

Connecting Microsoft Mobile Devices to Java Infrastructures

Introduction

If the task at hand is to connect a Pocket PC running the .Net Compact Framework to a Java back-end, and if Web Services are ruled out as an interoperability solution, there are not many viable options available. The one presented in this article may well be the only one.

This solution requires Middsol's MinCor.NET. The product is an object request broker for the Compact Framework written in C#. It supports Windows Mobile, Windows XP Embedded and Windows CE .NET.

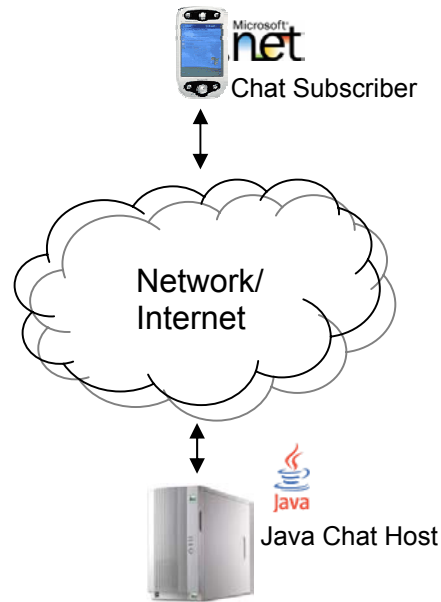
Requirements

A chat application is a good example for an interoperability scenario, because it does not require much business logic, while it still allows for some interesting features at the communication level. In this example, an existing Java server implements and exposes a basic chat service through RMI. A mobile client, to be written in .Net managed code, will consume the chat service. Chat subscribers must specify a user id and their favorite hobby. With the Naming Service's IP address, a subscriber can get a connection to the chat host and send messages that all subscribers will see on their displays. The example will demonstrate how a .Net developer who is unfamiliar with Java distributed object technology can

- generate an IDL file from a JAR (Java Archive) file that describes the interfaces in platform-independent terms
- generate .Net remote object stub code from the IDL (Interface Definition Language) file
- write a .Net application that consumes services provided by remote Java objects
- expose methods in .Net code that will subsequently be invoked by Java objects

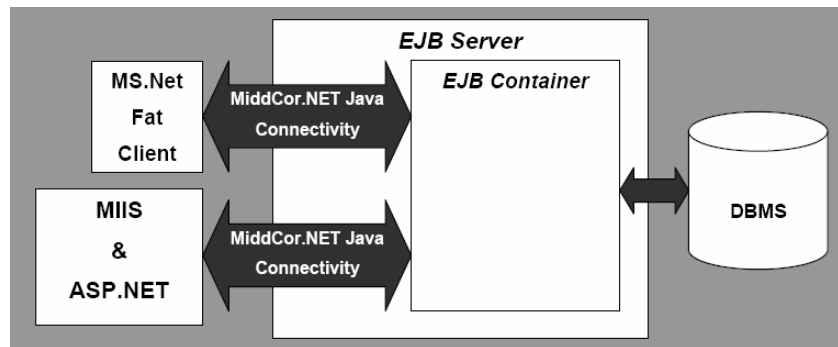
In order to compile and run the example files, the following software is necessary:

- Visual Studio .Net 2003
- Sun Java Development Kit Standard Edition 1.4
- Middsol MinCor.Net Evaluation License
- Microsoft ActiveSync (optional, for deployment to a Pocket PC device)



Alphabet Soup: CORBA, RMI, IIOP

Because RMI (Remote Method Invocation, Java’s own standard for distributed object interoperability) is based on the CORBA/IIOP protocol, it is possible to connect to an RMI server from any CORBA client, including a mobile .Net application. On the Java side, this can be accomplished with regular Java classes that implement the `java.rmi.Remote` interface or by writing Enterprise Beans (EJBs). Today’s application servers even expose JNDI (Java Naming and Directory Interface) through a CORBA naming service provider, which can be used by the .Net client to locate the host.



Step 1: Generate the IDL

Because one of the assumptions was that the server code already exists, it will (for the most part) be considered a black box. The chat interfaces must be exposed through RMI, of course:

```
public interface Host extends Remote
{
    java.util.Hashtable signOn( SubscriberAccount newSubscriber)
        throws RemoteException, Exception;

    void signOff( String name)
        throws RemoteException;

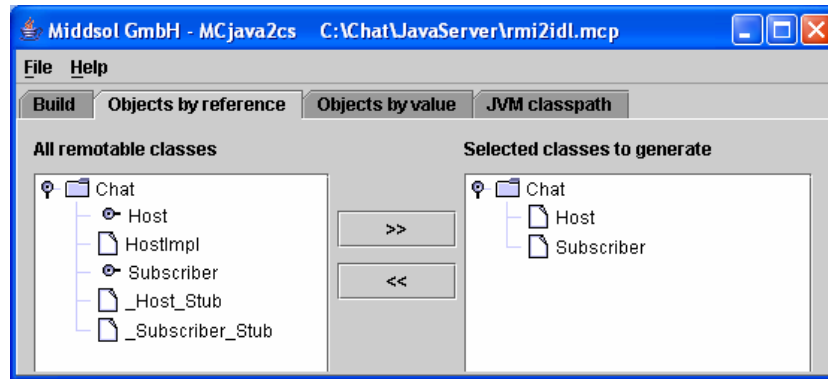
    void sendMessage( String senderName, String message)
        throws RemoteException;
}

public interface Subscriber extends Remote
{
    void displayMessage( String senderName, String message)
        throws RemoteException;
}
```

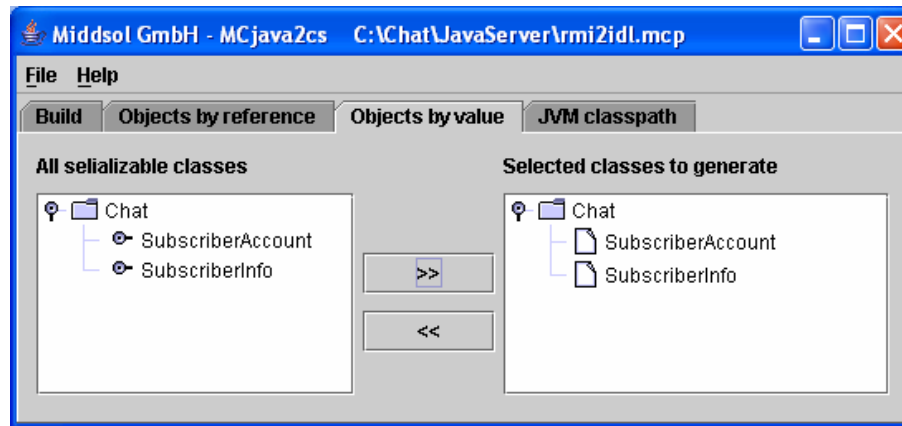
The IDL file is the glue between the host and the subscriber code. It contains a generic description of the calling interfaces with abstract data types that can be mapped to the actual data types of the target platform. In the chat example, the subscriber and the host will both act as a server and a client. Please note that the term ‘subscriber’ better describes the part of the application that runs on the mobile device than ‘client’ because of the potential confusion with the client at the communication layer, which can be either side. The subscriber will initialize the session by sending a registration packet to the host, which the host will acknowledge by returning a list of subscribers. Once registered, the client will then send messages to the host as needed. The host will call other subscribers to distribute the message to all participating devices.

When a subscriber wishes to deregister, he must call the host again.

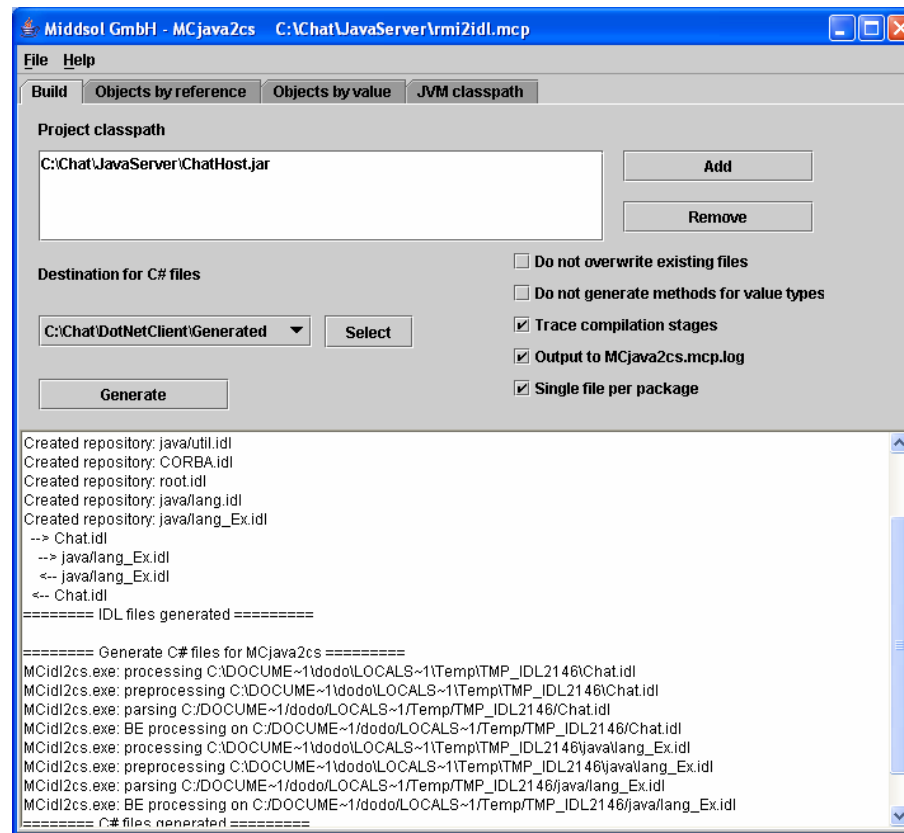
MinCor comes with a utility called *MCjava2cs.exe*. This little program will do most of the glue work for you. It opens existing JAR files (Java Archive files) and lists all the RMI and EJB interfaces found for selection:



For the screen shot above, the tool was pointed to the ChatHost.jar of the example. The `Host` and `Subscriber` interfaces were selected. *MCjava2cs* further lists serializable Classes (passed by value) for selection:



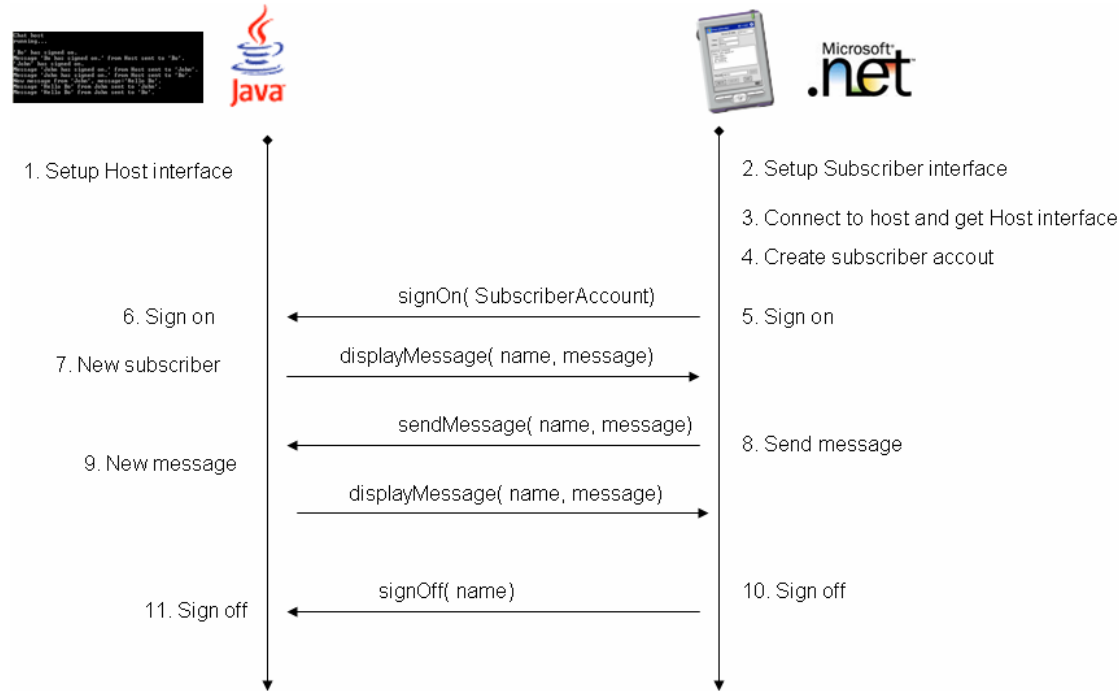
The example contains two classes of this type that were both selected above. The last step is to run the generation process which creates C# code for the selected interfaces:



The tool actually performs two distinct tasks. First, it extracts the IDL definition from the Java code. Second, it generates C# code based on the IDL generated during step 1. The IDL file is transient because it is not needed once the C# code has been generated. The tool created a file named *host.cs*, based on the name space of the Java interface. It contains the host class skeleton code and some helper methods. If this sounds like something you don't want know, you can relax. *MCjava2cs.exe* allows you to write CORBA code without any knowledge of IDL syntax and IDL types.

When I used the tool, I experienced problems with JVM version 1.4.2_04-b04, but it worked with JVM 1.4.2_05-b04. Also, make sure that there are no spaces in the path to the JAR file and the C# output path. These issues will likely be fixed by the time you download the trial software.

To better understand the purpose of the methods that the Java host makes available, it is beneficial to take a detailed look at the communication between the host and a subscriber.



In the sequence diagram above, the host interface exposes three methods that will be called by subscribers: `signOn()`, `signOff()` and `sendMessage()`. The method `signOn()` accepts one parameter (`subscriberAccount`) and returns a `HashTable` with all subscriber names. `sendMessage()` is called by the subscriber to send a message and requires the sender's name and the message as parameters. `signOff()` is called by the subscriber to end the chat session. Since the subscriber name is used as a key, it must be unique.

The subscriber, on the other hand, must allow the host to update the display, by exposing its interface to the host for a method call: `displayMessage()` requires two parameters – the name of the sender and the message itself.

The class `subscriberAccount` is known to both the host and the subscriber and is passed by value. It holds information about a subscriber's name, hobby and `Subscriber` interface. The latter must be exposed to the host so that the host can call the subscriber's `displayMessage()` method. The Java implementation looks as follows:

```

public class SubscriberAccount implements Serializable
{
    private String      name;
    private String      hobby;
    private Subscriber  subscriber;
  }
  
```

```

public SubscriberAccount( String _name, String _hobby, Subscriber _subscriber)
{
    name          = _name;
    hobby         = _hobby;
    subscriber    = _subscriber;
}

public String queryName()
{
    return name;
}

public String queryHobby()
{
    return hobby;
}

public Subscriber queryInterface()
{
    return subscriber;
}
}

```

The Client Project

With the server in place and the stub code already generated, what's left to do is to write the subscriber presentation layer (*MainDialog.cs*), the implementation of the Subscriber interface (*SubscriberImpl.cs*) and the code for application initialization and connection management (*Connection.cs*). The code for the client user interface is of limited interest in the context of this article, with one exception. In the .Net Compact Framework, only the main thread of an application is allowed to access UI controls directly. Because of this restriction, the host cannot simply invoke the method `displayMessage()` to write to the subscriber's display (the ORB is multithreaded and `displayMessage()` runs as a separate thread). The main dialog's interface `AsyncUIAccess` provides a workaround. The implementation of the method `writeLog()` writes to a queue. A timer empties the queue on a regular basis and writes all content to the display. This interface is defined in *MainDialog.cs*. *FrmChatSubscriber* implements the `writelog()` method:

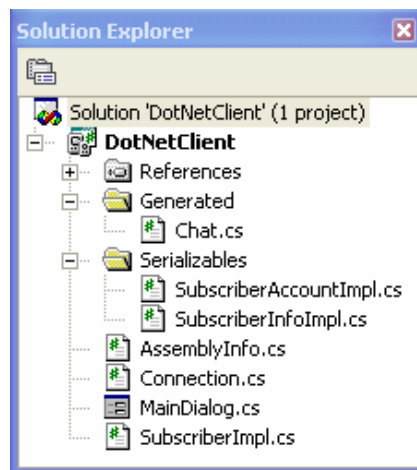
```

public interface AsyncUIAccess
{
    void writeLog( string a_strMsg);
}

```

The rest of the presentation code should be straight-forward to read and understand and will not be discussed here.

Instead, the article will walk through the Subscriber interface and the Connection class that provides a wrapper for the remote objects and manages the connection.



The folder /Serializables in the Visual Studio project helps to organize the solution. It contains the implementations of all serializable CORBA classes, or CORBA *valuetypes*. In this case, they are `SubscriberAccountImpl` and `SubscriberInfoImpl`. They are derived from `Chat.SubscriberAccount` (`SubscriberInfo`, respectively), which are generated classes in `Chat.cs`. These classes are used to pass subscriber information between the client and the server. `SubscriberInfo` does not expose the subscriber's interface, which is only needed by the host, and can therefore be safely passed to other clients. The suffix `Impl` is a naming convention that serves as a reminder of this fact.

The generated parent class already defines the properties and has abstract getter- and setter-method declarations. Therefore, the implementation is rather simple:

```
namespace Chat
{
    public class SubscriberAccountImpl: Chat.SubscriberAccount
    {
        public SubscriberAccountImpl()
        {
        }

        public SubscriberAccountImpl( string _name, string _hobby,
                                     Subscriber _subscriber)
        {
            name = _name;
            hobby = _hobby;
            subscriber = _subscriber;
        }
    }
}
```

```

public override string queryName( )
{
    return name;
}

public override string queryHobby( )
{
    return hobby;
}

public override Chat.Subscriber queryInterface( )
{
    return subscriber;
}
}
}

```

The implementation of the subscriber interface exposes one method, `displayMessage()`, that the host can call.

```

namespace DotNetClient
{
    public class SubscriberImpl: Chat.SubscriberPOA
    {
        private AsyncUIAccess m_oUserInterface;

        public SubscriberImpl( AsyncUIAccess a_oUserInterface)
        {
            m_oUserInterface = a_oUserInterface;
        }

        public override void displayMessage( string a_strName, string a_strMsg)
        {
            m_oUserInterface.writeLog( "<" + a_strName + "> '" + a_strMsg + "'");
        }
    }
}

```

`AsyncUIAccess` is the implementation of the display queue for the timer that was mentioned earlier.

`SubscriberPOA`, the parent of the `SubscriberImpl`, is also a generated class. POA stands for Portable Object Adapter. When a client invokes a server object, the POA helps the request broker to activate the appropriate object and to deliver requests to it.

The last module on the client side that deserves attention is `Connection.cs`. This class contains the methods that manage the communication between subscriber and host. It maintains a reference to the host and to its own interface. It also instantiates an object request broker instance (ORB).

```
public class Connection
{
    private string[] m_strORBInit =
        {"-ORBInitRef NameService=corbaloc:iiop:1.2@<ServerAddress>:1050/NameService",
        "-ORBDebug",
        "-AddAssembly DotNetClient.exe, JavaBuiltinTypesCF.dll, MinCorCF.dll"
        };

    private Chat.Host m_oChatHost;
    private Chat.SubscriberAccount m_oAccount;
    private Midsol.CORBA.ORB m_oOrb;
```

`signOn()` is called by the user interface when the client wants to connect to a server.

```
public void signOn( AsyncUIAccess a_oAsyncUIAccess, string a_strIpAddr,
    string a_strName, string a_strHobby)
{
    // Instantiate local ORB, find remote ORB and connect
    initializeConnection( a_oAsyncUIAccess, a_strIpAddr);

    // Instantiate local Subscriber interface
    Chat.Subscriber oSubscriber = new SubscriberImpl( a_oAsyncUIAccess)._this();

    // create local subscriber's account object
    m_oAccount = new Chat.SubscriberAccountImpl( a_strName, a_strHobby, oSubscriber);

    try
    {
        // sign on and display returned list of participants
        displayListOfSubscribers( a_oAsyncUIAccess, m_oChatHost.signOn( m_oAccount));
    }
    catch( Midsol.java.lang.Ex ex)
    {
        a_oAsyncUIAccess.writeLog("+++ Sign-on failed. +++");
        a_oAsyncUIAccess.writeLog("+++ Please try another name.+++");
        deinitializeConnection();
    }
}
```

signOff() calls this very method on the host in order to remove this instance from the list of subscribers.

```
public void signOff( )
{
    if( m_oOrb != null)
    {
        m_oChatHost.signOff( m_oAccount.queryName());

        deinitializeConnection();
    }
}
```

sendMessage() calls this very method on the host.

```
public void sendMessage( string a_strMessage)
{
    m_oChatHost.sendMessage( m_oAccount.queryName(), a_strMessage);
}
```

The initialize() method initializes the ORB.

```
private void initializeConnection( AsyncUIAccess a_oAsyncUIAccess, string a_strIpAddr)
{
    try
    {
        //Initialize the CORBA framework
        initCORBA( a_oAsyncUIAccess, a_strIpAddr);

        connectToHost( a_oAsyncUIAccess);
    }
    catch(Middsol.CosNaming.NamingContextPackage.NotFound ex)
    {
        //Name service not running or not accessible
        a_oAsyncUIAccess.writeLog( "NS Manager Error:" + ex.why);
        throw new System.Exception();
    }
    catch( Middsol.CORBA.SystemException exSys)
    {
        a_oAsyncUIAccess.writeLog( "Server exception caught (Middsol.CORBA.SystemException).");
        a_oAsyncUIAccess.writeLog( "ID:" + exSys.ID);
        deinitializeConnection();
    }
}
```

`initCORBA()` contains standard CORBA initialization code. It uses the .Net DNS API to get the local device's IP address, instantiates a new ORB, retrieves a reference to the newly created root portable object adapter (POA) and "narrows" it to the correct type (`Middsol.PortableServer.POA`). Finally, it activates the POA that has been in a holding state.

```
private void initCORBA( AsyncUIAccess a_oAsyncUIAccess, string a_strIpAddr)
{
    m_strORBInit[0] = m_strORBInit[0].Replace("<ServerAddress>", a_strIpAddr);

    m_oOrb = Middsol.CORBA._ORB.init( m_strORBInit, null);

    Middsol.PortableServer.POA oRootPOA =
        Middsol.PortableServer.POAHelper.narrow( m_oOrb.resolve_initial_references( "RootPOA" ));
    oRootPOA.the_POAManager.activate();
}
```

`deinitializeConnection()` destroys the ORB.

```
private void deinitializeConnection()
{
    if( m_oOrb != null)
    {
        m_oOrb.destroy();
    }
    m_oOrb = null;
}
```

`connectToHost()` uses the Java-Naming Services to locate the host. It requires an IP address, connects to the name server on this host and requests the location of a server called "ChatHost". `AsyncUIAccess`, while not needed by some of the method, is passed anyways to enable future improvements.

```
private void connectToHost( AsyncUIAccess a_oAsyncUIAccess)
{
    Middsol.CosNaming.NamingContextExt oNsCtx =
        Middsol.CosNaming.NamingContextExtHelper.narrow(
            m_oOrb.resolve_initial_references("NameService"));

    m_oChatHost = Chat.HostHelper.narrow( oNsCtx.resolve_str("ChatHost"));
    a_oAsyncUIAccess.writeLog( "Chat host connected");
}
```

`displayListOfSubscribers()` iterates through the list of subscribers that the host returned and writes each entry to the list box on the screen.

```
private void displayListOfSubscribers( AsyncUIAccess a_oAsyncUIAccess,
                                     System.Collections.Hashtable a_oSubscriberList)
{
    System.Collections.IDictionaryEnumerator enumerator = a_oSubscriberList.GetEnumerator();

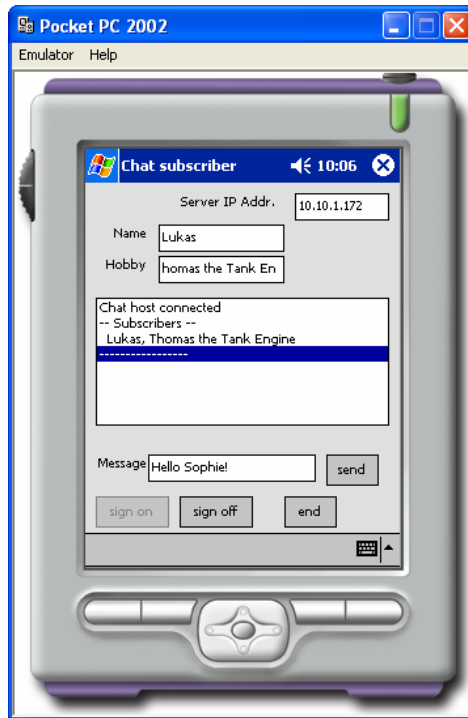
    a_oAsyncUIAccess.writeLog("-- Subscribers --");

    while( enumerator.MoveNext() )
    {
        Chat.SubscriberInfo oInfo = (Chat.SubscriberInfo)enumerator.Value;

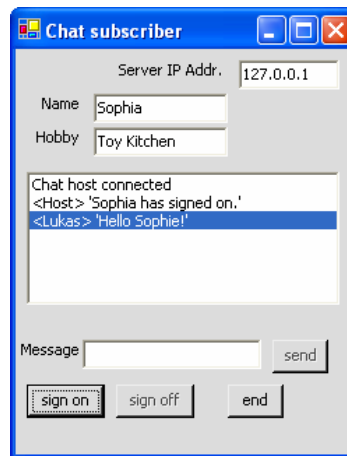
        a_oAsyncUIAccess.writeLog("  " + oInfo.queryName() + ", " + oInfo.queryHobby());
    }
    a_oAsyncUIAccess.writeLog("-----");
}
```

Deploy and Test

Once the solution files are downloaded, unpacked, and loaded into Visual Studio, the binaries for the subscriber can be built and tested. The host requires the Sun naming Service (orbd.exe) to be running. The subscriber client can be tested without a mobile device. In order to run the client in debug mode, though, it must either be deployed to a device or to the Pocket PC emulator (see screenshot below). Visual Studio will prompt you for a target device and if everything was configured properly, the emulator will be one of the choices. Unfortunately, it does not support the callback functionality, which means that the `displayMessage()` method will not work and eventually time out. In the example below, the user Lukas is running the emulator and he does not see any messages, neither his own nor from Sophia who is running the subscriber natively.



The client executable runs as a Windows application, too, which makes it very easy to test.



```
C:\SunAppServer\jdk\bin\java.exe
Chat host
running...

'Lukas' has signed on.
cannot distribute message to Lukas
'Sophia' has signed on.
Message 'Sophia has signed on.' from Host sent to 'Sophia'.
cannot distribute message to Lukas
New message from 'Lukas', message:'Hello Sophie!'.
Message 'Hello Sophie!' from Lukas sent to 'Sophia'.
cannot distribute message to Lukas
```

With these restrictions in mind, use the following steps to build and test the applications:

- Run the supplied *buildChatHost.bat* to compile the host sources
This batch creates the class files, RMI stubs, the jar file. It also builds the IDL and calls *MCidl2cs* to generate the .Net code. It is not necessary to run *MCjava2cs.exe*.
- Load the subscriber solution into Visual Studio and build the binaries (after fixing the references to the Middsol libraries)
- Start the Sun ORB demon
`orbd.exe -ORBInitialPort 1050`, or use the supplied batch file *startNameService.bat*
- Run *startChatHost.bat* to start the host service:
`start java -cp .;ChatHost.jar -
Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNctxFactory -
Djava.naming.provider.url=iiop://localhost:1050 ChatHost`
- Start the subscriber executable (*DotNetClient.exe*)
- Type in the Naming Service's IP address, subscriber name and a hobby
- Click on the sign-on button

Please feel free to improve the code and direct all questions or comments to the author at (pubs2004@donners.com).

Conclusion

This article demonstrated that it is possible to integrate with a Java back-end from a .Net application, specifically from the Compact Framework, with very little knowledge of the technical details of the middleware. Middsol's MinCor.NET, although a young product and still a little rough around the edges, is a powerful tool that connects your .Net application to a vast selection of CORBA-enabled targets, including J2EE Enterprise beans and RMI objects, without any further tools.

About the Author

Christian Donner is a Senior Consultant and Application Architect with Molecular Inc., a premier technology consulting firm based in Watertown, Massachusetts. When he does not tinker with mobile technology, he designs and develops enterprise-level web solutions for Fortune 1000 clients.

Resources

Introduction to Development Tools for Windows Mobile-based Pocket PCs and Smartphones (recommended!)

<http://msdn.microsoft.com/mobility/windowsmobile/default.aspx?pull=/library/en-us/dnppcgen/html/devtoolsmobileapps.asp>

Fundamentals of Microsoft .NET Compact Framework Development for the Microsoft .NET Framework Developer

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetcomp/html/net_vs_netcf.asp

Download the MinCor.Net evaluation copy

www.middsol.com