C#, .NET 1.1, .NET 1.0, Java
Posted 27 Aug 2003

Accessing an EJB from .NET Using IIOP.NET: an Example

By Patrik Reali

Accessing an EJB from .NET Using IIOP.NET: an Example

- Download demo project - 173 Kb
- Download IIOP.NET from sourceforge

Introduction

Enterprise Java Beans (EJB) [1] are an established technology for implementing software components on a Java platform; a lot of software exists that relies on them. Many IT key-players are providing application servers for hosting EBJs, in particular IBM's WebSphere, BEA's WebLogic, and the opensource JBoss. EBJs can be used to create distributed object systems, and rely on Java RMI/IIOP to exchange messages; EJBs can also be exposed as Web Services.

Although using Web Services to interoperate is currently trendy, they also have their share of limitations. Web Services are great when it comes to integrate heterogeneous loosely coupled systems, but they have no support for remote object references. In practice, they are stateless and closer to a remote method call than to a distributed object system. Furthermore, SOAP and XML are by no means a compressed format and tend to be quite verbose.

In a previous article [2], I did show how to access a .NET component from Java using IIOP.NET; this article presents the opposite direction: how to access a Java EJB service from a .NET client using the IIOP.NET remoting channel. No modification on the EJB site is required for this purpose.

About IIOP.NET

IIOP.NET [3] is a .NET remoting channel based on the IIOP protocol, the same used by Java's RMI/IIOP [4]. IIOP is part of the CORBA stardard [5]. IIOP.NET acts as an ORB (a CORBA object request broker): it makes objects defined in your .NET application accessible to other remote ORBs, and vice-versa. Java RMI/IIOP implements a subset of the CORBA type system (due to some limitations in Java's type system) and roughly provides the same features as IIOP.NET for the J2EE platform.

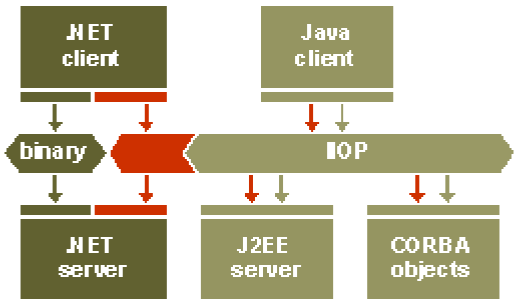


Figure 1: Overview of an IIOP-based distributed object system

Using IIOP.NET is almost as simple as using .NET's built-in remoting. IIOP.NET is an opensource project hosted on sourceforge (http://iiop-net.sourceforge.net/). It was developed by Dominic Ullmann as part of his master thesis at ETH-Z; further work is now sponsored by his current employer ELCA Informatique SA [6], where IIOP.NET is used to let the Java and .NET flavours of its LEAF framework [7] interoperate.

Not surprisingly, IIOP.NET is not the only software you can use for this purpose. The open-source project Remoting.Corba [8] is quite similar in its goals but doesn't support EJBs at the moment, and Janeva [9] from Borland promises to do the same, but is not free.

The Example: an EJB-based Chatroom

To show you how to access an EJB from .NET, we will use a simple non-trivial example: a chat service. The service is an EJB, which allows users to register and unregister a listener for receiving the messages submitted to the chatroom; the EJB manages the list of clients and dispatches the messages to all registered clients. The followig Java interfaces and classes are used to communicate with the service; they will be converted to IDL files to allow access from other CORBA clients.

A `Message` contains the name of the sender and the message itself. The message is mapped to a CORBA valuetype, i.e. an object that is serialized and sent to a remote machine, instead of being remotely accessed.

```
public class Message implements Serializable {
  private String m_originator;
  private String m_msg;

  public Message() { ... }
  public Message(String msg, String originator) { ... }
  public String getMsg() { ... }
  public String getOriginator() { ... }
}
```

The `MessageListener` interface must be implemented by all chatroom clients wishing to receive the chat messages. The interface extends `java.rmi.Remote` because the clients are remote and the communication will be performed with RMI/IIOP.

```
public interface MessageListener extends java.rmi.Remote {
  /*  notify the listener, that a new message has arrived. */
  public void notifyMessage(Message msg) throws java.rmi.RemoteException;
}
```

Finally, the `Chatroom` interface defines the features supported by the stateless bean, which allows sending messages and administrating the list of listeners. The home interface contains only the call to create a new component, and is not shown here.

```
public interface Chatroom extends EJBObject {
  /* post message in chat-room */
  public void broadCast(Message msg)
         throws java.rmi.RemoteException;

  /* register a client, interested in chatroom messages */
  public void registerMe(MessageListener listener, String forUser)
         throws java.rmi.RemoteException, AlreadyRegisteredException;

  /* unregister the client with the name userName. */
  public void unregisterMe(String userName)
         throws java.rmi.RemoteException, NotRegisteredException;
}
```

The chatroom works according to the following UML sequence diagram:

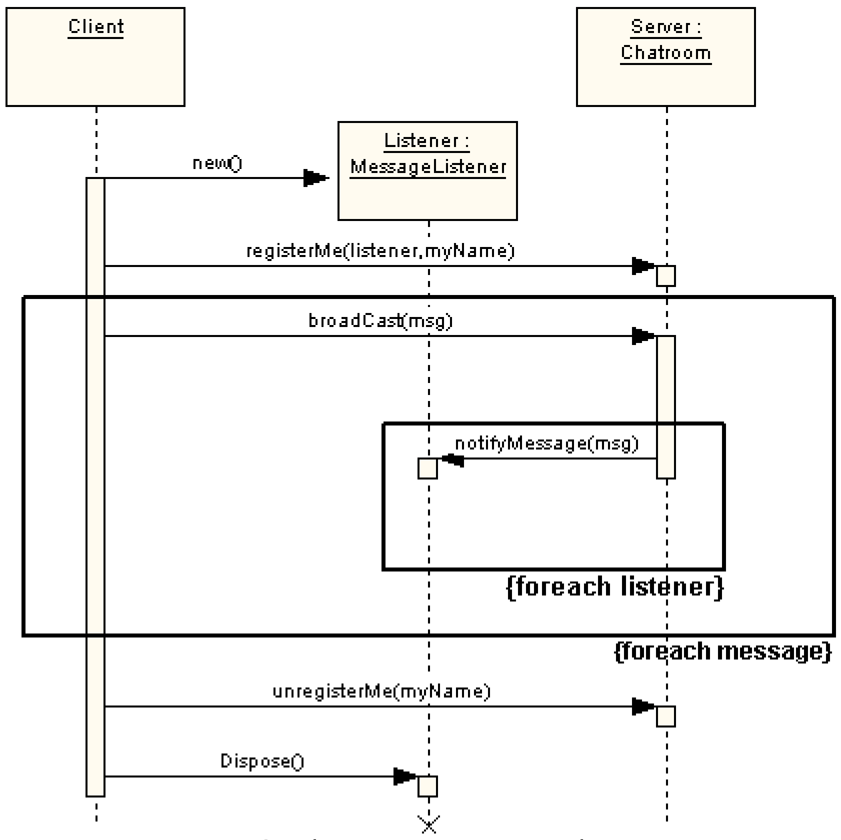


Figure 2: chatroom sequence diagram

In the attached example, the EJB is implemented and configured for WebSphere 5.0 [10]; the IIOP.NET release also contains a version for WebLogic 6.1 [11] , and for JBoss 3.2.1 [12] . Although written for WebSphere, we will try to keep the code in this article as generic as possible, assuming you know your application server well enough to configure the EJB for it.

We not start the realization of the EJB chatroom server and of the .NET client. This requires 6 steps.

Step 1: Install IIOP.NET

To build and execute this example, you need at least a Java SDK 1.3, an application server (we assume IBM WebSphere 5.0), the Microsoft .NET Framework SDK 1.0 or 1.1 [13], the Microsoft J# SDK version 1.0 or 1.1, and IIOP.NET (at least 1.3.1). Installing the first four is out of this article's scope. To install IIOP.NET, download the latest version from sourceforge.

The IIOP.NET project contains a few directories:

- `IIOPChannel` contains the channel code
- `CLSToIDLGenerator` contains a generator to create the IDL definition files from a .NET assembly
- `IDLToCLSCompiler` contains a generator to create CLS files (.NET multimodule assemblies) from the IDL definitions
- `Examples` contains examples and tutorials. The one presented in this article is `EJBChatRoom\WebSphere_5`; the WebLogic version is in `EJBChatRoom\WLS6.1`;the JBoss version is in `Examples\EJBChatRoom\JBoss3.2.1\`

Before building IIOP.NET, copy the files `lib\ir.idl` and `lib\orb.idl` from your Java SDK directory into IIOP.NET's `IDL` directory, and set the environment variable WAS_HOME to the

WebSphere application server directory. Compile everything by executing nmake.

Step 2: Implementing the EJB

Given the previous definitions, the implementation of the EJB is quite straightforward. We create the class Message, and the interfaces MessageListener and Chatroom containing the definitions shown above.

For the bean implementation, three files are provided: the bean home interface is in the interface ChatroomHome, the bean itself in ChatroomBean, and the implementation of the client management in ChatroomServer. The management is realized in a separate singleton class because this must be the same for each bean (whereas each client gets a different bean instance), and the access to the structures must be synchronized; this pattern doesn't follow the recommended EJB scalability practices (avoid synchronization), but a "correct" implementation would be far too complex to fit in such a small example. The bean forwards the calls to the client management class:

```
public void registerMe(MessageListener listener, String forUser) throws AlreadyRegistere
  ChatroomServer server = ChatroomServer.getSingleton();
  server.addClient(listener, forUser);
}

public void unregisterMe(String userName) throws NotRegisteredException {
  ChatroomServer server = ChatroomServer.getSingleton();
  server.removeClient(userName);
}

public void broadCast(Message msg) {
  ChatroomServer server = ChatroomServer.getSingleton();
  MessageListener[] listeners = server.getClients();
  for (int i = 0; i < listeners.length; i++) {
    try {
      listeners[i].notifyMessage(msg);
    } catch (Exception e) {
      System.err.println("error sending msg: " + e);
      System.err.println("--> removing listener");
      server.removeListener(listeners[i]);
    }
  }
}
```

Broadcasting the message is done in the Java bean itself to minimize the time spent in the chatroom (during which the chatroom is locked). Sending the messages can take a long time, in particular when a client is not available. The ideal solution here would be a CORBA one-way asynchronous call, but this is not possible in EJB; the proposed way of implementing this requires using the Java Message Service (JMS) [14], which is out of this article's scope; thus, for the sake of simplicity, we will let each bean send out the messages.

The implementation of ChatroomClient consists in a singleton instance handling the list of clients:

```
public class ChatroomServer {
  private static ChatroomServer s_chatroomServer = new ChatroomServer();
  private Hashtable m_clients = new Hashtable();

  private ChatroomServer() { super(); }

  public static ChatroomServer getSingleton() { return s_chatroomServer; }
  public synchronized void addClient(MessageListener ml, String forUser)
                           throws AlreadyRegisteredException {
    if (!m_clients.containsKey(forUser)) {
      m_clients.put(forUser, ml);
    } else {
      throw new AlreadyRegisteredException("a message listener is already registered for
                                           + forUser);
    }
  }

  public synchronized void removeClient(String forUser) throws NotRegisteredException {
    if (m_clients.containsKey(forUser)) {
      m_clients.remove(forUser);
    } else {
      throw new NotRegisteredException("no message listener registered for the user: " +
    }
  }

  public synchronized void removeListener(MessageListener listener) {
    m_clients.values().remove(listener);
  }

  public synchronized MessageListener[] getClients() {
    MessageListener[] result = new MessageListener[m_clients.size()];
    result = (MessageListener[])(m_clients.values().toArray(result)); return result;
  }
}
```

Note the `getClients()` method implementation, which returns a copy of the listener list to avoid synchronization issues.

Step 3: Generating the IDL files

Once the service is implemented, the next step is the generation of the IDL files, which describe the service interface using the CORBA model.

Each application server provides its own way to generate the IDL, because each application server works with different versions of the EJB specifications, which in turn have different interfaces.

Refer to your application server documentation for the exact procedure to generate the IDL files.

Step 4: Generating the CLS modules from the IDL files

From the IDL files, the `IDLToCLSCompiler` tool generates a CLS multimodule assembly containing the classes and interfaces needed to access the EJB-based service.

Why does the generator create a netmodule instead of a C# stub? Well, there are a few reasons, but the most important ones are simplicity and portability. First, netmodules are quite easy to generate using the reflection-emit interface of .NET; generating source code would require some pretty-printing algorithm. Second, the netmodules contain the definitions in the CLS form, which is understood by all .NET-compliant languages, thus it doesn't matter whether you code is in C# or Visual Basic (or even Oberon).

Invoke the generator specifying the output directory (-o) and the idl files to use.

```
IDLToCLSCompiler.exe -o ..\bin chatroom
        ch\elca\iiop\demo\ejbChatroom\Chatroom.idl
        ch\elca\iiop\demo\ejbChatroom\ChatroomHome.idl
```

The generator will remind you that you must provide the implementation for some classes: these are the classes implementing the CORBA valuetypes; in .NET these classes are defined with the `Serializable` attribute (do not confuse CORBA's valuetypes with .NET valuetype class). The content of these classes is cloned to the target system, thus the methods they provide must also be available on the remote system. Because the IDL contains no code, you have to provide this code (don't worry! this effort is usually limited to the class constructor).

In our example, the classes to be implemented are the `NotRegisteredException`, `AlreadyRegisteredException`, and `Message` defined by our service; and `Throwable`, `_Exception`, `CreateException`, `RemoveException` (defined by Java, RMI, or the EJB). Some EJBs may require the definition of additional classes.

Our implementation of those classes is in `ExceptionImpl.cs` and `MessageImpl.cs`.

```
using System;
namespace ch.elca.iiop.demo.ejbChatroom {
  ///<SUMMARY>/// Implementation of the CORBA value type Message /// </SUMMARY>

  [Serializable] public class MessageImpl : Message {
    public MessageImpl() { }
    public MessageImpl(string originator, string msg) {
      m_originator = originator;
      m_msg = msg;
    }
    public override string fromUser {
      get { return m_originator; }
    }
    public override string msg {
      get { return m_msg; }
    }
  }
}
```

Keep in mind that IIOP.NET will search for class `*Class*Impl` as implementation for CORBA valuetype `*Class*`. Thus, the classes to provide are named `NotRegisteredExceptionImpl`, `AlreadyRegisteredExceptionImpl`, and so on.

Step 5: Implementing the C# client

The client provides a user interface to collect the user's messages, invoke the service, and display the information sent by the service. A simple GUI is used in the example; independently of the user interface, there are a few important things to do. First, register the IIOP.NET channel, connect to the EJB, and get an instance of the service.

```
// register IIOP.NET channel
IiopChannel channel = new IiopChannel(callbackPort);
ChannelServices.RegisterChannel(channel);

// get the naming service
RmiIiopInit init = new RmiIiopInit(ejbNameServiceHost, ejbNameServicePort);
NamingContext nameService = (NamingContext)init.GetService("NameServiceServerRoot");

NameComponent[] name = new NameComponent[] { new NameComponent("demo", ""),
                                             new NameComponent("chatroomHome", "") };

// get the chatroom home interface
ChatroomHome home = (ChatroomHome) nameService.resolve(name);
Chatroom chatroom = home.create();
```

The `ejbNameServiceHost` and `ejbNameServicePort`, and the component name heavily depend on the application server and the way the service is configured and registered there.

To be able to receive messages, the client must register a listener, i.e. a remotable object implementing the MessageListener interface.

```
m_listener = new MessageListenerImpl(m_usernameTextbox.Text, this);
m_chatroom.registerMe(m_listener, m_listener.userName);
```

Now the chatroom is ready to be used. Sending a message to the room is simple:

```
MessageImpl msg = new MessageImpl(m_listener.userName, m_messageTextbox.Text);
try {
  m_chatroom.broadCast(msg);
} catch (Exception ex) {
  Console.WriteLine("exception encountered, while broadcasting: " + ex);
  MessageBox.Show("an exception occured, while broadcasting!");
}
```

This code performs a synchronous method invocation, i.e. it waits for the server to complete the broadcast and return. However, this is not mandatory, as the call returns no result and the client has no need to synchronize with the server. Thus, an asynchronous invocation is also possible:

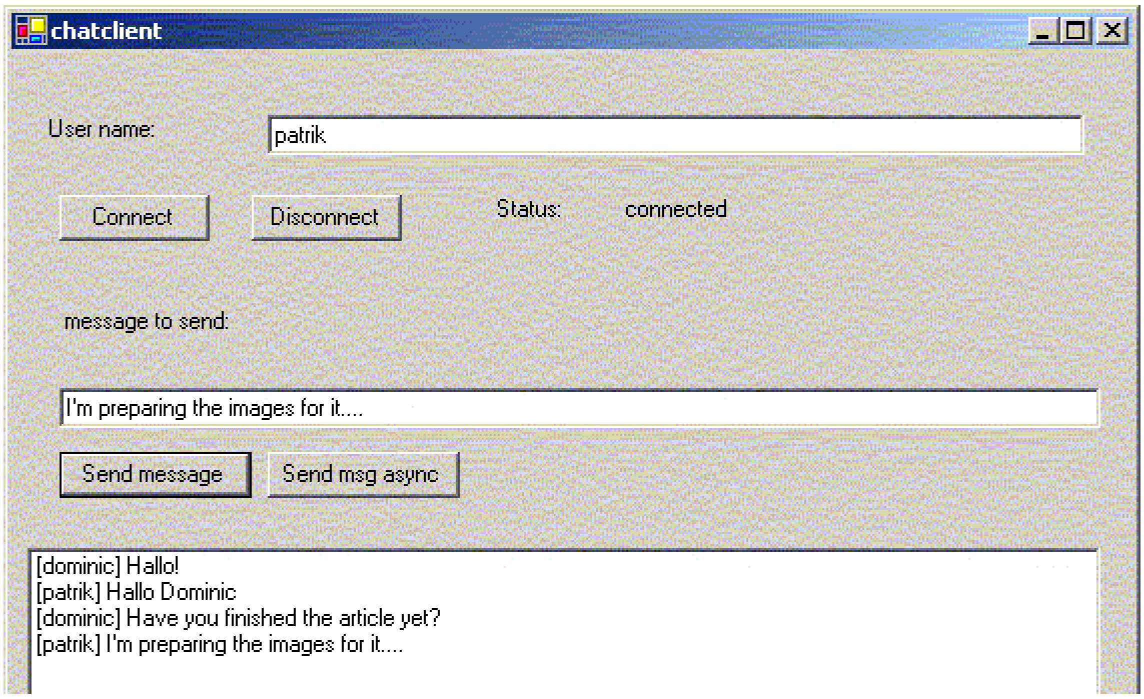
```
delegate void BroadCastDelegate(Message msg);

MessageImpl msg = new MessageImpl(m_listener.userName, m_messageTextbox.Text);
try {
  BroadCastDelegate bcd = new BroadCastDelegate(m_chatroom.broadCast);
  // async call to broadcast
  bcd.BeginInvoke(msg, null, null);
  // do not wait for response
} catch (Exception ex) {
  Console.WriteLine("exception encountered, while broadcasting: " + ex);
  MessageBox.Show("an exception occured, while broadcasting!");
}
```

The listener's notifyMessage() method will be called by the EJB service whenever a new message is available. This method implements the processing of incoming calls by the client.

Step 6: Run the example

Running the example is the last step. On the server side, the Makefile for the Websphere-hosted EJB service automatically registers it to the application server, so you just need to start the server (in case it is not already running).

On the client side, launch the application; then set your name, connect to the server, and you will be able to send and receive messages.



patrik

connected

I'm preparing the images for it....

[dominic] Hallo!
[patrik] Hallo Dominic
[dominic] Have you finished the article yet?
[patrik] I'm preparing the images for it....

Figure 3: the .NET Client UI

Problems and Pitfalls

The presented example is conceptually simple. Nevertheless, a few pitfalls may ruin your day and prevent the code from running. Most problems are related to the various application servers and their cumbersome configuration (many errors show only at execution-time, causing a tedious and time-consuming try-and-correct loop).

In IIOP.NET, CORBA valuetypes prevent the fully automated generation of proxies. When invoking a method on a remote object, the call is forwarded to the machine holding the object instance by the remoting infrastructure. Valuetypes are cloned and all invocations are processed locally, thus their implementation must also be available locally. Because the IDL contains solely the interface definitions, the implementation must be provided by hand. This is not needed when remoting from .NET to .NET: the DLL containing the definitions also contains the implementation of the classes.

Synchronization is another problem specific to this example. When using the graphical user interface, each operation (i.e. clicking on a button) locks the whole frame during the execution of the associated code. When sending a message to the server, the server will obviously forward the same message to the client, whereas the incoming message arrives in a different thread, and the frame lock prevents it from being delivered (causing a classical deadlock). Various solutions are possible: special handling of own messages is one approach, asynchronous calls are the other one: asynchronous calls can be made on the client side while sending or receiving the message, and on the server side while dispatching the message. In this examples we implemented asynchronous calls on message reception; furthermore, the client allows choosing between synchronous and asynchrounous message sending.

Conclusions

This article presents an example in which the IIOP.NET channel is used to provide access to a Java EJB from a .NET-based client. The various steps involved are presented and detailed. The IIOP protocol used to access the object is well established and quite efficient compared to the fancier and less powerful Web Services. The example itself is quite elaborate: it includes the .NET and J2EE platforms, and the CORBA technology to allow them to interoperate.

The development of such an application includes some pitfalls, often caused by the complex and non-trivial configuration and use of the various application servers hosting the EJB services.In particular, the service naming and location may heavily differ among the various servers.

Interacting with the various application servers was quite an eye-opener: we discovered many different interpretations of the same IIOP specifications, far more than we could imagine in our wildest dreams! A big fat thank goes here to the fast-growing IIOP.NET community, which helped us testing the channel on the many different servers in just a few weeks time.

Nevertheless, IIOP.NET has been tested with the most common application servers; code and configuration examples are available for WebSphere 5, WebLogic 6.1, and JBoss 3.2.1.

Links and References

[1] Java EJB
http://java.sun.com/products/ejb/

[2] Patrik Reali; Building a distributed object system with .NET and J2EE using IIOP.NET; July 2003; The Code Project
http://www.codeproject.com/csharp/dist_object_system.asp

[3] IIOP.NET
http://iiop-net.sourceforge.net/

[4] Java RMI/IIOP
http://java.sun.com/products/rmi-iiop/

[5] CORBA
http://www.corba.org/

[6] ELCA Informatique SA
http://www.elca.ch/

[7] ELCA LEAF and LEAF.NET
http://www.elca.ch/Home/Solutions/Technology_Frameworks/LEAF/index.php

[8] Remoting.Corba
http://remoting-corba.sourceforge.net/

[9] Borland Janeva
http://www.borland.com/janeva/

[10] IBM WebSphere Application Server
http://www.ibm.com/software/webservers/appserv/

[11] BEA WebLogic
http://www.bea.com/products/weblogic/server/

[12] JBoss
http://www.jboss.org/

[13] Microsoft .NET Framework
http://msdn.microsoft.com/netframework/

[14] Java JMS
http://java.sun.com/products/jms/

About Patrik Reali

Patrik Reali is a senior software engineer for ELCA Informatique (http://www.elca.ch/) in Zurich, Switzerland. He is specialized in systems and programming languages, with a strong interest for the .NET and Java platforms.

He got a PhD in computer science at ETH-Zurich working on the research operating system and language Oberon.

Together with Dominic Ullmann, he administers the open source project IIOP.NET.

Click here to view Patrik Reali's online profile.

All Topics, C#, .NET >> C# Programming >> General
Updated: 27 Aug 2003