

An Introduction to WCF for Device Developers

Chris Tacke
OpenNETCF Consulting, LLC
November, 2007

Introduction

With the release of Visual Studio 2008 to MSDN subscribers last week, I decided that it was probably safe enough to install the tool and start using it, plus I'm working on a large project where we're soon going to need to be able to communicate between a PC and a CE device.

Conceptually (and at only a very high level) I had an idea what Windows Communication Foundation (WCF) is and my guess was that it could be used to help solve the problem at hand, which is to generate some form of remoting capability.

Of course I'd never actually created anything that used WCF, nor even looked at WCF code except in some conference sessions, so the first step was to head to the bookstore. I picked up *Inside Windows Communication Foundation* from Microsoft Press and tried to give it a read. As is typical for me, I found most of it analogous to listening to a white noise machine. It did little to further my understanding of the subject, but left me annoyed and tired.

In all fairness I can't say it's a bad book, I just learn far better from concrete examples and hand-on work than I do from reading about doing something. Chapters 6 and 7 which cover Channels actually was interesting and useful and I actually read them rather than skimming. Chapter 6 gave me enough to say that the \$40 I spent on it was worth the investment.

So once I was done with the book and a couple beers, I decided it was time to roll up my sleeves and start searching the web for some concrete examples so I could walk through some code in a debugger and learn by breaking stuff.

Little did I know that while there's a lot of talk about WCF, finding a simple example that comes complete with source code for both ends is not a trivial task. This is especially true when it comes to WCF for devices – there simply is no full sample (that I could find anyway) that explained how to go from nothing but Studio to two devices talking to one another, step by step, with all the code and directions needed.

So after a day and a half of stumbling around and cussing, I finally ended up with something worked and that probably should have taken only an hour to develop. This white paper is a collection of the knowledge I gained, and that single source of a WCF sample that some other developer is now, or will be looking for in the near future.

Requirements

Before you get started, make sure that you have everything tool-wise that you're going to need.

1. *Visual Studio 2008 (used to be called Orcas) RTM*. I downloaded mine from MSDN. If you got yours elsewhere, then your mileage may vary through this article.
2. *Power Toys for .NET Compact Framework 3.5 CTP (September 2007)*. This one wasn't obvious until I'd done a fair bit of searching. You **will** need it, so download and install it now. The current link is <http://www.microsoft.com/downloads/details.aspx?FamilyID=C8174C14-A27D-4148-BF01-86C2E0953EAB&displaylang=en> but if you've been browsing Microsoft's site for years like I have, you know that that link may be dead next week, so search for title text if it fails.
3. *A Windows Mobile 5.0 device*. In theory you should be able to use any device that is supported by CF 3.5 (Windows CE 5.0 or 6.0 or WM 5.0 or 6.0), but I used an Axim x51. I can say that I tried for some time to get them WM 5.0 emulator working and failed, so you should at least start with a physical device to remove that variable.
4. *Windows XP development environment*. While this isn't necessarily a requirement, I'm running XP, not Vista, so if there are any UAC steps that are needed to get this working you're not going to see them in this article. If you're using Vista, I wish you luck. Please post any necessary deviations as comments.

The PC-side WCFServer application

To make things fairly simple we're going to use a transport shipped with WCF for devices and set up the server application to run on the PC (sure I need the reverse in my target solution, but as Bill Murray said in What About Bob – baby steps).

So the first thing we need is to create a service that does something on the desktop. We're not going to go overboard and make it complex, we're going to have a single function that takes in two integers and returns the sum of them. My thinking is that once you see that working, getting more complex stuff working is simply additional baby steps on top of that.

So first, create a desktop Console application (if you were in a rush and already created a WinForms app before you got this far in the article, delete the Form and change the project type in the Project Properties just like I ended up doing when I was creating this app).

Add the following reference to the project:
System.ServiceModel

Now add a new Class (well it will be an interface, but using the Add Class menu item and then changing the code is less clicking) document named ICalculator and replace its code with the code below. This is the interface that defines the contract our service exposes. It should all be self explanatory, so I won't bore you with a walk-through.

```
using System;
using System.ServiceModel;

namespace OpenNETCF.WCF.Sample
{
    [ServiceContract(Namespace = "http://opennetcf.wcf.sample")]
    public interface ICalculator
    {
        [OperationContract]
        int Add(int a, int b);
    }
}
```

```
}  
}
```

Now we need to create an implementation of that interface to expose as our actual service. Add another Class to the project called CalculatorService and overwrite the generated code in the document with the code below. Again, it's not rocket science, so we're not going to cover what it does.

```
using System;  
  
namespace OpenNETCF.WCF.Sample  
{  
    public class CalculatorService : ICalculator  
    {  
        public int Add(int a, int b)  
        {  
            Console.WriteLine(string.Format(  
                "Received 'Add({0}, {1})' returning {2}", a, b, a + b));  
            return a + b;  
        }  
    }  
}
```

The last remaining piece of our service app is the logic that actually creates an instance of our CalculatorService implementation and exposes it. This code is a little more complex (though not a whole lot), so we will look at what it's doing, but first, overwrite all the code in your existing Program.cs file with the following code:

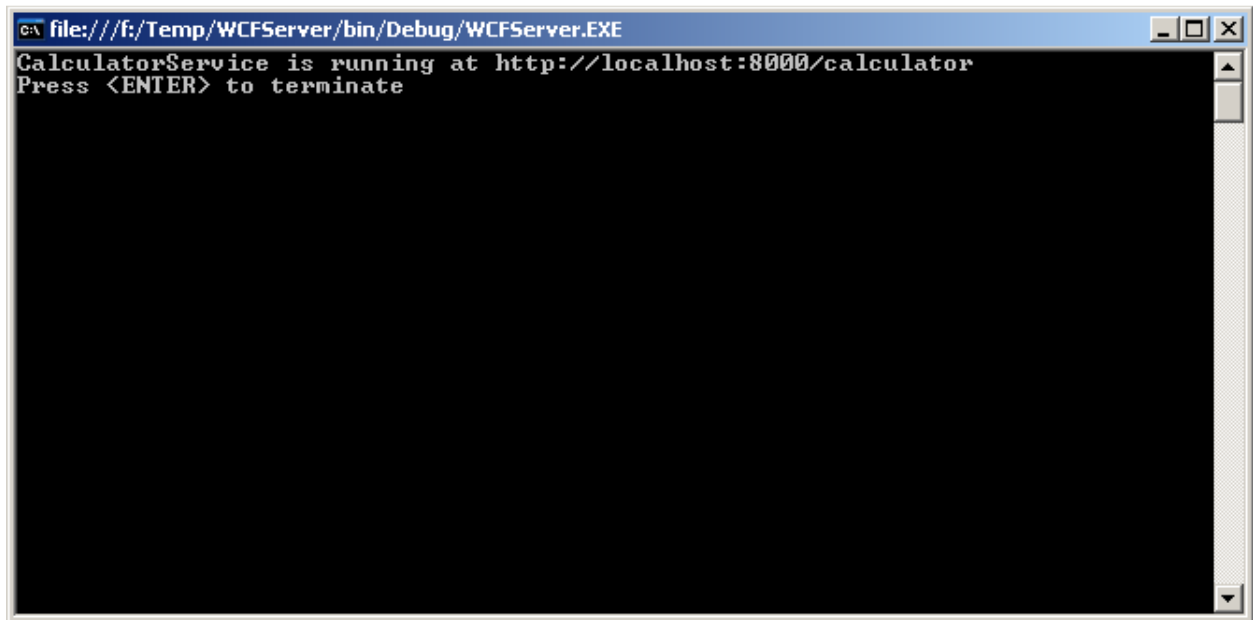
```
using System;  
using System.ServiceModel;  
using System.ServiceModel.Description;  
using System.Net;  
  
namespace OpenNETCF.WCF.Sample  
{  
    class Server  
    {  
        static void Main(string[] args)  
        {  
  
            string hostIP = Dns.GetHostEntry(  
                Dns.GetHostName()).AddressList[0].ToString();  
  
#if CLIENT_DISCOVERY_BUILD  
            Uri address = new Uri(string.Format(  
                "http://localhost:8000/calculator", hostIP));  
#else  
            Uri address = new Uri(  
                string.Format("http://{0}:8000/calculator", hostIP));  
#endif  
  
            ServiceHost serviceHost = new ServiceHost(  
                typeof(CalculatorService), address);  
  
            try  
            {  
                // Add a service endpoint  
                serviceHost.AddServiceEndpoint(  

```


Next we add the ability to exchange metadata only if we're in the Discovery configuration. The reason for this is that the tool (wait for it!) can only connect to our service and generate code if this is turned on. I turn it off otherwise simply because it seems like a potential security hole if you leave something like that on in the wild. Maybe not, but we'll play it safe until experience (or someone who knows more about WCF) tells us that we should do otherwise.

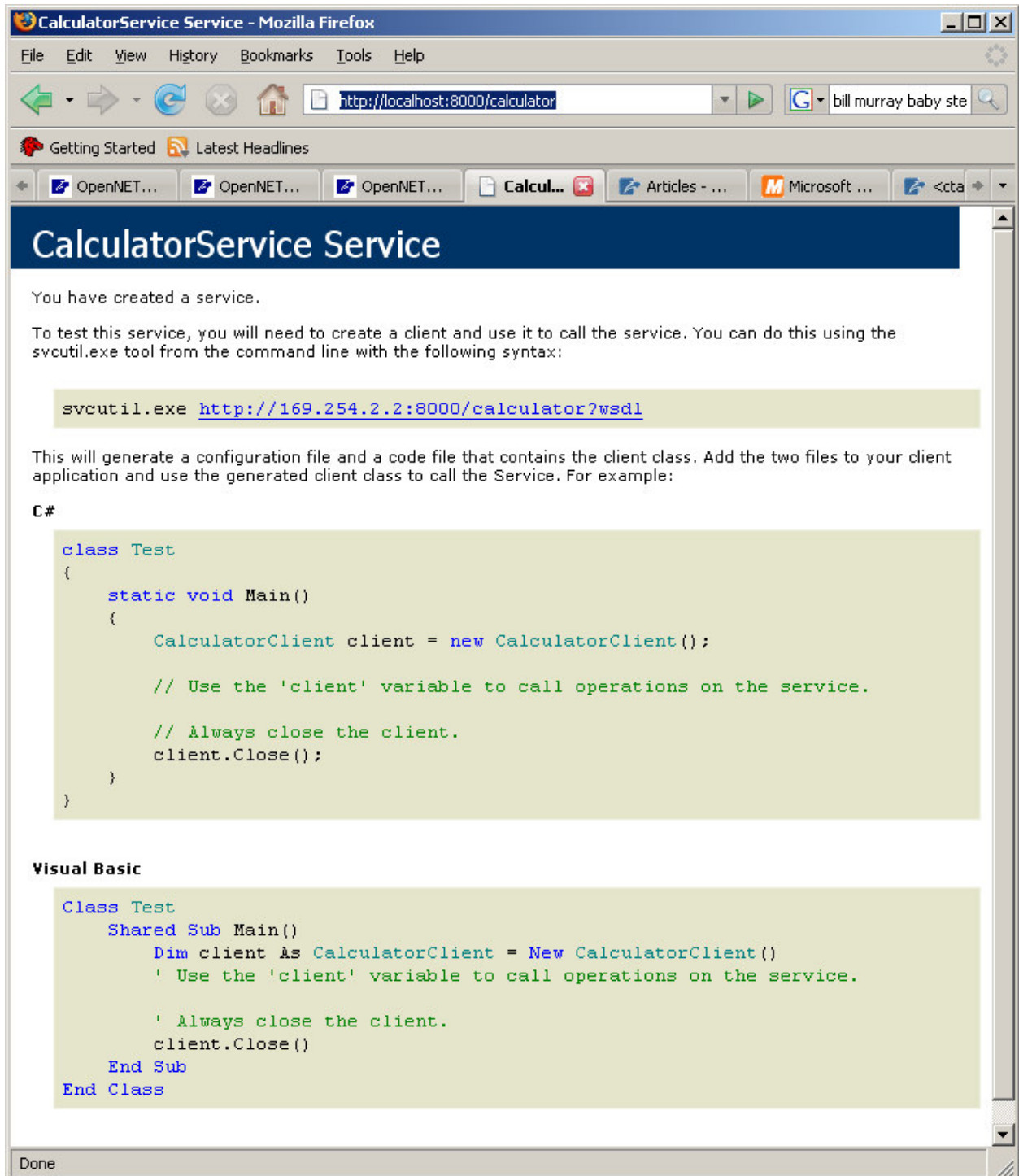
Finally, we open the service and output some stuff on the Console, waiting for the user to hit Enter to end the app.

That's all there is to it, so make sure that `CLIENT_DISCOVERY_BUILD` is set and run your app, and you'll get something exciting like the screen shot below.



```
C:\file:///f:/Temp/WCFServer/bin/Debug/WCFServer.EXE
CalculatorService is running at http://localhost:8000/calculator
Press <ENTER> to terminate
```

Now let's do a quick sanity check to make sure it's actually exposing itself as a service. Open a browser on your PC and browse to the address that the service reported (<http://localhost:8000/calculator>). You should see something like this:



Tell everyone around you to stand in awe – you have created the mythical WCF Service.

The NetCFSvcUtil.exe Tool

Yes, now we're going to actually talk about the tool that we talked about. The Power Toys for .NET Compact Framework 3.5 ship with a tool called NetCFSvcUtil.exe which is a lot like its desktop

counterpart (that you won't be using here – believe me, I tried). This tool will connect to your service using the metadata exchange that you enabled in your code and output some code for your CF client application to use.

Now I'm not big on code generators – having seen them evolve for many years, one thing that has typically been a truism is that they produce less-than-optimal code, but a quick glance at what this one generates looks reasonable and besides, we're here to create a WCF app, not criticize code optimization.

Open a command windows and browse to the folder that contains NetCFSvcUtil.exe. The default installation of the Power Toys will put it at *C:\Program Files\Microsoft.NET\SDK\CompactFramework\v3.5\bin*. Next, run the following command (again, the service must be running when you do this)

```
netcfSvcUtil.exe /language:cs http://localhost:8000/calculator
```

You should get an output that looks like this:

```
C:\Program Files\Microsoft.NET\SDK\CompactFramework\v3.5\bin>netcfSvcUtil.exe /language:cs http://localhost:8000/calculator
Microsoft (R) .NET Compact Framework Service Model Metadata Tool
[Microsoft (R) Windows (R) Communication Foundation, Version 3.5.0.0]
Copyright (c) Microsoft Corporation. All rights reserved.

Attempting to download metadata from 'http://localhost:8000/calculator' using WS-Metadata Exchange or DISCO.
Generating files...
C:\Program Files\Microsoft.NET\SDK\CompactFramework\v3.5\bin\CalculatorService.cs
C:\Program Files\Microsoft.NET\SDK\CompactFramework\v3.5\bin\CFClientBase.cs

C:\Program Files\Microsoft.NET\SDK\CompactFramework\v3.5\bin>
```

And 2 files will be generated in the same folder as NetCFSvcUtil.exe: CalculatorService.cs and CFClientBase.cs. You'll be using these files in the next step.

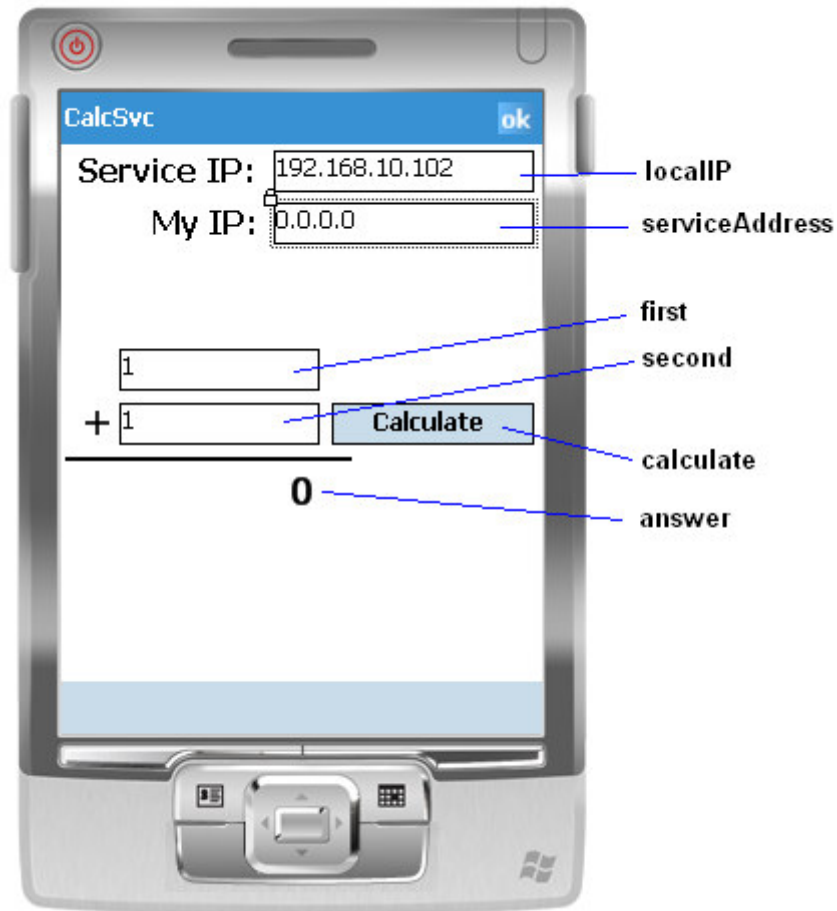
The CF Client Application

Now create a Smart Device Windows application named WCFClient and add the following references (failure to add these references will lead to lots of colorful language when the code previously generated by NetCFSvcUtil.exe fails to compile):

- System.ServiceModel
- System.Runtime.Serialization

Next add the two files created by NetCFSvcUtil.exe to the project. It's probably best to copy them to the project folder first to prevent cluttering the SDK bin directory with all of your generated source.

Next update Form1 to look something like the figure below. The names of the controls are on the right and if you want to just copy and paste code, you'll need your control names to match these.



NOTE

I was never able to get the emulator to connect to service. I was able to ping the emulator from the PC and it was able to browse to a web site, but it wouldn't connect to the service. I then tried it on a real device and it worked, so I never bothered to go back and try to figure out why the emulator didn't work. I highly suggest that you do run this sample on a real device.

Now let's look at implementing the client. The first step I did was to display the device's local IP address in the Form constructor. I did this when I was debugging because the emulator would not connect to the service and I was wondering if it had a valid IP address.

Update your Form's constructor code to look like this:

```
public Form1()
{
    InitializeComponent();

    try
    {
        localIP.Text = Dns.GetHostEntry(
            Dns.GetHostName()).AddressList[0].ToString();
    }
}
```

```

    }
    catch (Exception ex)
    {
        MessageBox.Show("No NIC found?");
    }
}

```

Now we add all of the service “work” in a Click event handler for the calculate button. Add a handler to the button and add the following code to the handler:

```

private void calculate_Click(object sender, EventArgs e)
{
    int a = 0;
    int b = 0;

    try
    {
        a = int.Parse(first.Text);
        b = int.Parse(second.Text);
    }
    catch
    {
        // parsing failed, just bail out
        return;
    }

    SMC.Binding binding = CalculatorClient.CreateDefaultBinding();
    string remoteAddress = CalculatorClient.EndpointAddress.Uri.ToString();
    remoteAddress = remoteAddress.Replace("localhost", serviceAddress.Text);
    EndpointAddress endpoint = new EndpointAddress(remoteAddress);

    CalculatorClient client = new CalculatorClient(binding, endpoint);

    try
    {
        answer.Text = client.Add(a, b).ToString();
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message);
    }
}

```

The important point in this code are that we don’t use the default constructor for the CalculatorClient, because that would end up using a remoteAddress with “localhost” in it, because that’s what the code generator hard-codes it to be. Instead, we extract that value (it’s a static field of the generated CalculatorClient) and then use string substitution to replace “localhost” with the IP address of the PC running the service.

Once we’ve created the CalculatorClient instance, calling the service is trivially simple.

Conclusion

So there you have it - a complete, working, end-to-end example of how to create a WCF service running on a PC and how to consume it using a Compact Framework 3.5 application. As you've seen, there isn't a lot of code to it, and it's not overly complex to get set up – it's just not very well documented in any one place.

Now where might you use something like this? One thing that comes to mind is the ability to put functionality on a machine somewhere within your network that can then be called by multiple devices without having to go through all of the configuration and setup required to get an XML Web Service under IIS running. You can then update or fix the service code in one place and all devices using the function immediately get the benefit without having to redeploy or roll out the fix. You could also do computationally-heavy work on a desktop machine with more power, but still do the collection and display on a lightweight mobile device.

At any rate, I hope that this white paper saves someone else a bit of the headache I went through in trying to get a WCF application running. Some additional links that I found at least somewhat useful in researching this are below. Enjoy.

CF WCF Intro materials:

<http://blogs.msdn.com/markprenticems/archive/2007/03/27/introduction-to-windows-communication-foundation-for-the-net-compact-framework-messaging-stack.aspx>

The Lunch Launcher

<http://blogs.msdn.com/davidklinems/archive/2007/11/12/the-journey-of-the-lunch-launcher-part-8-what-did-i-learn.aspx>

Using NetCFSvcUtil.exe

<http://blogs.msdn.com/danhorbatt/archive/2007/09/12/using-compact-svcutil-to-interact-with-servicecontracts.aspx>

Calling WCF Service from CF 3.5

<http://blogs.msdn.com/andrewarnottms/archive/2007/09/13/calling-wcf-services-from-netcf-3-5-using-compact-wcf-and-netcfsvcutil-exe.aspx>

The CF 3.5 subset of WCF

<http://blogs.msdn.com/andrewarnottms/archive/2007/08/21/the-wcf-subset-supported-by-netcf.aspx>

Remote Logging WCF

<http://blogs.msdn.com/danhorbatt/archive/2007/11/01/remote-logging-wcf-on-net-compact-framework.aspx>