server.

Based on this IDL, the following CORBA entities are derived:

- a CORBA client (`test.client.Client.java`),
- a CORBA server (`test.server.ServerApp.java`) which instantiates a CORBA object and registers it with the `osagent`,

- a CORBA object implementation (`test.server.Server1Impl.java`) whose functionality is described above.

A CORBA server (class `test.server.ServerApp`) which contains a `main()` method instantiates an instance of the above CORBA object and registers it with the `osagent`. A configuration to run this server is available in the accompanying JBuilder project as `ServerApplication`, which is equivalent to manually executing the command:

```
prompt> vbj -VBJclasspath exampleNM.jar test.server.ServerApp
```

The Client class (`test.client.Client.java`) which contains a `main()` method will locate an instance of a CORBA object which implements the `Server1Interface` and will bind to it. It will then invoke `operation1()` five times using the parameters values supplied to it on the command-line at startup.

Two runtime configurations are provided in the JBuilder project which launch the client: "Client - normal return" and "Client - exception return". These are equivalent to the following:

```
prompt> vbj -VBJclasspath exampleNM.jar test.client.Client 5000 10.0
```

and

```
prompt> vbj -VBJclasspath exampleNM.jar test.client.Client 5000 0.0
```

respectively.

Note. The logic of the Client class does not contain any of the proposed solution. That code is placed entirely within the Object Wrapper (`test.client.Server1NMObjectWrapper`).

Executing the test in an unreliable network environment

An unreliable environment of the type described in the problem statement above can be simulated by running the server and client on separate hosts and disconnecting one of the hosts from the network during the test run. When this is done, behaviour like the following will be observed. In this test, the network was disconnected during the third invocation to the server; the first two were successful.

```
prompt>
prompt> vbj -VBJclasspath exampleNM.jar test.client.Client 5000 10.0
result: Server returned value as expected - 10.0
result: Server returned value as expected - 10.0
org.omg.CORBA.COMM_FAILURE: vmcid: 0x0 minor code: 0 completed: No
at com.inprise.vbroker.orb.DelegatImpl.verifyConnection(DelegatImpl.java:369)
  at com.inprise.vbroker.orb.DelegatImpl.is_local(DelegatImpl.java:561)
  at org.omg.CORBA.portable.ObjectImpl._is_local(ObjectImpl.java:354)
  at test.CORBADefs._Server1InterfaceStub._vis_operation1(_Server1InterfaceStub.java:67)
  at test.CORBADefs._Server1InterfaceStub.operation1(_Server1InterfaceStub.java:59)
  at test.client.Client.operation1(Client.java:101)
  at test.client.Client.testOp1(Client.java:106)
  at test.client.Client.main(Client.java:46)
Exception in thread "main"
prompt>
```

Table 1. Exception resulting from interrupted network communication

This result showed that the client received a CORBA.COMM_FAILURE exception as soon as it detected the network connectivity was lost. Also, the reply from the server for the third invocation was lost.

Extending the test to use Native Messaging

To this code base is applied the solution outlined earlier in this document - using Visibroker Native Messaging and Object Wrappers.

Client Side Changes

To avoid making changes to the client side code (a constraint of the problem), Visibroker Typed Object Wrappers are used to host the Native Messaging capabilities. Typed Object Wrappers are generated by the IDL compiler during the compilation process. This is enabled by specifying the `-obj_wrapper` flag to `idl2java`.

Each CORBA object which is to be contacted via the Native Messaging service (using this solution) will require a Typed Object Wrapper. (Note. Additional overhead is incurred when contacting a CORBA server using Native Messaging. Therefore this approach should be applied only to those CORBA servers which merit the additional overhead.)

Since there is only one CORBA object in this example, there will be only one Typed Object Wrapper generated, and will require the development of only one custom wrapper. In the test package, the custom wrapper which hosts this functionality is the class `test.client.Server1NMObjectWrapper` which extends the generated class `Server1InterfaceObjectWrapper`. Because this is a typed object wrapper, it must override each remote operation which is to use Native Messaging. In this case, that is the method `operation1()`.

Server Side Changes

Using the Native Messaging Service with the Server doesn't require any changes to its code base. However, the NM RequestAgent must be started on the network. This is done using the command

```
prompt> requestagent
IOR written to: requestagent.iior
Request Agent started ...
```

By default, the RequestAgent will start on port 5555 (configurable using `vbroker.se.iiop_tp.scm.iiop_tp.listener.port`) and is given the name RequestAgent (configurable using `vbroker.requestagent.name`).

Executing the test in an unreliable communications environment with Native Messaging

Re-running the test with Native Messaging capabilities enabled under the same test conditions yields different results. Again, the network connection was terminated after the request of the third invocation had been sent to the server but before the response was received. Network connectivity was not re-established for 30 seconds which was after the server had responded with its result. Below is the observed behaviour.

```
prompt> vbj -DORBInitRef=RequestAgent=corbaloc::jorr2k:5555/RequestAgent
-Dvbroker.orb.dynamicLibs=com.borland.s1.nm.Init -VBJclasspath exampleNM.jar
test.client.Client 5000 10.0
result: Server returned value as expected - 10.0
result: Server returned value as expected - 10.0
result: Server returned value as expected - 10.0
result: Server returned value as expected - 10.0
result: Server returned value as expected - 10.0
prompt>
```

Table 2. Use of Native Messaging to overcome network interruptions

This result shows that the temporary loss of network connectivity did not affect the application's processing. It was able to continue to operate normally; eventually receiving the server's reply to the third invocation when communications were restored even though at the time the reply was available there was no communication link between the client and server. Furthermore, after the connection was re-established, the client completed its normal operations - submitting the fourth and fifth invocations - because there were no exceptions raised to it.

This test run demonstrates that the code has satisfied the three functional requirements demanded of the solution.

Appendix A - Code Listing:

test.client.Server1NMOBJECTWrapper

```
package test.client;

/**
 * <p>Title: S1 Native Messaging Project</p>
 * <p>Description: The Server1NMOBJECTWrapper is a Typed Object Wrapper which
 * overrides the application-level operations of Server1Interface (which in
 * this case is just the method operation1) to dispatch the invocation to a
 * Visibroker Native Messaging Request Agent which in turn submits the invctn
 * to the intended target. By communicating with the Request Agent instead of
 * the server directly, the client can recover from transient communication
 * failures and receive the queued replies from the Request Agent rather than
 * having to resubmit the original request again.</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Borland Software Corp</p>
 * @author Brenton Camac
 * @version 1.0
 */
import com.borland.sl.nm.*;
import com.borland.vbroker.NativeMessaging.*;
import test.CORBADefs.*;

public class Server1NMOBJECTWrapper
    extends Server1InterfaceOBJECTWrapper {

    // CORBA object reference to the remote Visibroker Native Messaging
    // Request Agent service
    private RequestAgent reqAgent;

    public Server1NMOBJECTWrapper() {
        reqAgent = NativeMessagingUtils.getRequestAgent();
    }

    /**
     * An application defined operation. Each such operation needs to be
     * overridden using the same pattern shown here.
     * @param replyDelaymsec
     * @param param2
     * @return
     * @throws test.CORBADefs.Server1InterfacePackage.ApplicationException
     */
    public float operation1(int replyDelaymsec, float param2)
        throws test.CORBADefs.Server1InterfacePackage.ApplicationException {

        // Native Messaging Request Agent proxies are only good for one
        // invocation. Therefore, create a new one for this invocation.
        // NB. Be sure it gets created on the call stack (as it is here) and not
        // in heap memory to ensure its local to this thread and not shared with
        // other threads.
        test.CORBADefs.Server1InterfaceOperations proxy = createRequestProxy();

        // Determine when is the time to quit waiting for a response and yield
        // control back to the caller
        long quittingTime_ms = System.currentTimeMillis() +
            NativeMessagingUtils.overallTimeout_ms;
    }
}
```

```
// remainingTimeInterval_ms is milliseconds before quittingTime_ms is
// reached. Updated while progressing through the task
long remainingTimeInterval_ms;

while (true) {
    try {
        // dispatch the request to the server and poll for the response
        return proxy.operation1(replyDelaymsec, param2);
    }
    catch (org.omg.CORBA.NO_RESPONSE ex) {

        // Request Agent was contacted ...
        if (ex.minor == REPLY_NOT_AVAILABLE.value) {

            // Request Agent indicated that response is not yet available
            remainingTimeInterval_ms = quittingTime_ms -
                System.currentTimeMillis();

            if (remainingTimeInterval_ms > 0) {

                // but still have time available to wait for a response
                try {

                    // sleep for PollingPeriod amount of time or less if that were
                    // to go past the quittingTime event
                    Thread.sleep(NativeMessagingUtils.pollingPeriod_ms <
                        remainingTimeInterval_ms
                        ? NativeMessagingUtils.pollingPeriod_ms
                        : remainingTimeInterval_ms
                    );

                    // try Request Agent again for a response
                    continue;
                }
                catch (InterruptedException ex1) {
                    // try Request Agent again for a response
                    continue;
                }
            }
            else {
                // Exceeded total amount of time allowed to wait for the response
                // Propagate exception to caller
                throw ex;
            }
        }
        else {
            // Weren't expecting this type of NO_RESPONSE response from RA!
            // Propagate NO_RESPONSE to caller
            throw ex;
        }
    }
    catch (org.omg.CORBA.COMM_FAILURE ex) {
        // couldn't contact the Request Agent

        remainingTimeInterval_ms = quittingTime_ms -
            System.currentTimeMillis();
        if (remainingTimeInterval_ms > 0) {
            // but still have time available to wait for a response or send a
            // request if it hasn't gone out already.

```

```
try {
    // sleep for PollingPeriod amount of time or less if that were to
    // go past the quittingTime event
    Thread.sleep(NativeMessagingUtils.pollingPeriod_ms <
        remainingTimeInterval_ms
        ? NativeMessagingUtils.pollingPeriod_ms
        : remainingTimeInterval_ms
    );

    // poll Request Agent again for a response
    continue;
}
catch (InterruptedException ex1) {
    // poll Request Agent again for a response
    continue;
}
}
else {
    // Exceeded total amount of time allowed to wait for the response
    // Propagate COMM_FAILURE exception to caller.
    throw ex;
}
}
}
}

/**
 * A private method used by all application-level operations overridden in
 * this Typed Object Wrapper.
 * It creates a one-shot Native Messaging proxy to send an asynchronous
 * request to the target identified by this ObjectWrapper.
 * @return a stub which can send typed invocations to the target
 */
private test.CORBADefs.Server1InterfaceOperations createRequestProxy() {

    // construct a proxy within the Request Agent which is to forward this
    // request to the desired target
    org.omg.CORBA.Object req_proxy = NativeMessagingUtils.createNMRequest(
        getTarget(),
        getTarget().__repository_id());

    /* Need to give req_proxy stub an appropriate application level interface.
     * But can't do this in the normal way using narrow: e.g.
     * test.CORBADefs.Server1Interface typedRequestTarget = test.CORBADefs.
     * Server1InterfaceHelper.narrow(req_proxy);
     * because that would wrap the req_proxy in another Object Wrapper and
     * cause infinite recursion on the operation1() method.
     * So do the equivalent explicitly -
     */

    // create a stub which won't invoke the registered ObjectWrappers
    test.CORBADefs._Server1InterfaceStub unwrappedStub = new test.CORBADefs.
        _Server1InterfaceStub(true);

    // set the new stub's delegate to be the req_target's delegate
    unwrappedStub.__set_delegate(
        (org.omg.CORBA.portable.ObjectImpl) req_proxy).__get_delegate() );

    return unwrappedStub;
}
}
```

Appendix B - Code Listing:

com.borland.s1.nm.NativeMessagingUtils

```
package com.borland.s1.nm;

/**
 * <p>Title: S1 Native Messaging Project</p>
 * <p>Description: Contains two utility methods that are frequently used when
 * dealing with the Visibroker Native Messaging service.</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Borland Software Corp</p>
 * @author Brenton Camac
 * @version 1.0
 */

import org.omg.CORBA.ORB;
import com.borland.vbroker.NativeMessaging.*;
import org.omg.CORBA.ORBPackage.InvalidName;
import com.borland.vbroker.NativeMessaging.RequestAgentPackage.
    DuplicatedRequestTag;

public class NativeMessagingUtils {
    private static RequestAgent request_agent = null;

    public static long overallTimeout_ms = 120000; // 2 minutes
    public static long pollingPeriod_ms = 3000; // 3 seconds

    /**
     * This method obtains a reference to the Visibroker Native Messaging
     * RequestAgent via a resolve_initial_references on the supplied ORB. This
     * requires that the ORB was configured with a RequestAgent reference.
     * Typically this is done by supplying a command-line argument such as the
     * following: -DORBInitRef=RequestAgent=corbaloc::hostname:5555/RequestAgent
     * where hostname is the name (or IP address) of the machine executing the
     * Visibroker Native Messaging Request Agent.
     * @param orb An ORB configured with a Visibroker Native Messaging Request
     * Agent ORB service reference
     */
    public static void init(ORB orb) {
        try {
            request_agent = RequestAgentHelper.narrow(
                orb.resolve_initial_references("RequestAgent"));
        }
        catch (InvalidName ex) {
            System.err.println(ex);
            System.exit(1);
        }
    }

    /**
     * Returns a reference to the Visibroker Native Messaging Request Agent
     * @return a CORBA object reference to a Visibroker Native Messaging
     * RequestAgent
     */
    public static RequestAgent getRequestAgent() {
        return request_agent;
    }
}
```

```
}

/**
 * Create a delegate within the RequestAgent which will forward and handle
 * a future application-level request.
 * @param target the CORBA object which ultimately is to receive the request
 * @param reposID the repository ID of the above mentioned target
 * @return a Native Messaging delegate created by the Request Agent
 */
public static org.omg.CORBA.Object createNMRequest(org.omg.CORBA.Object
    target,
    String reposID) {

    RequestAgent req_agent = getRequestAgent();
    RequestDesc rdesc = new RequestDesc();

    rdesc.target = target;
    rdesc.repository_id = reposID;
    rdesc.reply_recipient = null;
    rdesc.the_cookie = new byte[0];
    rdesc.polling_group = "";
    rdesc.request_tag = new byte[0];
    rdesc.properties = new com.borland.vbroker.NativeMessaging.Property[0];

    org.omg.CORBA.Object _NMDelegate = null;
    try {
        _NMDelegate = req_agent.create_request(rdesc);
    }
    catch (DuplicatedRequestTag ex) {
    }
    return _NMDelegate;
}

/**
 * Accessors, mutators and constructors
 */
private NativeMessagingUtils() {
}

public void setOverallTimeout_ms(long i) {
    if (i > 0) {
        overallTimeout_ms = i;
    }
}

public long getOverallTimeout_ms() {
    return overallTimeout_ms;
}

public void setPollingPeriod_ms(long i) {
    if (i > 0) {
        pollingPeriod_ms = i;
    }
}

public long getPollingPeriod_ms() {
    return pollingPeriod_ms;
}
}
```

Appendix C - Code Listing: com.borland.s1.nm.Init

```
package com.borland.s1.nm;

/**
 * <p>Title: S1 Native Messaging Project</p>
 * <p>Description: This class is used to install the Typed Object Wrappers
 * (Stubs) used to communicate with the Visibroker Native Messaging
 * service</p>
 * <p>Copyright: Copyright (c) 2003</p>
 * <p>Company: Borland Software Corp</p>
 * @author Brenton Camac
 * @version 1.0
 */

import java.util.*;
import com.inprise.vbroker.orb.ORB;
import com.inprise.vbroker.properties.PropertyManager;
import com.inprise.vbroker.interceptor.*;
import test.CORBADefs.*;
import test.client.Server1NObjectWrapper;
import com.borland.s1.*;

public class Init
    implements ServiceLoader {
    com.inprise.vbroker.orb.ORB _orb;

    public void init(final org.omg.CORBA.ORB orb) {
    }

    public void init_complete(org.omg.CORBA.ORB orb) {
        _orb = (ORB) orb;

        // initialize NativeMessagingUtils
        NativeMessagingUtils.init(orb);

        // install typed object wrapper
        Server1InterfaceHelper.addClientObjectWrapperClass(orb,
            Server1NObjectWrapper.class);
    }

    public void shutdown(org.omg.CORBA.ORB orb) {}
}
```