7 Networking with Windows Phone

A networked device is much more useful than one that is just free standing. Once a system is connected to the internet all kinds of things become possible. Mobile phones are around the most connected devices there are, in that they have a whole variety of connection possibilities. However, they also bring challenges to the application developer in that any, or perhaps all, of these network connections could be removed at any instant, and the user of our application will expect it to handle this automatically.

In this section we will look at the network connection options available to programs that we write. We will also give some consideration to how we can make sure that our applications can respond gracefully to network problems.

All Windows Phones support network connectivity via the mobile telephone network and also by means of their built in WIFI. As far as our programs are concerned the two network technologies are interchangeable. The programs we are going to write will initiate connections to a data service and the underlying phone systems will make the connection the best way that they can.

Of course, if there is no mobile signal or WIFI available these connections will fail. Applications that we write must be able to deal with this situation gracefully. At the very least this means displaying a “Sorry we can’t connect you just now, please try later” message. A more advanced program may store data locally and then send it later. It is up to you how your program will work in this respect.

The Windows phone emulator has the same networking abilities as the real device. It uses the network connectivity of the host PC it is running on. This means that we can use a desktop PC to test any Windows Phone applications we are writing.

However it is also important to test programs on a real device. On a real Windows Phone we can disable network services and test our applications to ensure that they perform properly over lower speed connections and when the network connection is suddenly removed.

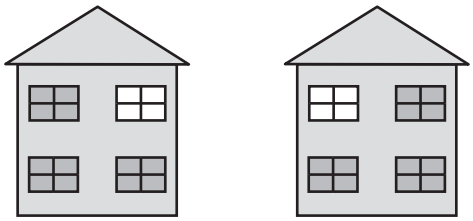
7.1 Networking Overview

Before we look at how Windows Phone uses network connections, we need to learn a little bit about networks. This is not a detailed examination of the field, but it should give you enough background to understand how our programs are going to work.

Starting with the Signal

The first computer networks used wires to send their data signals, although more modern networks can use radio or fibre optic cables. Whatever the medium is, the fundamental principle is that you have some hardware that can put data onto the medium in the form of bits and get it off again. A bit is either 0 or 1 (or you can think of a bit as either true or false) and can be signalled by the presence or absence of a voltage, a light from a light-emitting diode (LED), or a radio signal.

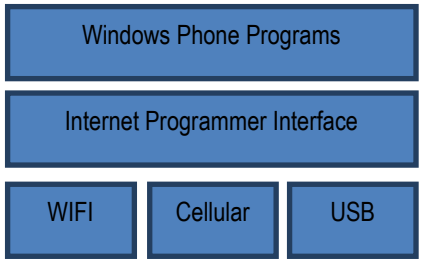
If you imagine signalling your friend in the house across the road by flashing your bedroom light on and off, you have an idea of the starting point of network communications.



Once we have this raw ability to send a signal from one place to another, we can start to transfer useful data.

Signals in Windows Phone

In the case of Windows Phone there are a whole range of signal types that it can use to connect with a network. There is the WIFI connection, the mobile phone cellular connection and even the USB connection to the Zune software on the PC. All of these technologies can be used to pass data between programs on the phone and the Internet.

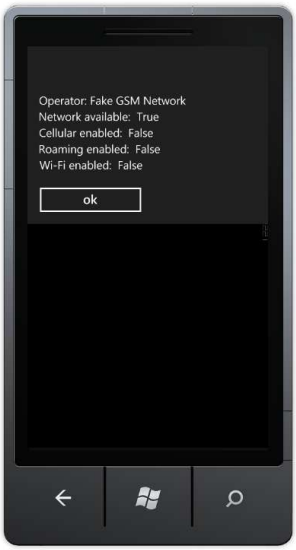


This is actually made to work as you see above. The Windows Phone programs talk to the Internet Programmer Interface (the thing we are going to learn how to use) and the Internet Programmer Interface talks to the three connection technologies. The good news is that the programs we write don't need to be changed to work over the different types of connection. However, there are things that we need to remember, in that if our phone user goes abroad they may not allow data roaming, so the cellular data connection will be disabled. Even worse for our programs, a user may put their phone into "aircraft" mode and disable all network connections for a while.

A Windows Phone program can determine which connections are available by using methods in the `DeviceNetworkInformation` class.

```
using Microsoft.Phone.Net.NetworkInformation;

...

System.Text.StringBuilder sb = new System.Text.StringBuilder();

sb.Append("Operator: " );
sb.AppendLine(DeviceNetworkInformation.CellularMobileOperator);

sb.Append("Network available:  ");
sb.AppendLine(DeviceNetworkInformation.IsNetworkAvailable.ToString());

sb.Append("Cellular enabled:  ");
sb.AppendLine(DeviceNetworkInformation.IsCellularDataEnabled.ToString());

sb.Append("Roaming enabled:  ");
sb.AppendLine(DeviceNetworkInformation.IsCellularDataRoamingEnabled.ToString())
;

sb.Append("Wi-Fi enabled:  ");
sb.AppendLine(DeviceNetworkInformation.IsWiFiEnabled.ToString());
```

The above code assembles a message that describes the status of the phone which can then be displayed. We can also use these values to control the way that our application behaves.



The solution in *Demo 01 NetworkDiagnostics* contains a program that displays the network status for a phone or emulator.

Building Up to Packets

Just flashing your light to your friend willy-nilly does not allow you to send much information. To communicate useful signals you have to agree on a message format. You could say, “If my light is off and I flash it twice, it means it is safe to come round because my sister is out. If I flash it once, it means don’t come, and if I flash it three times, it means come and bring pizza with you.” This is the basis of a thing called a protocol, which is an arrangement by parties on the construction and meaning of messages.

A modern network sends data in 8 bit chunks that are called *octets* by the people who design them. In C# we can hold the value of a single octet in a variable of type `byte`. If we have 8 data bits we can arrange them in 256 different patterns, from 00000000 (all the bits clear) to 11111111 (all the bits set). An octet is the smallest unit of data that is transferred by a network. A give octet might be a number (in the range 0 to 255) a single character (a letter digit or punctuation) or part of a larger item, for example a floating point number or a picture file. On the internet a data packet can be several thousand octets in size.

Addressing Packets

Your bedroom light communication system would be more complicated if you had two friends on your street who want to use your bedroom light network. You would have to agree with them that you would send two sets of flashes for each message. The first one would indicate who the message was for, and the second would be the message itself. Computer networks function in exactly the same way. Every station on a physical network must have a unique address. Packets sent to that address are picked up by the network hardware in that station.

Networks also have what is called a broadcast address. This allows a system to send a message which will be picked up by every system. This is the network equivalent of “Calling all cars . . .” In our communication network, this could be used to warn everyone that your sister has come home and brought her boyfriend, so your house is to be avoided at all costs. In computer networks a broadcast is how a new computer can find out the addresses of important systems on the network. It can send out a “Hi. I’m

new around here!" message and get a response from a system that can then send it configuration information.

Everyone on a particular network can receive and act on a broadcast sent around it. In fact, if it wanted to, a station could listen to all the messages traveling down its part of the wire or WiFi channel. This illustrates a problem with networks. Just as both of your friends can see all the messages from your bedroom light, including ones not meant for them, there is nothing to stop someone from eavesdropping on your network traffic. When you connect to a secure Web site, your computer is encoding all the messages that it sends out so that someone listening other than the intended recipient would not be able to learn anything.

Internet Addressing

We have already seen the networks we are using can send 8 bit sized items. This would mean that if we used a single octet to address stations on the network we could not connect more than 256 systems. To get around this the present version of the Internet (IP V4) uses four 8 bit values which are strung together to allow 32 bits to be used as an address. In theory this allows for over four thousand million different systems on the internet. However, this large number is hard to achieve in practice and so a new version of the Internet (IP V6) is being rolled out which has 128 bit addresses (16 octets in each address). This would allow for a huge increase in the number of possible stations, but since it is a fundamental change to way the network functions it is taking a while to put into practice. At the moment the networking software in the Windows Phone does not support IPV6 addressing.

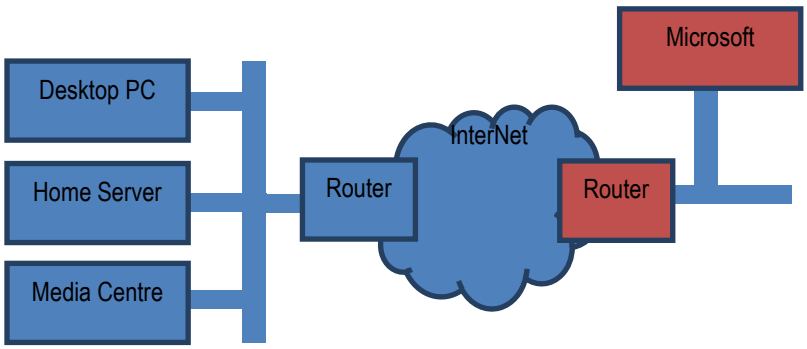
Routing

Not everything on the internet is connected to the same physical network. Signals sent around the wires in my house to not make as far as the network operated by my next door neighbour. Instead we have to view the Internet as a large number of separate networks which are connected together. This means that to get messages from one physical network to another we have to introduce the idea of routing.

If you had a friend on the next block, she might not be able to see your bedroom light. But she might be able to see the light of your friend across the road. This means that you could ask your friend across the road to receive messages and then send them on for you. Your friend across the road would read the address of the message coming in, and if it was for your friend on the next block, she would transmit it again. Figure 16-2 shows how this works. Your friend uses the window on the left to talk to you and the window on the right to relay messages to your more distant friend.

You can think of your friend in the middle as performing a routing role. She has a connection to both "networks", the people you can see and also the people that your distant friend can see. She is therefore in a position to take messages from one network and resend them on the other one. Note that this is not like passing a letter from one person to another. Instead your friend is receiving your message and then re-transmitting it to your distant friend.

An address of a given system on the Internet is made up of two elements, the address of the local network containing the system and the address of that system on the network. The important thing to remember about a local network is that the machines in it are actually connected to the same physical media.



The diagram shows how this works. The machines on your home network are physically connected and can talk directly to each other. The Desktop PC can fetch files directly from the Home Server. However, if the Desktop PC needs to contact Microsoft the messages must leave the local network and travel via the Internet. Just before the Desktop PC sends any message out it looks at the network address of the system it is talking to. If the remote system is on the same network it will connect directly the system. However if, like Microsoft, the system is on a different network the Desktop PC will send the message to the router which sends such messages into the internet, which delivers them to the router at the Microsoft.

Packets that you send from your home PC to distant machines are sent to the network at your Internet service provider (ISP) which then re-transmits them to the next system in the path to the destination. Packets might have to be sent over several machines to reach their destination. The Internet constantly changes the routes that packets take. This makes it very reliable and able to manage sudden surges in traffic and failures of parts of the network but it can lead to situations where a packet arrives before another which was sent first. Sometimes packets can get lost (although this is fairly rare), so you can't be sure that one has arrived until you receive an acknowledgement. One thing you should remember is that you do not really "connect" your system to the Internet. Whenever your system is connected, it actually becomes part of the Internet.

You can regard a local-level network as the same as the internal mail that is used in many organizations, including my university at Hull. If I want to send a message to our chemistry department, I just put the address "Chemistry Department" on the envelope and drop it in the internal mail. There is a local protocol (called the internal mail system) that makes sure that the message gets there. However, if I want to send a message to the chemistry department at York University, I must put a longer address on the envelope. When the letter gets to the mailroom the staff notice that the destination is not local, and they route it out to the postal system, which sends it to York. This is the "Internet Protocol" for letters.

The Internet is powered by a local protocol (Transport Control Protocol, or TCP) and an internetwork protocol (Internet Protocol, or IP). Put these together, and you have the familiar TCP/IP name that refers to the combination.

You can also use the TCP/IP protocol to connect machines without linking them to the Internet. If you plug your Windows Phone into your PC it actually creates a "tiny internet" between the two machines so that the Zune software can transfer media to and from your phone.

Anything that you send via the Internet will be transferred using one or more individual packets. Each packet contains the address of the destination and each packet is numbered, so that missing packets can be detected and packets can be put into the right order when they are received (if you want that). If you need to transfer a large file, this will be broken down into a number of packets.

Networks and Media

Note that links between different parts of a network can use different media. A laptop could connect via WIFI to a home network which would be connected via the service

provider to the Internet backbone. The internet backbone is connected to your mobile phone provider which then sets up a network connection from a cell phone tower to your Windows Phone. This is how you can use your Windows Phone to message your friends. Furthermore many devices have multiple network connections. Your phone can talk to the Internet via a cell connection to your mobile phone provider or it can use your home WIFI. The good news as far as our programs are concerned is that the way we make connections is independent of the precise connection.

Networks and Protocols

A protocol is a set of rules that tells you how to behave in a certain situation. There is a protocol that tells you which knife and fork to use in a posh banquet and another that tells you to kiss a maiden on her hand having just rescued her from a dragon. You already have one with your friend, where you have agreed on the meaning of the various messages that you send with your bedroom lights.

In networking terms, a protocol sets out the design of all the messages and how stations in a network should cooperate to move data around. There are essentially two levels at which this must take place. There must be a "local-level" protocol that lets local stations (ones on the same piece of physical media) exchange data, and there must be an "internetwork" protocol that allows messages to be sent from one local network to another. The protocol should also extend to cover how systems on the network are addressed and discovered by other systems.

The Internet is based on a large number of different protocols which all work together to describe how systems can work together to exchanged data.

Finding Addresses on the Internet using DNS

The Internet addresses everything by their 32 bit network/system values, which are usually expressed as four "octet" values. For example the address on the internet of the University of Hull web server is `150.237.176.24`. This 32 bit value is what Internet messages use to find their way to the web server system at our university. An address is split into `150.237` which is the address of the university network in the world and `176.24` which is the address on the Hull network of our web server. This form of a network address is called the Internet Protocol or *IP* address of a system.

However, nobody wants to have to remember an IP address like this just because they want to see our university web pages – beautiful though they are. People would much rather use a name like `www.hull.ac.uk` to find the site. To solve this problem a computer on the Internet will connect to a *name server* which will convert hostnames into IP addresses. The system behind this is called the *Domain Name System*. It is a collection of servers who pass naming requests around amongst themselves until they find a system with authority for a particular set of addresses who can match the name with the required address. We can think of a name server as a kind of "directory enquiries" for computers. In the old days if I wanted to know the phone number of the local movie house I would ring up directory enquiries who would tell me this. When a computer wants to know the IP address of a web site a user wants to visit it will ask its Domain Name service to provide the result.

Addresses and Subnets

You might think that if every device on the internet has an IP address it should be possible to uniquely address that specific device. In other words, if you get hold of the IP address of my phone you should be able to send network messages directly to it. After all, every phone in the world has a unique number, doesn't it?

This is not actually the case though. Because of the worldwide shortage of IP addresses many networks are arranged into subnets. In this arrangement systems on the subnet have IP addresses that work on that subnet but are not visible to machines outside the subnet. The subnet router, which has an IP address on the "proper" internet, sends messages on behalf of all the machines on the subnet. When messages come from the

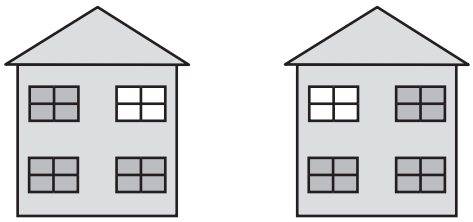
internet the router performs some address translation to ensure the packet is sent to the correct local system. This arrangement works well, and is used by companies and home networks to reduce the number of “worldwide” IP addresses that they need. It is also used by mobile phone carriers. This rather like a company having an internal telephone system with many staff telephones and only a few external phone numbers. Calls to the company phone number are switched to particular phones inside the company.

Unfortunately this internet addressing limitation can make it difficult for two mobile devices to directly exchange data, as their addresses are not meaningful to each other. The only way to solve this problem is to use a third party machine which is on the wired network and has a “proper” IP address. Both phones can originate a connection to this third machine and use it to pass data backwards and forwards.

Of course, if the phones are both connected to the same WIFI network when you are at home it should be possible for them to connect directly as they will both be on the same physical network.

Networks and Ports

We now know how the Internet sends messages from one computer to another. Each computer has an address that identifies the internet the network the computer is connected to and the address of that computer on the network. Now we have to add another layer to the protocol, and consider ports.



If we go back to our original network, where we are communicating with our friend by flashing our bedroom light, we have a problem if we want to talk about different topics at the same time. If we wanted to send messages about what is coming up on TV, and also order food, we would have to improve our protocol so that some messages could be flagged as being about TV shows, and others identify the type of pizza we want.

In the case of a computer system we have exactly the same problem. A given computer server can provide a huge variety of different services to the clients that connect to it. The server might be sending web pages to one user, sharing video with another and hosting a multiplayer game all at the same time. The different clients need a way of locating the service that they want on from the server.

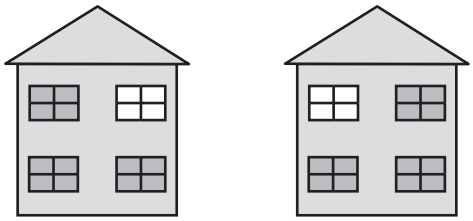
The Internet achieves this by using “ports”. A port is a number which identifies a particular service provided by a computer. Some of the ports are “well known”, for example port number 80 is traditionally the one used for web pages. In other words, when your browser connects to a web page it is using the internet address of the server to find the actual computer and then connecting to port 80 to get the web page from that server.

When a program starts a service it tells the Internet software which port that service is sitting behind. When messages arrive for that port the program is passed these messages. If you think about it, the Internet is really just a way that we can make one program talk to another on a distant computer. The program (perhaps a web server) you want to talk to sits behind a port on a computer connected to the Internet. You run another program (perhaps a web browser) that creates a connection to that port which lets you request web pages and display them on your computer.

Programmers can write programs that use any port number, but many networks use *firewalls* that only allow particular packets addressed to particular well know ports to be passed through. This reduces the chance of systems on the network coming under attack from malicious network programs.

Connections and Datagrams

The Internet provides two ways systems can exchange information: connections and datagrams. A datagram is a single message that is sent from one system to another, in the same way that we could just flash the lights in our inter-house network to deliver a message to someone who may or may not be watching. This is like flashing your bedroom light three times to ask for pizza and then just waiting for someone to turn up with your favourite pepperoni thin and crispy.



If you want to be sure that a message has got through you could agree with your friend that she would flash her light once to indicate that she has received it. Then perhaps you could send another message asking for drinks. When she was leaving to fetch the pizza, she could flash her light twice to indicate that she was "going off the air." This would be the basis of a *connection* between the two of you.

When two systems are in a connection, they have to perform extra work to manage the connection itself. When one system sends a message that is part of a connection the network either confirms the message was successfully transferred (once the network has received an acknowledgement) or gives an error saying that it could not be delivered.

Connections are used when it is important that the entire message gets through. When your browser is loading a Web page, your computer and the Web server share a connection across the network. This makes sure that all parts of the Web page are delivered and that any pieces that don't arrive first time are retransmitted. The effort of setting up and managing the connection and requesting retransmission when things fail to turn up means that data in connections will transferred more slowly. Managing a connection places heavier demands on the systems communicating this way.

The Internet has a protocol, called Transmission Control Protocol (or TCP), which describes how to set up and manage a connection between to network stations.

Datagrams are used for sending data in situations where you don't need an acknowledgement. If you were flashing your bedroom light every ten seconds to show the score in a football game it wouldn't matter if anyone receiving your messages missed a message as they would just have to wait for the next one to arrive. If a datagram is lost by the network, there is no point in asking for it again because by the time the replacement arrives, it will be out of date. Datagrams work well when you want to send data as fast as you can and it doesn't matter if some gets lost on the way. They are used for streaming media, where moving video is being sent and the priority is to get the signal delivered as fast as possible.

The internet has a protocol called User Datagram Protocol (or UDP) that describes how systems can send datagrams between each other.

The Windows Phone networking system can send datagrams or create connections, depending on what you want to do. You need to decide which connection type you should use in a particular situation.

7.2 Creating a User Datagram Protocol (UDP) Connection

We are going to start by creating a program that will use UDP to exchange messages with a remote service. The remote service we are going to use is called "echo". As the name implies, we send echo a message and the service just sends the same message right back to us. The echo service is very useful for testing a connection. It was one of the first ever Internet services and was given port number 7. Nowadays not all systems

have this service running, but we can start the echo service on our PC and use this to explore how datagrams are sent and received. We will see how to set up this service later in the section, when we actually have a program that we want to test.

We are going to write a method that uses UDP to send a message to a distant server. The user will type in the message that they want to send and identify the host they want to send to and the program will do the rest. The text will be entered into TextBox elements in our Silverlight program. The method will look like this:

```
/// <summary>
/// Send a message to a host using UDP
/// </summary>
/// <param name="message">message to send</param>
/// <param name="hostUrl">Url of host</param>
/// <param name="portNumber">number of port</param>
/// <returns>empty string or error message</returns>
public string SendMessageUDP(string message, string hostUrl,
                                              int portNumber)
{
    // Creates a socket and sends the message
    // returns an error message anything fails
    // returns an empty string if the message was sent OK
}
```

We can use the method like this:

```
const int ECHO_PORT = 7;
string sendResponse = SendMessageUDP(MessageTextBox.Text,
                                     HostTextBox.Text, ECHO_PORT);
```

This method takes text from to Silverlight TextBoxes that contain the message text and the name of the host and sends them to the ECHO_PORT on the remote server. The above call will send to port number 7, but it could be used to send a datagram to any port.

To manage the connection between the program and the network the program will create an instance of the `Socket` class.

The Socket class

We are well used to objects standing in for something that has meaning in the real world. We use Silverlight elements to represent user interface components and we use streams to represent connections to files. The `Socket` class is how a program represents a connection to an internet resource. If a program wants to connect to a number of different network services at the same time it must create a number of socket instances.

Note that although a Windows Phone has many different ways of physically connecting to the Internet a program will use the same socket technology to manage the connection. The layer underneath the `Socket` will select the most appropriate medium for the connection. Normally this will mean using WIFI if that is available. If there is no WIFI the connection will be made using the slower and less reliable cellular phone network.

Unlike desktop PCs with their fixed, wired, connections there is always the possibility that a Windows Phone has no network coverage, or the user has selected “flight” mode and turned off the network completely. In these situations it is not acceptable for a program to just crash. Instead it must either work around the missing network or give the user sensible status messages.

The `Socket` class is described in the `System.Net.Sockets` namespace, so we need to include this in any C# program that wants to use this class:

```
using System.Net.Sockets;
```

The network library files are already part of the project, so there is no need to add any extra resources.

The starting point for a network connection is the creation of a socket instance to represent the connection that the program wants to make.

```
Socket hostSocket;
```

When a socket is created the constructor is given the settings to be used for that particular socket type.

```
hostSocket = new Socket(AddressFamily.InterNetwork,
                        SocketType.Dgram, ProtocolType.Udp);
```

This creates a new `Socket` which uses internetwork V4 addressing and is *datagram* based using the UDP protocol. This is the basis of a datagram connection. We will use the tcp connection type later on.

Sockets and the SocketAsyncEventArgs class

Now that we have our socket we need to configure it and give it the data that we want to send. To do this we create an instance of the `SocketAsyncEventArgs` class and use this to configure the `hostSocket` value.

The `SocketAsyncEventArgs` class contains the properties for a socket which will be used to create an *asynchronous* connection.

```
// Create the argument object for this request
SocketAsyncEventArgs socketEventArgs = new SocketAsyncEventArgs();
```

The first thing we need to do is set the *endpoint* of the connection to identify the system and port that we want to communicate with.

Setting the EndPoint of a `SocketAsyncEventArgs` *value*

An endpoint is the thing that we want to talk to over the internet. It is made up of a host address and the number of the port on that host. In effect this is identifying the program on the distant machine that our program is trying to connect to. The endpoint with the internet address `www.hull.ac.uk` and the port number `80` actually identifies the Web server program on a machine on the University of Hull campus. We want our endpoint to identify an echo program (behind port 7) on a server.

The `SocketAsyncEventArgs` class exposes a property called `RemoteEndPoint` that we set to describe the connection endpoint we want the `Socket` to use.

```
socketEventArgs.RemoteEndPoint =
                        new DnsEndPoint(hostUrl, portNumber);
```

The statement above creates a new `DnsEndPoint` value from the `hostUrl` (the internet address of the host) and the `portNumber` (number of the port on that host) that we want to use. The `DnsEndPoint` class is very useful, as it will perform DNS lookups for us. We have already seen that systems on the internet use the Domain Naming System to perform a mapping between the names we want to use (`www.hull.ac.uk`) and the addresses that are used to route messages (`150.237.176.24`).

When a `DnsEndPoint` instance is created it is given the text url of the target system and will use the Doman Name System to find the actual IP address that is required by the socket to make the connection. If this address cannot be found the send will fail and the program will be given an appropriate error code.

Encoding the message we want to send

The next thing we need to do is load a message into the `SocketAsyncEventArgs` that describes what we want the `Socket` to do. This is the actual payload of our network action. When your browser fetches a web page from a server the data payload of the

message is the text of the web request that identifies the page you want to read. In our echo program it will be the text that we want the echo service to send back to us.

```
// Put the message into the buffer
byte[] messageBytes = Encoding.UTF8.GetBytes(message);
socketEventArgs.SetBuffer(messageBytes, 0, messageBytes.Length);
```

The internet works in terms of “octets” which are the 8 bit data values it moves around. This means that we can use an internet connection to transfer any kind of data, whether it is images, sounds, video or text, as long as we can break that data down into 8 bit values. However, it also means that before we can send any kind of data we have to encode it into a sequence of octet values.

The Echo service that we are going to use returns the octet sequence that it receives. This means that it could echo an image, a sound or anything else that we send in the form of data.

To convert text into 8 bit values we can make use of the `Encoding` class which is described in the `System.Text` namespace:

```
using System.Text;
```

This class contains a number of encoding methods that can take a string and convert it into a sequence of 8 bit values in an array of bytes. The `UTF8` encoding scheme provides a mapping for `UNICODE` characters (like the ones manipulated by the Windows Phone) into sequences of bytes.

The `GetBytes` method takes a string and returns the contents of the string encoded as an array of 8 bit byte values that can be transferred over the internet. When we get the response back from the echo server we will have to reverse this translation.

Handing the response from the Socket connection

We have now identified the end point we are sending the message to and also set the message that we want to send. The final thing we need to do is specify the method that will be called when the socket completes the action. We need to do this because the `SocketAsyncEventArgs` class, as its name implies, describes a connection that is managed asynchronously.

There are two ways you can do things, synchronous and asynchronous. When you do something synchronously what we are really saying is “while you wait”. If I have my car serviced “synchronously” it means that I stand in the garage waiting for the “service my car” method to complete, at which point I can take my car and drive home. If I have my car serviced “asynchronously” this means that I go off and do some shopping while the job is being done. At some point the garage will ring me up and deliver a “service complete” event. At which point I go and pick up my car and drive home.

Making asynchronous use of data services is very sensible, particularly on a device such as a Windows Phone. The limitations of the network connection to the phone mean that a network connection may take several seconds to deliver an answer. A program should not be stopped for the time it takes for the server to respond.

We are used to writing programs that respond to events generated by display elements. You make a program respond to events from network requests in exactly the same way. The `SocketAsyncEventArgs` class contains a `Completed` property which generates events when a network action completes. When an action completes we need to run some code that will check to see if the network request has worked. This code will be connected to the `Completed` property.

```
public string SendMessageUDP(string message, string hostUrl,
                                            int portNumber)
{

    // String to hold the response message for a caller
    // to SendMessage
    string response = "";

    // Create the argument object for this request
    SocketAsyncEventArgs socketEventArgs =
                                    new SocketAsyncEventArgs();

    // Make an endpoint that connects to the server
    socketEventArgs.RemoteEndPoint = new DnsEndPoint(hostUrl,
                          portNumber, AddressFamily.InterNetwork);

    // Put the message into the buffer
    byte[] messageBytes = Encoding.UTF8.GetBytes(message);
    socketEventArgs.SetBuffer(messageBytes, 0,
                                         messageBytes.Length);


       // Connect an event handler to run when a request completes
       socketEventArgs.Completed +=
                    new EventHandler<SocketAsyncEventArgs>
                        (delegate(object s, SocketAsyncEventArgs e)
        {
            if (e.SocketError == SocketError.Success)
            {
                response = "";
            }
            else
            {
                response = e.SocketError.ToString();
            }
            // Enable any threads waiting on this thread
            transferDoneFlag.Set();
        });

    ...

}
```

We can bind a method to the `Completed` event which is exposed by a `SocketAsyncEventArgs` value. The code above does this binding, but it works in a way you might not have seen before. It binds a block of code directly to the event itself. This is a good idea because it means that the event hander code can access local variables in the `SendMessageUDP` method, in this case the string variable called `response` that we are using to hold error reports. This can be a bit confusing though. Remember that the statements in the event handler (those shaded in the above code) are the ones that run when the event fires. This code is not run when `SendMessageUDP` is called, only when a socket action has completed.

The `Completed` event is passed a reference to the `SocketAsyncEventArgs` value that we set up to describe the transaction. This exposes a status flag called `SocketError` which our program can test to determine whether or not the message was sent successfully.

```
if (e.SocketError != SocketError.Success)
{
    response = "";
}
else
{
    response = e.SocketError.ToString();
}
```

The code above tests the result flag and if the message did not get through it sets the string `response` to a description of the error that was detected. I'm using an error reporting technique that I call "Silence is Golden". If the method completes with the `response` variable holding an empty string this means the message has been sent successfully. If `response` holds anything it means bad news.

Flags and Threads

The second thing that the event handler does is set a flag to indicate that our program has had a response from the socket:

```
// Enable any threads waiting on this thread
transferDoneFlag.Set();
```

To understand what this means we have to think a bit about how asynchronous operations work. An asynchronous operation happens "in the background". At some point the action completes and calls an event method to indicate that it has finished. This is just like me dropping my car off at the garage for a service. The mechanic will work on my car while I go off to do something else, perhaps relax in a café with a good book. At some point my car will be ready, and the garage will give me a call to let me know.

In .NET asynchronous actions are performed by threads of execution. When the phone is switched on it actually contains many different threads that are looking after different aspects of the phone behaviour. Our program is just one more thread that is active on the device. Programs can start further threads to perform asynchronous activities and sometimes a thread will need to wait for another one to finish (rather like me sitting reading a book and waiting for a call from the garage). The .Net framework provides a means by which threads can communicate by using the `ManualResetEvent` class.

```
static ManualResetEvent transferDoneFlag =
                                    new ManualResetEvent(false);
```

The program contains a variable called `transferDoneFlag`. This can be used as a flag. This has been made `static` so that only one flag will exist for all instances of the transfer class. This is because we can only create a limited number of `ManualResetEvent` instances on a given system. Note that this could lead to problems if we had multiple socket requests active at the same time, since they would all be using the same flag variable. In this situation we would have to add extra code to manage the different requests.

The `transferDoneFlag` variable has two possible states. It is initially set to `false` (i.e. not set). A thread can wait for another thread to set this flag (change it to `true`) by using the `WaitOne` method:

```
transferDoneFlag.WaitOne();
```

This will stop the present thread until another thread sets the `transferDoneFlag` flag:

```
transferDoneFlag.Set();
```

In "waiting for the car to be serviced terms we would use the flags as follows:

```
serviceDoneFlag.WaitOne();  // Me drinking coffee until car ready
pickUpCar();                // method to pick up the car

...

doingCarService();       // Garage doing service on car
serviceDoneFlag.Set() ;  // Garage calling me to say car is ready
```

If the two threads are arranged like this it means that the `pickUpCar` method in the "me" thread will not run until the `doingCarService` at the garage has completed.

One problem with the above design is that the `WaitOne` method will cause a thread to pause until the flag is set. If the flag is never set the thread will be paused for ever. The same thing would happen if the garage forgot to call me, or I had given them the wrong phone number. With the code above I would be sat waiting for ever, which would not be good. In real life I'd probably sit for an hour or so and if I'd not heard anything from the garage I'd give them a call. This is called a timeout, and we can do just the same thing with the `WaitOne` method. We can give the `WaitOne` method a timeout value in milliseconds.

```
transferDoneFlag.WaitOne(1000);   // Wait for 1 second
```

The waiting thread will continue if the flag is set by another thread or after one second. This is actually rather impatient. Network requests can take a while to complete, especially when you add the time taken to look up the address of a system, with this in mind a timeout of several seconds is appropriate.

This approach is especially useful on the phone platform, as there is a much higher chance of a network request failing on a mobile device. When you write code that uses the network you must make sure that you include timeouts and your program does sensible things when a connection is not possible.

You might be thinking that making one thread wait for another is a bad idea for performance reasons. However this is not the case. When a thread is waiting for a flag it is actually suspended and using no system resources. The operating system manages the threads and will "wake up" any threads waiting for flags.

Resetting the Flag

The technique we are using is that the "foreground" thread (the one running in the `SendMessageUDP` method) will start a socket request and then wait for the "background" thread (the one running in the `Completed` event handler) to set the flag indicating it has completed.

If the foreground thread is going to wait for the flag to be set it must first clear the flag.

```
transferDoneFlag.Reset();
```

If we didn't do this it might mean that our program will not wait when it should, as the flag might already be set when the `WaitOne` method is called.

Sending the message and waiting for a response

Now that we have set up the address of the endpoint of the message, the contents of the message and what to do when the message completes we can send the message to the remote system:

```
// Send the message
hostSocket.SendToAsync(socketEventArgs);

// Wait on the event
// Will continue after the timeout or
// if a response is received
transferDoneFlag.WaitOne(MESSAGE_TIMEOUT_MSECS);
```

The Complete Method

```
static ManualResetEvent transferDoneFlag =
                                    new ManualResetEvent(false);
const int MESSAGE_TIMEOUT_MSECS = 10000;

public string SendMessageUDP(string message, string hostUrl,
                                                int portNumber)
{
    string response = "Timed Out";

    SocketAsyncEventArgs socketEventArgs =
                                     new SocketAsyncEventArgs();

    socketEventArgs.RemoteEndPoint =
                     new DnsEndPoint(hostUrl, portNumber,
                                       AddressFamily.InterNetwork);

    byte[] messageBytes = Encoding.UTF8.GetBytes(message);
    socketEventArgs.SetBuffer(messageBytes, 0,
                                       messageBytes.Length);


    socketEventArgs.Completed +=
             new EventHandler<SocketAsyncEventArgs>
                    (delegate(object s, SocketAsyncEventArgs e)
    {
        if (e.SocketError == SocketError.Success) {
            response = "";
        }
        else {
            response = e.SocketError.ToString();
        }
        transferDoneFlag.Set();
    });

    transferDoneFlag.Reset();

    hostSocket.SendToAsync(socketEventArgs);

    transferDoneFlag.WaitOne(MESSAGE_TIMEOUT_MSECS);

    return response;
}
```

This is the complete method that we have been working on over the last few pages. The response string is initially set to “Timed Out”. If the socket does not respond before the timeout, this is the string that is returned from the method. If the socket request does respond and the response indicates success, the method will return an empty string.

Sending a Message

The method can be called by giving it the text to send, the name of the host and the port to use. It will return a response which we can then test in our program:

```
string sendResponse = SendMessageUDP(
                MessageTextBox.Text, HostTextBox.Text, ECHO_PORT);
```

If the address of the host is invalid the message will fail. However, that is pretty much the only way that this method call will fail. Remember that this is a datagram. We have no way of knowing if the response was received by the remote system unless we get something back from it.

Receiving a Message

If we send a message to a host that has the echo service running behind its echo port we would expect to get a response back from the host containing a copy of the message we sent. We therefore need to set up another socket connection to the remote host to read back the response. We could write another method to send this read request:

```
public string ReceiveMessageUDP(int portNumber, out string result)
{
    string response = "Error: Request Timed Out";
    string message ="";

    SocketAsyncEventArgs socketEventArgs =
                                new SocketAsyncEventArgs();
    socketEventArgs.RemoteEndPoint =
                    new IPEndPoint(IPAddress.Any, portNumber);

    byte  [] responseBytes = new Byte[MAX_BUFFER_SIZE];
    socketEventArgs.SetBuffer(responseBytes, 0, MAX_BUFFER_SIZE);

    socketEventArgs.Completed +=
           new EventHandler<SocketAsyncEventArgs>
              (delegate(object s, SocketAsyncEventArgs e)
    {
        if (e.SocketError == SocketError.Success) {
            response = "";
            message = Encoding.UTF8.GetString(e.Buffer, e.Offset,
                                              e.BytesTransferred);
            message = message.Trim('\0');
        }
        else {
            response = e.SocketError.ToString();
        }

        transferDoneFlag.Set();
    });

    transferDoneFlag.Reset();

    hostSocket.ReceiveFromAsync(socketEventArgs);

    transferDoneFlag.WaitOne(MESSAGE_TIMEOUT_MSECS);

    result = message;
    return response;
}
```

This method looks very similar to the send message method. It will listen on a particular port for a response from the network. It differs from the send method in a few respects though:

```
socketEventArgs.RemoteEndPoint =
                        new IPEndPoint(IPAddress.Any, portNumber);
```

The endpoint for a receive request can be set to an `IPEndPoint` value which can receive on any IP Address. This would mean that any remote system could send a message to the socket on our system and we would receive it. If we only want to receive messages from one particular system we could put the address of that system here.

Since we are requesting some data from a distant system we need to create a buffer to hold the bytes that are sent back to us.

```
byte [] responseBytes = new Byte[MAX_BUFFER_SIZE];
socketEventArgs.SetBuffer(responseBytes, 0, MAX_BUFFER_SIZE);
```

These two statements create a byte array to receive the message and then set the buffer on the `socketEventArgs` value to this array.

We also need to make some changes to the code that runs when the receive request completes. When the response is received the program must fetch the data that the remote system has sent us.

```
socketEventArgs.Completed +=
                new EventHandler<SocketAsyncEventArgs>
                   (delegate(object s, SocketAsyncEventArgs e)
{
    if (e.SocketError == SocketError.Success) {
        response = "";
        // Retrieve the data from the buffer
        message = Encoding.UTF8.GetString(e.Buffer, e.Offset,
                                          e.BytesTransferred);
        message = message.Trim('\0');
    }
    else {
        response = e.SocketError.ToString();
    }
    transferDoneFlag.Set();
});
```

The string `response` is set to the result of the request as before. The response is an empty string if the request succeeded or an error message if something went wrong.

If the receive request succeeds the program must convert the stream of 8 bit values that have been sent into the data our program will work with. The code uses the `Encoding.UTF` class to convert the buffer that has been received into a string that can be displayed. Finally the event handler sets a flag to allow a thread to synchronise on the text received event. Remember that, as with the send request we saw earlier, this code runs when the asynchronous request to the socket has completed.

Finally the method must return the string of text that was received.

```
// Clear the flag this is set when the request is completed
transferDoneFlag.Reset();

// Send off the read request to the socket
hostSocket.ReceiveFromAsync(socketEventArgs);

// Pause this thread until the socket responds or the wait
// times out
transferDoneFlag.WaitOne(MESSAGE_TIMEOUT_MSECS);

// copy the text that was received into the out result
result = message;
// Return status of our request, an empty string if it worked
return response;
```

This sequence of statements resets the thread synchronisation flag and then makes a Socket request to receive some data. It then waits for the socket to complete and, when it does, it sets the result parameter to the message that was received and returns the response.

A call of this method looks like this:

```
string receiveResponse;
string message;

receiveResponse = ReceiveMessageUDP(ECHO_PORT, out message);
```

Sending a datagram to the Echo service

We can now put together some statements that will make a call to an Echo service. We will first call `SendMessageUDP` to send a message to the service and then call `ReceiveMessageUDP` to get the echo back.

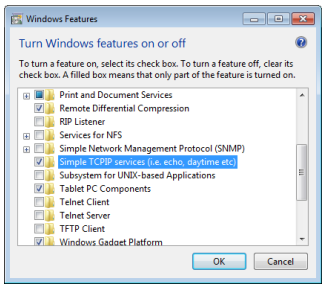
```
string sendResponse = SendMessageUDP(MessageTextBox.Text,
                                     HostTextBox.Text, ECHO_PORT);

if (sendResponse.Length == 0) {
    string receiveResponse;
    string message;

    receiveResponse = ReceiveMessageUDP(ECHO_PORT, out message);
    if (receiveResponse.Length == 0) {
        ResponseTextBox.Text = message;
    }
    else {
        ErrorTextBox.Text = "Receive Error: " + receiveResponse;
    }
}
else {
    ErrorTextBox.Text = "Send Error: " + sendResponse;
}
```

If the `SendMessageUDP` returns an empty string it means that the message was sent successfully and the program can call the `ReceiveMessageUDP` method to get the text back from the Echo server.

The solution in *Demo 02 UDPNetworkClient* contains a program that connects to the echo service on a host. You can type in some text and this will be sent to the echo port on the host that you identify. The program will display any errors that are detected.

Setting up an Echo Server

To test the example program above you need to have a system that provides an echo server. Fortunately a Windows PC can work as an echo server, but this service has to be enabled before you can use it. To turn on the service you first need to open the Windows Control panel and open “Turn Windows Features on or off”. This will open the Windows Features dialog.



Find the “Simple TCPIP services” option and ensure that it is selected. Then click OK. You will now have to reboot your PC to start the services running. Once you have done this you will be able to enter your computer name into the address for a service and it will respond to echo requests.

This shows the echo client working on my machine once I had started the service. My machine is called “martini”. If you happen to know any servers on the internet which provide the echo service you can enter their names in the Host dialog box.

7.3 Creating a Transmission Control Protocol (TCP) Connection

The problem with User Datagram Protocol (UDP) is that systems that use it to transfer data do not know if the recipient received anything. If you use the demo program above to connect to a machine that is on the Internet but not running the echo service your outgoing message will be sent with no reported errors. However, the as there is not a service listening behind port 7 at the destination it will not receive anything back in response. This means that the receive request will timeout when the host does not respond with the echoed message.

If we want to create reliable connections we have to use a different protocol. This is the Transmission Control Protocol we saw earlier. This provides a connection between two machines. You can compare Transmission Control Protocol (TCP) with User Datagram Protocol (UDP) by comparing SMS text messages with phone calls. If I send someone a text message I have no idea whether or not they received it. I will only know that they have got the message when I get a response from them. This is the service provided via UDP. However if I call them on the phone we make a connection. I can send and receive messages during the call and be sure that they are at the other end listening. This the service provided by TCP.

When we use TCP we actually create an on-going connection. Messages are created and transferred in exactly the same way, but each message is part of a call, rather than being an individual event. The call will exist until one of the parties in it decides to terminate the conversation. The response that we receive from sending a message will allow our programs to determine whether the message was delivered successfully or not. At the end of the call one of the parties will close the connection.

Reading a Web Page

The method we are going to create will look like this:

```
private string RequestWebPage(string url, string page,
                              out string pageContent)
```

It is given the web address of the server, the name of the page to fetch and places the text in the third parameter. It returns a response that is an empty string if the request worked or an error description if the request failed. The following call would get the home page of my blog.

```
string resp;
string page;
resp = RequestWebPage ("www.robmiles.com",
                        "index.html", out page);
```

The World Wide Web uses a protocol called Hyper Text Transfer Protocol or HTTP. This protocol defines the language used by a browser to request particular pages of content from a web server. For example, to get a web page from the server the browser will send the command GET, along with the location of the web page on the server.

Because it is important that all the parts of a web page are delivered to the client the HTTP protocol uses a TCP connection. To fetch a web page a program must do the following:

1. Create a TCP connection to the web server using port 80, which is the one reserved for HTTP requests.
2. Send the GET request to the server on this connection.
3. Read the web page content from the server. This may require several read operations if the page is large.
4. Close the connection.

Setting up the TCP Connection

We can use the same format for the connection methods as we did before.

```
private string ConnectTCP(string host, int port)
{
    string response = "Connect timed out";

    DnsEndPoint hostEntry = new DnsEndPoint(host, port);

    hostSocket = new Socket(AddressFamily.InterNetwork,
                            SocketType.Stream, ProtocolType.Tcp);

    SocketAsyncEventArgs socketEventArgs =
                                    new SocketAsyncEventArgs();
    socketEventArgs.RemoteEndPoint = hostEntry;

    socketEventArgs.Completed +=
                    new EventHandler<SocketAsyncEventArgs>
                      (delegate(object s, SocketAsyncEventArgs e)
    {
        if (e.SocketError == SocketError.Success) {
            response = "";
        }
        else {
            // Retrieve the result of this request
            response = e.SocketError.ToString();
            // Ensure the socket is flagged unavailable
            hostSocket = null;
        }
        transferDoneFlag.Set();
    });

    transferDoneFlag.Reset();

    hostSocket.ConnectAsync(socketEventArgs);

    transferDoneFlag.WaitOne(MESSAGE_TIMEOUT_MSECS);
    return response;
}
```

This code is very similar to that we used to send a UDP message. We set up a `SocketAsyncEventArgs` value and then use this to configure the call that we make to the socket. However, in this case we call the `ConnectAsync` method:

```
hostSocket.ConnectAsync(socketEventArg);
```

If this method completes successfully it will return with a TCP connection set up to the remote system. We can then use this to send and receive messages until the connection is closed by either party. However, note that the connection only lasts for the transmission of a single page of content. If a user navigates to a different page this will require the creation of another TCP connection to fetch the new page.

Sending a Message to a TCP Connection

To send a message we simply need to assemble the message text and then use the socket to send it. The socket that we have set up contains properties that we can use to configure the message transfer:

```
private string SendMessageTCP(string message)
{
    if (hostSocket == null) return "Send socket not open";

    string response = "Connect timed out";

    SocketAsyncEventArgs socketEventArg =
                                    new SocketAsyncEventArgs();

    // Use the endpoint in the socket to set up the arguments
    socketEventArg.RemoteEndPoint = hostSocket.RemoteEndPoint;

    byte[] messageBytes = Encoding.UTF8.GetBytes(message);
    socketEventArg.SetBuffer(messageBytes, 0, messageBytes.Length);

    socketEventArg.Completed +=
             new EventHandler<SocketAsyncEventArgs>
               (delegate(object s, SocketAsyncEventArgs e)
    {
        if (e.SocketError == SocketError.Success) {
            response = "";
        }
        else {
            response = e.SocketError.ToString();
            CloseTCP();
        }

        transferDoneFlag.Set();
    });

    transferDoneFlag.Reset();

    hostSocket.SendAsync(socketEventArg);

    transferDoneFlag.WaitOne(MESSAGE_TIMEOUT_MSECS);

    return response;
}
```

This code is almost identical to the send message that we created earlier for UDP connections. This version checks to make sure that the socket has been set up before it uses it:

```
if (hostSocket == null) return "Send socket not open";
```

It also closes the socket if the request returns with an error. The `CloseTCP` method just closes the socket if it exists:

```
void CloseTCP()
{
    if (hostSocket != null) {
        hostSocket.Close();
        hostSocket = null;
    }
}
```

Receiving a Message from a TCP Connection

The final method that we need to read a web page using a TCP connection is one to receive a message. Again, this looks remarkably like the ones that we have used to read UDP messages:

```
private string ReceiveMessageTCP(out string message)
{
    message = "";

    if (hostSocket == null) return "Send socket not open";

    string response = "Receive timed out";
    string result = "";

    SocketAsyncEventArgs socketEventArg =
                                    new SocketAsyncEventArgs();
    socketEventArg.RemoteEndPoint = hostSocket.RemoteEndPoint;

    // Create a buffer for the response
    byte[] responseBytes = new Byte[MAX_BUFFER_SIZE];
    socketEventArg.SetBuffer(responseBytes, 0, MAX_BUFFER_SIZE);

    socketEventArg.Completed +=
              new EventHandler<SocketAsyncEventArgs>
                (delegate(object s, SocketAsyncEventArgs e)
    {
        if (e.SocketError == SocketError.Success) {
            response = "";
            if (e.BytesTransferred > 0) {
                result = Encoding.UTF8.GetString(e.Buffer,
                              e.Offset, e.BytesTransferred);
                result = result.Trim('\0');
            }
        }
        else {
            response = e.SocketError.ToString();
        }
        transferDoneFlag.Set();
    });

    transferDoneFlag.Reset();

    hostSocket.ReceiveAsync(socketEventArg);

    transferDoneFlag.WaitOne(MESSAGE_TIMEOUT_MSECS);

    message = result;

    return response;
}
```

This version of the method checks to see if any bytes have been transferred from the server and if they have it decodes the bytes and returns them as a string result.

Building the Complete Request

Now that we have the methods that we need to build a request for a web page we can put them together to do the job. This method will send the request and then assemble the web page text from the responses. It knows when the transfer is complete as it will receive an empty buffer from the read method call.

```
private string RequestWebPage(string url, string page,
                              out string pageContent)
{
    pageContent = "";

    // Set up the connectiopn
    string response = ConnectTCP(url, 80);

    if (response != "") {
        return response;
    }

    // Send the page request using the HTTP Get command
    response = SendMessageTCP("GET " + page + " HTTP/1.1\r\nHost: "
                        + url + "\r\nConnection: Close\r\n\r\n");

    if (response != "") {
        CloseTCP();
        return response;
    }

    // Repeatedly ask the server for packets until we
    // get an empty one sent back
    string wholePage = "";
    string fetchText;

    do {
        response = ReceiveMessageTCP(out fetchText);

        if (response != ""){
            CloseTCP();
            return response;
        }

        if (fetchText == "") break;

        wholePage = wholePage + fetchText;
    } while (true);

    pageContent = wholePage;

    CloseTCP();

    return response;
}
```

This method calls each of our methods in turn, building up the text of the complete web page from successive responses. If any of the methods returns an error response the method will close the TCP connection and return.

The method is called when the user presses the Load button in our application:

```
private void LoadButton_Click(object sender, RoutedEventArgs e)
{
    string response;
    string webPageText;

    response = RequestWebPage(UrlTextBox.Text, PageTextBox.Text,
                              out webPageText);

    StatusMessage.Text = response;
    ResponseTextBlock.Text = webPageText;
}
```

This loads the url and the page name from the textboxes and then calls the method to requeset the web page. It then displays any response that was received.

This shows the program in operation. We can see the web page content that has been loaded.

The solution in *Demo 03 TCPNetworkClient* contains a program will fetch web pages for you using the methods described above.

Page Requests and Threads

At the moment all the activity for the page loading takes place in the button event handler. This means that the button pressed event will not complete until either the page has been fetched and displayed or something has gone wrong. We can see this in the user interface of the program above; the button seems to take a while to return to the “un-pressed” state when we perform a load. This is not necessarily a problem, but if we wanted the user to be able to do other things while our page was loading this would not be possible.

If we want to allow the user to perform actions in parallel – for example load a web page and use the echo server at the same time we have to use multiple threads to do the work. This makes our life much more difficult as developers, but the user will prefer this approach. If you want to do this kind of thing you need to find out more about threading, which works in exactly the same way on Windows Phone as on any other .NET platform.

Data Chunks and the UTF8 Encoding

If you use the program above you might notice that every now and then it will corrupt a web page. This is because of the way that it works. We are using socket methods that fetch an array of bytes from a web server and then convert that array into a string of

text. The encoding method converts the text that we have typed into one or more 8 bit values that are transferred for us. This would work fine if the web server sent us the entire page in response to a single read request, but this is not how networks send things. Networks transfer large amounts of data by breaking it down into smaller chunks. And the network is completely unaware of the nature of the data that it is sending.

This gives us a problem, in that if a UTF8 character is split over two packets it will become corrupted. As an example, the UTF8 character for the pound currency sign (£) is the pair of bytes 0xC2 and 0xA3 in sequence. If the 0xC2 is transferred as the very last character of a transfer and the 0xA3 sent as the first character of the next one my code above will cause this character to be corrupted.

The HTML protocol itself will normally encode these kinds of characters into values that should not cause problems like this, but if we decide to transfer text other than web pages we need to address this. There are two ways to solve it, we can either keep track of "trailing" bytes that the UTF8 decoder did not understand or we can create an array of byte values from the incoming messages and decode the whole array at once. I've written some sample code that does the latter, it assembles a large byte array. This is actually quite a useful thing to have, in that it makes it easy for us to receive other kinds of binary data that may be sent from a web host.

The solution in *Demo 04 Binary TCPNetwork Client*contains a program will fetch web pages for you and will assemble the message in bytes and decode them all at once.

7.4 Connecting to a Data Source

We have seen that we can create a TCP connection and read text from a web page. Reading information from network datasources is something that programs do a lot. However, this is rather tedious to do. Fortunately there is a much easier way of reading web pages. We can use the `WebClient` class.

Using the WebClient class

The `WebClient` class is provided as part of the Windows .NET networking library, so if you learn how to use it on the Windows Phone you can also use it in your Windows desktop programs. We can use the class to perform an HTTP (Hyper Text Transfer Protocol) interaction with a remote web server. There are a number of different ways we can use a `WebClient`, we are going to look at one of the simplest.

We are going to create a `WebClient` variable in the `MainPage.xaml.cs` of our application. When the user presses the `loadButton` our application will load the url from a `TextBox` and then display the text content of that page.

Above you can see the program in action, displaying one of the best pages on the web. Note that this is exactly what a browser does, except that a browser will then take the HTML content received from the server and render it along with any associated graphical content.

The first thing we need to do is declare our `WebClient` as a member of the `MainPage` class. This will be used manage the web requests that we are going to make.

```
WebClient client;
```

At the moment we just have the reference to a `WebClient` object. The next thing we need to do is create an actual instance of a `WebClient`. We can do this in the constructor for the page class:

```
public MainPage()
{
    InitializeComponent();

    client = new WebClient();
    client.DownloadStringCompleted +=
                new DownloadStringCompletedEventHandler(
                    client_DownloadStringCompleted);
}
```

The constructor creates a new `WebClient` and then binds a method to the `DownloadStringCompleted` event that the `WebClient` exposes. The method, called `client_DownloadStringCompleted`, will display the web page text in the in the `pageTextBlock`:

```
void client_DownloadStringCompleted(object sender,
                                    DownloadStringCompletedEventArgs e)
{
    if (e.Error == null)
    {
        pageTextBlock.Text = e.Result;
    }
}
```

When this method is called it tests to see if any errors were reported. If there are no errors the method displays the result of the download in the `pageTextBlock` element.

The last thing we need to add to our program is the event handler for the button that will start the download of a page at the url that the user has typed in:

```
private void loadButton_Click(object sender, RoutedEventArgs e)
{
    client.DownloadStringAsync(new Uri(urlTextBox.Text));
}
```

When the load button is clicked the following sequence takes place:

1. The text from the `urlTextBox` is used to create a new `Uri` which describes the web site to be loaded.
2. The `DownloadStringAsync` method is called on the `WebClient`. This starts a new web transaction. This method returns immediately, so the button event handler is now completed.
3. Some-time later the `WebClient` will have fetched the string of content at the url. It fires the `DownloadStringCompleted` event to indicate that it has finished, and to deliver the result.
4. The client_`DownloadStringCompleted` method now runs, which checks for errors and displays the received text in the TextBlock.

The solution in *Demo 05 Web Scraper* contains this code. You can type in any url and download the html from that page. The program will work on either the emulator or a real device.

Note that this is a very basic web page reading program (as you might expect for only fifteen or so lines of code). It does not stop the user from pressing the Load button repeatedly and interrupting downloads and it does not provide any form of time out or progress display to the user.

7.5 Using LINQ to Read from an XML Stream

The ability to read from the web can be used for much more than just loading the text part of a web page. We can also interact with many other data services, for example Twitter. This uses a REST (Representational State Transfer) protocol to expose the activities of Twitter users. REST makes use of the structure of the url being used to denote the required action. As an example, if you point a web browser at the following url:

`http://twitter.com/statuses/user_timeline/robmiles.xml`

- you will be rewarded with an XML document that contains my most recent Twitter posts. By replacing robmiles with the name of another Twitter user you can read their feed.

This makes it very easy to make a slightly modified version of our WebClient program that assembles a web request for the Twitter server and uses that to read feed information from it. The only change we have to make is to the code that produces the web request:

```
private void loadButton_Click(object sender, RoutedEventArgs e)
{
    string url = "http://twitter.com/statuses/user_timeline/" +
                 nameTextBox.Text + ".xml";

    client.DownloadStringAsync(new Uri(url));
}
```

The solution in *Demo 06 Twitter Reader* contains this code. You can type in any twitter username and download the feed from that user. The program will work on either the emulator or a real device.

Structured Data and LINQ

Twitter returns status reports in the form of an XML formatted document. XML is something we know and love (a bit). What we want to do now is pull all the information out of the XML content we get from Twitter and display this information

properly. As great programmers we could of course write some software to do this. The good news is that we don't have to. We can use LINQ to do this for us.

An XML document is very like a database, in that contains a number of elements each of which has property information attached to it. Along with tools to create database queries and get structured information from them, LINQ also contains tools that can create objects from XML data. We are going to use these to read our Twitter feed.

The Twitter feed as a data source

The feed information that you get from Twitter in response to a status request is a structured document that contains attributes and properties that describe the Twitter content. If you have been paying attention during the discussions of XAML and XML you should find this very familiar.

```
<?xml version="1.0" encoding="UTF-8"?>
<statuses type="array">
<status>
  <created_at>Tue Oct 11 11:57:37 +0000 2011</created_at>
  <id>27131245083</id>
  <text>Had 12,000 hits on www.csharpcourse.com for my C# Yellow Book. No idea why.</text>
  <source>web</source>
  <truncated>false</truncated>
  <in_reply_to_status_id></in_reply_to_status_id>
  <in_reply_to_user_id></in_reply_to_user_id>
  <favorited>false</favorited>
  <in_reply_to_screen_name></in_reply_to_screen_name>
  <retweet_count>0</retweet_count>
  <retweeted>false</retweeted>
  <user>
    <id>2479801</id>
    <name>Rob Miles</name>
    <screen_name>robmiles</screen_name>
    <location>Hull, UK</location>
    <description>A Computer Science lecturer and Microsoft MVP.</description>
    <profile_image_url>http://a3.twimg.com/prof_im/157/me2_normal.jpg</profile_image_url>
    <url>http://www.robmiles.com</url>
    <protected>false</protected>
    <friends_count>21</friends_count>
    <created_at>Tue Mar 27 12:35:28 +0000 2007</created_at>
    <favourites_count>0</favourites_count>
    <utc_offset>0</utc_offset>
    <time_zone>London</time_zone>
    <profile_background_image_url>http://s.twimg.com/a/1286/images/themes/theme1/bg.png
      </profile_background_image_url>
    <notifications>false</notifications>
    <geo_enabled>true</geo_enabled>
    <verified>false</verified>
    <following>true</following>
    <statuses_count>731</statuses_count>
    <lang>en</lang>
    <contributors_enabled>false</contributors_enabled>
    <follow_request_sent>false</follow_request_sent>
    <listed_count>61</listed_count>
    <show_all_inline_media>false</show_all_inline_media>
  </user>
  <geo/>
  <coordinates/>
  <place/>
  <contributors/>
</status>
</statuses>
```

I've removed some elements to make it fit nicely on the page, but if you look carefully you can find the text of the tweet itself, along with lots of other stuff. The real document contains a number of status elements. Above I have only shown one.

In the same way as we can ask LINQ to query a database and return a collection of objects it has found we can ask LINQ to create a construction that represents the contents of an XML document:

```
XElement TwitterElement = XElement.Parse(twitterText);
```

The `XElement` class is the part of LINQ that represents an XML element. The **`twitterText`** string contains the text that we have loaded from the Twitter web site using our **`WebClient`**. Once this statement has completed we now have an `XElement` that contains all the Twitter status information as a structured document. The next thing we need to do is convert this information into a form that we can display. We are going to pull just the status items that we need out of the **`TwitterElement`** and use these to create a collection of items that we can display using data binding.

Creating Posts for Display

We know that it is very easy to bind a C# object to display elements on a Silverlight page. So now we have to make some objects that we can use in this way.

The items we are going to create will expose properties that our Silverlight elements will use to display their values. The three things we want to display for each post are the image of the Twitter user making the post, the date of the post and the post text itself. We can make a class that holds this information.

```
public class TwitterPost
{
    public string PostText { get; set; }

    public string DatePosted { get; set; }

    public string UserImage { get; set; }
}
```

These properties are all "one way" in that the **`TwitterPost`** object does not want to be informed of changes in them. This means that we can just expose them via properties and all will be well. The **`UserImage`** property is a string because it will actually give the uri of the image on the internet.

The next thing we have to do is to use LINQ to get an object collection from the `XElement` that was built from the Twitter feed we loaded.

```
var postList =
    from tweet in twitterElements.Descendants("status")
        select new TwitterPost
        {
            UserImage = tweet.Element("user").Element("profile_image_url").Value,
            PostText = tweet.Element("text").Value,
            DatePosted = tweet.Element("created_at").Value
        };
```

This code will do exactly that. It is well worth a close look. The `from` keyword requests an iteration through a collection of elements in the document. The elements are identified by their name. The program will work through each status element in turn in the document. Each time round the loop the variable **`tweet`** will hold the information from the next status element in the `XElement`.

The `select` keyword identifies what is to be returned from each iteration. For each selected tweet we want to return a new `TwitterPost` instance with settings pulled from that **`tweet`**. The **`Element`** method returns an element with a particular name. As you can see we can call the **`Element`** method on other elements. That is how we get the **`image_profile_url`** out of the **`user`** element for a tweet.

The variable **postList** is set to the result of all this activity. We can make it var type, but actually it is a collection of TwitterPost references. It must be a collection, since the from keyword is going to generate an iteration through something.

We now have a collection of TwitterPost values that we want to put onto the screen.

Laying out the post display using XAML

We have seen how easy it is to put a Silverlight element on a page and then bind object properties to it. However, we have a little extra complication with our program. We want to display lots of posts, not just one value. Ideally we want to create something that shows the layout for a single item and use that to create lots of display elements, each of which is bound to a different TwitterPost value. This turns out to be quite easy.

Above you can see what I want an individual post to look like. It will have an image on the side, with the date and post text on the right, one above each other.

From a Silverlight point of view this is three panels, arranged as shown above. We can get Silverlight to do a lot of the work in laying these out for us by using the StackPanel container element.

The first StackPanel runs horizontally. This contains the user image and then another StackPanel that runs vertically and holds the data and time above the post text.

```
<StackPanel Orientation="Horizontal" Height="132">
    <Image Source="{Binding UserImage}" Height="73" Width="73"
           VerticalAlignment="Top" Margin="0,10,8,0"/>
    <StackPanel Width="370">
        <TextBlock Text="{Binding DatePosted}" Foreground="#FFC8AB14" FontSize="22" />
        <TextBlock Text="{Binding PostText}" TextWrapping="Wrap" FontSize="24" />
    </StackPanel>
</StackPanel>
```

Above you can see the XAML that expresses all this. The StackPanel class is a lovely way of laying out items automatically. We don't have to do lots of calculations about where things should be placed, instead we just give the items and the StackPanel puts them all in the right place. You can also see the bindings that link the items to the properties in the TwitterPost class. The **UserImage** property in TwitterPost is bound to the Source property of the image. If we add the complete XAML above to our main page we have an area that can hold a list of Twitter posts.

Creating a complete layout

```
<ListBox Height="442" HorizontalAlignment="Left" Name="tweetsListBox" VerticalAlignment="Top"
                                                 Width="468">
  <ListBox.ItemTemplate>
      <DataTemplate>
          <StackPanel Orientation="Horizontal" Height="132">
              <Image Source="{Binding UserImage}" Height="73" Width="73"
                     VerticalAlignment="Top" Margin="0,10,8,0"/>
              <StackPanel Width="370">
                  <TextBlock Text="{Binding DatePosted}" Foreground="#FFC8AB14" FontSize="22" />
                  <TextBlock Text="{Binding PostText}" TextWrapping="Wrap" FontSize="24" />
              </StackPanel>
          </StackPanel>
      </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

This is exactly how we do it. This XAML repays a lot of study. The first thing we notice is that this is a `ListBox` element. We've not seem these before but they are, as you might expect, a way of holding a list of boxes. Within a `ListBox` we can put a template that describes what each item in the list will look like. This includes the data binding that will link element properties to our `TwitterPost` instances. It also includes the layout elements that will describe how the post will look.

Creating Posts for Display

The next thing our Twitter reader must do is get the posts data onto the display. Before we do this, it is worth stepping back and considering what we are about to do. This is worth doing because it will help us to understand how Silverlight binds collections to displays.

We could write some test code that made us a collection of `TwitterPost` items:

```
TwitterPost p1 = new TwitterPost
{
    DatePosted = "Tue Oct 11 11:57:37 +0000 2011",
    UserImage = "http://a3.twimg.com/profile_images/150108107/me2_normal.jpg",
    PostText = "This is a test post from Rob"
};

TwitterPost p2 = new TwitterPost
{
    DatePosted = "Wed Oct 12 14:21:04 +0000 2011",
    UserImage = "http://a3.twimg.com/profile_images/150108107/me2_normal.jpg",
    PostText = "This is another test post from Rob"
};

List<TwitterPost> posts = new List<TwitterPost>();
posts.Add(p1);
posts.Add(p2);
```

This code creates two **`TwitterPost`** instances and then adds them to a list of posts. This is not particularly interesting code. It becomes interesting when we do this:

```
tweetsListBox.ItemsSource = posts;
```

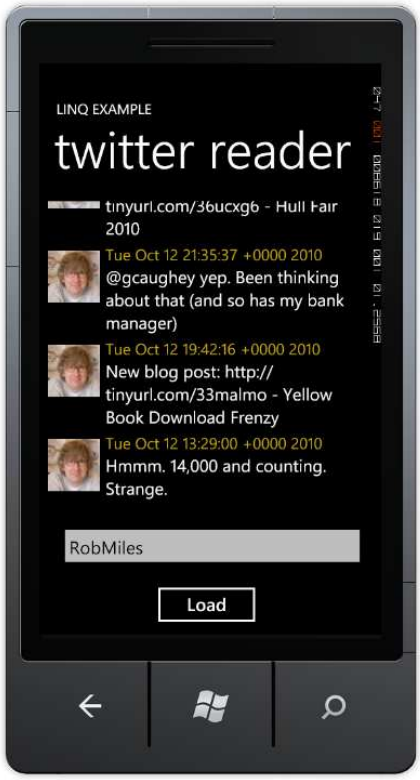
The magic of Silverlight data binding takes the list of posts and the template above, and does this with it:

This is very nice. It means that you can display any collection of data in a list on the page just by setting them as the source for a **`ListBox`**. All you have to do is create a data template that binds the properties of each instance to display elements and then assign a list of these instances to the **`ItemSource`** property of the **`ListBox`**.

Since we know that we can assign a collection of **`TwitterPost`** references to a **`ListBox`** and get this displayed the next statement is not going to come as much of a surprise.

```
tweetsListBox.ItemsSource = postList;
```

At this point our work is done, and the Twitter posts are displayed.



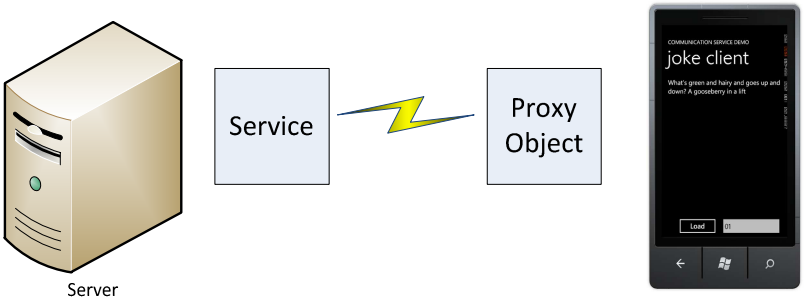
The **`ListBox`** provides a lot of useful functionality. If you run this program you will find that you can scroll the list up and down by dragging them with your finger. The scroll action will slow down and the items will even "scrunch" and bounce as they move.

The solution in *Demo 7 Complete Twitter Reader* contains this code. You can type in any twitter username and download the feed from that user and display the feed on the screen. When the program starts it is pre-loaded with the test tweets shown above.

7.6 Using Network Services

We can create Windows Phone applications that interact with web servers. If we want to pass data to and from a server using web protocols we can also use the GET and POST elements of HTTP (Hyper Text Transfer Protocol) . In this section we are going to go beyond that and investigate how we can create network based services and then consume them on the Windows Phone device.

WCF (Windows Communication Foundation) services provide a way that two processes can communicate over a network. They take care of all the transfer of messages between two systems and can work over a variety of communication media. The services are exposed by a server and then accessed by a client program. In our case the client program will be running on a Windows Phone.



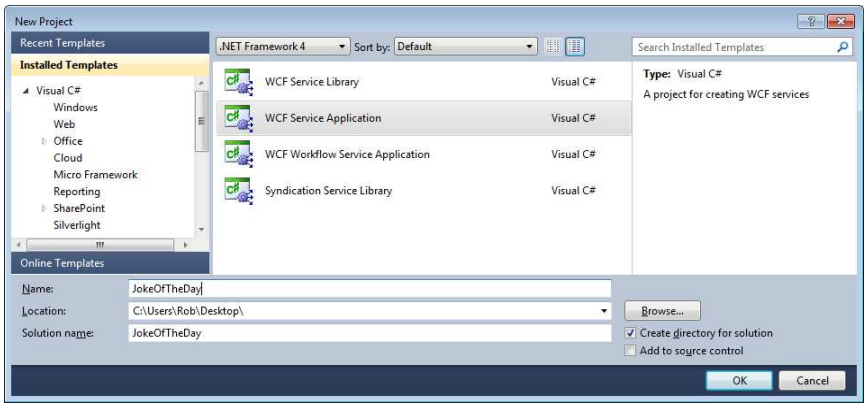
The diagram shows how this works. Software on the phone interacts with a “proxy object” that contains the methods provided by the service. A call to a method on the proxy object will cause a network message to be created that is sent to the service on the server. When the server receives the message it will then call the method code in a server object. The result to be returned by the method will then be packaged up into another network message which is sent back to the client which then sends the return value back to the calling software. We don’t need to worry about how the messages are constructed and transferred. We just have to create our server methods and calls from the client systems.

The service also exposes a description of the methods that it provides. This is used by the development tool (in our case Visual Studio) to actually create the proxy object in the client program. This means that we can create a client application very easily.

Joke of the Day service

To discover how all this works we are going to create a “Joke of the Day” service. This will return a string containing something suitably rib-tickling on request from the client. The user will be able to select the “strength” of the joke, ranging from 0 to 2 where 0 is mildly amusing and 2 is a guaranteed roll on the floor laughing experience.

The application will be made up of two parts, the server program that exports the service and the client program that uses it. The server will be created as a WCF (Windows Communication Foundation) Service application. The client will be a Windows Phone application that connects to the service and requests a joke.



Services are created as Visual Studio projects. They can run inside Visual Studio in a test environment. After testing the project files would be loaded onto a production server.

Services and Clients

Services and clients look like methods. When we write the server we write a method that does something and returns a result. When we write the client we call a method and something comes back. The communication service underneath the programs takes care of packaging the parameters and the method return value. When we create the service we create an interface that defines the method the service will provide.

```
[ServiceContract]
public interface IJokeOfTheDayService
{

    [OperationContract]
    string GetJoke(int jokeStrength);

}
```

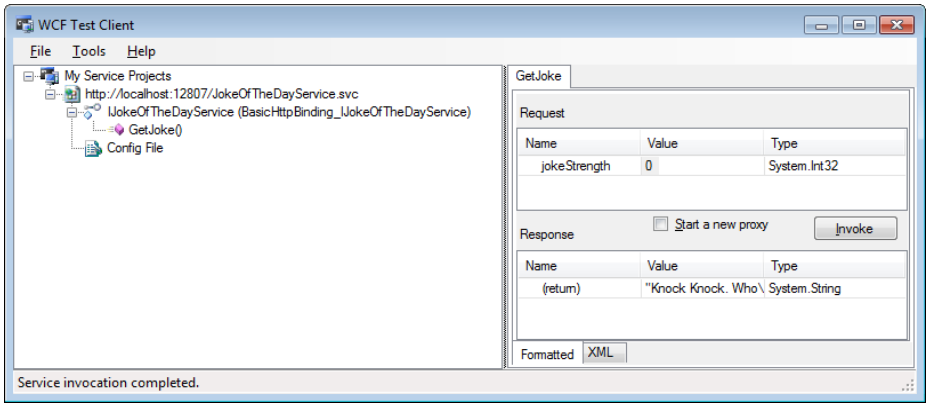
We are only going to create one method in our service, but a service could provide many methods if required. This method only accepts a single parameter and returns a result, but we could create more complex methods if we wish.

Once we have the interface we can now create the method to provide the service that the service describes. The **`[ServiceContract]`** and **`[OperationContract]`** attributes on the classes and methods provide the means by which the build process can generate the service descriptions that will be used by clients to discover and bind to these services.

```
public class JokeOfTheDayService : IJokeOfTheDayService
{
    public string GetJoke(int jokeStrength)
    {
        string result = "Invalid strength";
        switch (jokeStrength)
        {
            case 0:
                result =
"Knock Knock. Who's there? Oh, you've heard it";
                break;
            case 1:
                result =
"What's green and hairy and goes up and down? A gooseberry in a lift";
                break;
            case 2:
                result =
"A horse walks into a bar and the barman asks 'Why the long face?";
                break;
        }
        return result;
    }
}
```

As you can see from above, the method is quite simple. As are the jokes. The input parameter is used to select one of three jokes and return them. If there is no match to the input the message “Invalid strength” is returned.

If we create a service like this we can use the WCF Test Client to invoke the methods and view the results.

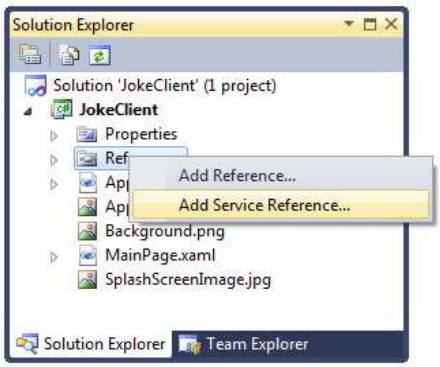


This tool lets us call methods in the service and view the results that they return. You can see the results of a call to the method with a parameter of 0. We can also view the service description in a browser:

This gives a link to the service description, as well some sample code that shows how to use it.

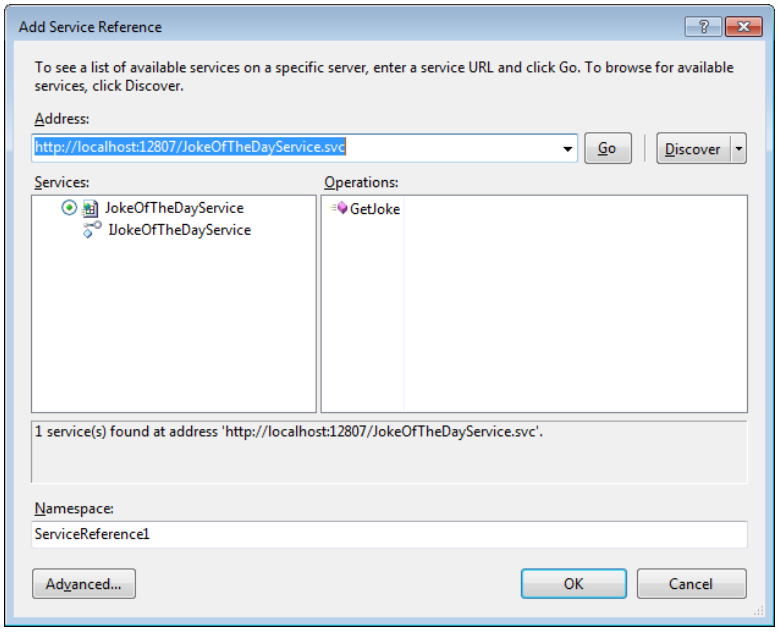
Joke of the Day Client

The client application needs to have a connection to the **`JokeOfTheDayService`**. This is added as a resource like any other, by using the Solution Explorer pane in Visual Studio:

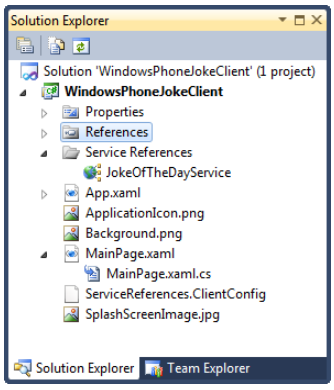


Rather than being a reference to a library assembly we instead start to create a Service reference. At this point Visual Studio needs to find the description of the service that is to be used. The Add Service Reference dialog lets us type in the network address of a service and it will then read the service description provided by the service.

It is very easy to create and test WCF services as these can be run on your Windows PC using a Development Server which is provided as part of Visual Studio. Once we have the service running and have obtained the url of the service on from the development server we just have to type this into the “Add Service Reference” dialog as shown below.



The Add Service Reference dialog will read the service description and show the operations available from that service. Above you can see that the **`JokeOfTheDay`** service only provides one method. At the bottom of this dialog you can see the namespace that we want the service to have in our Windows Phone client application. We can change this name to **`JokeOfTheDayService`**.



Once the service has been added it now takes its place in the solution. Our application must now create a proxy object that will be used to invoke the methods in the service. The best place to do this is in the constructor of the main page:

```
JokeOfTheDayService.JokeOfTheDayServiceClient jokeService;

// Constructor
public MainPage()
{
    InitializeComponent();

    jokeService = new JokeOfTheDayService.JokeOfTheDayServiceClient();

    jokeService.GetJokeCompleted +=
        new EventHandler<JokeOfTheDayService.GetJokeCompletedEventArgs>
            (jokeService_GetJokeCompleted);
}
```

This code creates a service instance and binds a method to the “completed” event that is fired when a service call returns. You might remember that when we got content from a web server the action was performed asynchronously, in that the program sent off a request to read the service and then another method was called when the data had

returned. Calls to WCF methods are exactly the same. When the program wants a new joke it must call the **GetJoke** method. This will not return with the joke, instead it will start off the process of fetching a joke. At some point in the future, when the joke arrives back from the server, a method will be called to indicate this. All the method needs to do is display the joke on the screen:

```
void jokeService_GetJokeCompleted(object sender,
    JokeOfTheDayService.GetJokeCompletedEventArgs e)
{
    if (!e.Cancelled)
    {
        jokeTextBlock.Text = e.Result;
    }
}
```

This method just checks to see if the call was successful and if it was the method takes the result from the call and puts it on the screen.

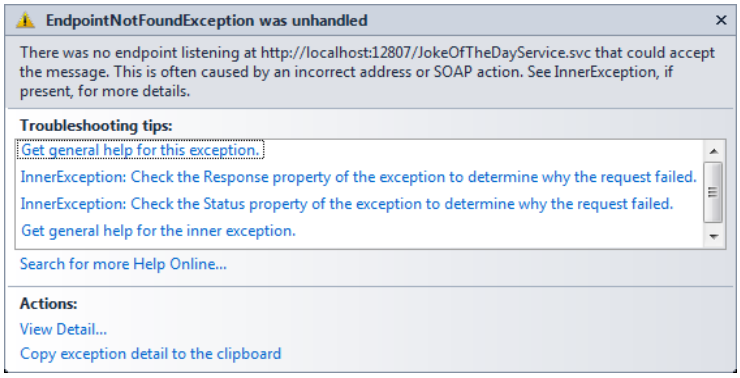
The final piece of code that we need to add is the button event handler that will actually call the WCF method to ask the server to do something:

```
private void getJokeButton_Click(object sender, RoutedEventArgs e)
{
    int strength = 0;

    if (int.TryParse(strengthTextBox.Text, out strength))
    {
        jokeService.GetJokeAsync(strength);
    }
}
```

This method gets the strength value and then uses this in a call of the **GetJokeAsync** method. Note that **TryParse** will return false if the text in **strengthTextBox** could not be converted into an integer.

Errors and TimeOuts

If the service cannot be accessed the call to **GetJokeAsync** will throw an exception:



We know that the connectivity of a Windows Phone is not guaranteed. If they do not have a network signal when they run this program then it will throw the above exception. In production code you must make sure that these exceptions never get passed through to the user. This means that you need to enclose all your attempts to use these facilities in **try** – **catch** constructions and provide appropriate feedback to the user when these events happen.

Using Network Services

For the purpose of the demonstration above I used a network server that runs on the same machine as the client. In reality you will be connecting to different systems on the internet which may be a long way away. This is not a problem. As long as Visual Studio can locate the service and download the service information it can then build the

correct proxy objects to use the service correctly. If you want to test a service you can connect to a local service initially and then re-configure the service uri later to connect to the actual service.

The solution in *Demo 08a JokeService* contains a server which provides the joke service as described above. You can run this service on your machine and use the solution in *Demo 08b WindowsPhoneJokeClient* to connect to this service and read jokes from it. When you run the program you need to make sure that the development server url assigned to the service matches that of the service in the client.

We can now create systems that use Windows Phones as clients to servers. A given Windows Phone client can connect to multiple remote services at the same time if required. This is just an introduction to the things you can do with network services. Later we will look at some of the Windows Phone controls that build on these abilities.

What We Have Learned

1. Computer networks provide a means by which one system can send a message to another. A given system is physically connected to its *local* network. Each local network also contains a *router* component which will send packets off the local network to remote destinations. The routers are connected by data transfer devices which provide *inter-network* connections.
2. The full address of a system on a network contains the address of that station on its physical network along with the address of their physical network on the internet.
3. Data can be transferred between systems on the basis of unacknowledged datagrams (UDP) or as part of a connection set up between two systems (TCP).
4. Individual services on a system can be addressed by the use of ports, some of which have “well known” address values, for example an HTTP web server will typically be located behind port 80 on a host.
5. Windows Phone applications can create and use objects that represent a connection to a network service. The use of these objects is *asynchronous*, in that the results of a request are not returned to that request, but delivered later by a further call made by the network client.
6. A socket provides an abstraction of a connection to a network.
7. Some network calls may return structured data that contains XML describing the content.
8. The LINQ (Language INtegrated Query) library provided for use on the phone can create a representation of structured data which can then be traversed to create structured data for use in a program.
9. LINQ can used be to provide an automatic transfer of data from the rows and tables in a database into objects that can be used in a program.
10. XAML allows the creation of display templates which can be bound to data elements. These templates can include lists, so that a collection of structured data can be bound to a list.
11. Windows Communication Foundation (WCF) provides a means by which servers can be set up that expose methods for clients to call.
12. A client application can read the service description provided by a WCF server and use this to create a *proxy object* that contains the methods provided by the service. When code in the client calls the method in the proxy object a network transaction is performed that transfers the parameters of the call into the server. The server method is then invoked and a further network transaction delivers the result back to the caller.