Michele Leroux Bustamante and Nickolas Landry, May 2009

Contents

Mobile devices have become a staple of everyday life, not only for professional knowledge workers but also for average consumers and end-users who have grown accustomed to a connected lifestyle. Mobile devices are routinely used to make phone calls, send and receive emails, exchange text messages and manage busy schedules with calendars and reminders. Most devices can perform much more than typical “electronic organizer” and phone features. They literally are small computers that can fit in a pocket, lab coat or purse. Just like any computer, to get the most value out of devices, you need additional software.

When it comes to writing mobile software, there are two main types of applications you can build to run on devices: standalone applications, and connected smart clients. Standalone applications have to deal with multiple constraints introduced by the device form factor - such as limited screen real estate, less memory, CPU power and available storage, limited input methods such as tactile screens and hardware keys on the device, and dependency on battery life. Smart clients on the other hand have to deal with the same constraints as standalone mobile applications with the addition of concerns related to communications with remote server resources such as sporadic connectivity, slower connection speeds, high latency networks, wireless security threats and much more.

The ability to reach application servers and backend systems and databases is what finally enabled mobile devices as full class enterprise citizens. Enterprise applications have long been accessible to office users and knowledge workers in various ways. Whether they are deployed locally as rich clients or accessed centrally as Web clients running within a browser, enterprise applications connect people, systems, processes and data in order to achieve enterprise agility. This is done by relying on systems that enhance productivity, lower operational costs, and enable new ways to work with data, resulting in a competitive advantage.

Great technological advances in the field of mobility have allowed us to reach even farther out, beyond the physical boundaries of the enterprise. Faster wireless networks, broader coverage, more powerful devices, standardized positioning and location-based services, different form factors and state of the art development tools have all contributed to the mobile revolution currently in progress.

Most of the users that have been long forgotten by the world of information technology are now able to participate in automated business processes once again. Both internal and external mobile and field workers can access corporate data from the road, alter it or collect new data, and submit it back to the central office.

Spurred by technologies like the Windows Mobile platform and the Microsoft .NET Compact Framework which brings managed code to mobile devices, a new breed of mobile smart client applications is emerging and changing the way we think about software development. These new smart clients are based on best practices learned by merging rich clients and Web development, bringing the best of both worlds to the realm of mobile devices.

Michele Leroux Bustamante and Nickolas Landry, May 2009

Windows Mobile 6.1 is the latest operating platform from Microsoft for running mobile applications – be they standalone or connected. Mobile smart clients built for Windows Mobile using the .NET Compact Framework can use a number of different technologies for remote communications including text messaging, email-based transports, sockets, and web services. Today, Windows Communication Foundation (WCF) is the preferred technology for building connected applications – a subset of which is available in the .NET Compact Framework 3.5. The goal of this whitepaper is to help developers build mobile smart clients for Windows Mobile 6 using WCF to communicate with remote server resources. Specifically, this paper will explain how to design and implement WCF services so that they can be consumed by .NET Compact Framework applications running on Windows Mobile devices and show you how to work with the subset of WCF on mobile devices to consume those services.

For those new to mobile development we will provide an overview of the platform tools available along with instructions to set up your development environment as it relates to the scenarios we will be describing in this whitepaper. We'll also walk you through creating and hosting a simple WCF service and generating a proxy to consume that service from a mobile device – so if you are new to WCF you will be able to follow along. We will then discuss specific considerations related to WCF and mobile development including contract design and serialization, binding configuration for endpoints, hosting options, proxy generation, exception handling, and security. In addition, we will discuss how to consume REST-based WCF services from your mobile applications.

The State of Windows Mobile Today

Mobile devices, just like any desktop or server computer, require an operating system to run. Mobile .NET applications run as managed code under the care of the .NET Compact Framework, which in turn runs on top of a specific embedded operating system. Just like the .NET Framework was designed and built to run on top of various flavors of Windows, like Windows Vista, Windows 7 or Windows Server 2008 - the .NET Compact Framework was designed and built to run on top of mobile and embedded device-specific versions of Windows: Windows CE.

Microsoft Windows CE is a scaled-down and modular version of the Windows NT core operating system targeted at a wide range of innovative, small-footprint devices including consumer electronics, gateways, industrial controllers, mobile handheld devices, Internet Protocol (IP) set-top boxes, voice over Internet protocol (VoIP) phones and thin clients. Windows CE features a multithreaded 32-bit kernel and an open architecture optimized for real-time, and has the power to drive serious applications in any kind of device. With its scalable model, Windows CE can scale up to power feature-rich devices and appliances, and scale down to less than 1 MB to provide basic but powerful system services in very compact form factors.

Windows CE's modularity also allows device manufacturers to decide which components of the OS they want to include. The device manufactures can thus streamline their respective devices by excluding unnecessary modules and libraries that would add little value for the end user.

Michele Leroux Bustamante and Nickolas Landry, May 2009

Because of the modular nature of Windows CE, it has become difficult for mobile application developers to write generic solutions. Each mobile device might have a slightly different variant of Windows CE, and the developer could never know in advance whether an OS component existed in the final device or not. This inconsistency and uncertainty greatly limited deployment scenarios because applications had to be device-specific.

Microsoft solved that problem by introducing Windows CE-based specifications targeted at standard consumer devices and phones. A specification defines two sets of components:

- The core components necessary for the OS
- A set of shell-extensions which provide a uniform look and feel and uniform data entry mechanism for each group of device form factors.

In other words, each device must support a common, minimum set of components. Additional components are grouped into specific extensions. In order to unify the Windows CE versions to be used on generic consumer-grade Pocket PCs and Smartphones, Microsoft created Windows Mobile. Windows mobile is therefore a platform that meets a standard set of specifications for Windows CE.

Windows Mobile 6 is powered by Windows CE 5.0 and includes a number of mobile productivity features including an SDK with managed API's for working with Outlook Mobile and Telephony, a lightweight version of Office called Office Mobile (including Word Mobile, Excel Mobile and PowerPoint Mobile) and Media Player Mobile. Windows Mobile 6.0 added many security enhancement such as storage card security and better support for certificates, and a number of other productivity and networking features are also included to provide control for device specific capabilities, such as Caller Photo ID, GPS and Bluetooth. Windows Mobile 6.1 added a more streamlined Today screen and support for Microsoft App Center Mobile Device Manager, and the new version 6.5 introduces new user interface options as well as updated productivity applications.

Windows Mobile Devices

Windows Mobile devices are primarily consumer-level devices that fall under one of two categories: Pocket PCs and Smartphones.

Pocket PCs (Windows Mobile 6 Professional/Classic)

Pocket PCs are versatile PDAs (Personal Digital/Data Assistants) that act like miniaturized computers in terms of running applications on top of a mobile operating system, but they also provide the user with many key characteristics to facilitate mobile usage in the field. These characteristics include stylus-based input on a tactile screen, handwriting recognition, and a standard set of device keys (i.e. buttons) for quick and easy access while standing. Many Pocket PC devices today feature a full QWERTY keyboard, either at the base of the device under the screen, or on a panel at the bottom or on the side of the device.

As wireless networks evolved, Pocket PC devices started featuring standard wireless radios for access to Personal Area Networks (PAN, like Bluetooth-based networks), Wireless LANs (WLAN, like Wi-Fi-based

networks) and Wireless WANs (WWAN, like carrier-operated nationwide GPRS/EDGE or 1X networks). Fast 3G broadband networks like EV-DO and UMTS/WCDMA/HSDPA are also WWAN standards.

Microsoft revised the Pocket PC specification by introducing the Pocket PC Phone Edition, which is a Pocket PC device with standard digital phone and Internet access features. Pocket PC Phone Edition devices feature a telephony dialer software utility, telephony APIs and standard configuration tools for wireless access to the Internet. This made it easier for mobile software developers to write applications that rely on wireless connectivity and voice without requiring third-party APIs.

Under the Windows Mobile 6 generation, Pocket PC Phone devices are called **Windows Mobile 6 Professional** devices, whereas Pocket PCs without the phone feature are known as **Windows Mobile 6 Classic** devices. Both leverage the Windows Mobile 6 Professional SDK for application development.

Smartphones

Smartphone are digital Internet phones with PDA-style features and Microsoft released their first Smartphone specification in 2002. Now available for Windows Mobile 6 and higher, Smartphones pack many of the same features as a Pocket PC Phone Edition but in a (typically) smaller and altered form factor. For starters, many Smartphone devices look like a standard modern cell phone, with a large color screen that supports advanced graphics and animations and the standard numeric keypad that is ubiquitous to almost all phones. Navigation is driven via soft keys and a small cursor-style 4-way joystick. There is no tactile screen and no stylus-based input on Smartphone devices.

Under the Windows Mobile 6 generation, Smartphone devices are called **Windows Mobile 6 Standard** devices, and leverage the Windows Mobile 6 Standard SDK for application development. Note that despite the name, Windows Mobile 6 Professional devices are not meant to be more advanced than Windows Mobile 6 Standard devices. The primary input method (stylus vs. keys) is the primary differentiation factor.

Whereas Windows Mobile 6 Professional devices (i.e. Pocket PC Phone) were originally designed as PDAs first and phones second, the Microsoft Smartphone is a phone first and a PDA second. Text entry on Smartphones is often less convenient than on Pocket PCs as they are geared towards quick lookups and very minor data updates.

However, as the lines are blurring more and more between Pocket PC and Smartphone devices, the only sure giveaway to tell them apart nowadays is the tactile screen & stylus input on Pocket PCs and the rigid screen on Smartphones.

.NET Compact Framework

The .NET Compact Framework is the lightweight version of the .NET Framework, designed for use on resource-constrained Windows devices. The .NET Compact Framework shares many similarities with the full .NET Framework. However, operating system limitations, platform size and performance characteristics dictate that the .NET Compact Framework architecture differs somewhat from the full version.

The .NET Compact Framework provides operating system and CPU portability. You do not need to recompile applications for each CPU that runs the .NET Compact Framework. The .NET Compact Framework itself has targeted implementations of its Execution Engine for each of the CPUs found in various Windows mobile and Windows CE devices.

The .NET Compact Framework is a subset of the full .NET Framework. If you are familiar with the full .NET Framework, you will recognize most of the functions in the .NET Compact Framework and will be able to immediately apply them. For example, both Frameworks use the common language runtime with managed code execution, just in time (JIT) code compilation, and garbage collection. Both Frameworks use assemblies and access portable executable files. However, there are differences between the two Frameworks that affect how you develop applications.

The size of the .NET Compact Framework is limited to improve performance. The footprint of the common language runtime for the .NET Compact Framework is approximately eight percent the size of the full .NET Framework runtime and shares approximately one third of the classes in common with the full .NET Framework. As a result, there are many features not supported by the .NET Compact Framework. Furthermore, support for a specific namespace in the .NET Compact Framework does not necessarily mean all the sub-namespaces and classes associated with the parent namespace are supported. Also note that the .NET Compact Framework version of a class might have fewer members than the full .NET Framework since not all properties, methods and events are supported for every class.

.NET Compact Framework 3.5 is the latest version, is integrated with Visual Studio 2008, and includes the following new features:

- Partial support for Windows Communication Foundation (WCF), which is the primary topic discussed in this paper
- Partial support for the Language Integrated Query (LINQ), including LINQ to Objects, XML and DataSets
- New Media classes to play sounds from managed code
- Various Windows Forms enhancements
- Compression support to provide basic compression and decompression services for streams via the new **System.IO.Compression** namespace
- Support for Client-side certificates
- Managed debugger fixes and a new CLR Profiler tool

WCF is not supported on earlier versions of the .NET Compact Framework, thus all the code samples for this whitepaper require the .NET Compact Framework 3.5.

Remote Communications with Mobile Devices

Communicating with server resources and the rest of your corporate network infrastructure is a key aspect of any mobile smart client application that participates in distributed enterprise architecture. Many alternatives are available to the mobile developer seeking an efficient communication scheme between device and server. Although this whitepaper focuses exclusively on techniques based on

Windows Communication Foundation (WCF) – it may help to compare and contrast the options as you are making decisions about the right architecture for your mobile applications.

Figure 1 shows a summary of remote communications methods available to mobile applications, along with a short description and a summary of typical usage scenarios. For this whitepaper we will primarily focus on WCF, along with some relevant uses for raw HTTP request/response style messaging.

Figure 1: Summary of remote communication methods for mobile applications

Method	Description	Recommended Usage
Web Services	Interoperable SOAP messages sent over HTTP	• Calling Web services on other platforms • Calling remote services over the Internet (HTTP) • No security (HTTP) or transport security (HTTPS/SSL) • Need background asynchronous calls via easy proxy generation
Sockets	Custom packet-based protocol over TCP/IP	• Fast transport & low overhead needed • Communications over internal/private networks • Custom protocol needed for niche scenarios
Direct Database Access	Live connection to SQL Server on the backend over TCP/IP or HTTP, using ADO.NET for data exchange	• Need access to all server data • Devices used in Wi-Fi or high wireless availability (cell) environments, online only • No service invocation model required, only data is downloaded/uploaded
Synchronization	Synchronization of data between the source SQL Server database and client-side SQL Server CE database using RDA, Merge Replication or ADO.NET Sync Services	• Need to pull frequent data snapshots from the server database • No service invocation model required, only data is downloaded/uploaded • Sporadic connectivity environment
Message Queuing	Asynchronous exchange of data messages with Message Queuing Services (MSMQ) over TCP/IP	• Wi-Fi environments where MSMQ already deployed • Communications over internal/private networks • Real-time exchange of messages/data needed • Need asynchronous model for offline support • No transport or message security requirements, authentication only
Email/SMS Messages	Asynchronous exchange via email or text messaging	• Need to push messages from the server to the mobile device • Need asynchronous model for offline support
HTTP	Raw communication over	• Calling remote services over the Internet

Request/Response	HTTP protocol	(HTTP) • Flexible protocol adaptable to a variety of messages and formats • Easy to secure at the transport level via SSL • Need to call REST-based services
WCF	SOAP-based or REST-based communications over HTTP	• Calling Web services on other platforms • Calling remote services over the Internet (HTTP) • Robust transport-level (HTTPS/SSL) or message-level (WSS) security needed • Need to push messages from the server to the mobile device

WCF for the Mobile Developer

WCF is Microsoft's platform for building distributed service-oriented applications for the enterprise and the web that are secure, reliable, and scalable. It supersedes previous technologies such as .NET Remoting, Enterprise Services and ASP.NET Web Services (ASMX) by offering a unified object model for building applications that support the same distributed computing scenarios. WCF supports scenarios such as:

- Classic client-server applications where clients access functionality on remote server machines.
- Distribution of services behind the firewall in support of a classic service-oriented architecture.
- Asynchronous or disconnected calls implemented with queued messaging patterns.
- Workflow services for long running business operations.
- Interoperable web services based on SOAP protocol and advanced WS* protocols.
- Web programming models with support for Plain-Old-XML (POX), Javascript Object Notation (JSON), Representational State Transfer (REST) and Syndication with RSS or AtomPub.

WCF was initially introduced with the .NET Framework 3.0 which released with Windows Vista in January 2007 –along with Windows Workflow Foundation (WF) and Windows Presentation Foundation (WPF). When the .NET Framework 3.5 released with Visual Studio 2008 in November 2007 additional WCF features were introduced including improvements to the web programming model and support for the latest WS* protocols. As mentioned earlier, as of the .NET Compact Framework 3.5 WCF is supported on mobile devices. To that end, mobile devices will typically consume WCF services as interoperable web services using a restricted set of protocols, or as POX and REST-based services – although nothing precludes you from calling services using JSON serialization, or syndication services – all it takes is a little elbow grease.

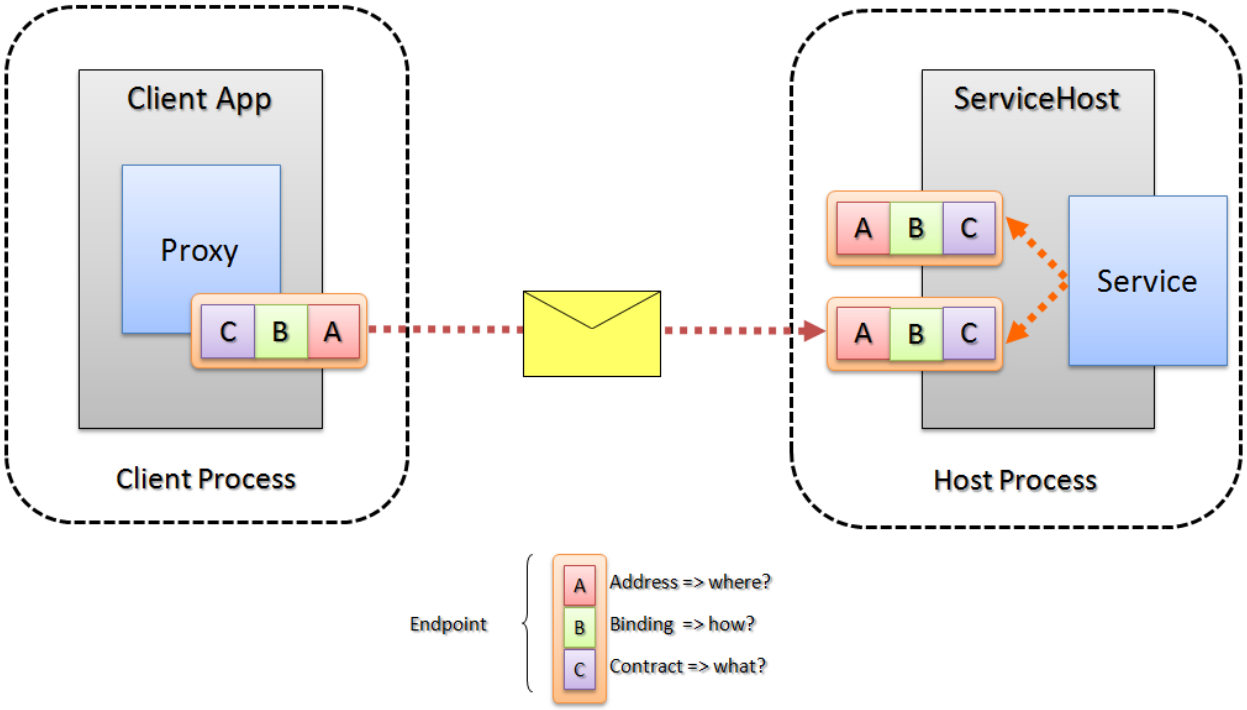
It is impossible to sum up a platform as vast and feature-rich as WCF in a short section however, it is possible to explain some fundamental concepts that will provide you with a foundation for subsequent sections in this paper. In this section we'll keep the discussion conceptual, since in later sections we will

Michele Leroux Bustamante and Nickolas Landry, May 2009

walk you through the steps required to accomplish creating, hosting and calling service in the context of mobile development.

Figure 2 provides a visual perspective on the fundamental architecture of a WCF service and WCF client. A service is a type that exposes one or more service operations (methods) for remote clients to call. These operations usually coordinate calls to the business and data tier to implement their logic, while the service itself is a boundary for distributed communications. Services can be hosted in any managed process – typically Internet Information Services (IIS) for mobile clients. Part of the hosting process is to initialize a ServiceHost instance, tell it which service type should be instantiated for incoming requests, and configure endpoints to tell it where requests should be sent. An endpoint comprises the address where messages should be sent, a binding which describes a set of protocols supported at the address, and a contract describing which operations can be called at the address and their associated metadata requirements. The ServiceHost can be initialized programmatically or by declarative configuration – in either case it will be initialized with one or endpoints that will trigger initializing the service type when request are received.

Figure 2: Fundamental architecture of a WCF implementation



In order for a client to call operations exposed by a hosted service it must have access to the service metadata (the contract) so that it can send messages with the required parameters, and process return values accordingly. Clients typically rely on proxy generation to improve their productivity for writing code to call remote services. Services enable metadata exchange in order to support proxy generation. The proxy generation tool (NetCFSvcUtil.exe for mobile clients) can look at the Web Service Description

Language (WSDL) document produced by the ServiceHost description – or interact with a live metadata exchange endpoint (based on WS-MetadataExchange protocol) – to produce a proxy for the client.

A proxy is a type that looks much like the service type, except that it does not include the implementation. This type is configured much the same as the ServiceHost – it requires access to one of the service endpoint addresses, the binding for that endpoint, and the contract. This way the client and service agree on where the proxy should send messages, what protocols should be used, and what data should be serialized to call the operation, along with what data will be serialized upon its return. It doesn't matter if the client proxy looks identical to the service type, nor if the supporting types are implemented exactly as they are at the service. What matters is wire-compatibility between the two so that either end can handle deserialization of the other's data.

There are a number of features to choose from when you configure a service and its bindings including:

- Message size quotas
- Sessions and timeouts
- Security including transport or message security and choice of credentials
- Reliable messaging
- Transactions

These features, along with other messaging protocols and behaviors all contribute to how messaging works between client and service. Since the .NET Compact Framework implements a subset of WCF features, services must also be careful not to require features that a mobile client is unable to communicate with. Of course this is a subject that we will address throughout this paper.

Setting Up Your Mobile Development Environment

Before doing any job, you have to get the right tools, and Windows Mobile development is no exception. If you're already a .NET developer, you already have the primary tool you need for .NET Compact Framework application development: Visual Studio 2008. Beyond that, you'll need additional libraries, emulators and resources to help you develop end-to-end mobile applications.

The list of tools you'll need includes:

- Visual Studio 2008 Professional (and higher)
- Windows Mobile 6 Professional and Standard Software Development Kits Refresh
- Microsoft Windows Mobile Device Center 6.1 for Windows Vista
- Power Toys for .NET Compact Framework 3.5

These tools, where to get them and how to install them are covered next.

Visual Studio 2008 Professional (and higher)

Microsoft Visual Studio 2008 is the primary development tool used when creating mobile device applications. The .NET Compact Framework is the runtime platform for your application development

system for Windows Mobile devices. Smart Device Projects for Visual Studio allow you to design, debug, and deploy .NET Compact Framework applications on many of today's Windows Mobile devices. All code samples based on this whitepaper require .NET Compact Framework 3.5.

Visual Studio 2008 provides the following for developing Smart Device applications:

- Project templates for creating Smart Device applications that target Pocket PC or Smartphone, and for creating supporting class libraries based on the .NET Compact Framework.
- Several high fidelity, onscreen device emulators – including emulators for Windows Mobile 5.0 Pocket PC and Smartphone, as well as Pocket PC and Smartphone devices running on Windows Mobile 2003. Additional emulators can be installed in Visual Studio 2008 via the Windows Mobile 6 SDKs.

Windows Mobile 6 Professional and Standard Software Development Kits Refresh

While Visual Studio supports Windows Mobile 6.0 development, you must install additional mobile development Software Development Kits (SDK) such as the Windows Mobile 6.0 Professional SDK for Pocket PC development and the Windows Mobile 6.0 Standard SDK for Smartphones development. These SDKs extend Visual Studio 2008 and the Smart Device Projects by providing additional libraries that can be used to access device specific features through managed code. The code samples in this white paper are designed for a Windows Mobile 6.0 Pocket PC device or emulator and therefore require the Windows Mobile 6.0 Professional SDK. Note that the techniques demonstrated in this white paper are easily usable on Windows Mobile 6 Standard devices (i.e., Smartphone).

Both SDK's can be downloaded from the Microsoft web site at http://www.microsoft.com/downloads/details.aspx?familyid=06111A3A-A651-4745-88EF-3D48091A390B&displaylang=en.

You can also download the latest Windows Mobile 6.1.4 emulator images from http://www.microsoft.com/downloads/details.aspx?familyid=1A7A6B52-F89E-4354-84CE-5D19C204498A&displaylang=en.

Microsoft Windows Mobile Device Center 6.1 for Windows Vista

The Windows Mobile Device Center (WMDC) sits on top of the Windows Vista Sync Center to manage partnerships, connectivity, and synchronization with Windows Mobile devices. WMDC is fully compatible with Windows Vista but requires a separate installation as it is not part of the standard operating system feature set.

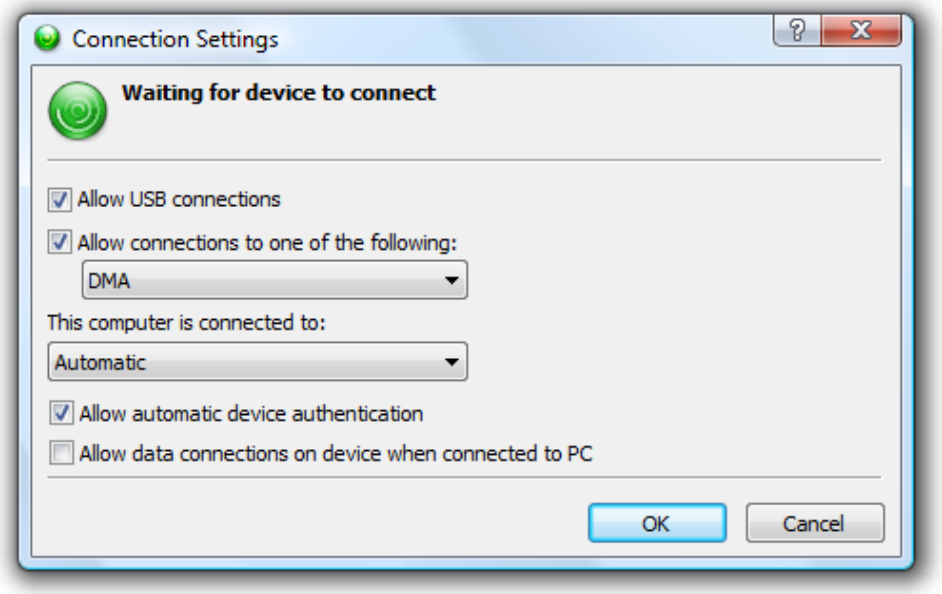
Download the 32-bit version of WMDC 6.1 for Vista at http://www.microsoft.com/downloads/details.aspx?familyid=46F72DF1-E46A-4A5F-A791-09F07AAA1914&displaylang=en.

Michele Leroux Bustamante and Nickolas Landry, May 2009
Download the 64-bit version of WMDC 6.1 for Vista at
http://www.microsoft.com/downloads/details.aspx?familyid=4F68EB56-7825-43B2-AC89-2030ED98ED95&displaylang=en

WMDC replaces ActiveSync, which was used in Windows XP to manage connectivity to Windows Mobile devices. Once you have installed WMDC, connect your Windows Mobile device. WMDC greets you with a Welcome screen and offers you a choice between setting up a device on your computer – which creates a new partnership with this device – or, connecting the device with no setup, similar to ActiveSync's "guest" mode. When connecting to device emulators or development devices in Visual Studio 2008, it is recommended to simply connect with no device setup (i.e., guest partnership).

For mobile development with Visual Studio, make sure the connection settings are as shown in Figure 3.

Figure 3: WMDC connection settings for mobile development



- **Allow USB connections** is checked
- For **Allow connections to one of the following**, select DMA in the dropdown list. Direct memory Access (DMA) is the fast native protocol used by the Visual Studio for Devices Debugger to establish connections for deployment and remote debugging with devices and emulators.
- For **This computer is connected to**, select Automatic to make sure pass-through connectivity is automatically enabled between your device or emulator and your development workstation. This pass-through connectivity allows the connected device or emulator to share your workstation's network and Internet connectivity, which is essential for Mobile WCF development.
- **Allow data connections on device when connected to PC** must be unchecked since this feature is only used when a Mobile PC notebook needs to "borrow" the wireless cellular connectivity of the device, thus acting as a wireless modem.

All the code examples in this document require connectivity between the device emulator and the development workstation. It is therefore essential that Windows Mobile Device Center be installed and functional prior to getting started with any of the samples presented here.

Michele Leroux Bustamante and Nickolas Landry, May 2009

Power Toys for .NET Compact Framework 3.5

The Power Toys for the .NET Compact Framework 3.5 provide several tools for Windows Mobile development. The most relevant to this document is the **ServiceModel Metadata Tool for the .NET Compact Framework** (NetCFSvcUtil). We'll use this to generate proxies to call WCF services, similar to SvcUtil for the full .NET Framework and WCF platform.

You can download the Power Toys for the .NET Compact Framework 3.5 at http://www.microsoft.com/downloads/details.aspx?FamilyID=c8174c14-a27d-4148-bf01-86c2e0953eab&DisplayLang=en.

Final Steps

After installing the tools just discussed you are now ready to start building Windows Mobile applications with .NET Compact Framework and Visual Studio 2008. The following are additional tips and considerations to keep in mind as you build mobile applications:

- It will be easier for you to develop mobile applications communicating with WCF services if you disable the User Account Control (UAC) feature in Windows Vista which is used to request an elevation of privilege for certain operations using an Allow/Deny dialog option. However do this at your own risk and make sure that your computer has all the latest Windows security patches and that you have an up-to-date security suite installed and running to prevent intrusions from viruses and other malware.

  To disable UAC, open the Control Panel, select **User Accounts**, and then **Turn user Account Control on or off**. Clear the checkbox and select **Ok**. A reboot will be required. Remember to turn UAC back on when you are done with your development session as a precaution.
- When running and testing your mobile applications on the device emulator, remember that when closing the emulator you should select the option to save the state when prompted. This prevents the long deployment and installation of runtime packages, such as .NET Compact Framework 3.5 or SQL Server Compact Edition, the next time you use that emulator for development. It also ensures that you save any files copied to the device for testing, such as certificates we ask you to install for security scenarios.

Getting Started: Your First Mobile + WCF Application

It's now time to create your first mobile client application that leverages a WCF service on the server. If you are new to WCF or mobile development it will be helpful to take a simple scenario that summarizes the steps to create a WCF service and consume it from a mobile application. This section will walk through the steps to do so assuming that you have set up your environment as discussed previously. We will create a greeting service that requests the user's name and sends it to the service to retrieve a personalized greeting message. The WCF service will be hosted on your desktop machine, and the mobile client on the device emulator.

Creating and Hosting the WCF Service

Let's start by creating a simple WCF service, discussing some of the design constraints for mobile scenarios. To keep things simple, this will be a self-hosted service (not hosted in IIS). The steps for designing, implementing and hosting a WCF service are as follows:

- Design the service contract
- Implement the service contract on a service type
- Supply one or more service endpoints, custom binding configurations if applicable, and configure runtime behaviors
- Initialize the ServiceHost for that service type

Consider the simple example of a service contract and service type implementation shown in Figure 3. The service contract is a CLR interface (IGreetingService) decorated with the ServiceContractAttribute, each method decorated with the OperationContractAttribute to include it in the service metadata. You typically supply a Namespace to the ServiceContractAttribute to disambiguate messages on the wire. So far, this is typical for any WCF service.

The service type (GreetingService) implements the service contract and in theory would coordinate calls to the business and data tier. In this case, it is a simple service and the implementation is embedded directly into the service type. The ServiceBehaviorAttribute is used to apply service behaviors that are tightly coupled to the implementation.

Figure 3: A simple service contract and implementation for mobile clients

```
[ServiceContract(Namespace = "urn:mobilewcf/samples/2009/04")]
public interface IGreetingService
{
    [OperationContract]
    string Greet(string firstname);
}

public class GreetingService : IGreetingService
{
    public string Greet(string firstname)
    {
        if (string.IsNullOrEmpty(firstname))
        {
            Console.WriteLine("ERROR: You must provide a valid name to
receive a greeting.");
            throw new InvalidOperationException("You must provide a valid
name to receive a greeting.");
        }

        string s = string.Format("Greetings {0}.", firstname);
        Console.WriteLine(s);
        return s;
    }
```

```
}
```

Once you have designed and implemented the service – keeping in mind mobile limitation – you can host the service and expose one or more endpoints for client applications. Assuming you are hosting on Windows Server 2003 or Windows Server 2008 machines – you will typically host services in Internet Information Services (IIS), Windows Process Activation Service (WAS) or self-host in a Windows Service. The only requirement is that you expose one or more HTTP-based endpoints compatible with mobile clients.

For testing purposes, in this example we will host the service in a simple Console application. Figure 4 shows the code to initialize the ServiceHost.

Figure 4: Initializing the ServiceHost for the GreetingService

```
ServiceHost host = new ServiceHost(typeof(GreetingService));
try
{
    host.Open();
    Console.ReadLine();
}
finally
{
    if (host.State == CommunicationState.Faulted)
        host.Abort();
    else
        host.Close();
}
```

Instead of hard-coding endpoints, this code assumes that endpoints will be configured in the app.config (or web.config for IIS and WAS hosting). Figure 5 shows the <system.serviceModel> section for the GreetingService described in Figure 3. A single endpoint is exposed over BasicHttpBinding – which means it is a simple SOAP 1.1 endpoint without security features. This is a good way to test that your mobile application can reach the service before you begin adding necessary security features – which is covered later. Additionally, you'll want to expose a metadata exchange endpoint, using MexHttpBinding (a variation of WSHttpBinding without security) so that you can generate a proxy for the mobile application.

The associated service behavior (see the <serviceBehaviors> section) enables the ServiceMetadataBehavior to support the metadata exchange endpoint, and to enable HTTP browsing (so that you can view the WSDL document in the browser). The debug setting to includeExceptionDetailsInFaults should be set to false in production deployments – and, frankly, this setting is not very useful for debugging mobile application since the mobile implementation of WCF in the .NET Compact Framework leaves much to be desired in the way of handling faults thrown by the service channel.

Figure 5: A simple example of a mobile-compatible service model configuration

```
<system.serviceModel>
  <services>
    <service name="Greetings.Services.GreetingService"
behaviorConfiguration="serviceBehavior">
      <endpoint address="GreetingService" binding="basicHttpBinding"
contract="Greetings.Services.IGreetingService" />
      <endpoint address="mex" binding="mexHttpBinding"
contract="IMetadataExchange" />
      <host>
        <baseAddresses>
          <add baseAddress="http://localhost:8000"/>
        </baseAddresses>
      </host>
    </service>
  </services>
  <behaviors>
    <serviceBehaviors>
      <behavior name="serviceBehavior">
        <serviceDebug includeExceptionDetailInFaults="false"/>
        <serviceMetadata httpGetEnabled="true"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

With this you now have a mobile-compatible service configuration and can begin to look at creating your first mobile application to consume this simple WCF service.

Creating a Simple Mobile Application

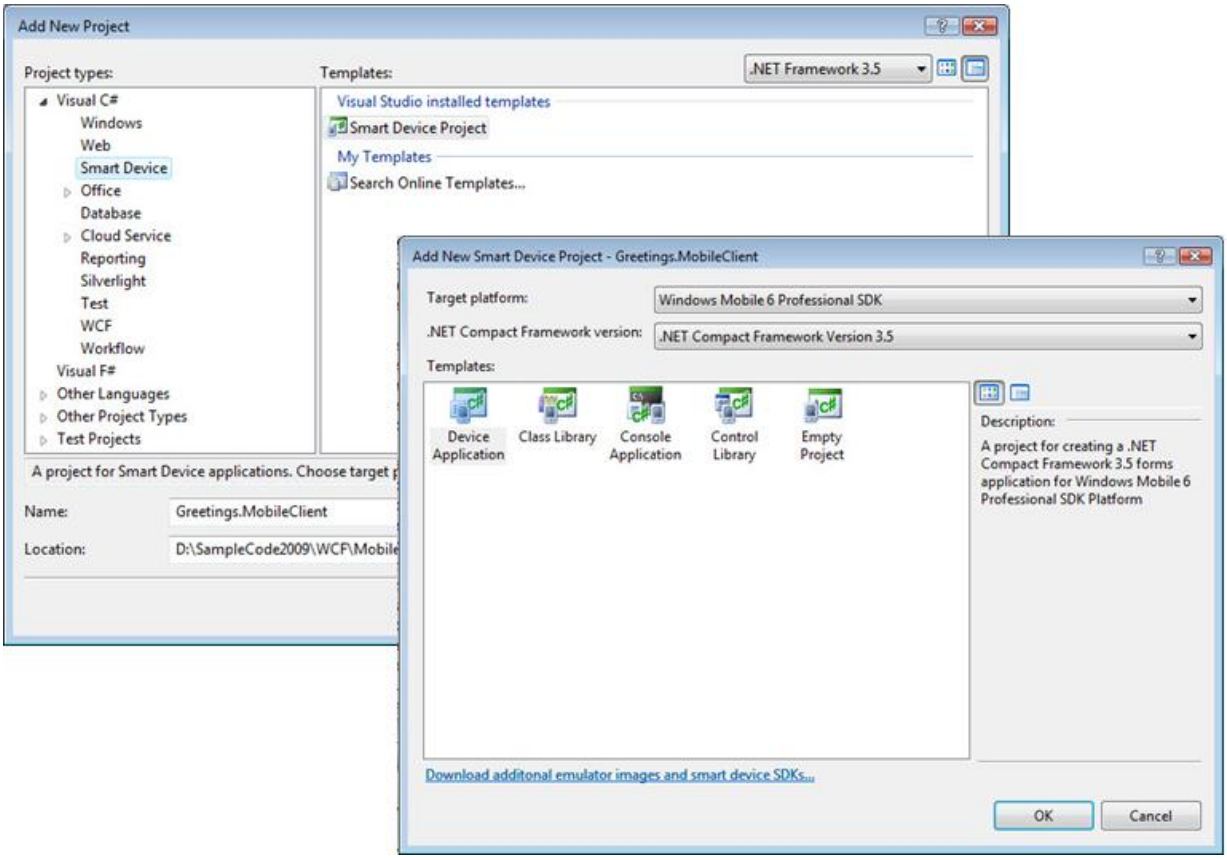
After installing the prerequisites described earlier you will have access to the latest templates and emulators to build and debug a mobile application. To begin creating your first mobile application, create a new project in Visual Studio 2008, select the Smart Device project type and select the Smart Device Project template.

This will lead to a second dialog (both shown in Figure 6 below) from which you can select from a few .NET Compact Framework project templates. It is important that you create mobile device projects using one of these templates as this restricts the project to using only .NET Compact Framework assemblies so that you are working with the right functionality sandbox. By default, the .NET Compact Framework version will be 3.5 – and this is what you want to support WCF client code. Version 2.0 of .NET Compact Framework is also supported in Visual Studio 2008, but you need version 3.5 for mobile WCF.

As for the target platform you can select from Windows Mobile 6 Professional or Standard SDK – depending on if you want to create a Pocket PC or Smartphone application. With regards to communication with WCF services, both support .NET Compact Framework 3.5. The code samples here use the Windows Mobile 6 Professional SDK (i.e. Pocket PC). Note that you can change the target SDK of

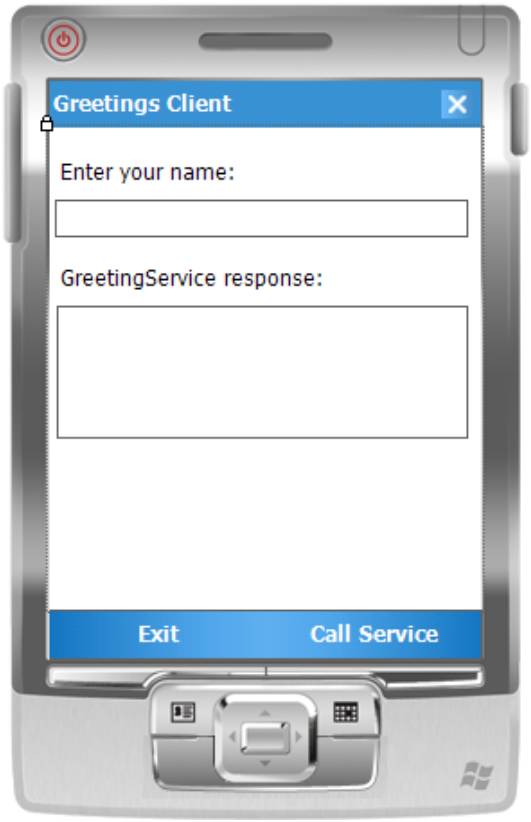
your project later in Visual Studio 2008 by selecting the Change Target Platform option under the Project menu.

Figure 6: Smart Device templates for creating mobile applications



In this case, you will use the Device Application template to create a mobile client, which is the mobile equivalent of a standard Windows Forms project. This generates a properly-sized form that presents a device image where you can drop controls as you would any Windows Forms or Windows Presentation Foundation (WPF) application to design the UI. Figure 7 shows the UI for the GreetingService sample.

Figure 7: A mobile client application for the GreetingService



The point of this section was to make sure you are aware of the requirement to use Smart Device templates for mobile clients. Next we'll explain how to generate a proxy to call a WCF service from a mobile application.

Proxy Generation for Mobile Applications

Traditionally you would use Add Service Reference from the Solution Explorer in Visual Studio 2008, or directly call SvcUtil.exe (SvcUtil) to generate proxies for WCF client applications. For mobile applications there isn't an Add Service Reference function available to device projects, and you shouldn't use SvcUtil to generate proxies since this does not generate code compatible with the subset of WCF functionality available to the .NET Compact Framework 3.5 on mobile devices. Fortunately, the Power Toys for .NET Compact Framework 3.5 (mentioned earlier) include a SvcUtil equivalent for .NET Compact Framework – NetCFSvcUtil.exe (NetCFSvcUtil).

NetCFSvcUtil can be used from a console command line, like SvcUtil, to generate a proxy and related contracts from a WSDL document or from a metadata exchange endpoint (like the one we enabled earlier). So, for example, you can write a batch file that generates a proxy for the GreetingService shown in Figure 3 by using metadata exchange against the base address shown configured in Figure 5. The following example illustrates generating a proxy named GreetingServiceClient for the GreetingService, including a namespace consistent with the client application for generated code:

```
netcfsvcutil.exe /namespace:*,Greetings.MobileClient
/out:GreetingServiceClient http://localhost:8000
```

Two source code files are generated from this command: CFClientBase.cs and GreetingServiceClient.cs. These files are saved in the same folder from where you ran the command. The contents of CFClientBase.cs are consistent to all services for which proxies are generated. Types generated in GreetingServiceClient.cs are specific to the GreetingService.

When you use NetCFSvcUtil to generate a proxy, it inspects the service endpoint configuration and its metadata to ensure they are mobile-compatible. If the target service is not compatible, an error will be displayed. For example, if the service doesn't expose any endpoints compatible with mobile binding requirements – a proxy cannot be generated.

Since mobile devices do not support application configuration files natively (i.e., due to the missing classes in System.Configuration), .NET Compact Framework applications do not have access to the code necessary to process the <system.serviceModel> configuration section – NetCFSvcUtil generates code to produce the endpoint address and binding configuration. For example, the GreetingServiceClient proxy exposes a static property called EndpointAddress as follows:

```
public static EndpointAddress EndpointAddress = new
EndpointAddress("http://localhost:8000/GreetingService");
```

In addition, a static method called CreateDefaultBinding() returns a runtime representation of the binding. In this case, a CustomBinding equivalent for BasicHttpBinding defaults is returned:

```
public static Binding CreateDefaultBinding()
{
    CustomBinding binding = new CustomBinding();
    binding.Elements.Add(new
TextMessageEncodingBindingElement(MessageVersion.Soap11, Encoding.UTF8));
    binding.Elements.Add(new HttpTransportBindingElement());
    return binding;
}
```

NOTE: One apparent limitation seems to be that NetCFSvcUtil only generates code for the first endpoint it finds that is compatible with mobile devices – during proxy generation.

Once you have generated the proxy using NetCFSvcUtil you can write code to call the service from your mobile application. The following code illustrates using the static binding and endpoint address exposed by the proxy, GreetingServiceClient, to initialize the proxy before calling the Greet() operation:

```
Binding binding = GreetingServiceClient.CreateDefaultBinding();
string address = GreetingServiceClient.EndpointAddress.Uri.ToString();

GreetingServiceClient m_proxy =
new GreetingServiceClient(binding, new EndpointAddress(address));
string result = m_proxy.Greet(txtName.Text).ToString();
```

NOTE: Mobile devices are unable to resolve "localhost" to the correct IP address or machine name where the service is hosted. This is due to the fact that the device or emulator are considered to be

separate computers, therefore localhost would resolve back to the mobile device itself, not the development workstation. A workaround for this issue to streamline development is discussed later in this document.

Testing the Solution

You have so far created a WCF service and hosted it with mobile-compatible endpoints, created a simple mobile application, generated a proxy using NetCFSvcUtil, and wrote code to invoke the service. Now, it is time to test the code! In order to do this there are yet a few more steps to be aware of for running your mobile application in a mobile device emulator.

The following steps need only be performed once to set up your testing environment for your mobile application. This is essential to make sure that your device emulator is "cradled" to Windows Mobile Device Center (WMDC), thus "borrowing" network and Internet connectivity from your development workstation to your device.

- Launch Windows Mobile Device Center via the Windows Vista Start menu or from the Control Panel.
- From Visual Studio 2008, select Connect to Device under the Tools menu. This lists all the available emulators compatible with your mobile project. Select the Windows Mobile 6 Professional Emulator and click Connect. Wait as the emulator is loaded and initialized.
- From Visual Studio 2008, launch the Device Emulator Manager (DEM) under the Tools menu. Locate the emulator you just launched in the list (it has a small "Play" icon next to it as shown in Figure 8).
- Select the active emulator you located, right-click it and pick Cradle (i.e., the equivalent of connecting a device to your computer via a USB cable) to establish a connection between the emulator and WMDC, which will show the device status as "Connected", and the "Play" icon next to your emulator in DEM will change to a "cradle" icon.
- The final step is to select the option to connect to the device without setting it up, from WMDC.
- Your emulator is now ready for deployment and has network and Internet access. At this point, you can use WMDC to interact with the device, such as copying files to and from the device for testing. This is useful for copying certificates for security scenarios, for example.

Figure 8: The Device Emulator Manager and Windows Mobile Device Center

Once you have executed these steps, the Device Emulator Manager and will continue to run in the background, and the selected emulator will also remain open in a separate window. Even if you close down all Visual Studio instances, these continue to run – which saves you time if you reload your project and wish to continue testing with the same emulator. You will only have to repeat these steps if you close the emulator window (which can be a force of habit while debugging) or if you shut down the DEM with brute force.

Now, with the emulator running you can test your code. It is helpful to test the mobile client from a separate Visual Studio instance. Run the service host first, and then run the mobile client application in debug mode. This will bring up a deployment dialog asking where you want to deploy your mobile application. Select Windows Mobile Professional Device. By selecting the device instead of the emulator you are deploying to whatever device the WMDC is connected to – which will be helpful when you are testing deployment to a physical cradled device. This dialog will be presented each time you debug the client.

Contract Design

Contract design is the first step in implementing a WCF service. Service contracts define the operations that will be exposed to clients, and in some cases place requirements on communications between clients and services. Service operation parameters and return values define the messaging requirements for each operation – and are usually serializable complex types such as data contracts. Service operations can also use message contracts to support advanced SOAP messaging requirements such as custom message headers. Fault contracts provide information to clients about the type of SOAP fault details – beyond a simple fault with an error message – that can be returned by service operations.

You can define contracts that are compatible with both non-mobile and mobile clients. This may not make sense in all cases because of mobile client considerations such as:

- Mobile clients prefer chunky over chatty interfaces. This optimizes transfer overhead and reduces the number of calls to the service given the latency of mobile communications.

- Mobile client applications are not likely to leverage the same number of application features as a non-mobile client – thus it may make sense to design service contracts that isolate what the mobile client needs in terms of functionality. The service implementation or the business components behind it should still be shared to reduce maintenance overhead but this reduces the noise for mobile developers and targets how they access remote functionality.

If contracts will be shared by non-mobile and mobile clients you must avoid requiring features not supported by the .NET Compact Framework. A short summary of features related to contract design is shown in Figure 9.

Figure 9: Contract-related features and support for mobile clients

Feature	Comments
Streaming	Streaming not supported. Service contract can use Stream parameters but they will be buffered, not streamed. Cannot enable streaming on the binding.
Sessions	Transport sessions not supported. Service contract can use SessionMode.Allowed. Service should use InstanceContextMode.PerCall behavior.
Duplex	Services designed for duplex communications with callback contracts cannot be called by mobile clients. Duplex requires a transport session.
Transactions	Service contract cannot require transactions for any service operations. Mobile clients cannot flow transactions.
Data Contracts and Serializable Types	Can freely employ data contracts or any other serializable types in the service contract definition. The mobile client will use XmlSerializer types that are wire compatible.
Message Contracts	Can freely employ message contracts in the service contract. If the message contract includes custom headers proxy generation will not work. The mobile client can work with headers with additional custom code.
Fault Contracts	Can freely employ fault contracts in the service contract. Proxy generation will not include fault contracts and does not handle faults well. Can write custom code for mobile client to work with faults.

The following sections will discuss these features and limitations with additional detail.

Service Contracts

Generally speaking you will approach service contract design for mobile clients as you would for any WCF service. As discussed earlier a service contract is typically implemented as a CLR interface with the ServiceContractAttribute. The OperationContractAttribute is also applied to all operations you want to expose as part of the service implementation. Figure 10 illustrates the service contract for the TodoListService.

Figure 10: Service contract for the TodoListService type, ITodoListService

```
[ServiceContract(Namespace="urn:mobilewcf/samples/2009/04")]
public interface ITodoListService
{
    [OperationContract]
    List<TodoItem> GetItems();

    [OperationContract]
    string CreateItem(TodoItem item);
    [OperationContract]
    void UpdateItem(TodoItem item);
    [OperationContract]
    void DeleteItem(string id);
}
```

The equivalent implementation at the client is an interface including a method for each service operation – minus the attributes. The proxy, TodoListServiceClient in this case, implements this interface and inherits the base proxy, CFClientBase, which handles the serialization dirty work. Figure 11 illustrates the client-side implementation (without the proxy implementation details).

Figure 11: Client-side equivalent for the ITodoListService service contract, also implemented on the proxy type TodoListServiceClient

```
public interface ITodoListService
{

    TodoItem[] GetItems();

    string CreateItem(TodoItem item);

    void UpdateItem(TodoItem item);

    void DeleteItem(string id);
}

public partial class TodoListServiceClient :
Microsoft.Tools.ServiceModel.CFClientBase<ITodoListService>, ITodoListService
{…}
```

There are some WCF features that are not supported in mobile scenarios which impacts the usage of these fundamental service contract attributes, in addition to other attributes that you might apply to a service contract or its operations. They are streaming, sessions, duplex and transactions.

Streaming

Only buffered messages are supported by the .NET Compact Framework. It is perfectly legal to use Stream parameters and return types in your service contracts, however the data will be buffered, not streamed. That's because Stream parameters are only streamed when exposed by an endpoint that supports streaming in the binding configuration – which is not supported. Since the behavior for a mobile client would be very different from that of a non-mobile client that supports streaming – you

may want to consider designing a new service contract for mobile clients so that the buffering semantics are clear.

Sessions

Mobile clients cannot maintain a session with a service – which means that only simple HTTP bindings are supported for mobile endpoints. With respect to service design, there are two implications. First, the service contract cannot require sessions – for if it does the contract cannot be exposed by a simple HTTP endpoint. Instead, service contracts should probably use SessionMode.Allowed (the default) which gives developers the freedom to expose service endpoints over Named Pipes, TCP or HTTP bindings that support session – for non-mobile clients – while still supporting mobile clients with simple HTTP endpoints.

```
[ServiceContract(Namespace = "urn:mobilewcf/samples/2009/04",
SessionMode=SessionMode.Allowed)]
public interface IGreetingService
```

Using SessionMode.NotAllowed is also acceptable, except that this precludes exposing endpoints over any binding that supports session – which is like saying you can only expose these service operations over simple HTTP.

Since sessions are not supported by mobile clients, it doesn't make sense to call a service that maintains session state between calls for clients. The default behavior for a service type is InstanceContextMode.PerSession, so to avoid unexpected results you should explicitly apply the instancing behavior InstanceContextMode.PerCall.

```
[ServiceBehavior(InstanceContextMode=InstanceContextMode.PerCall)]
public class GreetingService : IGreetingService
```

If the service is a singleton (not useful in most distributed environments) this is also acceptable, so long as session is not maintained as defined by the contract setting.

Duplex

Duplex communications require a transport session (Named Pipes, TCP or HTTP Dual) and this is not supported by NetCF. Service contracts designed for duplex communications cannot be called by mobile clients.

Transactions

Distributed transactions are not supported by the .NET Compact Framework Contract – so the mobile client can never flow transactions to the service. The implication of this is that the service contract cannot require transactions from the client. This means that service operations decorated with the TransactionFlowAttribute cannot use TransactionFlowOption.Mandatory. It is, however, acceptable for the service to support transactions if the client wants to send one. Thus, the same contract can be called by mobile and non-mobile clients if it uses TransactionFlowOption.Allowed.

```
[OperationContract]
[TransactionFlow(TransactionFlowOption.Allowed)]
string Greet(string firstname);
```

The default behavior is TransactionFlowOption.NotAllowed – which means you likely do not use the TransactionFlowAttribute unless you want to opt-in support for transactions at the service.

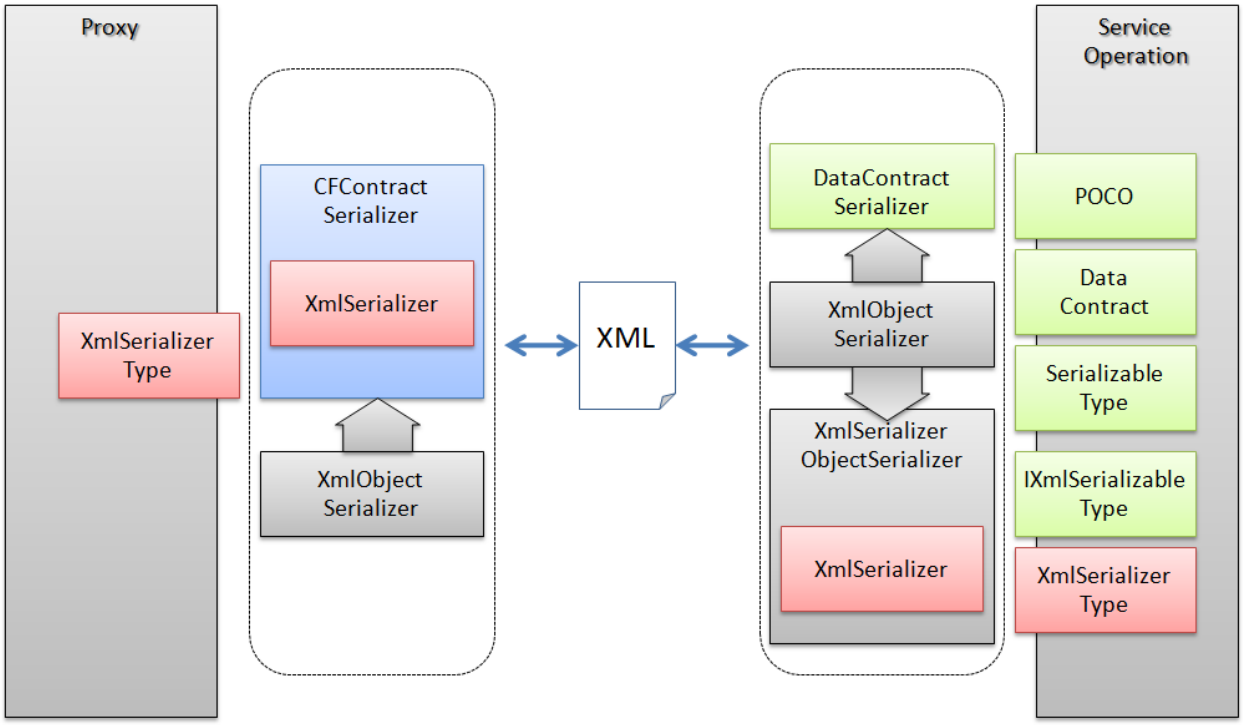
Complex Types and Serialization

Service contracts support many acceptable formats complex types as parameters or return values including Plain-Old-CLR-Objects (POCO), data contracts, serializable types, types that implement the IXmlSerializable interface, and XmlSerializer types – which we'll collectively refer to as serializable types.

Serialization Architecture

Figure 11 illustrates serialization of various complex types at the service, and the serialization equivalent for mobile clients. By default, the services use the DataContractSerializer to handle message serialization which includes mapping between serializable types and XML. All but XmlSerializer types are typically serialized by the DataContractSerializer. You can use the XmlSerializer as an alternative to the DataContractSerializer – usually when you have existing XmlSerializer types or when you need to customize how a type is serialized beyond what the DataContractSerializer supports. This means working with XmlSerializer types instead of the other serializable formats – although in theory any type with public fields or properties can be serialized with the XmlSerializer as well.

Figure 11: Serialization of complex types for mobile clients and services



At the mobile client, the XmlSerializer is always used for message serialization, and complex types are always XmlSerializer types. Proxy generation using NetCFSvcUtil produces the code to handle serialization through the XmlSerializer, and produces wire-compatible versions of complex types defined at the service.

The messaging layer of WCF requires you to supply a type that derives from XmlObjectSerializer for message serialization. The DataContractSerializer is an implementation of the XmlObjectSerializer, but the XmlSerializer is not. At the service, if you apply the XmlSerializerFormatAttribute to the service contract (or, service operation) the XmlSerializer is used – through an internal wrapper class that inherits XmlObjectSerializer – XmlSerializerObjectSerializer. Figure 11 also illustrates the relationship between these types at the service.

This detail is important to mobile clients since they do not use traditional service contracts for serialization – everything is done directly at the messaging layer of WCF inside the proxy. NetCFSvcUtil generates an implementation of the XmlObjectSerializer called CFContractSerializer. As illustrated in Figure x this type wraps the use of the XmlSerializer to handle message serialization, much like the equivalent at the service, the XmlSerializerObjectSerializer. When a message is created to send to the service, an instance of this serializer is supplied to handle serialization. Likewise, when a response is received an instance of this serializer is created to handle deserialization. Since the proxy base type CFClientBase handles this for you, the details of message and complex type serialization are not apparent to your mobile client code. You work with instances of the XmlSerializable complex types, and a proxy that looks much like the service type.

Complex Types and Wire Compatibility

As illustrated in Figure 11, at the service any number of complex type formats can be used to describe the messaging requirements for each service operation – although the preferred route is to use data contracts. Figure 12 lists each of the supported formats and reasons to use them.

Figure 12: Summary of complex type formats at the service

Serialization Format	Usage
POCO	Plain-Old-CLR-Type (POCO) aptly refers to CLR types not decorated with any serialization attributes. Only public properties and fields are included in serialization and the type must not be part of a type hierarchy that includes non-POCO types. This is useful for simple scenarios that do not require a long term strategy for contract versioning, nor for supporting other serializable types in the type hierarchy.
Data Contract	Data contracts are the preferred format for WCF services. Types are opted-in as data contracts using the DataContractAttribute and DataMemberAttribute (to be discussed). Data contracts provide control over the serialization namespace for types, serialization order, and required members.
Serializable	Serializable types are types decorated with the SerializableAttribute. Only fields are serialized (wire-level serialization) and there is no control over namespaces,

	order, or which fields should be serialized. These types are usually used only when you need to serialize existing types in assemblies that you don't own or can't recompile.
IXmlSerializable	IXmlSerializable types are an advanced type that provides control over how a type is mapped to and from XML. This is useful in scenarios where a very complex XSD schema must be supported for a service operation, which precludes relying on the DataContractSerializer or XmlSerializer to automate serialization. With this type you can provide the schema to be used for a particular parameter or return type and it will be included in the service metadata for proxy generation at the client.
XmlSerializer	The XmlSerializer is useful for handling serialization requirements that the DataContractSerializer doesn't support such as mapping properties to XML attributes instead of XML elements, or customizing the data type for serialization. This is often important for platform interoperability.

The complex type format used at the service is irrelevant to the mobile client. The service metadata and WSDL include XSD schema definitions for types – and proxy generation is based on the schema. As already mentioned, complex types are always represented as XmlSerializer types for the mobile client – but the result is wire-compatible with the service.

To provide some context, let's take a look at a few examples. Consider the service contract in Figure x which exposes operations to interact with a collection of "todo" items described by the complex type TodoItem. If the TodoItem type is implemented as the POCO type shown in Figure 13, the following are characteristics of the serialized type:

- All public fields and properties are serialized as XML elements
- Elements are ordered alphabetically
- No elements are considered required

Figure 13: The TodoItem type implemented as a POCO type

```
public class TodoItem
{
    public string Id { get; set; }
    public string Title { get; set; }
    public string Description { get; set; }
    public DateTime DueDate { get; set; }
    public string Tags { get; set; }
}
```

When NetCFSvcUtil generates a proxy for the TodoListService it also generates the XmlSerializer type shown in Figure 14 to represent the POCO type from Figure 13. Here are a few relevant points about the client-side type:

- It relies on XmlSerializer attributes such as XmlTypeAttribute and XmlElementAttribute to provide instructions for serialization.

- The XmlTypeAttribute indicates the namespace for the type, which matches the default namespace used by the POCO type (since the POCO type doesn't provide a namespace).
- Each public property from the POCO type is included as a public property in the generated type.
- The Order property of the XmlElementAttribute provides order matching the alphabetical order of the POCO type's properties.
- Since the DueDate cannot be nullable (it is a value type) a DueDateSpecified property is created and must be set to true for the DueDate to be serialized. DueDateSpecified will not be serialized because it is decorated with the XmlIgnoreAttribute.

The result is a wire-compatible type with the POCO type at the service.

Figure 14: XmlSerializer type definition, wire-compatible with the POCO type in Figure 13

```
[System.SerializableAttribute()]
[System.Xml.Serialization.XmlTypeAttribute(Namespace=
"http://schemas.datacontract.org/2004/07/Entities")]
public partial class TodoItem
{

    private string descriptionField;

    private System.DateTime dueDateField;

    private bool dueDateFieldSpecified;

    private string idField;

    private string tagsField;

    private string titleField;

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=0)]
    public string Description
    {
        get
        {
            return this.descriptionField;
        }
        set
        {
            this.descriptionField = value;
        }
    }

    [System.Xml.Serialization.XmlElementAttribute(Order=1)]
    public System.DateTime DueDate
    {
        get
        {
            return this.dueDateField;
        }
        set
        {
```

```
            this.dueDateField = value;
        }
    }

    [System.Xml.Serialization.XmlIgnoreAttribute()]
    public bool DueDateSpecified
    {
        get
        {
            return this.dueDateFieldSpecified;
        }
        set
        {
            this.dueDateFieldSpecified = value;
        }
    }

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=2)]
    public string Id
    {
        get
        {
            return this.idField;
        }
        set
        {
            this.idField = value;
        }
    }

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=3)]
    public string Tags
    {
        get
        {
            return this.tagsField;
        }
        set
        {
            this.tagsField = value;
        }
    }

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=4)]
    public string Title
    {
        get
        {
            return this.titleField;
        }
        set
        {
            this.titleField = value;
        }
    }
}
```

The limitation of POCO types includes lack of control over namespace and order, in addition to the limitation that all types in the inheritance tree of a POCO type must also be POCO (not data contracts or otherwise serializable types). Generally data contracts are preferred at the service.

Consider the data contract implementation of the TodoItem type shown in Figure 15. Recommended practices for a data contract include providing a namespace to disambiguate serialized types on the wire and to support a versioning strategy, explicitly indicating which properties are required and not, and explicitly enforcing order rather than accepting the default alphabetical order. NetCFSvcUtil generates a wire-compatible type for this data contract as shown in Figure 16. This time, the following characteristics are worth noting:

- The XmlTypeAttribute uses the same namespace as the data contract.
- The XmlElementAttribute for each property enforces order consistent with the data contract.
- Since DueDate is required there is no longer a DueDateSpecified field to support omitting serialization of the element.

Figure 15: The TodoItem type implemented as a data contract

```
[DataContract(Namespace="urn:mobilewcf/samples/2009/04/schemas")]
public class TodoItem
{
    [DataMember(IsRequired=false, Order=1)]
    public string Id { get; set; }
    [DataMember(IsRequired = true, Order = 2)]
    public string Title { get; set; }
    [DataMember(IsRequired = true, Order = 3)]
    public string Description { get; set; }
    [DataMember(IsRequired = true, Order = 4)]
    public DateTime DueDate { get; set; }
    [DataMember(IsRequired = true, Order = 5)]
    public string Tags { get; set; }
}
```

Figure 16: XmlSerializer type definition, wire-compatible with the data contract in Figure x

```
[System.SerializableAttribute()]
[System.Xml.Serialization.XmlTypeAttribute(Namespace=
"urn:mobilewcf/samples/2009/04/schemas")]
public partial class TodoItem
{

    private string idField;

    private string titleField;

    private string descriptionField;
```

```
    private System.DateTime dueDateField;

    private string tagsField;

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=0)]
    public string Id
    {…}

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=1)]
    public string Title
    {…}

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=2)]
    public string Description
    {…}

    [System.Xml.Serialization.XmlElementAttribute(Order=3)]
    public System.DateTime DueDate
    {…}

    [System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=4)]
    public string Tags
    {…}
}
```

Fault Contracts

Service operations can optionally include fault contract declarations in order to produce metadata that indicates the type of exception details that may be reported by each operation. This is done by applying one or more FaultContractAttribute to the operation definition as follows:

```
[OperationContract]
[FaultContract(typeof(FaultDetail))]
List<TodoItem> GetItems();
```

The fault contract type can be any serializable type – but is usually a custom data contract that exposes properties to collect useful information to report to clients beyond a simple error message. The .NET Compact Framework does not support the FaultContractAttribute and so when NetCFSvcUtil is used to generate a proxy it ignores the presence of this metadata. In the Exception Handling section we will discuss strategies for working with faults at the mobile client.

Message Contracts

Message contracts are types that represent the SOAP message including message headers and the message body. They can be used in place of complex types in the service contract enabling the following scenarios:

- Adding custom message headers to message requirements for an operation – and publishing the custom header requirement with the service metadata
- Returning multiple complex types in an operation response.
- Serializing unwrapped messages, usually for compatibility with other platforms.
- Controlling which message header or message body elements should be signed or encrypted.

Most of these are advanced scenarios, the most popular of which is supporting custom headers.

Consider the following service operation that uses two message contracts for its parameter and return type – GetItemsRequest and GetItemsResponse, respectively:

```
[OperationContract]
GetItemsResponse GetItems(GetItemsRequest requestMessage);
```

The message contract for the request should include one or more message body member, one for each parameter that would have been passed to the operation. The response message contract should include at least a message body member for the return type. Message body members are properties decorated with the MessageBodyMemberAttribute. For each custom header to be received by a request, or to be returned with a response (less common) a property should be added and decorated with the MessageHeaderAttribute. All header and body members should be either POCO types or serializable types – usually data contracts.

Figure 17 shows the request and reply message contracts for the GetItems() operation shown previously. GetItems() returns the collection of TodoItem elements, thus GetItemsRequest does not have a body member. The response, GetItemsResponse, includes a single message body member which is the collection of TodoItem types to return.

Figure 17: Message contracts for the request and reply of GetItems()

```
[MessageContract]
public class GetItemsRequest
{
}

[MessageContract]
public class GetItemsResponse
{
    [MessageBodyMember]
    public List<TodoItem> Items { get; set; }
}
```

The service operation implementation can access body members through the message contract parameter or return type. In Figure 18, the request message contract has no body members, however to return the response an instance of GetItemsResponse must be created and its body member (the Items property) initialized.

Figure 18: Message contracts in the GetItems() implementation

```
public GetItemsResponse GetItems(GetItemsRequest requestMessage)
{
    return new GetItemsResponse{ Items=m_globalTodoList };
}
```

The presence of a message contract in a service operation definition is irrelevant to the mobile client. In fact, when a proxy is generated for any service contract an XmlSerializer type is created to represent the request and reply for each and every operation – regardless if the service used message contracts. Within these pseudo message contract types are the parameters or return types expected for the operation, which are also (as has been discussed) XmlSerializer types. Figure 19 illustrates the request and reply type for GetItems() at the client.

Figure 19: Pseudo message contracts generated by NetCFSvcUtil for the GetItems() operation

```
[System.Xml.Serialization.XmlRootAttribute(ElementName="GetItems",
Namespace="urn:mobilewcf/samples/2009/04")]
public partial class GetItemsRequest
{

    public GetItemsRequest()
    {
    }
}

[System.Xml.Serialization.XmlRootAttribute(ElementName="GetItemsResponse",
Namespace="urn:mobilewcf/samples/2009/04")]
public partial class GetItemsResponse
{

    [System.Xml.Serialization.XmlArrayAttribute(IsNullable=true,
Namespace="urn:mobilewcf/samples/2009/04", Order=0)]

[System.Xml.Serialization.XmlArrayItemAttribute(Namespace="http://schemas.dat
acontract.org/2004/07/Entities")]
    public TodoItem[] GetItemsResult;

    public GetItemsResponse()
    {
    }

    public GetItemsResponse(TodoItem[] GetItemsResult)
    {
        this.GetItemsResult = GetItemsResult;
    }
}
```

If a custom header were to be included in the message contract at the service, a header will also be added to the pseudo message contract generated for the client. However, a warning will be issued by during proxy generation and the header will not properly serialize. The details for working with custom headers will be discussed in a later section.

Proxy Generation

Earlier we walked you through the process of generating a proxy with NetCFSvcUtil for the GreetingService. Since then, we have discussed how service contracts, complex types and message

contracts are represented by the proxy; how the proxy leverages the XmlSerializer to handle message serialization; and some of the limitations of proxy generation from a high level. In this section we'll explore the options you have for generating proxies with NetCFSvcUtil and provide you with an architectural view of the resulting proxy and related files.

Working with NetCFSvcUtil

You can easily discover all of the options for NetCFSvcUtil by passing the /? option as shown here:

```
NetCFSvcUtil /?
```

The purpose of this section is not to regurgitate the long list of options, but to discuss the most common ways to use the tool. Figure 20 summarizes the most useful options available to NetCFSvcUtil.

Figure 20: Useful NetCFSvcUtil options for proxy generation

Option (Short Form)	Usage
/language (/l)	By default the tool generates C# code. To generate Visual Basic (VB) code, use this option.
/out (/o)	Controls the filename of the generated proxy. One useful pattern for this is to name this for the service type with the "Client" suffix. For example, if the service type is TodoListService, this option would use TodoListServiceClient.
/cfClientBase (/cb)	This option lets you rename the file that contains the proxy base type, CFClientBase, however this does not change the type name inside the file. Usually it is better to name file with the same name as the type for clarity so this option is not typically useful.
/namespace (/n)	By default custom proxy types generated by the tool are not wrapped in a namespace. This option wraps types in a CLR namespace of your choosing. Typically you will use a namespace that matches that of the assembly where you will include these files.
/reference (/r)	You can use this option to have the proxy reference types from a preexisting assembly instead of generating those types. This can be useful for sharing libraries between multiple projects.
/collectionType (/ct)	You should be able to use this option to control which array and dictionary types the proxy uses. For example, to use the generic type List<T> for arrays. Unfortunately this option does not appear to work so you would have to manually edit the generated proxy to modify arrays to use List<T>.
/enableDataBinding (/edb)	This option adds an implementation of the IPropertyNotifyChanged interface on complex types imported through proxy generation. This supports one-way or two-way data binding with UI components.
/messageContract (/mc)	This option should instruct the tool to generate message contracts (wrapper types for parameters and return values) for each service operation. It appears that the tool does this by default thus this option is not necessary.

First and foremost, you should always use the /namespace and /out options – perhaps also /language if you want to generate VB output. The following illustrates how to supply these options:

```
netcfsvcutil /language:vb /namespace:*,MobileClient
/out:TodoListServiceClient http://localhost:8000
```

In this case, VB code will be generated and the proxy type will be named TodoListServiceClient. The proxy and all supporting types (excluding the base functionality of CFClientBase) will be generated within the namespace MobileClient – which purposely matches the namespace of the client application. You can optionally map each XSD namespace from the service metadata to a different CLR namespace, but this is usually not necessary therefore by supplying "*" all types share the same namespace.

If you have a library that includes XmlSerializer types that can be used in place of generated types, you can supply the /reference options as shown here:

```
netcfsvcutil /namespace:*,MobileClient /out:TodoListServiceClient
/reference:sharedtypes.dll  http://mobilewcf.com:8000
```

The previous two examples illustrate generating a proxy for a self-hosted service at the base address http://mobilewcf.com:8000. That's because most of the code samples for this paper use a host header to provide an alternate name for the service hostname (this will be discussed further). You could also generate proxies for http://localhost:8000, or, use the IP address in place of localhost. If the service is hosted in IIS, your based address is the .svc endpoint as follows:

```
netcfsvcutil /namespace:*,MobileClient /out:TodoListServiceClient
http://mobilewcf.com/TodoListServiceWebHost/TodoListService.svc
```

Once again, localhost could be used in place of mobilewcf.com if you do not configure the IIS application to use an alternate hostname. We think for mobile development the custom hostname is the better approach so we'll describe how to achieve that later.

Since there is not the equivalent of Add Service Reference for NetCFSvcUtil, the most productive thing to do is create a command file that includes the instructions to generate a proxy for your mobile clients, and place it in the solution root so you can easily find it, generate the files, and copy them to the client app once you are sure they were correctly generated. The samples with this whitepaper include command files for this purpose.

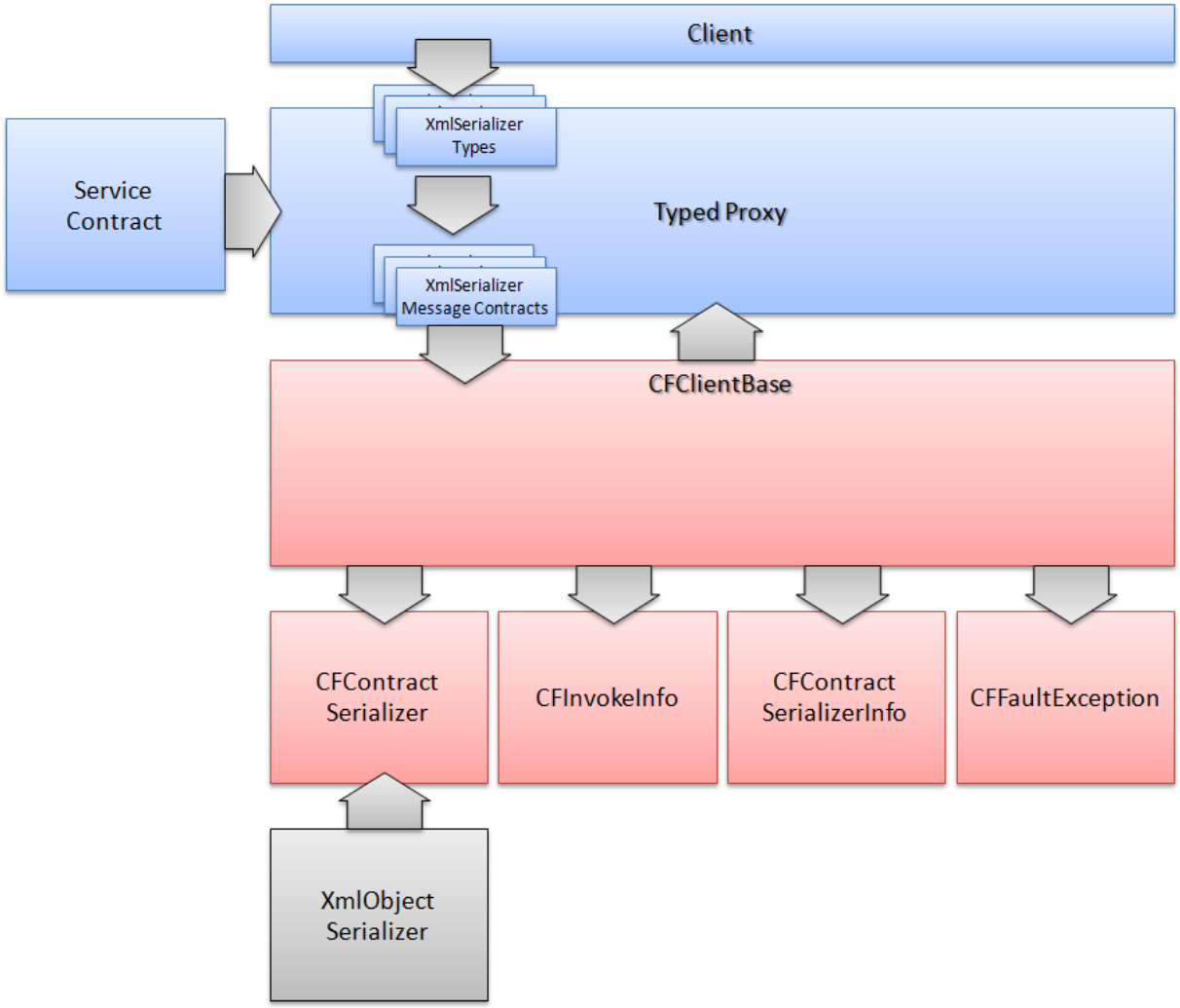
Proxy Architecture

NetCFSvcUtil generates core base functionality and a typed proxy with associated XmlSerializer types based on the service metadata. The result is that developers can work with a friendly object model to communicate with services – unless customizations are necessary. Unfortunately, there are times that you will need to modify the resulting files in order to support exception handling, custom headers, binding configurations not supported by NetCFSvcUtil and potentially other requirements specific to your implementation. In this section we'll discuss the architecture of the proxy so that later discussions that customize the proxy will be clear.

Figure 21 provides a high level view of the types generated by NetCFSvcUtil for the TodoListService described in earlier sections. The core, reusable functionality is provided by these types: CFClientBase,

CFContractSerializer, CFContractSerializerInfo, CFInvokeInfo and CFFaultException. Specific to the TodoListService are TodoListServiceClient (the typed proxy), ITodoListService (an interface that matches the service contract description), TodoItem (the only complex type used by service operations), and a number of message contracts, one for each operation (an example was described earlier).

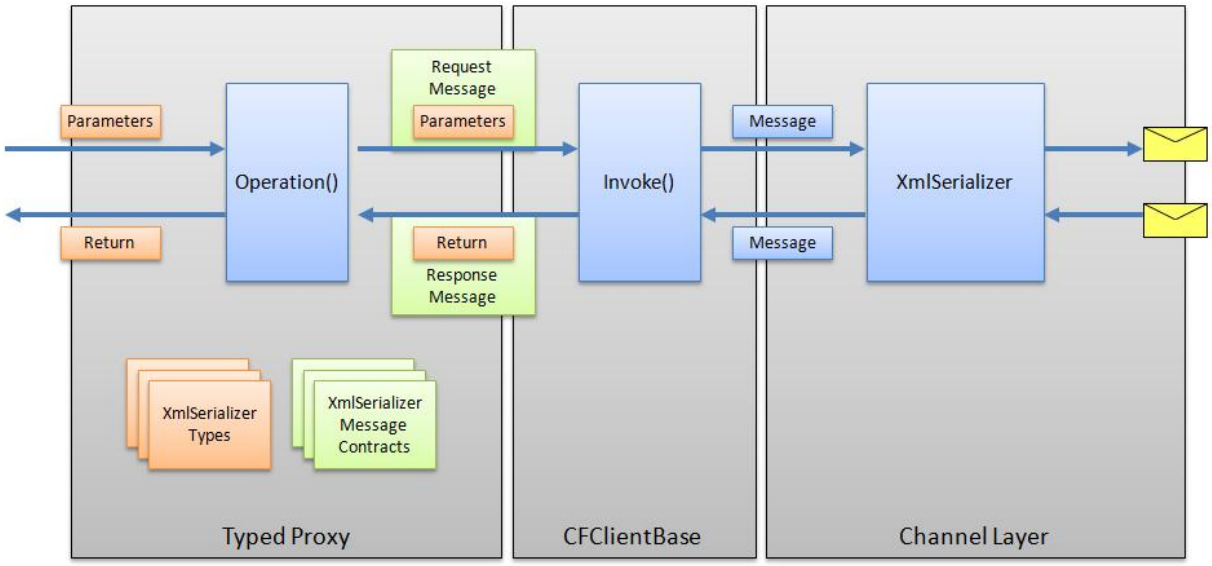
Figure 21: Relationship between files generated by NetCFSvcUtil



CFClientBase wraps the channel layer and the functionality necessary to serialize and deserialize messages. It exposes two Invoke() methods: one for calls that return a value, another for calls that do not return a value. Invoke() is the core function of the proxy base type which drives communications with the channel layer. CFContractSerializer is the XmlObjectSerializer implementation that wraps the XmlSerializer as discussed earlier. An instance of this type is provided to the channel layer to serialize and deserialize messages. CFContractSerializerInfo is a type that collects settings to initialize the CFContractSerializer, and CFInvokeInfo is a type that collects settings to build the message. CFFaultException is a wrapper class that references the XML returned when a fault is returned by an operation.

As for the actual proxy, the main type (TodoListServiceClient in this example) inherits CFClientBase and exposes operations for each method exposed by the service contract. It coordinates calls to the base type Invoke(). The client code is responsible for creating complex types appropriate for the service operation being called, and the proxy will wrap those parameters in a request message contract. The response is returned as a response message contract – carrying the return value. Figure 22 illustrates this flow.

Figure 22: Flow of communication through the proxy to the channel layer for a single operation



In later sections some scenarios will be discussed where proxy generation fails, generates a warning, or otherwise does not generate the code necessary to work with a particular service endpoint. These high level details will come in handy to understanding the required changes to satisfy these scenarios.

Bindings

When you configure endpoints for a service the binding controls which protocols are supported between clients and services. In order to support mobile clients you must expose at least one endpoint that employs protocols supported by the channel layer in the .NET Compact Framework. In this section we'll discuss which bindings and configurations are supported by the .NET Compact Framework, discuss how bindings are initialized for the generated proxy, and review some scenarios where you might modify the generated code.

Supported Binding Features

There are many standard bindings to choose from – each encapsulating specific settings to address common communications scenarios. The only standard binding supported by mobile clients is BasicHttpBinding. Other bindings each default to using features not supported by the .NET Compact Framework. You can also use a CustomBinding to enable features not supported by BasicHttpBInding that are supported by the .NET Compact Framework.

NOTE: There are in fact two other standard bindings, although they are only used by the .NET Compact Framework. They are WindowsMobileMailBinding and ExchangeWebServiceMailBinding. An overview of the scenarios where these bindings are employed can be found later in this paper.

Figure 23 provides a list of binding features supported by the .NET Compact Framework – with a description of how you would configure either BasicHttpBInding or a CustomBInding to enable the feature.

NOTE: This and other WCF support on the .NET Compact Framework has also been summarize on Andrew Arnott's blog (see resources).

Figure 23: Binding features available to mobile clients

Feature	BasicHttpBinding	CustomBinding
HTTP Transport	This is the default.	Use HttpTransportBindingElement.
HTTPS Transport	Configure Transport security mode.	Use HttpsTransportBindingElement
Buffered Messages	This is the default transfer mode. Cannot use any of the streamed transfer options.	This is the default transfer mode for HttpTransportBindingElement and HttpsTransportBindingElement. Cannot use any of the streamed transfer options.
Text Encoding	This is the default encoding format. Cannot use Mtom.	Use TextMessageEncodingBindingElement. Cannot use other encoding binding elements.
SOAP 1.1	This is the default message version setting.	This is the default message version setting for TextMessageEncodingBIndingElement.
SOAP 1.2 + WS-Addressing 1.0	NA	Set message version of the TextMessageEncodingBindingElement to Soap12WSAddressing10.
Basic Authentication	Set security mode to Transport and configure transport security client credential type as Basic.	Use HttpsTransportBindingElement and set authentication scheme to Basic.
Mutual Certificate Security	Set security mode to Message and configure message security client credential type as Certificate.	Use the AsymmetricSecurityBindingElement (or, MutualCertificate authentication mode).

There are some features that you may have expected to see in this list that are not supported by the .NET Compact Framework 3.5. Here is a summary of the most notable features that are missing:

- WS-Security UserName Token credentials are not supported. In theory you could roll your own implementation and pass it as a custom header over SSL. Custom headers cannot be encrypted using message security.

- Streaming is not supported. Contracts that expose Stream parameters are supported but the information will be buffered.

Client Binding Configurations

When you generate a proxy using NetCFSvcUtil it looks for endpoints that are compatible with the tool and uses the first compatible endpoint to generate the client-side binding configuration. If an endpoint is not found to be compatible with the tool, an exception is thrown. You should be aware of the following implications:

- If the service exposes multiple bindings for mobile clients, a binding configuration is only created for the first endpoint. You would have to manually create a binding for other endpoints if the client needs access to more than one (or, a preferred) option.
- It is possible that the tool cannot generate a proxy for an endpoint, and yet the .NET Compact Framework may support the features exposed by the endpoint.

Since the .NET Compact Framework does not provide support for declarative binding configuration, the NetCFSvcUtil generates a function called CreateDefaultBinding() for every proxy. This function returns an instance of a Binding type compatible with the mobile-compatible service endpoint selected during proxy generation. This code always creates a CustomBinding with the required features enabled. For example, if the service exposed an endpoint using the defaults for BasicHttpBInding:

```
<endpoint address="GreetingService" binding="basicHttpBinding"
contract="Greetings.Services.IGreetingService" />
```

CreateDefaultBInding() will include code to create a CustomBInding instance with HttpTransportBindingElement and TextMessageEncodingBIndingElement as follows:

```
public static Binding CreateDefaultBinding()
{
    CustomBinding binding = new CustomBinding();
    binding.Elements.Add(new
TextMessageEncodingBindingElement(MessageVersion.Soap11, Encoding.UTF8));
    binding.Elements.Add(new HttpTransportBindingElement());
    return binding;
}
```

To support SOAP 1.2 the .NET Compact Framework requires that you also use WS-Addressing. To configure a SOAP 1.2 endpoint at the service the endpoint and custom binding configuration look like this:

```
<endpoint address="GreetingService" binding="customBinding"
bindingConfiguration="Soap12WSAddressing10"
contract="Greetings.Services.IGreetingService" />
```

```
<customBinding>
  <binding name="Soap12WSAddressing10">
    <textMessageEncoding messageVersion="Soap11WSAddressing10" />
    <httpTransport />
  </binding>
</customBinding>
```

Unfortunately NetCFSvcUtil cannot generate a proxy for SOAP 1.2 endpoints, so an exception will be thrown by the tool. At the client you would have to pass MessageVersion.Soap11WSAddressing10 to the TextMessageEncodingBIndingElement initialization:

```
public static Binding CreateDefaultBinding()
{
    CustomBinding binding = new CustomBinding();
    binding.Elements.Add(new
TextMessageEncodingBindingElement(MessageVersion.Soap11WSAddressing10,
Encoding.UTF8));
    binding.Elements.Add(new HttpTransportBindingElement());
    return binding;
}
```

Security scenarios become a little more intricate but those will be discussed in a later section.

Debugging and Host Name Configuration

In the section “Your First Mobile + WCF Application” we walked you through generating a proxy for a simple WCF service, and modifying the endpoint address hard-coded in the proxy by hand – replacing “localhost” with the IP address of the machine where the service is running. This is a cumbersome step to repeat every time you create a new proxy and if you are sharing files with other developers each developer will have to update the value when they work on the application.

An easy solution to this problem is to provide a friendly host name for mobile clients and services to work with. This is an issue in particular for debugging scenarios. Since the mobile client runs as a separate machine in the emulator, “localhost” is not an acceptable host name to reach services hosted on the desktop.

For self-hosted services (services not hosted in IIS) you can provide a friendly host name with an entry in the hosts file. For IIS-hosted services you can edit the bindings of the web site. In either case, the end result is that you can expose services using a friendly host name instead of localhost, the service metadata (as in the WSDL document) will use that host name, and when a proxy is generated for the service it also uses the same host name. Most importantly, the mobile client will be able to use that host name to reach services on the desktop during debugging scenarios.

Modifying the Hosts File

The hosts file maps IP addresses to a host name (or, domain name). On Vista machines it is usually located at: C:\Windows\System32\drivers\etc\hosts. It doesn’t have a file extension but you can open it

with NotePad to make your changes. By default, the hosts file includes these IPv4 and IPv6 entries for localhost:

```
127.0.0.1       localhost

::1             localhost
```

Once you have decided on a host name to use for your development environment (since, this will inevitably change as you move source through staging, test and production) you can add an IPv4 and IPv6 entry for that host name. For example, in the samples for this whitepaper we use “mobilewcf.com” as the host name. For this to work on your machine, add the following to your hosts file:

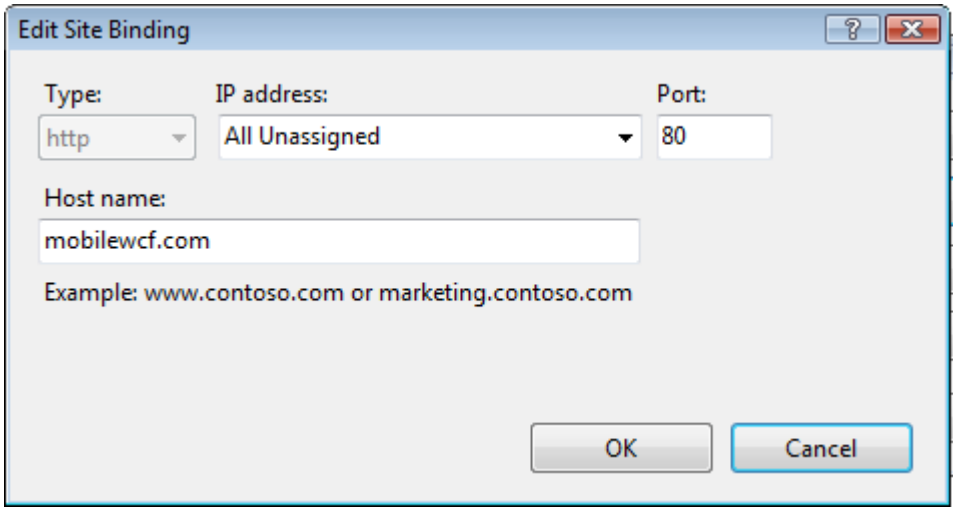
```
192.168.0.1       mobilewcf.com

::1               mobilewcf.com
```

You must replace 192.168.0.1 with your actual IPv4 address. This must be the desktop’s actual IP address on the network – don’t use 127.0.0.1. Also, for the device to communicate with your desktop you must have network connectivity.

Configuring IIS 7 Bindings

To modify the host name for an IIS 7 web site, select the web site node and from the Actions pane select Bindings. From the Site Bindings dialog edit the entry for HTTP binding and supply your custom host name. Figure 24 illustrates setting the name to mobilewcf.com as used in the code samples.

Figure 24: Modifying the site binding for HTTP to include use an alternate host name



You can also do this from the Visual Studio command line as follows:

```
cscript //nologo %systemdrive%\inetpub\adminscripts\adsutil.vbs set
W3SVC/1/ServerBindings ":80:mobilewcf.com"
```

To confirm that it worked, use this command line instruction:

```
cscript //nologo %systemdrive%\inetpub\adminscripts\adsutil.vbs get
W3SVC/1/ServerBindings
```

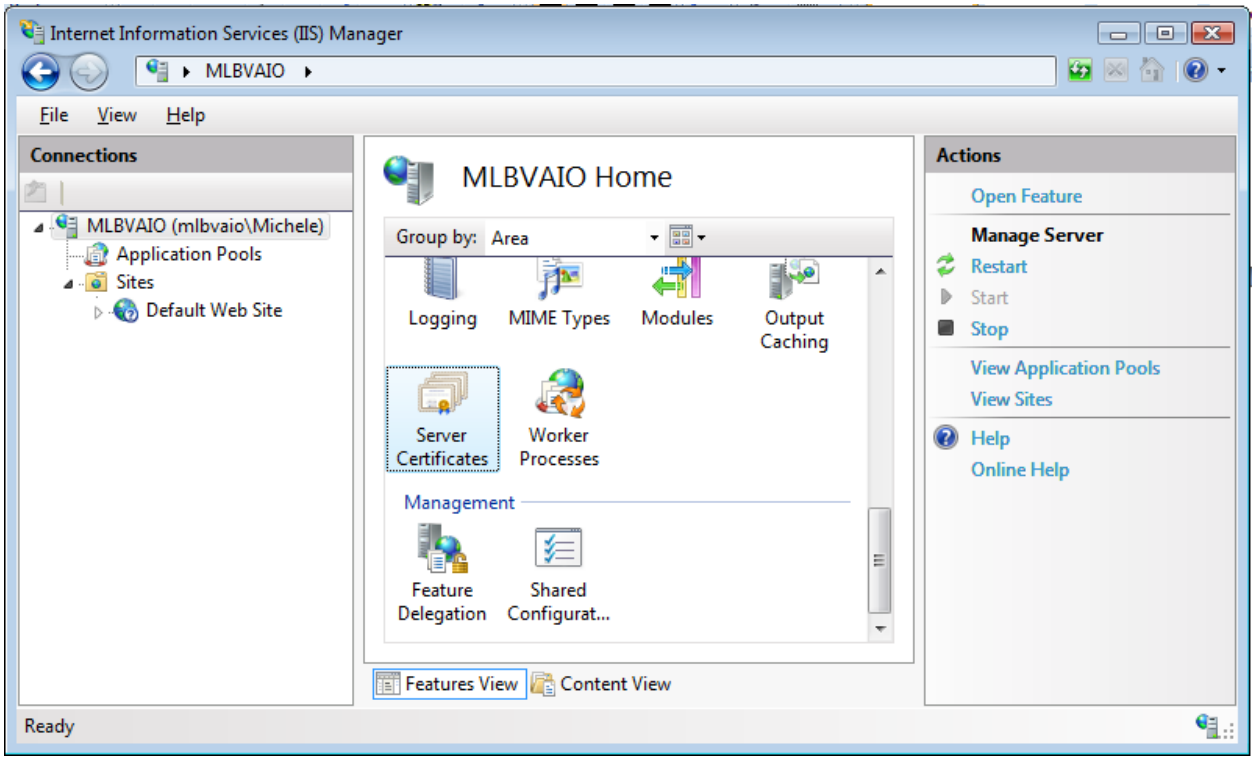
The output should be:

```
ServerBindings                    : (LIST) (1 Items)

":80:mobilewcf.com"
```

NOTE: This command line instruction modifies the host name for the Default Web Site in IIS as indicated by the number "1" in "W3SVC/1/ServerBindings". You should probably create a new web site for the samples so that your Default Web Site can remain as "localhost" for other development efforts. In that case, assuming that you only have two web sites, change the command line to: "W3SVC/2/ServerBindings".

To modify the host name for HTTPS requests you must use a command line instruction. First, create a certificate using makecert.exe that matches the host name you will be using and install this certificate to your local machine Personal certificate store (see code download instructions for details on how to do this). For convenience we have included a certificate for our samples named mobilewcf.com. Add this cert to the Server Certificates for the machine using IIS Manager. Select the machine node and the Server Certificates feature as shown in Figure 25. Import mobilewcf.com.pfx (the password is "mobilewcf.com").

Figure 25: Server Certificates feature in IIS Manager



From IIS Manager, select the web site and from the Actions pane select Bindings. Select the HTTPS binding and select the mobilewcf.com certificate to match the host name as shown in Figure 26. Notice that you cannot edit the host name (thus, we used the command line to set it). You can verify once again the correct host name is being used from the Site Bindings dialog as shown in Figure 27.

Figure 26: Modifying the SSL certificate for the HTTPS binding

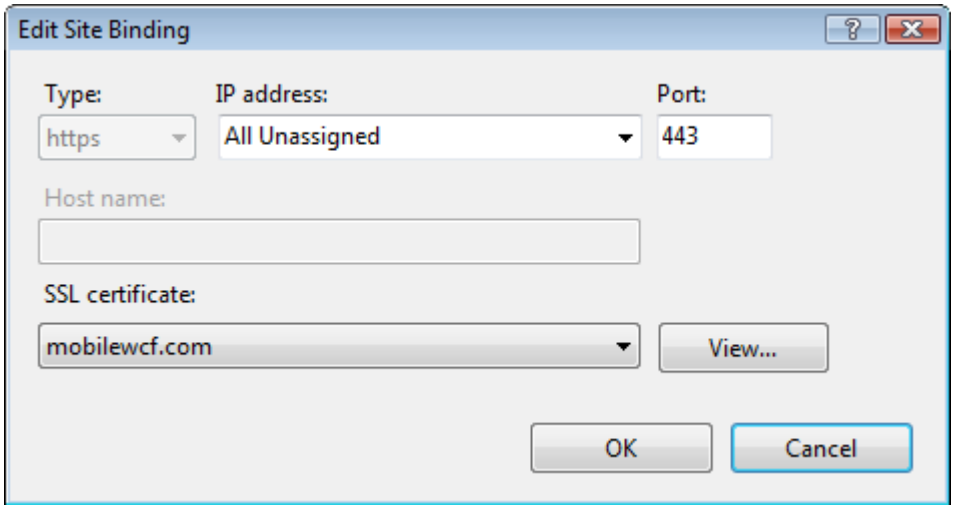
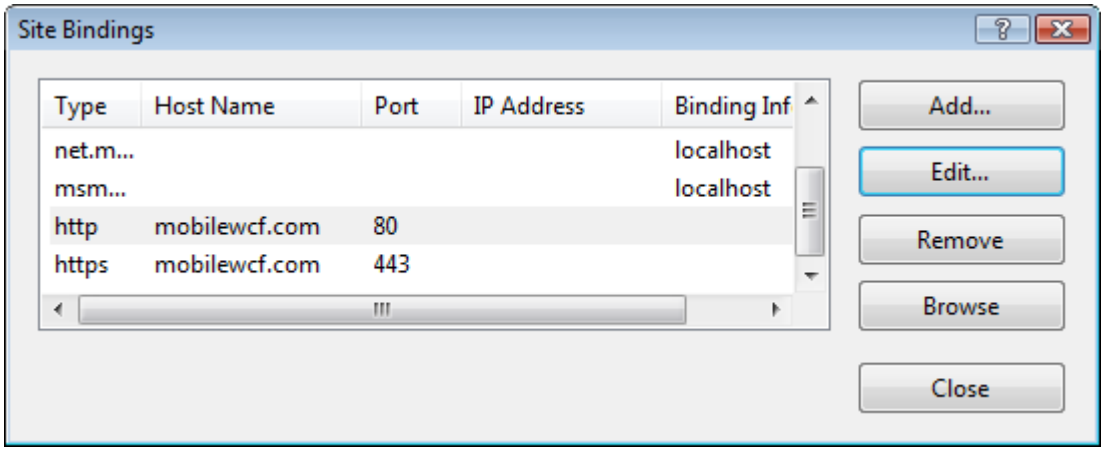


Figure 27: Site Bindings for HTTP and HTTPS using the designated certificate



Next, execute the following command line to modify the host name:

```
cscript //nologo %systemdrive%\inetpub\adminscripts\adsutil.vbs set
W3SVC/1/SecureBindings ":443:mobilewcf.com"
```

To verify the host name was changed, execute this command line:

```
cscript //nologo %systemdrive%\inetpub\adminscripts\adsutil.vbs get
W3SVC/1/SecureBindings
```

Your output should be similar to the following:

```
SecureBindings                    : (LIST) (1 Items)

":443:mobilewcf.com"
```

NOTE: As mentioned for the HTTP binding, if you create a new web site, modify the number in "W3SVC/1/SecureBIndings" to match the ordinal number of the web site.

Verify from the site bindings dialog once more that the HTTPS binding shows the correct host name, and that the certificate is indeed still set to mobilewcf.com. Reset IIS after making these changes. From the Visual Studio command line type "iisreset".

NOTE: The code download includes a detailed set of instructions for working with certificates including generating certificates with makecert.exe, installing them into the various certificate stores, and working with IIS Manager for web site certificates.

Exception Handling

In this section we'll explore how WCF services handle exceptions and faults, and more importantly, how to update the code generated by the NetCFSvcUtil so that mobile clients can work with faults and report useful information to the user.

Exceptions and Faults

Services can report exceptions in a number of ways. They can throw CLR exceptions, they can throw simple fault exceptions, or they can throw detailed faults. When a service throws a CLR exception the service model suppresses reporting that information to the client – to prevent from sharing exception details that could compromise the security of the service. Instead, a general fault is presented (some namespace details omitted for simplicity):

```
<s:Fault xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">

  <faultcode>a:InternalServiceFault</faultcode>

  <faultstring>

The server was unable to process the request due to an internal error.  For
more information about the error, either turn on
IncludeExceptionDetailInFaults (either from ServiceBehaviorAttribute or from
the &lt;serviceDebug&gt; configuration behavior) on the server in order to
send the exception information back to the client, or turn on tracing as per
the Microsoft .NET Framework 3.0 SDK documentation and inspect the server
trace logs.

  </faultstring>

</s:Fault>
```

You can enable the debugging behavior to include exception details in faults reported by the service as follows:

```
<behaviors>
  <serviceBehaviors>
    <behavior name="serviceBehavior">
      <serviceDebug includeExceptionDetailInFaults="true"/>
    </behavior>
  </serviceBehaviors>
</behaviors>
```

This is not a solution, however, to reporting useful information to the client since it includes the stack trace and other information that may not be appropriate for clients to view. Instead, services should throw faults. To throw a fault, the service can use the FaultException type. At a minimum an error message should be reported, but you can also include a fault code which provides additional details to the client. The following illustrates how to throw a fault without the fault code, and one that includes the fault code:

```
throw new FaultException("You must provide a valid name to receive a
greeting.");

throw new FaultException("You must provide a valid name to receive a
greeting.", FaultCode.CreateSenderFaultCode("SenderFault",
"urn:mobilewcf.com/samples/2009/04"));
```

A simple fault will result in a SOAP fault similar to the following (based on SOAP 1.1):

```
<s:Fault xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">

  <faultcode>s:Client</faultcode>

  <faultstring xml:lang="en-US">

You must provide a valid name to receive a greeting.

  </faultstring>

</s:Fault>
```

Typically, the service can get away with throwing a simple FaultException without a code – because most clients require only a string message to present to the user. At times, however, the service implements a rich error handling strategy that returns additional details to the client, beyond a simple error message and error code. In this case, the service can use declared faults.

Declared Faults

When the service wants to share additional details with the client in the event of an exception, they throw faults using FaultException<T>. The generic type parameter, T, can be any serializable type that includes details to share with the client. Typically, this is a custom type, not a CLR exception – since the former is interoperable and the latter is not. Figure 28 shows an example of a data contract named FaultDetail that exposes properties for information about the original CLR exception. This is just one example of the type of additional information the service might want to provide with the fault. The code

required to throw the fault uses FaultException<FaultDetail> and supplies an instance of the data contract along with a fault message and fault code.

Figure 28: Throwing a fault exception with a detail element described by a data contract

```
[DataContract(Namespace="urn:mobilewcf/samples/2009/04/schemas")]
public class FaultDetail
{
    public string OriginalExceptionType { get; set; }
    public string OriginalExceptionMessage { get; set; }
}


throw new FaultException<FaultDetail>(new FaultDetail { OriginalExceptionType
= "InvalidOperationException", OriginalExceptionMessage = "You must provide a
valid name to receive a greeting." }, "Name missing.",
FaultCode.CreateSenderFaultCode("SenderFault",
"urn:mobilewcf.com/samples/2009/04"));
```

If the service throws a fault with FaultException<FaultDetail>, a SOAP fault is returned including an additional detail element containing the serialized version of the FaultDetail. In order to produce metadata for clients so that they can process this detail – services usually declare faults in the service contract by applying the FaultContractAttribute to operations that can throw this type of fault:

```
[OperationContract]
[FaultContract(typeof(FaultDetail))]
string Greet(string firstname);
```

This is included in the metadata for the service.

Fault Exceptions and NetCFSvcUtil

Regardless of the technique used by the service – the client will be sent a SOAP fault formatted according to the protocol used for the request: SOAP 1.1 or SOAP 1.2. The client's job is to process this fault and present useful information to the user. The service can improve this experience by throwing faults instead of CLR exceptions – which means catching CLR exceptions thrown by the business and data layers and generating faults in their place instead of allowing the exception to propagate up to the channel layer at the service.

When NetCFSvcUtil generates a proxy from service metadata it produces a CFFaultException type as a wrapper for a fault. When the reply from a service call is processed, the generated proxy inspects the message and if it contains a fault throws a CFFaultException instance. Figure 29 shows definition of CFFaultException. The FaultMessage property is initialized with the raw fault XML as shown in Figure 30.

Figure 29: CFFaultException as generated by NetCFSvcUtil

```
public class CFFaultException : System.ServiceModel.CommunicationException
```

```
{

    private string _faultMessage;

    public CFFaultException(string faultMessage)
    {
        this._faultMessage = faultMessage;
    }

    public string FaultMessage
    {
        get
        {
            return this._faultMessage;
        }
    }
}
```

Figure 30: CFClientBase implementation of processReply()

```
private void processReply(System.ServiceModel.Channels.Message reply)
{
    if (reply.IsFault)
    {
        XmlDictionaryReader reader = reply.GetReaderAtBodyContents();
        try
        {
            throw new CFFaultException(reader.ReadOuterXml());
        }
        finally
        {
            reader.Close();
        }
    }
}
```

The problem with this is that the based type, CommunicationException, is not provided with a friendly value for its Message property. Thus, the user will always see "CFFaultException" as the error message with the proxy-generated implementation. In order to provide even the most basic error handling for mobile clients – changes must be made to CFClientBase and CFFaultException. In addition, a few new supporting types must be created to compliment the new version of CFFaultException.

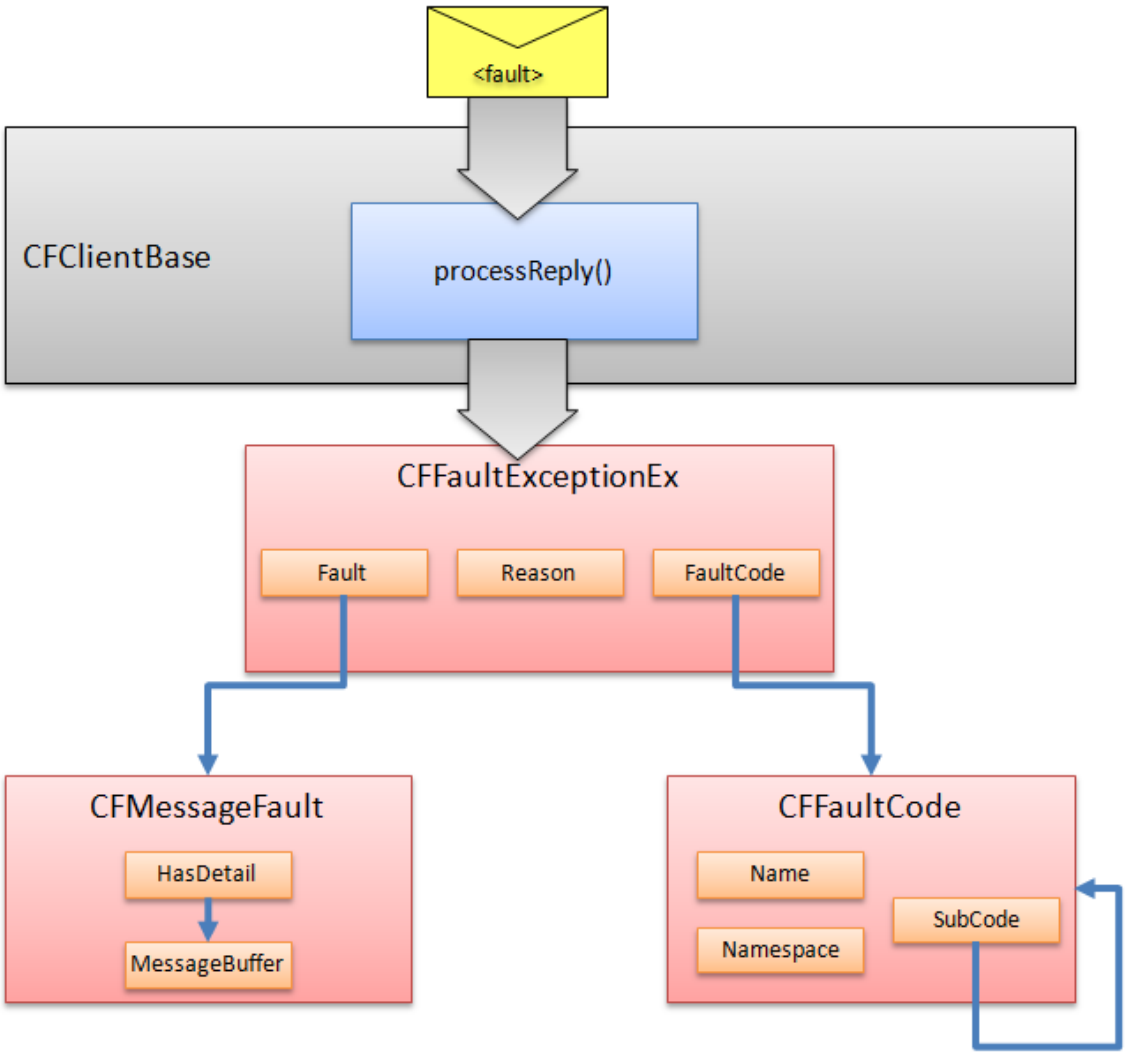
Fault Processing Improvements to CFClientBase

In this section we will summarize improvements made to CFClientBase and CFFaultException to provide a better exception handling experience for mobile clients. Figure 31 provides a high level view of the new components and key properties. Here are the notable changes:

- CFClientBaseEx is a new incarnation of CFClientBase that your proxy should inherit that throws CFFaultExceptionEx instead of CFFaultException.
- CFFaultExceptionEx is a new fault exception type that provides access to the fault reason, fault code and the original MessageFault.

- CFFaultCode is a lightweight implementation of a deserialized fault code.
- CFMessageFault is a lightweight implementation of a wrapper for the entire SOAP fault. It is responsible for deserializing the fault into individual properties, and providing access to the fault detail if any.

Figure 31: New components supporting an improved fault processing experience



Changes to CFClientBase

A message can only be read once, so in order to provide an unread copy of the message to CFMessageFault we must first buffer the message prior to attempting to read the reply. The MessaegBuffer instance is used to create a new copy of the message to process the reply, and again (if necessary) to create the CFMessageFault. The CFMessageFault instance is used to initialize an instance of CFFaultExceptionEx, instead of CFFaultException. Changes to CFClientBase are highlighted in Figure 32.

Figure 32: Changes to processReply() to create CFFaultExceptionEx

```
private void processReply(System.ServiceModel.Channels.Message reply)
{
    System.Diagnostics.Debug.Assert((reply != null));
    if (reply.IsFault)
    {
        MessageBuffer buffer = reply.CreateBufferedCopy(int.MaxValue);
        reply = buffer.CreateMessage();

        System.Xml.XmlDictionaryReader reader =
reply.GetReaderAtBodyContents();
        try
        {
            CFMessageFault faultMessage =
CFMessageFault.GetMessageFault(buffer.CreateMessage());
            throw new CFFaultExceptionEx(faultMessage);

        }
        finally
        {
            reader.Close();
        }
    }
}
```

CFMessageFault exposes a GetMessagFault() method which does the work of deserializing the SOAP fault. The code must handle the subtle differences between SOAP 1.1 and SOAP 1.2 faults and initialize a set of properties that represent the fault code, reason, and any detail element if present. Figure 33 shows the implementation details of CFMessageFault (omitting some details for brevity).

Figure 33: CFMessageFault implementation

```
public class CFMessageFault
{
    public string Reason { get; private set; }
    public CFFaultCode FaultCode { get; private set; }

    public bool HasDetail { get; private set; }
    protected MessageBuffer MessageBuffer { get; private set; }

    // other constructors

    public CFMessageFault(string reason, CFFaultCode faultCode, Message
faultMessage)
    {
        this.Reason = reason;
        this.FaultCode = faultCode;
        if (faultMessage != null)
        {
            this.MessageBuffer =
faultMessage.CreateBufferedCopy(int.MaxValue);
            this.HasDetail = true;
```

```
        }
    }

    public static CFMessageFault GetMessageFault(Message faultMessage)
    {
        CFMessageFault messageFault = null;

        string reason;
        CFFaultCode faultCode;
        bool hasDetail = false;
        MessageBuffer buffer;

        if (!faultMessage.IsFault)
            throw new InvalidOperationException("Invalid use of
CFFaultExceptionEx. Expecting Message.IsFault to return true.");

        buffer = faultMessage.CreateBufferedCopy(int.MaxValue);

        faultMessage = buffer.CreateMessage();
        XmlDictionaryReader reader = faultMessage.GetReaderAtBodyContents();

        if (faultMessage.Version == MessageVersion.Soap11)
        {
            reader.ReadStartElement("Fault", Constants.SoapVersion11);
            reader.ReadStartElement("faultcode");

            string localname;
            string ns;
            ParseCode(reader, reader.ReadContentAsString(), out localname,
out ns);
            faultCode = new CFFaultCode(localname, ns);
            reader.ReadEndElement();
            reader.MoveToContent();
            reason = reader.ReadElementContentAsString("faultstring", "");

            if (reader.IsStartElement("detail", ""))
                hasDetail = true;

            reader.Close();

            if (hasDetail)
                messageFault = new CFMessageFault(reason, faultCode,
buffer.CreateMessage());
            else
                messageFault = new CFMessageFault(reason, faultCode);

        }
        else if (faultMessage.Version == MessageVersion.Soap12WSAddressing10)
        {
            // SOAP 1.2 fault is slightly different
        }

        return messageFault;

    }
    private static void ParseCode(XmlReader reader, string codeValue, out
string localname, out string ns)
```

```
    {
        // extracts the namespace and name from the code
    }

}
```

The reason for creating the CFMessageFault type is so that the message can be processed and the fault reason (the error message) extracted before constructing an instance of CFFaultExceptionEx. This is the only way to provide a reasonable error message to the base type, CommunicationException. This value must be provided to the constructor as it is read-only, thus both CFFaultExceptionEx constructors call down to the base constructor. Figure 34 shows the listing of both CFFaultExceptionEx and CFFaultCode.

Figure 34: CFFaultExceptionEx and CFFaultCode implementation

```
public class CFFaultExceptionEx : CommunicationException
{
    public string Reason { get; private set; }
    public CFFaultCode FaultCode { get; private set; }
    public bool HasDetail { get; private set; }
    public CFMessageFault Fault { get; private set; }

    public CFFaultExceptionEx(string reason) : base(reason)
    {
        this.FaultCode = CFFaultCode.DefaultFaultCode;
        this.Reason = reason;
        this.HasDetail = false;
    }

    public CFFaultExceptionEx(CFMessageFault fault) : base(fault.Reason)
    {
        this.FaultCode = fault.FaultCode;
        this.Reason = fault.Reason;
        this.HasDetail = fault.HasDetail;
        this.Fault = fault;

    }
}

public class CFFaultCode
{

    public string Name { get; private set; }
    public string Namespace { get; private set; }
    public FaultCode SubCode { get; private set; }

    // other constructors

    public CFFaultCode(string name, string ns, FaultCode subCode)
    {
        if (String.IsNullOrEmpty(name))
            throw new InvalidOperationException("The parameter 'name' cannot
be null or empty");
        this.Name = name;
        this.Namespace = ns;
        this.SubCode = subCode;
```

```
    }

    public static CFFaultCode DefaultFaultCode
    {
        get {return new CFFaultCode("CFClientBaseEx"); }
    }

}
```

The end result of these modifications is that your client code can catch CFFaultExceptionEx and show the fault code and reason, or optionally catch Exception and show the error message as follows:

```
catch (CFFaultExceptionEx faultEx)
{
    MessageBox.Show(string.Format("FaultCode: {0}\r\nFaultReason: {1}",
faultEx.FaultCode, faultEx.Reason));
}
catch (Exception ex)
{
    MessageBox.Show(ex.Message);
}
```

To use these types you will add a reference to our custom library, Mobile.ServiceModelEx, which contains these and other enhanced files to support mobile clients. You will then modify your proxy to inherit CFClientBaseEx instead of CFClientBase. The rest is done for you!

NOTE: You can find Mobile.ServiceModelEx in the code download for this whitepaper.

Working with Message Headers

Custom message headers are a useful way to pass supporting information tangential to the functionality of a service operation. A perfect example of this is security headers required for authenticating callers to each operation. Another example could be an account identifier or license key that is useful for identifying the account holder separate from the user identified in security headers. Passing data in message headers prevents cluttering operations with unnecessary additional parameters, and can boost performance in that headers can be evaluated prior to reading the message body to determine the course of action for a message. WCF supplies built-in protocol channels to process security headers according to WS* specifications – but many of these protocols are not supported by the .NET Compact Framework. For this reason it is sometimes necessary to roll custom solutions to pass credentials to service operations – in the form of custom message headers. Likewise, any number of custom applications requirements could lead to a service design that relies on custom headers.

In this section we'll explore how to work with custom message headers in your mobile applications.

Michele Leroux Bustamante and Nickolas Landry, May 2009

Message Contracts

Before discussing the implications on your mobile applications, let's review how message contracts are used by a WCF service implementation to require custom headers. Message contracts are an easy way to implement operations that require custom headers at the service. A message contract is a type that represents the SOAP message – the message body and any custom message headers. Typically a message contract is used for both the request and reply in an operation definition as shown in the following service contract for the GetItems() operation:

```
 [ServiceContract(Namespace="urn:mobilewcf/samples/2009/04")]
public interface ITodoListService
{
    [OperationContract]
    GetItemsResponse GetItems(GetItemsRequest requestMessage);

    // other operations

}
```

GetItemsRequest and GetItemsResponse are each types decorated with the MessageContractAttribute. GetItemsRequest includes a single message header named LicenseKey, and no message body members since the GetItems() operation requires no parameters:

```
[MessageContract]
public class GetItemsRequest
{
    [MessageHeader]
    public string LicenseKey { get; set; }
}
```

GetItemsResponse includes a single message body member named header named Items, but no message headers:

```
[MessageContract]
public class GetItemsResponse
{
    [MessageBodyMember]
    public List<TodoItem> Items { get; set; }
}
```

In the service contract implementation the LicenseKey header is accessed as a property of the deserialized GetItemsRequest instance. If the LicenseKey property is valid, an instance of the GetItemsResponse type is constructed to return the item collection:

```
public GetItemsResponse GetItems(GetItemsRequest requestMessage)
{
    if (requestMessage.LicenseKey != "XXX")
    {
```

```
        throw new FaultException(string.Format("Invalid license key: {0}",
requestMessage.LicenseKey));
    }

    return new GetItemsResponse{ Items=m_globalTodoList };
}
```

The metadata for this service describes the requirement to pass the LicenseKey header to aide in proxy generation.

NetCFSvcUtil and Custom Headers

When you try to generate a proxy for a service that requires a custom message header be passed to one or more operation, NetCFSvcUtil throws the following exception:

```
Warning: .NET Compact Framework does not support 'MessageHeaderAttribute' found on
'GetItemsRequest.LicenseKey'.  Client proxy will support a reduced service contract.
```

Despite this message, at first blush it may appear that the proxy supports passing the LicenseKey header to the GetItems() operation. The proxy indeed exposes a GetItems() method that takes a single parameter, the LicenseKey string value:

```
TodoItem[] todoList = proxy.GetItems("XXX");
```

The implementation in the proxy also appears to pass this value to an internal GetItems() method that relies on similar message contracts: GetItemsRequest and GetItemsResponse:

```
public TodoItem[] GetItems(string LicenseKey)
{
  GetItemsRequest request = new GetItemsRequest(LicenseKey);
  GetItemsResponse response = this.GetItems(request);
  return response.Items;
}
```

This pattern is consistent with the service contract implementation using message contracts, except that the outgoing and incoming message types are defined as XML serializable types. As the NetCFSvcUtil warning stated, however, the proxy only supports a reduced set of functionality for calling the service. Specifically, the internals of the generated proxy does not include code to handle message headers.

Indeed the GetItemsRequest type is initialized with the LicenseKey value, however since message contracts are not supported, there is no mechanism for indicating that this value is to be serialized as a message header. To prevent serializing the value as part of the message body, the LicenseKey property is decorated with the XmlIgnoreAttribute:

```
[System.Xml.Serialization.XmlRootAttribute(ElementName="GetItemsRequest",
Namespace="urn:mobilewcf/samples/2009/04")]
public partial class GetItemsRequest
{
```

```
  // MessageHeaderAttribute and MessageHeaderArrayAttribute members not
supported by the .NET Compact Framework.
  [System.Xml.Serialization.XmlElementAttribute(IsNullable=true)]
  [System.Xml.Serialization.XmlIgnoreAttribute()]
  public string LicenseKey;

  public GetItemsRequest()
  {
  }

  public GetItemsRequest(string LicenseKey)
  {
    this.LicenseKey = LicenseKey;
  }
}
```

Removing this attribute would not achieve the goal of serializing the value as a message header. It would merely add another element to the SOAP body serialization which would be ignored by the service. Additional code is required to include this property as a message header for the outgoing message.

Ultimately, one of the two Invoke() methods exposed by CFClientBase (one is for two-way operations, the other for one-way operations) handles building the message to be sent to the service operation. Invoke() is a generic method that receives two parameters:

- A CFInvokeInfo type which provides information to the serializer such as the type to serialize
- A request parameter typed for the message contract type for this particular operation – which describes the message body

The two-way implementation of Invoke() is shown here:

```
protected TRESPONSE Invoke<TREQUEST, TRESPONSE>(CFInvokeInfo info, TREQUEST
request)
{
    CFContractSerializerInfo serializerInfo = new CFContractSerializerInfo();
    serializerInfo.MessageContractType = typeof(TREQUEST);
    serializerInfo.IsWrapped = info.RequestIsWrapped;
    serializerInfo.ExtraTypes = info.ExtraTypes;
    serializerInfo.UseEncoded = info.UseEncoded;
    Message msg = Message.CreateMessage(this.binding.MessageVersion,
info.Action, request, GetContractSerializer(serializerInfo));

    return this.getResult<TRESPONSE>(this.getReply(msg), info);
}
```

What you ultimately want is to add message headers to the Message type that is created during Invoke(). If the LicenseKey header were to be the same for all calls, you could add code to write the

header directly inside the Invoke() method. This would require creating a serializer instance for the header and then adding the header to the Headers collection of the Message type:

```
CFContractSerializerInfo headerSerializationInfo = new
CFContractSerializerInfo();
headerSerializationInfo.MessageContractType = typeof(string);
headerSerializationInfo.IsWrapped = info.RequestIsWrapped;
headerSerializationInfo.ExtraTypes = info.ExtraTypes;
headerSerializationInfo.UseEncoded = info.UseEncoded;

CFContractSerializer headerSerializer = new
CFContractSerializer(headerSerializationInfo);

msg.Headers.Add(MessageHeader.CreateHeader("LicenseKey",
"urn:mobilewcf/samples/2009/04", "XXX", headerSerializer));
```

Note that this assumes that the LicenseKey header name, namespace and value are all known to the proxy base type, CFClientBase. To achieve reuse there must be a way to pass the header to the Invoke() method in a way that it can generalize creating the header.

Enhancing CFClientBase to Support Headers

A number of changes must be made to the generated proxy in order to support a reusable mechanism for serializing outgoing headers in a message. Here is a summary of the required changes to CFClientBase and related classes:

- CFClientBase includes three child types: CFInvokeInfo, CFContractSerializerInfo, and CFContractSerializer. These types are be extracted from CFClientBase and made public types so that the proxy implementation can access them freely.
- CFInvokeInfo exposes a MessageHeaders collection type so that headers can be passed to the Invoke() method explicitly.
- CFContractSerializer implements the WriteObjectContent() method since for header serialization this operation is called.
- CFClientBase exposes a MessageVersion property so that the proxy implementation can access this information to create message headers for methods that require it.
- Both Invoke() methods exposed by CFClientBase use the updated CFInvokeInfo type to add headers to the Message type before calling the service operation.

In the Mobile.ServiceModelEx implementation (introduced earlier) these changes have been applied to the new type CFClientBaseEx, and enhancements described to the other types are included in this library as well. Figure 35 highlights these changes.

Figure 35: Changes to the generated CFClientBase proxy and related types to accommodate outgoing header serialization

```
public partial class CFClientBaseEx<TChannel> where TChannel : class
{
```

```
        public MessageVersion MessageVersion
        {
            get
            {
                return this.binding.MessageVersion;
            }
        }

        protected TRESPONSE Invoke<TREQUEST, TRESPONSE>(CFInvokeInfo info,
TREQUEST request)
        {
            CFContractSerializerInfo serializerInfo = new
CFContractSerializerInfo();
            serializerInfo.MessageContractType = typeof(TREQUEST);
            serializerInfo.IsWrapped = info.RequestIsWrapped;
            serializerInfo.ExtraTypes = info.ExtraTypes;
            serializerInfo.UseEncoded = info.UseEncoded;
            System.ServiceModel.Channels.Message msg =
System.ServiceModel.Channels.Message.CreateMessage(this.binding.MessageVersio
n, info.Action, request, GetContractSerializer(serializerInfo));

            if (info.Headers != null)
                msg.Headers.CopyHeadersFrom(info.Headers);

            return this.getResult<TRESPONSE>(this.getReply(msg), info);
        }

        protected void Invoke<TREQUEST>(CFInvokeInfo info, TREQUEST request)

        {
            CFContractSerializerInfo serializerInfo = new
CFContractSerializerInfo();
            serializerInfo.MessageContractType = typeof(TREQUEST);
            serializerInfo.IsWrapped = info.RequestIsWrapped;
            serializerInfo.ExtraTypes = info.ExtraTypes;
            serializerInfo.UseEncoded = info.UseEncoded;
            System.ServiceModel.Channels.Message msg =
System.ServiceModel.Channels.Message.CreateMessage(this.binding.MessageVersio
n, info.Action, request, GetContractSerializer(serializerInfo));

            if (info.Headers != null)
                msg.Headers.CopyHeadersFrom(info.Headers);


            if (info.IsOneWay)
            {
                if ((this._outputChannelFactory != null))
                {
                    this.postOneWayMessage(msg);
                }
                else
                {
                    this.getReply(msg);
                }
            }
            else
```

```
            {
                this.processReply(this.getReply(msg));
            }
        }

        // other members


    }


    public class CFContractSerializer : XmlObjectSerializer
    {
        public override void WriteObject(System.Xml.XmlDictionaryWriter
writer, object graph)
        {
            if (this.info.IsWrapped)
            {
                this.serializer.Serialize(writer, graph);
            }
            else
            {
                WriteObjectContent(writer, graph);

            }
        }

        public override void
WriteObjectContent(System.Xml.XmlDictionaryWriter writer, object graph)
        {
            System.IO.MemoryStream ms = new System.IO.MemoryStream();
            System.Xml.XmlWriterSettings settings = new
System.Xml.XmlWriterSettings();
            settings.OmitXmlDeclaration = true;
            System.Xml.XmlWriter innerWriter =
System.Xml.XmlDictionaryWriter.Create(ms, settings);
            this.serializer.Serialize(innerWriter, graph);
            innerWriter.Close();
            ms.Position = 0;
            System.Xml.XmlReader innerReader =
System.Xml.XmlDictionaryReader.Create(ms);
            innerReader.Read();
            writer.WriteAttributes(innerReader, false);
            if ((innerReader.IsEmptyElement == false))
            {
                innerReader.Read();
                for (
                ; ((innerReader.NodeType ==
System.Xml.XmlNodeType.EndElement)
                            == false);
                )
                {
                    writer.WriteNode(innerReader, false);
                }
            }
            innerReader.Close();
```

```
        }

        // other members

    }

    public struct CFContractSerializerInfo
    {
      // members
    }

    public class CFInvokeInfo
    {
        public MessageHeaders Headers { get; set; }

        // other members

    }
}
```

The result is that the proxy implementation can, for each operation that requires a header, initialize the CFInvokeInfo type with that information. Figure 36 illustrates the changes made to the customized version of the GetItems() method to pass the LicenseKey header.

Figure 36: Adding headers to the CFInvokeInfo type for serialization

```
private MobileClient.GetItemsResponse GetItems(MobileClient.GetItemsRequest
request)
{

    CFInvokeInfo info = new CFInvokeInfo();
    info.Action = "urn:mobilewcf/samples/2009/04/ITodoListService/GetItems";
    info.RequestIsWrapped = true;
    info.ReplyAction =
"urn:mobilewcf/samples/2009/04/ITodoListService/GetItemsResponse";
    info.ResponseIsWrapped = true;


    CFContractSerializerInfo headerSerializationInfo = new
CFContractSerializerInfo();
    headerSerializationInfo.MessageContractType = typeof(string);
    headerSerializationInfo.IsWrapped = info.RequestIsWrapped;
    headerSerializationInfo.ExtraTypes = info.ExtraTypes;
    headerSerializationInfo.UseEncoded = info.UseEncoded;

    CFContractSerializer headerSerializer = new
CFContractSerializer(headerSerializationInfo);

    info.Headers = new MessageHeaders(base.MessageVersion);
    info.Headers.Add(MessageHeader.CreateHeader("LicenseKey",
"urn:mobilewcf/samples/2009/04", request.LicenseKey, headerSerializer));
```

```
    MobileClient.GetItemsResponse retVal =
base.Invoke<MobileClient.GetItemsRequest,
MobileClient.GetItemsResponse>(info, request);
    return retVal;
}
```

Unfortunately there is no way to avoid modifying the generated proxy to support custom headers but you can minimize impact by using the CFClientBaseEx.

NOTE: You can find Mobile.ServiceModelEx in the code download for this whitepaper.

Communicating with REST-Based Services

The majority of this whitepaper focuses on mobile clients communicating with SOAP-based WCF services. In this section we will discuss how mobile clients communicate with REST-based WCF services. First, we'll review how to implement a REST-based service, followed by a discussion of the code required to call that service.

Implementing a REST-Based Service

SOAP-based services are implemented by designing a service contract, implementing it on a service type, configuring endpoints for each contract, and initializing a ServiceHost type either in a self-hosted environment or by associating the service type with a .svc endpoint in IIS. The same steps are followed for REST-based services, but there are a few key differences:

- Additional attributes are applied to each service operation in the service contract to define the HTTP verb used to invoke the operation, and to describe the Uri that will be mapped to the operation at runtime.
- The only binding supported for REST-based services is the WebHttpBinding. Much like BasicHttpBinding it is a simple binding with limited features.
- Instead of using the ServiceHost and ServiceHostFactory, the WebServiceHost and WebServiceHostFactory are employed. That means that for self-hosted environments (not IIS) the WebServiceHost is initialized in code, and for IIS hosting the WebServiceHostFactory is associated with the .svc endpoint.

Figure 36 show the listing for the ITodoListService contract using WebGetAttribute and WebInvokeAttribute to describe REST-based requirements. Operations that return a value use the WebGetAttribute which means an HTTP GET will be used to call the operation. Other operations use the WebInvokeAttribute specifying the HTTP verb POST (to create items), PUT (to update items), or DELETE (to delete items). In this case both attributes are instructed to use XML format for communications instead of JSON – since support for the latter is not build-in to the .NET Compact Framework.

Figure 36: REST-based service contract definition for ITodoListService

```
 [ServiceContract(Namespace="urn:mobilewcf/samples/2009/04")]
public interface ITodoListService
```

```
{
    [OperationContract]
    [WebGet(RequestFormat=WebMessageFormat.Xml,
ResponseFormat=WebMessageFormat.Xml, UriTemplate="Items")]
    List<TodoItem> GetItems();

    [OperationContract]
    [WebInvoke(RequestFormat = WebMessageFormat.Xml, ResponseFormat =
WebMessageFormat.Xml, UriTemplate = "Items", Method="POST")]
    string CreateItem(TodoItem item);
    [OperationContract]
    [WebInvoke(RequestFormat = WebMessageFormat.Xml, ResponseFormat =
WebMessageFormat.Xml, UriTemplate = "Items", Method = "PUT")]
    void UpdateItem(TodoItem item);
    [OperationContract]
    [WebInvoke(RequestFormat = WebMessageFormat.Xml, ResponseFormat =
WebMessageFormat.Xml, UriTemplate = "Items/{id}", Method = "DELETE")]
    void DeleteItem(string id);
}
```

The .svc endpoint uses the WebServiceHostFactory to activate the service using the WebServiceHost which installs behaviors that bypass traditional SOAP filtering.

```
<%@ ServiceHost Service="TodoList.TodoListService"
Factory="System.ServiceModel.Activation.WebServiceHostFactory" %>
```

As for the binding configuration for the REST-based endpoint, the defaults for WebHttpBinding are used in this simple example.

```
<system.serviceModel>
  <services>
    <service name="TodoList.TodoListService">
      <endpoint address="" binding="webHttpBinding"
contract="Contracts.ITodoListService" />
    </service>
  </services>
</system.serviceModel>
```

In a later section we'll explore securing a REST-based endpoint.

Using HttpWebRequest

There is no concept of proxy generation for REST-based services and so the HttpWebRequest object is used. That means you need to know the Url for the service endpoint, the Uri to invoke each operation, the required XML to pass to operations, and the expected XML to receive back from operations.

Figure 37 shows how to use HttpWebRequest to call the GetItems() operation exposed by the TodoListService. Here are some points worth noting:

- You provide the Url to the service when you create the HttpWebRequest instance. The suffix /Items matches the template Uri indicated in the WebGetAttribute from Figure x.
- Since this is an HTTP GET request, the Content-Length header is set to 0. For a POST or PUT request this header should be set to the size of the data being passed.

- The Accept header indicates that you are expecting XML content. It the client also supported JSON, for example, you can add additional types and let the service decide what to return that is compatible.
- The response string is the resulting XML and can be processed using an XmlReader or the subset if LINQ supported by the .NET Compact Framework.

Figure 37: Using HttpWebRequest to call a REST-based service

```
string url =
"http://mobilewcf.com/TodoListServiceRestWebHost/TodoListService.svc/Items";

HttpWebRequest webRequest = HttpWebRequest.Create(url) as HttpWebRequest;

webRequest.ContentLength = 0;
webRequest.Accept = "application/xml";
webRequest.Method = "GET";

string response = String.Empty;
using (WebResponse webResponse = webRequest.GetResponse())
{
  using (Stream responseStream = webResponse.GetResponseStream())
  {
    using (StreamReader reader = new StreamReader(responseStream))
    {
      response = reader.ReadToEnd();

      // parse response
    }
  }
}
```

Using HttpWebRequest is very simple, but it lacks the convenience of automatic serialization and deserialization that would be provided by a WCF proxy.

Securing Mobile + WCF Communications

Now that we have talked about how to design WCF services for mobile clients, and how to consume those services be they SOAP-based or REST-based, it's time to discuss the choices available for securing communications between those clients and services. WCF in the .NET Framework includes a very rich set of security options from basic authentication to federated security – but the subset in the .NET Compact Framework includes only a very limited set of security options. In this section we will describe available features by scenario, and describe how you should set up your environment to implement each scenario if applicable.

SSL Transfer Security

The most common way to secure the transfer of information between mobile clients and WCF services is using SSL. This not only secure the body of the message, but also any headers passed along with it. This provides a way to securely pass a custom username and password header with each message, for example.

First and foremost is it a more realistic test to host your services in IIS when testing SSL transfer security. In addition, to ensure a successful SSL handshake between the mobile client and the WCF service you must use a friendly host name. Thus, you should follow the instructions described earlier to configure IIS for a friendly host name, and use a certificate matching that host name. In addition, the mobile client must use the same host name to connect.

To require an SSL connection with the service you can configure BasicHttpBInding to use transport security. This requires the service to be hosted over HTTPS. Figure 38 shows this configuration.

Figure 38: BasicHttpBInding using SSL encryption

```
<system.serviceModel>
  <services>
    <service name="TodoList.TodoListService">
      <endpoint address="" binding="basicHttpBinding"
bindingConfiguration="basicSSL" contract="Contracts.ITodoListService" />
    </service>
  </services>
  <bindings>
    <basicHttpBinding>
      <binding name="basicSSL">
        <security mode="Transport" >
          <transport clientCredentialType="None"/>
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
</system.serviceModel>
```

At the client, the proxy will generate a CustomBinding equivalent as follows:

```
CustomBinding binding = new CustomBinding();
binding.Elements.Add(new
TextMessageEncodingBindingElement(MessageVersion.Soap11,
System.Text.Encoding.UTF8));
HttpsTransportBindingElement https = new HttpsTransportBindingElement();
https.RequireClientCertificate = false;
binding.Elements.Add(https);
```

Since the client isn't passing credentials in this case, no additional initialization is required for the proxy to call the service endpoint.

Basic Authentication

With basic authentication a username and password are passed in HTTP headers, in the clear. For this reason you will usually rely on an SSL connection to protect the credentials. To support basic authentication without SSL encryption at the WCF service, you can configure BasicHttpBinding as follows:

```
<basicHttpBinding>
  <binding name ="basicAuth">
```

```
    <security mode="TransportCredentialOnly">
      <transport clientCredentialType="Basic" />
    </security>
  </binding>
</basicHttpBinding>
```

To support basic authentication the preferred way, with SSL encryption, use the following configuration:

```
<basicHttpBinding>
  <binding name ="basicAuth">
    <security mode="Transport">
      <transport clientCredentialType="Basic" />
    </security>
  </binding>
</basicHttpBinding>
```

In addition, your IIS application must support Basic Authentication.

Unfortunately the .NET Compact Framework does not support basic authentication scenarios using the channel layer. You can work around this by using HttpWebRequest to communicate with the SOAP-based service, with some modifications to how we discussed using this for REST-based services earlier including:

- All requests use the HTTP POST verb
- The Content-Type header is set to “text/xml” instead of “application/xml”
- For SOAP 1.1 requests you must add a SOAPAction HTTP header indicating the operation to invoke
- You must manually build the XML to POST for each request including the SOAP envelope and namespaces according to the SOAP version to be used

Figure 39 illustrates using the HttpWebRequest to call the GetItems() operation exposed by the TodoListService over the binding illustrated previously. In order to pass credentials using basic authentication a new NetworkCredentials instance is created and assigned to the Credentials property of the HttpWebRequest instance. To pass credentials you will also set PreAuthenticate to true to prepopulate authentication headers on the request and avoid redirects, and set AllowWriteStreamBuffering to true as this is a requirement after setting the Credentials property.

A SOAP message is generated according to the requirements of the operation. In this case, an empty element <GetItems> is passed within the SOAP body. The message is written to the request stream first, and then GetResponse() leads you to the response stream that contains the result of the call. This response will contain a SOAP fault if the call failed, otherwise it will contain XML according to the operation result.

Figure 39: HttpWebRequest invoking a SOAP-based service using basic authentication

```
HttpWebRequest webRequest = HttpWebRequest.Create(
```

```
"http://mobilewcf.com/TodoListServiceWebHostBasicAuth/TodoListService.svc")
as HttpWebRequest;

webRequest = HttpWebRequest.Create(url) as HttpWebRequest;
webRequest.Credentials = new NetworkCredential("username", "password");
webRequest.PreAuthenticate = true;
webRequest.AllowWriteStreamBuffering = true;

webRequest.ContentType = "text/xml; charset=utf-8";
webRequest.Headers.Add("SOAPAction",
"urn:mobilewcf/samples/2009/04/ITodoListService/GetItems");

webRequest.Method = "POST";

string soapBody = "<s:Envelope
xmlns:s=\"http://schemas.xmlsoap.org/soap/envelope/\"><s:Body><GetItems
xmlns=\"urn:mobilewcf/samples/2009/04\"/></s:Body></s:Envelope>";
webRequest.ContentLength = soapBody.Length;

string response = String.Empty;
using (Stream requestStream = webRequest.GetRequestStream())
{
  using (StreamWriter writer = new StreamWriter(requestStream))
  {
    writer.Write(soapBody);
  }
}

using (WebResponse webResponse = webRequest.GetResponse())
{
  using (Stream responseStream = webResponse.GetResponseStream())
  {
    using (StreamReader reader = new StreamReader(responseStream))
    {
      response = reader.ReadToEnd();
      // process XML response
    }
  }
}
```

Digest Authentication

With digest authentication is implemented with a challenge/response protocol whereby the service sends material to the client to be used to hash the password before sending a username and password to the server. The server then compares the hashed password with the hashed password at the server to authenticate the caller. To support digest authentication without SSL encryption at the WCF service, you can configure BasicHttpBinding as follows:

```
<basicHttpBinding>
  <binding name ="basicAuth">
    <security mode="TransportCredentialOnly">
      <transport clientCredentialType="Digest" />
    </security>
  </binding>
```

```
</basicHttpBinding>
```

To support digest authentication with SSL encryption, use the following configuration:

```
<basicHttpBinding>
  <binding name ="basicAuth">
    <security mode="Transport">
      <transport clientCredentialType="Digest" />
    </security>
  </binding>
</basicHttpBinding>
```

In addition, your IIS application must support Digest Authentication.

You should follow the instructions discussed in the previous section for basic authentication, to use the HttpWebRequest for digest authentication.

Securing REST-Based Services

Any of the three methods just discussed can be used to secure REST-based services:

- SSL encryption without credentials
- Basic authentication with or without SSL
- Digest authentication with or without SSL

You will initialize the security features of HttpWebRequest as you did to secure calls to SOAP-based services, and handle messaging payload for GET, POST, PUT and DELETE calls to service operations according to the earlier discussion about REST-based services and HttpWebRequest.

Mutual Certificate Authentication

The .NET Compact Framework supports a subset of WS-Security 1.0. Specifically, it supports SOAP Message Security which means that mutual certificate authentication is possible between mobile clients and WCF services. With mutual certificate authentication, the client uses a private key to authenticate to the service, and the service identifies itself with a private key. Messages to the service are encrypted with the service public key – which means that it must be provided to the client a priori. The result is that messages between the client and service are encrypted.

To configure the service you can use BasicHttpBinding with message security mode. The client credential type should be set to Certificate, and the algorithm suite must be set to Basic256Rsa15 – the only algorithm supported by the channel layer in the .NET Compact Framework.

Figure 40 shows the service model configuration for a service exposing a mutual certificate security endpoint. The service behavior includes a <serviceCredentials> section that indicates the certificate used to identify the service- mobilewcf.com and describes how client certificates will be authenticated. Typically, client certificates are authenticated with PeerTrust, which means that the certificate is explicitly installed in the local machine's TrustedPeople store. This step also acts as authorization for the

request however if there are a large number of certificates, the service can configure a more dynamic mode of authentication and authorization.

Figure 40: Service configuration for mutual certificate security

```
<system.serviceModel>
  <services>
    <service name="Greetings.Services.GreetingService"
behaviorConfiguration="serviceBehavior">
      <endpoint binding="basicHttpBinding" bindingConfiguration="MutualCert"
contract="Greetings.Services.IGreetingService" />
    </service>
  </services>
  <bindings>
    <basicHttpBinding>
      <binding name="MutualCert">
        <security mode="Message">
          <message algorithmSuite="Basic256Rsa15"
clientCredentialType="Certificate"/>
        </security>
      </binding>
    </basicHttpBinding>
  </bindings>
  <behaviors>
    <serviceBehaviors>
      <behavior name="serviceBehavior">
        <serviceCredentials>
          <clientCertificate>
            <authentication certificateValidationMode="PeerTrust"
revocationMode="Online" trustedStoreLocation="LocalMachine"/>
          </clientCertificate>
          <serviceCertificate findValue="mobilewcf.com"
storeLocation="LocalMachine" storeName="My"
x509FindType="FindBySubjectName"/>
        </serviceCredentials>
        <serviceAuthorization principalPermissionMode="None"/>
      </behavior>
    </serviceBehaviors>
  </behaviors>
</system.serviceModel>
```

The service can also use a CustomBInding to describe the same configuration. This is sometimes necessary when features not exposed by BasicHttpBinding must also be configured. For example, to expose an endpoint that supports SOAP 1.2 with WS-Addressing 1.0 the configuration in Figure 41 can be employed. The <security> element illustrates the only allowed settings for defaultAlgorithmSuite, authenticationMode and messageSecurityVersion for mobile clients.

Figure 41: Custom binding equivalent for mutual certificate security

```
<customBinding>
  <binding name="MutualCertCustom">
    <security defaultAlgorithmSuite="Basic256Rsa15"
authenticationMode="MutualCertificate" messageSecurityVersion=
```

```
"WSSecurity10WSTrustFebruary2005WSSecureConversationFebruary2005WSSecurityPol
icy11BasicSecurityProfile10" requireDerivedKeys="false" />
    <textMessageEncoding messageVersion="Soap12WSAddressing10" />
    <httpTransport />
  </binding>
</customBinding>
```

Through proxy generation, the binding equivalent of the configuration in Figure 41 is shown in Figure 42. To initialize the proxy you must provide a client certificate to authenticate to the service, and a service certificate to use for encrypting messages. That means installing the certificates to the device ahead of time (to be discussed) and initializing the proxy's ClientCredentials – the ClientCertificate and ServiceCertificate properties. This is illustrated in Figure 43. The client certificate is initialized from the My store, the service certificate from the Root store.

Figure 42: Programmatic configuration for mutual certificate security at the mobile client

```
CustomBinding binding = new CustomBinding();
binding.Elements.Add(new
TextMessageEncodingBindingElement(MessageVersion.Soap11, Encoding.UTF8));
AsymmetricSecurityBindingElement asbe = new
AsymmetricSecurityBindingElement(new
X509SecurityTokenParameters(X509KeyIdentifierClauseType.Any,
SecurityTokenInclusionMode.Never), new
X509SecurityTokenParameters(X509KeyIdentifierClauseType.Any,
SecurityTokenInclusionMode.AlwaysToRecipient));
    asbe.MessageSecurityVersion =
MessageSecurityVersion.WSSecurity10WSTrustFebruary2005WSSecureConversationFeb
ruary2005WSSecurityPolicy11BasicSecurityProfile10;
    asbe.LocalClientSettings.DetectReplays = false;
    asbe.LocalServiceSettings.DetectReplays = false;
    asbe.DefaultAlgorithmSuite = SecurityAlgorithmSuite.Basic256Rsa15;
    asbe.SetKeyDerivation(false);
    binding.Elements.Add(asbe);
    binding.Elements.Add(new HttpTransportBindingElement());
    return binding;
```

Figure 43: Initializing CFClientBase with certificates for mutual certificate authentication

```
Binding binding = GreetingServiceClient.CreateDefaultBinding();

EndpointAddress ep = new EndpointAddress(
GreetingServiceClient.EndpointAddress.Uri);

GreetingServiceClient m_proxy = new GreetingServiceClient(binding, ep);

m_proxy.ClientCredentials.ClientCertificate.SetCertificate(
StoreLocation.CurrentUser, StoreName.My, X509FindType.FindBySubjectName,
"mobileclient");

m_proxy.ClientCredentials.ServiceCertificate.SetDefaultCertificate(
StoreLocation.CurrentUser, StoreName.Root, X509FindType.FindBySubjectName,
"mobilewcf.com");
```

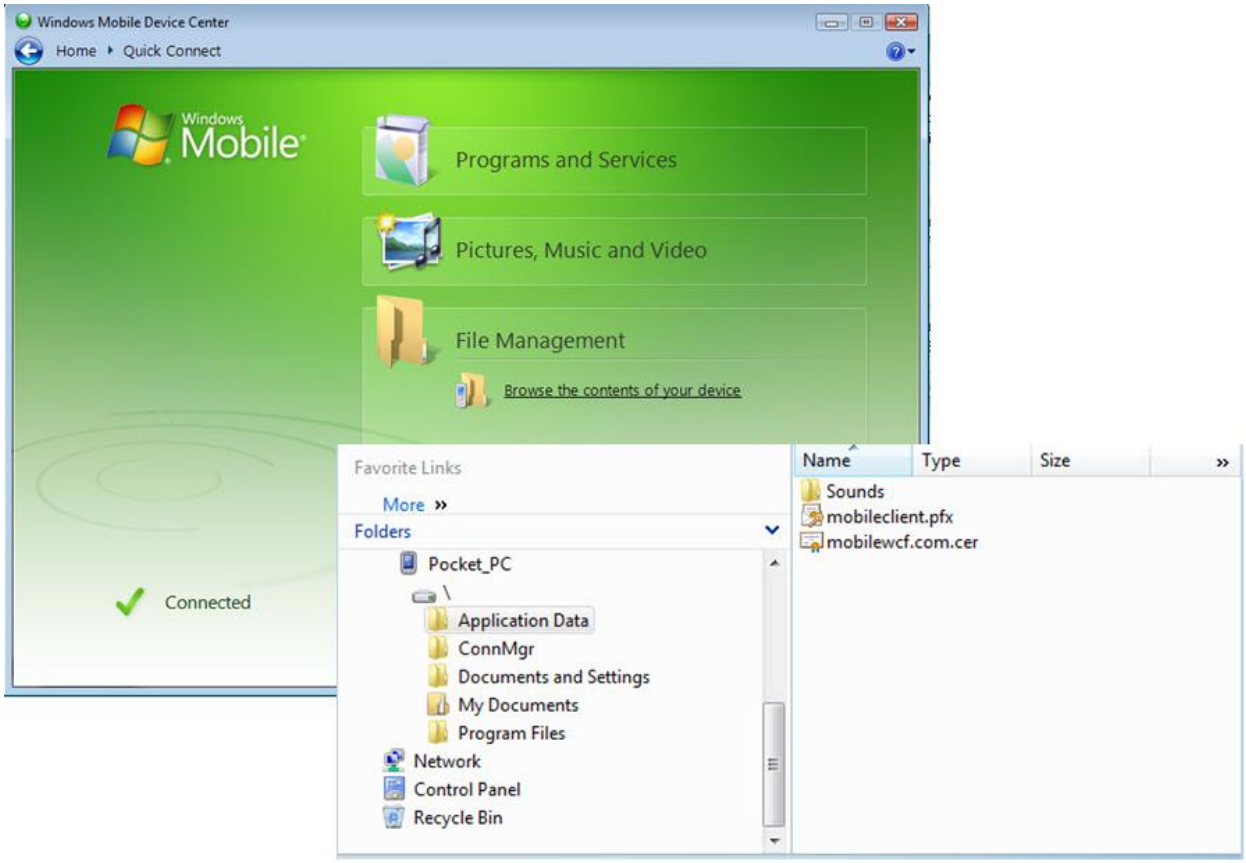
Michele Leroux Bustamante and Nickolas Landry, May 2009

Installing Certificates to a Mobile Device

Although it is relatively easy to install certificates to a mobile device, there is little documentation on how to achieve this. This section will quickly summarize the steps necessary to install certificates for the mutual certificate authentication scenario.

From the WMDC you can browse the file system of the device (see Figure 44) and simply drag and drop files from the desktop. For the sample code with this paper you would copy the mobileclient.pfx and mobilewcf.com.cer files to the device. After the files have been copied you can install the certificates to their respective certificate stores through the device emulator.

Figure 44: Viewing the device file system from the WMDC



From the device emulator Start menu, select the File Explorer to browse for the certificates. Figure 45 shows the files copied to the Application Data directory. From here, simply click on each file to install them into the certificate store. In the case of mobileclient.pfx, a password will be requested (in this case the password is “mobileclient”). You will be presented with a message indicating that the certificate has been successfully installed. Private keys are installed to the My certificate store, and certificates are installed to the Root certificate store.

Figure 45: Installing certificates copied to the device

Deployment Considerations

Pushing new applications and updates to rich clients on the desktop has always been a challenge in itself, but what do you do when the physical deployment device is not even connected to the network and out of the office most of the time? Web-based applications gained a lot of popularity thanks to centralized management and deployment, but when connectivity cannot be taken for granted in the field the majority of mobile scenarios have to adopt a smart client deployment model.

One such “smart” feature is the ability of a smart client to automatically deploy itself without requiring setup kits and manual or scripted deployments. Web clients have always been popular thanks to the centralized nature of the application on the Web server where no client deployment is needed. Smart clients reuse this model by living inside a Web server as well, ready to be initiated. When first accessed by a new user/client, the smart client downloads itself to the client machine or device along with all its dependencies, therefore eliminating the need for a local installation before the application can be used. Technologies like ClickOnce in .NET Framework 2.0 and higher can provide such functionality for desktop scenarios, but unfortunately not for mobile device applications. Mobile smart clients typically need to be deployed to the device first, and then a bootstrapper can manage updates as they become available on the deployment server.

By the same token automated deployment features can also make sure the latest version of all assemblies are always available on the client device. Whereas the application downloads the required dependencies on first use, future updates posted on the central Web server are also detected the next time the application is used by comparing assembly dates, times and version numbers. Once it is identified that a newer version exists, a local bootstrapper downloads the new files, performs the proper configuration operations and then executes them. The end-user might notice a slower initialization routine whenever such updates are being processed, but the whole process is transparent and does not require any human interaction.

Visual Studio 2008 provides you with a project template to easily create a deployment package for smart device applications. Refer to the MSDN documentation at http://msdn.microsoft.com/en-us/library/zcebx8f8.aspx for a step-by-step walkthrough on packaging a smart device solution for deployment.

As you prepare your WCF-enabled mobile application for deployment, some of the considerations to keep in mind are:

- Make sure that .NET Compact Framework 3.5 is deployed and installed since Windows Mobile 6.1 & 6.5 devices only come with .NET Compact Framework 2.0 pre-installed in device ROM.
- When using message-based security, make sure your certificate files are properly deployed and installed in the appropriate certificate store on the device.
- Remember to deploy and install the Daylight Savings Time Update for Windows Mobile to avoid exceptions tied to the clock skew between the mobile client and the server in WS-Security scenarios.

Acknowledgements

Very few resources exist for mobile applications communicating with WCF services. We would like to recognize the few people who have online resources on the subject of mobile development with WCF including Andrew Arnott and Chris Tacke. In addition, there are several people at Microsoft who were a great help to us as we pulled together this information including Mahathi Mahabhashyam, Amit Chopra, and Kirill Gavrylyuk.