Connecting Microsoft Mobile Devices to Java Infrastructures

Integrating a .NET Compact Framework application with a Java back end

BY CHRISTIAN DONNER

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, but 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 do the following:

- 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 Interface Definition Language (IDL) 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

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 Remote Method Invocation (RMI; 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; see Figure 1). Today's application servers even expose Java Naming and Directory Interfaces (JNDIs) through a CORBA naming service provider, which can be used by the .NET client to locate the host.

Because one of the assumptions was that the server code already exists, it will (for the most part) be considered a black box. The Java chat interface must be exposed through RMI, of course (see Listing 1; the listings and source code are online at www.sys-con.com/dotnet/source.cfm).

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 act both 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." This is 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 subscriber then sends messages to the host as needed. The host calls other subscribers to distribute the message to all participating devices. When a subscriber wishes to deregister, he

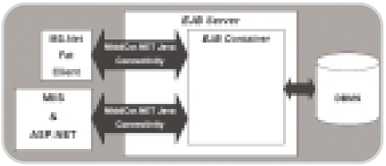


Figure 1: *EJB container and remote connections*

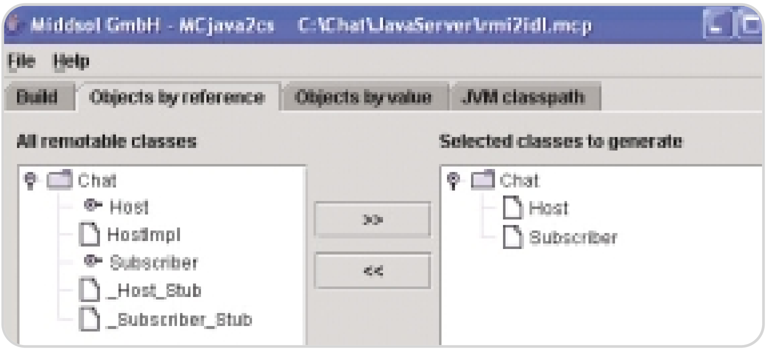


Figure 2: *MCjava2cs.exe*

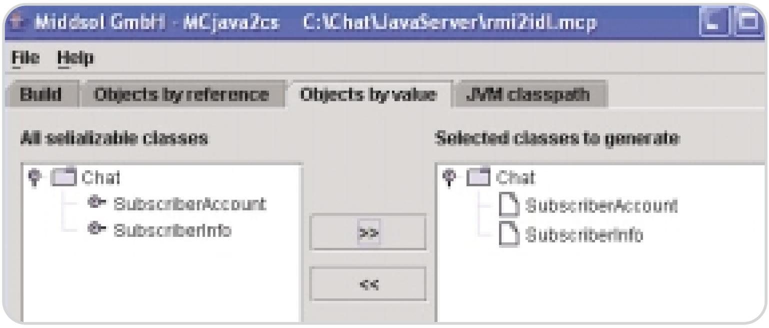


Figure 3: *Serializable classes to be generated*

must call the host again.. MinCor.NET 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 (see Figure 2). In Figure 2, the tool is pointed to the ChatHost.jar of the example. The Host and Subscriber interfaces were selected. MCjava2cs further lists serializable Classes (passed by value) or selection (see Figure 3).

The example contains two classes of this type that were both selected. The last step is to run the generation process that creates C# code for the selected interfaces (see Figure 4).

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 Figure 5, 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. Because 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. See Listing 2 for how the Java implementation looks.

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).

Figure 4: *Running the generation process, creating C# code for the selected interfaces*

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 supposed to access UI controls directly. Because of this restriction, the host cannot simply

AUTHOR BIO:
Christian Donner is a senior consultant and application architect with Molecular Inc., a premier technology consulting firm based in Watertown, Massachusetts. When he isn't tinkering with mobile technology, he designs and develops enterprise-level Web solutions for Fortune 1000 clients.

pubs2004@donners.com

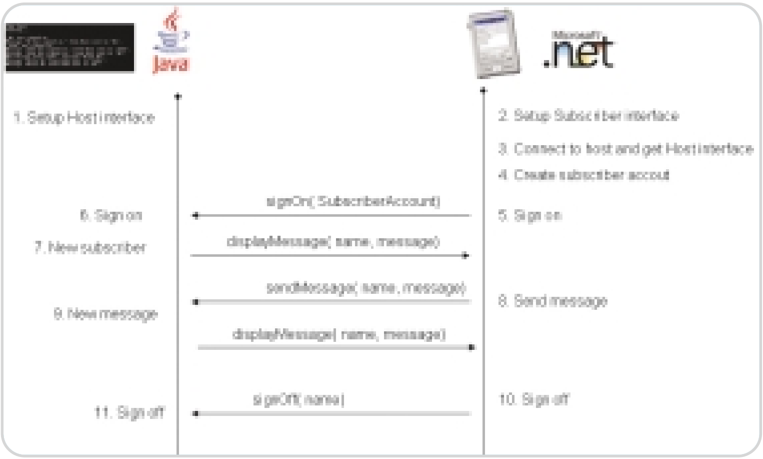
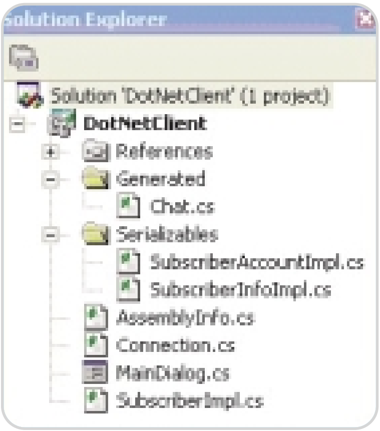

Java
Microsoft .net
1. Setup Host interface
2. Setup Subscriber interface
3. Connect to host and get Host interface
4. Create subscriber accout
signOn(SubscriberAccount)
6. Sign on
5. Sign on
displayMessage(name, message)
7. New subscriber
sendMessage(name, message)
8. Send message
9. New message
displayMessage(name, message)
signOff(name)
11. Sign off
10. Sign off

Figure 5: *The host interface exposes three methods*


Solution Explorer

Solution 'DotNetClient' (1 project)
- DotNetClient
  - References
  - Generated
    - Chat.cs
  - Serializables
    - SubscriberAccountImpl.cs
    - SubscriberInfoImpl.cs
  - AssemblyInfo.cs
  - Connection.cs
  - MainDialog.cs
  - SubscriberImpl.cs

Figure 6: *Visual Studio Solution Explorer*

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 straightforward to read and understand and will not be discussed.

Instead, the article will walk you 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 (see Figure 6). 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.Subscriber-Account (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 in a hashtable. 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 (see Listing 3). The implementation of the subscriber interface exposes one method, displayMessage(), that the host can call (see Listing 4).

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.

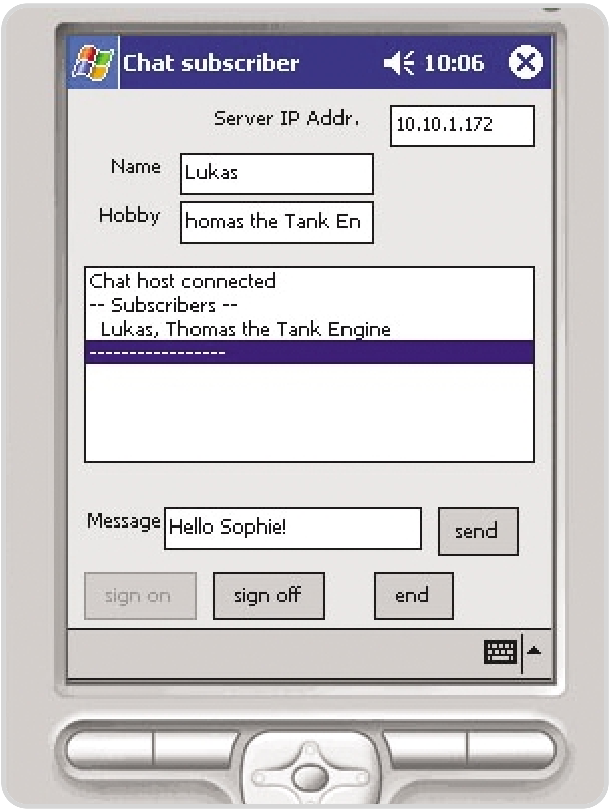
The last module on the client side that deserves attention is Connection.cs. This class contains the methods that manage the communication between subscriber and host. It maintains a reference to the host and to its own interface. It also instantiates an object request broker instance (ORB; see Listing 5).

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.

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, although not needed by some of the methods, is passed to enable future improvements.displayListOf-Subscribers() iterates through the list of subscribers that the host returned and writes each entry to the list box on the screen.

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. To run the client in debug mode, though, it must either be deployed to a device or to the Pocket PC emulator (see Figure 7). 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 Figure 7, the user


Chat subscriber 10:06

Server IP Addr. 10.10.1.172

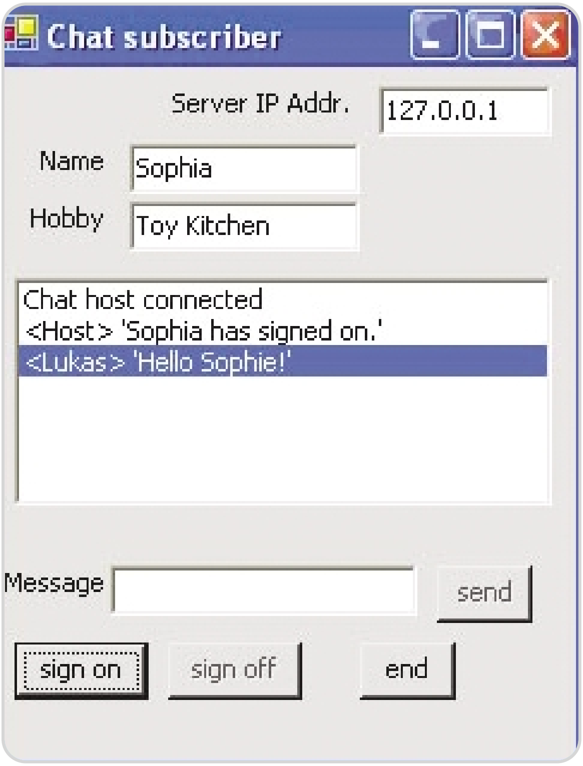
Name Lukas

Hobby homas the Tank En

Chat host connected
-- Subscribers --
 Lukas, Thomas the Tank Engine

Message Hello Sophie!

f7 **Figure 7:** Using the emulator to chat


Chat subscriber

Server IP Addr. 127.0.0.1

Name Sophia

Hobby Toy Kitchen

Chat host connected
<Host> 'Sophia has signed on.'
<Lukas> 'Hello Sophie!'

Message

f8 **Figure 8:** The client executable running as an XP application

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 (see Figure 8).

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

- Run the supplied buildChat-Host. bat to compile the host sources. This batch creates the class files, RMI stubs, and 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 start-NameService.bat.
- Run startChatHost.bat to start the host service, or type in the following command:

```
start java  -cp .;ChatHost.jar
   -Djava.naming.factory.initial=com.sun.jndi.cosnam-
ing.CNCtxFactory
   -    Djava.naming.provider.url=iiop://local-
host:1050 ChatHost
```

- Start the subscriber executable (DotNetClient.exe).
- Type in the Naming Service's IP address, subscriber name, and a hobby.
- Click the sign-on button.

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

Conclusion

This article demonstrated that it is possible to integrate a .NET Compact Framework application with a Java back end. This integration requires very little knowledge of the technical details of the middleware. MiddSol's MinCor.NET is a young product and still a little rough around the edges. However, it 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.

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
- *An Overview of the CORBA Portable Object Adapter:* www.cs.wustl.edu/~schmidt/PDF/POA.pdf ◉