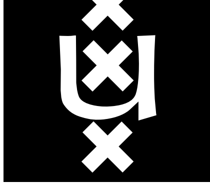


ISIS technical report series, Vol. 24, April, 2000

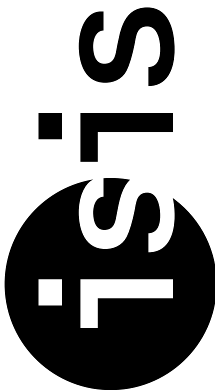
Practical tutorial for using Corba

A step-by-step introduction to the Common Object Request Broker Architecture

Jan van Gemert
Intelligent Sensory Information Systems
Department of Computer Science
University of Amsterdam
The Netherlands

A compact step-by-step tutorial for creating a CORBA object to get some hands-on experience with the Common Object Request Broker Architecture. Corba enables platform, language and network transparency. How to use a C++ object created on a Windows NT, in a Java program running on Unix.

Intelligent Sensory Information Systems
University of Amsterdam
The Netherlands



Contents

1 Introduction 1
1.1 Description of files used 2
1.2 Applications used 2

2 Install Orbacus 4.0 beta 2 3
2.1 Installing Orbacus for the C++ server 3
2.2 Installing Orbacus for the Java client 4

3 Environmental Changes 5
3.1 Variable settings 5
3.2 MS visual C++ settings 6
3.3 Orbacus configuration files 7

4 Specify the object in IDL 9

5 Setting up the C++ Server 10
5.1 Compile IDL file 10
5.2 Implement the servant object 10
5.2.1 Example of Count_i.h 11
5.2.2 Example of Count_i.cpp 12
5.3 Implement the server 13
5.3.1 Server.cpp code 14

6 Setting up the Java Client 15
6.1 Compile IDL file 15
6.2 Implement the client 15
6.2.1 Client.java code 16

7 Running the example 17

Intelligent Sensory Information Systems
Department of Computer Science
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

tel: +31 20 525 7463
fax: +31 20 525 7490
http://www.science.uva.nl/research/isis

Corresponding author:
Jan van Gemert
tel: +31(20)525 7507
jvgemert@science.uva.nl
www.science.uva.nl/~jvgemert

1 Introduction

This small practical tutorial is a doorway to the world of the Common Object Request Broker Architecture. Yes young Java Knight, this will give you the powers to use transparent inter-language client-server relationships between objects. This document provides a framework for implementing a simple count object, containing a number that can be set, read and incremented by 1. The idea of this simple object was found in [1]. The choice for such an object, is based on its simplicity as its display of client-server interaction.

What you have here, is no theoretical work, nor does it explain in detail how CORBA uses its magic. It's just a step by step practical tutorial for creating a CORBA C++ server with a Java client. Its small steps are backed up with some chunks of fast-food files, just waiting to be compiled and executed.

For further information about CORBA, check the included `OB-4.0b3.pdf.zip` file. Additionally, I recommend these sites:

`http://www.omg.org/` (creators of CORBA)
`http://www.ooc.com/` (implementers of this CORBA version)
`http://www.corba.net/`
`http://developer.java.sun.com/developer/onlineTraining/corba/`
`http://www.infosys.tuwien.ac.at/Research/Corba/homepage.html`

From now on, the CORBA fundamentals are presumed as known. You do not have to master the details of it, just a grasp of the basics is enough. Do you know what an IOR is? What is IDL used for? Skeletons? Stubs? Etc. Furthermore, some understanding of Java, C++ and Microsoft Visual C++ (MSVC) is required. For information about them, I recommend these sites:

`http://java.sun.com/docs/books/tutorial/`
`http://msdn.microsoft.com/visualc/technical/documentation.asp`

Here also, no expertise is really required. Just know how to create and compile a new Visual C++ project, compile and run Java code, and know how to write a simple C++ class.

Exception handling is mainly not done in this tutorial, in order to focus on the main goal of implementing a CORBA object. This document will allow you to create a working implementation of the CORBA architecture, in a small amount of time. The University of Amsterdam disclaims any warranty and takes no responsibility for damages of any kind for the software based upon this tutorial.

Section 1 will guide you through the installation and compilation of Orbacus 4.0 beta 2. Where section 2 will let you adapt the environmental conditions to let Orbacus flourish. section 3 specifies the Count object in the Interface Definition Language. Where the IDL definition is compiled to C++ in order to set up the server in section 4. The client is implemented in section 5. Whereas the execution of the example is described in section 6.

1.1 Description of files used

File:	Description:
LearnCorba.doc	This tutorial document;
Licence.doc	License agreement for free use of Orbacus;
OB-4_0b2.zip	Orbacus source code for C++;
JOB-4_0b2_jars.zip	Orbacus precompiled .Jar files for Java;
OB-4.0b3.pdf.zip	Documentation file for Orbacus ;
Count.idl	Language independent definition of the object;
Count_i[.cpp .h]	C++ Implementation of the count-object;
Server[.cpp .h]	Code of the C++ Server;
server.*	MS visual C++ project and workspace files;
orbacus.cfg	System Properties for Orbacus for C++;
orbacusVars.bat	Batch file for Sever Variable settings;
Client.*	Code of the Java Client;
orbacus.properties	System Properties for Orbacus for Java;
orbacusClient.bat	Batch file for Client variable settings.

1.2 Applications used

- MS Windows NT
- MS Word
- Acrobat reader
- WinZip
- Microsoft visual C++
- JDK1.2.2.

2 Install Orbacus 4.0 beta 2

The first thing to do, is to install Orbacus. Without it, not much corbaing can be done. The Orbacus files were downloaded from the Object Oriented Concepts site, the implementers of Orbacus. The URL of their site: `http://www.ooc.com/`. This ORB, (Object Request Broker, a Corba implementation) is free for non-commercial use, like in this tutorial. For more details about this, take a look at the file `licence.doc`.

2.1 Installing Orbacus for the C++ server

Supplemented with this package is a `.zip` file called `OB-4_0b2.zip`. This file contains the C++ source code of the Orbacus 4.0 for C++. To properly install Orbacus, we have to unzip, configure and compile the source code. [1]

To Do:	
1	Unzip the file `OB-4_0b2.zip` to a directory of your choice (e.g. `D:\OrbacusSrc`). A sub-directory is created, called `OB-4.0b2`;
2	Edit the file `config\Make.rules.mak` according to the instructions in the comments in that file and make the appropriate changes. At least set the installation directory. For example set it to `prefix = D:\OOCb4`;
3	Open a DOS box, and run the batch file `vcvars32.bat` (included in MSVC) to set compilation environmental variables;
4	In DOS, Go to the Orbacus source directory (`OB-4.0b2`) Compile Orbacus, using `'nmake /f Makefile.mak install'` (this will take some time, for me about 50 minutes, but you should be able to continue with this tutorial until IDL-file compilation). This will install Orbacus in the prefix directory (e.g. `D:\OOCb4`);
5	Recommended, use `'nmake /f Makefile.mak clean'` to delete files used during compilation which are no longer needed (will create some space on your drive).

If all goes well, you should now have a compiled pile of about 75 MB of fresh Orbacus in your installation directory. The source code is no longer necessary, so if youŕe lacking hard drive space you can delete at will.

For more information about installing Orbacus, see the files `INSTALL.WINDOWS`, `OB\README.WINDOWS` and the file `Makefile.mak`.

[1] According to the OOC help file, you need approximately 150 MB of free disk space (250 MB to be sure), and Windows NT (not win95/win98).

2.2 Installing Orbacus for the Java client

The file JOB-4_0b2_jars.zip contains all the pre-compiled .jar files needed to use Orbacus for Java. So, all we need to do is to unzip the Jars and adapt the Classpath variable used by Java to locate .jar files.

To Do:

1	Unzip the file JOB-4_0b2_jars.zip to a directory of your choice (e.g. D:\OOCb4\jars\);
2	Create a batchfile (e.g.OrbacusClient.bat) which adds the OB.jar and the OBNaming.jar to the Classpath variable, and for example contains this line: SET CLASSPATH= %CLASSPATH%;d:\oocb4\jars\ob.jar;d:\oocb4\jars\obnaming.jar;
3	Add to this bath file a line which inserts the executable directory in the search path (e.g. PATH=%PATH%;D:\OOCb4\BIN). For easy access to the executables in this directory.

3 Environmental Changes

The working environment has to be adapted, in order to let Orbacus and Microsoft visual C++ work together. Some variables need to be set, some MSVC settings got to change and a configuration file has to be created.

3.1 Variable settings

In order to let MSVC find Orbacus-libraries and includes, some variables have to be set. For this, the batch file `OrbacusVars.bat` is included in this package. This file:

- Sets a helpful MS-DOS variable `OBC_ROOT`, not needed for Orbacus but makes life easier in MSVC. This variable should contain the path to the installation directory of Orbacus (e.g. `D:\OOCb4`). This, so we can easy reference the path to Orbacus in MSVC;
- Sets MSVC environment variables, (calls `vcvars32.bat`, which is included in the bin directory of MSVC);
- Adds the Orbacus executables (`OBC_ROOT\bin`) in the search path;
- Sets the `ORBACUS_CONFIG` variable, which points to a configuration file used by Orbacus. (this configuration file isn´t standard included with Orbacus, so we have to create it ourselves, more about this in section 3.3).

```
OrbacusVars.bat:

@echo off
if not defined INCLUDE call vcvars32
echo Setting environment for using ORBacus 4.0 beta2 for C++
rem if not already done, set MS-visual C++ vars,
if not defined INCLUDE call vcvars32
rem set root directory to 1e parameter, or to the default
set OBC_ROOT=%1
if not defined OBC_ROOT set OBC_ROOT=D:\OOCb4
rem add Orbacus binaries to search-path
set PATH=%PATH%;%OBC_ROOT%\bin
rem set location of configuration file.
set ORBACUS_CONFIG=%OBC_ROOT%\etc\orbacus.cfg
```

To Do:	
1	Open a DOS box;
2	Call OrbacusVars.bat. Make sure vcvars32.bat is in your path;
3	Start MS Visual C++ (type msdev at the DOS prompt, so the dos-box environment settings (variable values) sustain within MSVC).

3.2 MS visual C++ settings

Microsoft Visual C++ also has to be configured, in order to work with Orbacus:

- The `OBC_ROOT` variable has to be added to the include-path;
- The corresponding libraries have to be added to the project;
- Multithreading DLL has to be set;
- Run Time Type checking has to be enabled.

To Do:	
1	Start an empty workspace and a new `win32 console` project (e.g. `server`)
2	Open the menu project settings;
3	Set `settings for:` to `all configurations` (both debug and release);
4	In tab `C/C++`, category: `Pre processor` Set additional include Directory to: `'., $(OBC_ROOT)\include'` (Yes, include the current directory (`'.'`));
5	In tab `C/C++`, category `code generation` change the setting `use runtime library` set to `Multithreaded DLL`;
6	In tab `C/C++`, category `C++ language` change the setting `Enable Run-time Type Information (RTTI)` to `yes`;
7	Orbacus uses `wsock32.lib` so, add this library to tab `Link`, category `Input` add to the setting `object/library modules` the library `wsock32.lib`;
8	Add to the project, for example in the folder `Resource files` the needed libraries `ob.lib` and `CosNaming.lib`. You can find these files in `OBC_ROOT\lib`.

Details:

The reason for including the current directory in the include directory path, is that Orbacus will generate files, which use the statement: `#include <count.h>` instead of `#include "count.h"`. This does mean that the source files have to be in a system directory of MSVC. By including the current directory, it is added to the system directories. The source files can now be in the same directory as the project files.

Because we compiled Orbacus with the default settings, the compiler setting `Multithreading DLL` was used. We have to keep doing this in our project. To change this default compilation option, change the file `config\Make.rules.mak` and re-compile Orbacus. For more details about multithreading options, refer to the MSVC documentation and the file: `OB\README.WINDOWS`.

Orbacus makes use of the wsock32.lib and uses Run Time Type checking Information.

3.3 Orbacus configuration files

In order to let Orbacus work with the naming service, which allows human object naming, instead of IOR strings, we have to specify some more settings.

The naming service converts an object name to an IOR. In order to let Orbacus find this naming service in its initial references, the IOR of the naming service has to be known. Also, the IP-port on which the naming service will listen has to be set. This can be any free IP-port (e.g. 8001). For an example of an Orbacus configuration file, see below. Remember that the exact IOR of your naming server will differ from the example specified below.

In the `\BIN` directory you can find the executable `Nameserv.exe` corresponding with the naming service. By typing `Nameserv -h` you can see the options. Furthermore, the program `nsadmin.exe` lets you manage the naming service from the commandline. Also, by typing `nsadmin -h` the options of the naming service administration program appear on the screen.

To Do:

1	Print the IOR of the name service listening on port 8001 by typing `Nameserv -i -OAport 8001` (this will automatically start the naming service, use ctrl-c to abort);
2	For the Server: Create a textfile with a filename specified as in the variable `ORBACUS_CONFIG` in the variable settings (e.g. `OBC_ROOT\etc\orbacus.cfg`);
3	Specify the `ooc.service.NameService=` to the IOR just read of the naming service in step 1;
4	Save the file, and make sure it has the same name and path as in the variable `ORBACUS_CONFIG`, and be sure not to break up the IOR, but leave it as a whole string;
5	For the Client: Create a textfile for example `Orbacus.properties` in for example the `java.home\lib` directory (e.g. `C:\Java\JRE\1.2\lib\orbacus.properties`);
6	Set Orbacus as the ORB, insert the following lines in this file: `ooc.omg.CORBA.ORBClass=com.ooc.CORBA.ORB` `ooc.omg.CORBA.ORBSingletonClass=com.ooc.CORBA.ORBSingleton`;
7	Also, specify in this file, the IOR of the name service in `ooc.orb.service.NameService=`. Just like the server configuration.

Two different configuration files (one for C++ and one for Java) have no apparent use on just one machine. When using a network, distributed file locations have their advantages over central stored settings.

The Client configuration file, is just a system property list for Java, which can and will easily be loaded using the method `java.util.Properties.load` (more on this in section 6).

The location of the configuration files is not compulsory, just an easy way to store it.

Example of the Orbacus configuration file for C++ sever:
(Lines starting with # are comment, and ignored)

file: `orbacus.cfg`:

```
# Initial references:
ooc.service.NameService=IOR:010000002a00000049444c3a6f6f632e636f6
d2f436f734e616d696e672f4f424e616d696e67436f6e746578743a312e300000
0002000000000000004c000000010102001600000070632d6373797330392e776
96e732e7576612e6e6c00411f24000000abacab305f526f6f74504f410049494f
504c4f43504f4100004e616d65536572766963650000000010000007c0000000
10102000100000001000006c00000001000000010001000a000000020001000 3
0001000400010005000100060001000700010008000100090001000100010520 0
00100090101000c00000000101000100010002000100030001000400010005 00
0100060001000700010008000100090001000100010520000100
```

Example of the Orbacus configuration file for the Java client:
(Lines starting with # are comment, and ignored)

file: `Orbacus.properties`:

```
# Using ORBacus ORB:
org.omg.CORBA.ORBClass=com.ooc.CORBA.ORB
org.omg.CORBA.ORBSingletonClass=com.ooc.CORBA.ORBSingleton

# set IOR of the nameservice:
ooc.orb.service.NameService=IOR:010000002a00000049444c3a6f6f632e6
36f6d2f436f734e616d696e672f4f424e616d696e67436f6e746578743a312e30
000000020000000000000004c000000010102001600000070632d637379733039 2
e77696e732e7576612e6e6c00411f24000000abacab305f526f6f74504f410049
494f504c4f43504f4100004e616d65536572766963650000000010000007c000
00001010200010000000100000 6c00000001000000010001000a000000020001
00030001000400010005000100060001000700010008000100090001000100010
52000010009010100 0c000000000101000100010002000100030001000400100
0500010006000100070001000800010009000100010001052000 0100
```

4 Specify the object in IDL

The methods and properties of an object have to be known to both the server as the client. The object has to be described in the *Interface Definition Language* (IDL). This description forms the foundation on which the server, client and objects are built. The IDL file (`Count.idl`):

```
module Counter
   { interface Count
           { attribute long sum;
             long increment();
           };
   };
```

The object described, is a Count object. It is a simple object with only 1 method and 1 attribute. An attribute in IDL is not like a real member variable of a class. It generates a read and a write function with the same name of the attribute specified in IDL, which effect the servant objectś attribute. This attribute is a long called sum, which can be incremented using the method increment, which also returns the new value of the attribute.

To Do:	
1	Create a text file `count.idl` and save it in the same directory as the project (this in order to let MSVC find the include files in the current directory, see section 3.2);
2	Add the IDL file to the project;
3	Choose `settings` of the IDL file;
4	Select the settings for `all configurations (both debug and release)`
5	Select `custom build` and add to `command`: `cd /D \$(InputDir)` `$(OBC_ROOT)\bin\idl $(InputPath)`;

6	Add to output:
	`$(InputDir)/$(InputName).h`
	`$(InputDir)/$(InputName).cpp`
	`$(InputDir)/$(InputName)_skel.h`
	`$(InputDir)/$(InputName)_skel.cpp`.

All this is necessary, to compile the IDL file with the Orbacus IDL to C++ compiler (`idl.exe`) within MSVC, and to update the generated files automatically.

5 Setting up the C++ Server

To set up a CORBA server the IDL file has to be compiled, in order to auto-generate Server Skeletons and Client Stubs. Also, the object implementation (servant) and the server itself have to be written.

5.1 Compile IDL file

Compilation of the IDL file with the IDL-compiler as provided with Orbacus (`BIN\idl.exe`) generates a number of files needed to successfully implement object transparency.

To Do:	
1	Compile IDL file, with `compile` button, in MSVC. (note, if you try to build the entire project, the linker will complain that it has nothing to link with);

2	Add following generated files to your project:
	`Count.h`
	`Count.cpp`
	`Count_skel.h`
	`Count_skel.cpp`;

`Count.*` is used for the Client and for the Server. `Count_skel.*` is used only for the server and contains the abstract class of the implementation of the object. `Count.h` contains definitions of the IDL types.

5.2 Implement the servant object

The implementation of the object, which by definition, is done by the server, is called a servant object. The task of the servant is to create the functionality required by the object, as defined in the IDL definition.

To do so, we need to create a class, which inherits from the abstract server class `Count_skel`. In the file `Count_skel.h` we can see the virtual functions which need implementing. A description of these functions:

- To request the value of the property sum: `virtual CORBA::Long sum() ;`
- To set the value of the property sum: `virtual void sum(CORBA::Long _itvar_sum) ;`
- To increment the value of the property sum, and return this value, the method `increment`: `virtual CORBA::Long increment()`.

Now, we need to create the code for the implementation. Conform naming conventions, we create the files `Count_i.h` and `Count_i.cpp`:

To Do:

1	Create a new file in the project directory, `Count_i.h`, and add it to the project;
2	Implement this headerfile. With above functions and a constructor (and facultative a destructor) inherit from the class `public POA_Counter::Count\verb` in the file `Count_Skel.h`. Include a private member to contain the sum-value. (an Example of this file is given below);
3	Create a new file in the project directory, `Count_i.cpp` and add it to the project;
4	Implement this file with the required code as specified in the headerfile. (An example of this file is given below.)

All files are included in the package, so you can either make your own implementation, or use the one supplied.

5.2.1 Example of Count_i.h

```
#include <OB/CORBA.h>
#include "Count_skel.h"
#include "iostream.h"

// inherit from the Server class Count
class Count_i : public POA_Counter::Count
{ public:
     // constructor
     Count_i() ;
     // destructor
     ~Count_i() ;
     // get sum value
     CORBA::Long sum() ;
     // set sum value
     void sum(CORBA::Long _itvar_sum)  ;
     // increment sum value
     CORBA::Long increment() ;
  private:
     long The_Sum ;
};
```

5.2.2 Example of Count_i.cpp

```
#include "Count_i.h"

Count_i::Count_i(){
cout << "Created Count object" << endl ;
this->The_Sum = 0 ;
}

Count_i::~Count_i(){
cout << "Gone is the Count object" << endl ;
}

CORBA::Long Count_i::sum(){
// get value of sum
return this->The_Sum ;
}

void Count_i::sum(CORBA::Long value){
// set value of sum
this->The_Sum = value;
}

CORBA::Long Count_i::increment(){
// increment value of sum by 1
this->The_Sum ++ ;
return this->The_Sum ;
}
```

5.3 Implement the server

To setup the server we have to:

- Initialize the ORB;
- Find a reference to the RootPOA.
  A portable object adapter (POA), provides the mechanism by which a server process maps CORBA objects to language-specific implementation (servants);
- Create the servant object. The servant object is the implementation of the object;
- Bind the servant object to a POA;
- Create and bind a name to the servant object;
  The namingService lets you associate names with objects. The namingService provides the IOR of an object;
- Activate the POA manager;
- Run the ORB.

See the next page, for an example of the server code.

Now, you should be able to build the server code. For running the example, refer to section 7.

5.3.1 Server.cpp code

```
#include <OB/CORBA.h>
#include <OB/CosNaming.h>
#include "Count_i.h"
#include <iostream.h>
int main( int argc, char **argv) {
          // initialise the ORB
     CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
          // finding rootPoa
     CORBA::Object_var poaObject ;
     PortableServer::POA_var poa ;
     poaObject = orb->resolve_initial_references("RootPOA");
     poa = PortableServer::POA::_narrow(poaObject) ;
          // Create CORBA object and use it to register
          // servant object reference with the POA
     Count_i servant ;
     poa->activate_object(&servant) ;
          // Declare naming Context
     CosNaming::NamingContextExt_var rootCxtExt ;
     CORBA::Object_var objVarNaming ;
          // get root NameService
     objVarNaming = orb->resolve_initial_references("NameService");
     rootCxtExt = CosNaming::NamingContextExt::_narrow(objVarNaming) ;
          // add the count name to the root NameService
     CosNaming::Name_var name ;
     name = rootCxtExt->to_name("The counter");
          // get object reference
     CORBA::Object_var c = poa->servant_to_reference(&servant) ;
          // add object reference to counter
     rootCxtExt->bind(name, c);
          // use REbind if the nameservice is already initialized
          // only use bind the first time you bind an object
          // rootCxtExt->rebind(name, c);
          // print IOR of object, just in case...
     CORBA::String_var str = orb->object_to_string(c);
     cout << str << "\n" << endl ;

     cout << "Activating the POA manager.\n" << endl;
     poa->the_POAManager()->activate() ;

     orb->run() ;
          // the ORB will be waiting for requests
     return 0 ;
}
```

6 Setting up the Java Client

To set up a CORBA Client the IDL file has to be compiled, in order to automaticly generate Client Stubs. Also, the client has to be written.

6.1 Compile IDL file

Compilation of the IDL file with the IDL-compiler as provided with Orbacus (`BIN\jidl.exe`) generates a number of files needed to successfully implement object transparency.

To Do:	
1	Compile IDL file, on the command line with `jidl --no-skeletons Count.idl` no skeletons are required, cause we already have a C++ server;

2	A directory `Counter` is created with the following files:
	`_CountStub.java`
	`Count.java`
	`CountHelper.java`
	`CountHolder.java`
	`CountOperations.java`.

6.2 Implement the client

To setup the client we have to:

- Initialize the ORB;
- Find a reference to the naming service;
- Create and use the object.

See the next page, for an example of the client code.

This client can be executed on any machine connected to the server. Just make sure the settings are correct.

Now, you should be able to compile the client code. For running this example, refer to section 7.

6.2.1 Client.java code

```
// importing the used Orbacus packages
 import org.omg.CORBA.* ;
 import org.omg.CosNaming.* ;
 import java.io.* ;
 import Counter.* ;   // importing the compiled idl-code

 public class Client {
   public static void setOrbacusProperties(java.util.Properties props)
     throws java.io.FileNotFoundException, java.io.IOException {
     String javaHome = System.getProperty("java.home");
     File propFile = new File("orbacus.properties");
     if(!propFile.exists())
         propFile = new File(javaHome+File.separator+"lib"+File.separator
                 + "orbacus.properties");
     if(!propFile.exists())
         System.out.println("Cannot find file: orbacus.properties");
     else      {
         FileInputStream fis = new FileInputStream(propFile);
         System.out.println("Loading "+propFile.getPath());
         props.load(fis);
         fis.close();      }
      }   // end setOrbacusProperties()

 public static void main(String args[]) {
 try{
   System.out.println("running client..\n");
         // load properties for Orbacus, and the location of the NamingService
   java.util.Properties props = System.getProperties();
   setOrbacusProperties(props) ;
   System.out.println("init ORB.\n");
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args, props);
        // connect to nameservice, get IOR of nameservice
   System.out.println("\n connecting to nameservice...\n");
   org.omg.CORBA.Object objNaming = orb.resolve_initial_references("NameService");
   NamingContext ncRef = NamingContextHelper.narrow(objNaming);
        // find object 'The counter'
   NameComponent nc = new NameComponent("The counter","");
   NameComponent path[] = {nc};
        // create a counter object
   Count counter = CountHelper.narrow(ncRef.resolve(path));
        // Set sum to initial value of 0
   System.out.println("Setting sum to 0");
   counter.sum((int)0);
        // Calculate Start time
   long startTime = System.currentTimeMillis();
        // Increment 1000 times
   System.out.println("Incrementing");
   for (int i = 0 ; i < 1000 ; i++ ) { counter.increment();     }
        // Calculate stop time; print out statistics
   long stopTime = System.currentTimeMillis();
   System.out.println("Avg = "+ ((stopTime - startTime)/1000f) + " msecs");
   System.out.println("Sum = " + counter.sum());
   } catch (Exception e) { System.out.println("!Exception...: " + e); }
  } // end main()
  }  // end class Client
```

7 Running the example

For running this example, we have to do several things:

- Run the naming service, on the port earlier specified (e.g. 8001);
- Run the Server;
- Run the client.

To Do:

1	Run `nameserv.exe -OAport 8001` in the `OBC_ROOT\BIN` directory;
2	Execute the server project in MSDEV;
3	Compile the `Client.java` by typing `javac Client.java`. Be sure to have set your CLASSPATH correctly;
4	Run the Client with `java Client`.

On the screen of the server something appears like:

```
init ORB.

Created Count object
connecting to namingservice..

IOR:01494f501600000049444c3a436f756e7465722f436f756e743a312e3000a
b3002000000000000004c0000000101025016000000070632d6373797330392e77
696e732e7576612e6e6c00ba0424000000abacab31393535343531373937005f5
26f6f74504f410000cafebabe38f309950000000000000000010000007c000000
0170820001000000010000006c00000001000000010001000a000000020001000
30001000400010005000100060001000700010008000100090001000100010520
000100090101000c00000000010100010001000200010003000100040001000050
001000600010007000100080001000900010001000105200000100
```

On the screen of the Client something appears like:

```
running client..

Loading orbacus.properties

init ORB.

connecting to nameservice...

Setting sum to 0
Incrementing
Avg = 1.188 msecs
Sum = 1000
```

References

[1] R Orfali and D Harkey. *Client/Server Programming with Java and CORBA*. Wiley, second edition, 1998.

ISIS reports

This report is in the series of ISIS technical reports. The series editor is Rein van den Boomgaard (reports-isis@science.uva.nl). Within this series the following titles are available:

References

[1] J.M. Geusebroek, A.W.M. Smeulders, F. Cornelissen, and H. Geerts. Segmentation of tissue architecture by distance graph matching. Technical Report 16, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[2] J.M. Geusebroek, F. Cornelissen, A.W.M. Smeulders, and H. Geerts. Robust autofocusing in microscopy. Technical Report 17, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[3] J.M. Geusebroek, R. van den Boomgaard, A.W.M. Smeulders, and A. Dev. Color and scale: The spatial structure of color images. Technical Report 18, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[4] J.M. Geusebroek. A physical basis for color constancy. Technical Report 19, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[5] J.M. Geusebroek, A.W.M. Smeulders, and R. van den Boomgaard. Color invariance. Technical Report 20, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[6] L. Todoran and M. Worring. Segmentation of color document images. Technical Report 21, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[7] A.W.M. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain. Content based image retrieval at the end of the early years. Technical Report 22, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[8] C.B.J. Bergsma, G.J. Streekstra, A.W.M. Smeulders, and E.M.M. Manders. Velocity estimation of spots in 3d confocal image sequences of living cells. Technical Report 23, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

[9] J.C. van Gemert. Practical tutorial for using corba. Technical Report 24, Intelligent Sensory Information Systems Group, University of Amsterdam, 2000.

You may order copies of the ISIS technical reports from the corresponding author or the series editor. Most of the reports can also be found on the web pages of the ISIS group (http://www.science.uva.nl/research/isis).

Intelligent Sensory Information Systems
University of Amsterdam
The Netherlands