Connecting a Smartphone 2003 Application to a Remoting Infrastructure

Despite the fact that remoting is not currently supported in .NET Compact Framework applications running on the Smartphone 2003 platform, by using a third-party object request broker (ORB) you can interact with remote objects.

by Christian Donner

In its current release, the .NET Compact Framework does not encourage the development of applications with a distributed object model because it doesn't support .NET remoting. That leaves only a single interoperability option: Web services. I'm not by any means attempting to establish a case in favor of or against the use of Web services; however, Web services may not be appropriate for some types of mobile applications. Fortunately, you can create your own remoting solution. The example chat application in this article will show you how to do that.

You will need two third-party products to compile and test the example application: MiddSol's MinCor.NET (an object request broker for the Compact Framework written in C#) and MiddCor (the counterpart that runs in a standard .NET environment). If you want to follow along, I recommend that you first download and install the sample Visual Studio solution code, and download the MiddCor and MinCor libraries from the Middsol Web site as well. When you load the solution into Visual Studio for the first time, you will need to fix the references to the MinCor and MiddCor libraries. To do that, right-click on any broken entries in the References sections of the Solution Explorer and select 'Remove'. Then right-click on 'References' and select 'Add Reference'. Make sure that there is a reference to MinCorSmartPhone.dll in the client project (usually in C:\Program Files\Middsol\MinCor\bin) and one to MiddCor.dll in the server project (usually found in C:\Program Files\Middsol\MiddCor\bin).

If you choose not to install the solution files, the links to code listings found throughout the text should give you a good idea of the techniques used; however, the listings do not contain all the code required to run the sample.

What You Need

Microsoft Visual Studio .NET 2003, MiddSol MinCor.NET Evaluation, MiddSol MiddCor.NET Evaluation, Microsoft Smartphone 2003 SDK, and (optionally) Microsoft ActiveSync.

Functional Requirements

A chat application is good for studying distributed objects because the business logic is relatively simple, yet the application still requires some non-trivial features at the communication level. The example application consists of a chat subscriber (a Smartphone 2003 device) and a chat host component (a .NET host application), connecting through the Internet (see Figure 1).

Chat subscribers specify a UserID and their favorite hobby. Given the chat host's IP address, subscribers can connect to that host and send messages that all subscribers will see on their displays.

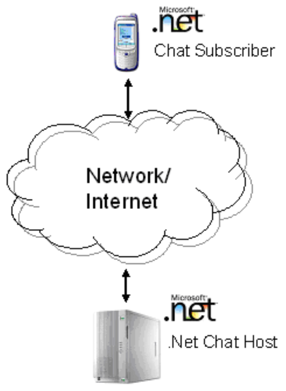

Figure 1. A Distributed Application: A Smartphone with a .NET Compact Framework client connects to a .NET server.

The Subscriber Project

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 straightforward to read and understand (see Listing 1). The remainder of the article will walk you through the class implementations, the subscriber interface, and the Connection class that provides a wrapper for the remote objects and manages the connection.

The folder /Serializables contains the implementations of the CORBA valuetypes. One of these is SubscriberAccountImpl (see Listing 2). While the class methods are trivial, the class declaration is not. The class is derived from the abstract class SubscriberAccount which was generated and is declared in Chat.cs. The suffix "Impl" is a naming convention that serves as a reminder that this is an implementation of an abstract class. The generated parent SubscriberAccount is derived from Middsol.CORBA.portable.StreamableValue—the MiddCor representation of the CORBA valuetype. The same applies to the class SubscriberInfoImpl in Listing 3.

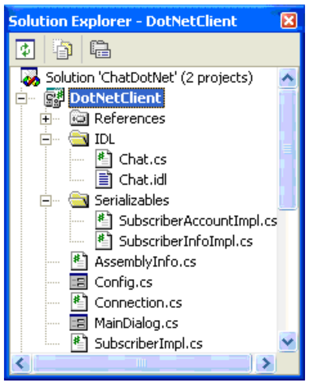

Figure 3. The Subscriber Project: The subscriber project contains the folders that help to organize the code.

The subscriber interface in SubscriberImpl.cs 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 + "'");
      }
   }
}
```

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

The last module deserving of attention on the client side is Connection.cs (see Listing 4). 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 creates an object request broker instance (ORB). It implements the methods shown in Table 1:

Table 1. Connection Class Methods: The connection class contains the methods that manage communications between the subscriber and the host.

Method	**Description**
signOn()	A public method called by the user interface when the subscriber wants to connect to a host.
signOff()	A public method that calls the same method on the host to remove the calling instance from the list of subscribers.
sendMessage()	A public method that makes a remote call to the same method on the host.
initialize()	Initializes the ORB.
initCORBA()	This method 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.
deinitCORBA()	Destroys the ORB.
connectToHost()	Uses the MiddSol Name Service 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". By default, the Name Service process listens on port 2809.
displayListOfSubscribers()	Iterates through the list of subscribers that the host returned and writes each entry to the list box on the screen.

The Glue: IDL
An IDL (Interface Definition Language) file provides the glue between the host and the subscriber code. The IDL file contains a generic description of the calling interfaces with abstract data types that must be mapped to the actual data types of the target platform.

In the chat example, the subscriber and the host each act as both server and client—the term "subscriber" better describes the part of the application that runs on the mobile device than the term 'client' because of the potential confusion with using the term "client" at the communication level, which can be either side.

The subscriber initializes the session by sending a registration packet to the host, which the host acknowledges by returning a list of current subscribers. Once registered, the client can then send messages to the host as needed. The host calls other subscribers to distribute the message to all participating devices. To unregister, a subscriber must call the host again (see Figure 2).

How do these requirements translate to an IDL file? The Host interface shown below exposes three methods that subscribers will call: signOn, sendMessage, and signOff.

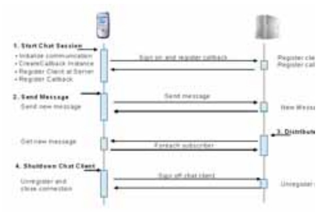

Figure 2. Sequence Diagram: The sequence diagram visualizes the interactions between subscriber and host at the object level.

```
interface Host
{
   SubscriberInfoList signOn(
      in::Chat::SubscriberAccount
         subscriberAccount)
      raises(SignOnException);

      void signOff(in wstring name);

      void sendMessage(in wstring name, in wstring message);
};
```

Table 2 shows how .NET maps data types between IDL and C#. You can find additional information about the IDL syntax and the IDL-to-C# compiler in the MiddSol documentation.

Table 2. The table shows how .NET maps data types between IDL and C#.

IDL Type	C# Type
boolean	Bool
char	Char
wchar	Char
string	String
wstring	String
octet	Byte
short	Short
unsigned short	Ushort
long	Int
unsigned long	Uint
long long	Long
unsigned long long	Ulong
float	Float
double	Double
long double	Double

The signOn() method accepts one parameter (subscriberAccount) and returns a SubscriberInfoList, which is a list of all registered subscribers. The subscriber uses the returned list to display all the participating subscribers. The signOff() and sendMessage() methods require the subscriber to pass in a subscriber name; sendMessage() also requires the message that is to be sent. Because the application uses subscriber names as keys, they must be unique. The host enforces this at registration by iterating through the list of current subscribers and rejecting a sign-on attempt for a user name that already exists.

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(). The displayMessage() method requires two parameters—the name of the sender and the message itself. Here's the IDL for the Subscriber interface.

```
interface Subscriber
{
   void displayMessage(in wstring
      from, in wstring message);
};
```

The Host interface above uses the class SubscriberAccount as a parameter—and that's not a primitive type. Because both host and subscriber use that type, it must be defined in the IDL as well. You use the IDL construct valuetype to represent such a class (do not confuse the IDL valuetype with the .NET valuetype—the latter is the .NET base type for primitives and structs).

CORBA valuetypes are serializable classes that can be transmitted "by value" between two communicating partners. In this case, the class is SubscriberAccount and extends the valuetype SubscriberInfo. The class contains information about a subscriber. The subscriber sends an instance of that class to the host during registration. Here's the relevant section of the IDL file

```
valuetype SubscriberInfo
{
   private wstring strHobby;
   private wstring strName;
   wstring queryName();
   wstring queryHobby();
};

valuetype SubscriberAccount : SubscriberInfo
{
   private ::Chat::Subscriber
      subscriber;
   ::Chat::Subscriber
      queryInterface();
};
```

Note that, in addition to the trivial get methods, the class contains a method called queryInterface() that returns the subscriber's interface. The host needs to know this to call the method displayMessage() on the subscriber.

SubscriberInfo was introduced as the base value for SubscriberAccount because publishing a subscriber's interface to all the other subscribers in the chat room would create a security risk. It would not be a good idea for the signOn() method to return a list of SubscriberAccount objects.

```
typedef sequence <SubscriberInfo> SubscriberInfoList;
```

Now that the IDL is complete (see Listing 5 for the code) you can run the MiddCor IDL compiler (MCidl2cs.exe) to generate C# stub code. You must add the resulting stub file (Chat.cs) to both the host and the subscriber projects in Visual Studio.

The server is a console application project. It includes the same implementations of the SubscriberAccount and SubscriberInfo classes as the subscriber (see Listing 2 and Listing 3). Because the primary focus of this article is on the mobile device, I'll discuss the host code only in general terms, but you can download the sample code to explore further.

The file ChatHost.cs (see Listing 6) contains the host's main() method, which instantiates a ChatHost object and then calls the method runHost(). This method, like the client code, creates a new ORB and registers its services with the MiddCor Name Service.

The class HostImpl maintains the list of subscribers and contains the implementations of the exposed methods signOn(), signOff(), and sendMessage(). The latter calls the private method distributeMessage() that iterates through the list of subscribers and calls displayMessage() for each one.

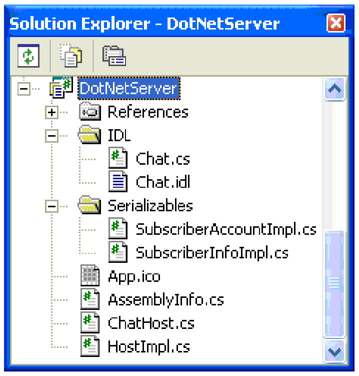

Figure 4. The Host Project: The Serializable classes and the generated stub code are shared between the host and the subscriber projects.

The signOn() method calls createSubscriberInfoList() and creates an instance of the SubscriberInfo[] array of subscribers for each new subscriber. It calls the method querySubscriberInfo() added to SubscriberAccount on the host to spawn a SubscriberInfo instance.

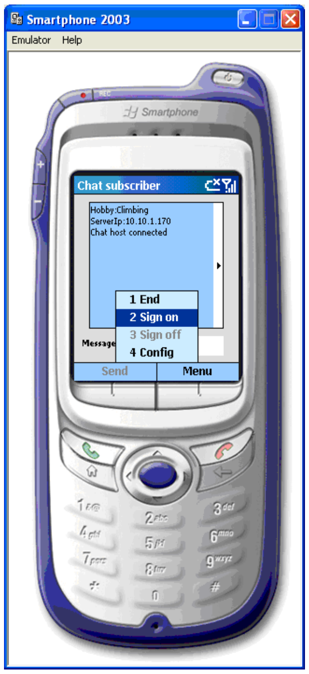

Figure 5. The Smartphone Emulator: The emulator is not the tool of choice to test a distributed application because it does not support callbacks.

Deploy and Test

The host executable requires the Name Service (MCns.exe) to be running. You can test the subscriber client without a mobile device; however, to run the client in debug mode, you must either deploy it to a physical device or to the SmartPhone 2003 emulator. Visual Studio will prompt you for a deployment target whenever you start the debugger. Note that the emulator has some limitations. For instance, it does not support the callback functionality, which means that the displayMessage() method will not work and will eventually time out.
With this restriction in mind, use the following steps to test the application:

- Start the MiddCor Name Service (MCns.exe)
- Start the host executable (DotNetServer.exe)
- Start the subscriber executable (DotNetServer.exe)
- Open the subscriber configuration dialog and provide the correct server IP address, your screen name and hobby
- Save your changes
- Sign on to the host

Security Considerations
Mobile communication devices are exposed to more security threats than stationary, wired devices. You shouldn't build and operate even a simple chat application like the one described in this article without first making a careful evaluation of the risks and implementing appropriate security measures, such as authentication and encryption.

Regardless of whether the application communicates via the cellular network or a WLAN (802.11) connection, the most common mechanisms for data encryption are available on the device. The mobile client can establish a VPN connection to the server using either Microsoft's Point-to-Point Tunneling Protocol (PPTP) or the standard-based IP Security Protocol (IPSEC). Both protocols create a secure tunnel between two communicating partners. IPSEC is considered the more robust option.

WLAN connections can also be secured by WEP, the Wireless Equivalent Privacy protocol. There are many reports about weaknesses and exploits of WEP, so don't rely on it too much. Both VPN and WEP are completely transparent to the application.

IIOP over SSL (Secure Socket Layer) is an interesting option for a secure mobile application. It is supported by most commercial application servers today, but MinCor does currently not implement SSL. According to MiddSol support it is one of the features planned for 2005.

Using a Smartphone device, Windows Mobile technology is ready to take your distributed object application to topologies that were previously out of reach. While the current version of the .NET Compact Framework erects some barriers for developers of tightly-coupled distributed applications, third-party products are available today that can overcome such limitations and allow you to expose your existing C# or VB.NET object to an increasing number of mobile devices.

Christian Donner *is a Senior Consultant and Technical Architect at Molecular, a premier technology consulting firm based in Watertown, Massachusetts. When he's not tinkering with mobile technology, he designs and develops enterprise-level Web solutions for Fortune 1000 clients.*