SAMPLE CHAPTER

Windows Store App Development

C# and XAML

Pete Brown

MANNING

Windows Store App Development

by Pete Brown

Chapter 18

brief contents

18 A chat app using sockets

This chapter covers

- Creating a socket server
- Connecting to a socket server
- Using TCP and UDP sockets

Higher-level networking approaches such as SOAP and RESTful services are great when you don't need to count milliseconds or when communication is primarily one way. But what about those times when you need to perform near real-time control of, say, a robot? How about synchronizing character or object movement for a game? Those are all performance-critical, often bidirectional, and sometimes peer-to-peer communications scenarios.

When apps need to communicate across a network as quickly and efficiently as possible, they use sockets. Socket communication is two-way communication between, in most cases, two endpoints. (Multicast/broadcast socket is the one-to-many or even many-to-many approach used in some other scenarios. I won't cover those approaches here because they aren't as commonly used, but I do build on the normal socket communication in this chapter.)

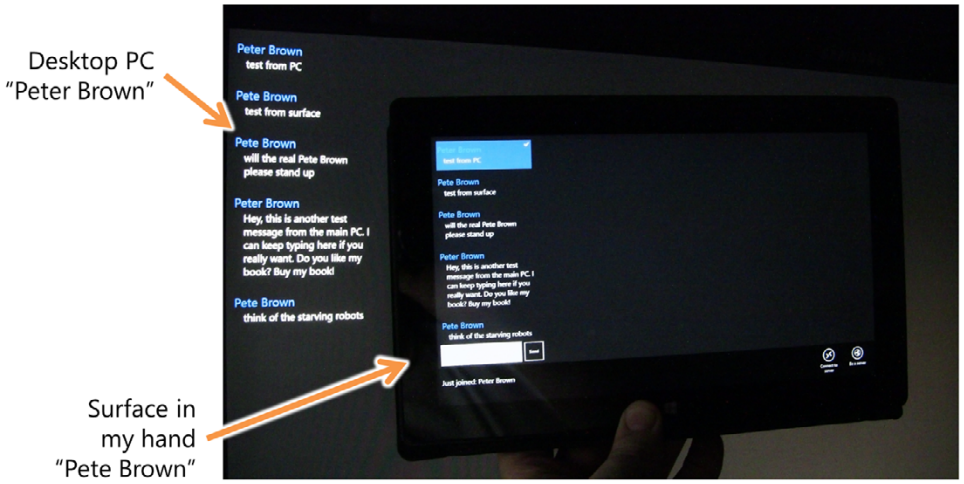

Figure 18.1 The app for this chapter running on my main PC as well as my Surface. The PC version has the same app bar as the Surface, but it's out of frame in the photo. The two machines are communicating over sockets. You'll have to trust me that this very large blob of dark print is, in fact, two separate screens running the app.

Sockets are what power the internet. Protocols such as HTTP are built on top of socket communication. Whenever you specify a port, you're specifying an endpoint for a socket, with the additional protocol built on top of it. If you want to implement your own protocol, you'll almost certainly start with socket communication.

In this chapter, you'll start building a socket-based communications app. The functionality introduced here will be simple peer-to-peer chat between two machines. Subsequent chapters will add even more game-like features that will build on the same communications and messaging infrastructure, but starting with chat makes it all easier to visualize and understand.

Figure 18.1 shows the peer-to-peer chat functionality of the app you'll build in this chapter. The app will work on all flavors of Windows 8, including Windows RT and the Microsoft Surface.

The app is both the client and the server, something that many don't realize can be done inside a Windows Store app. To make this work, you'll need to use two machines as the endpoints of the communication.

First, we'll turn to the MVVM pattern to help you structure the app. The functionality will be completely encapsulated within the viewmodel and will start with TCP streaming sockets. The UI that binds to the viewmodel will be kept simple in order to focus on learning the mechanics of socket communication. Once you get the basic chat app working, you'll refactor the code and add UDP socket support into the mix. By the end of this chapter, you'll have a simple chat app that can be used across two machines on a network. In my case, my two machines are my Windows 8 desktop PC and my Windows RT Surface.

Why two machines?

Two machines for a demo? This may seem like an attempt on my part to sell more licenses on behalf of my employer, but trust me, it's not. Really!

I was tempted to do my usual approach of using a .NET Micro Framework (NETMF) device (Netduino or Gadgeteer) but figured that would overflow the geek-o-meter for this book. You can definitely learn from the code even with a single Windows 8 machine, but sockets are point-to-point and you need two points for a conversation.

Why didn't I set up a console app on the desktop or have communication between two Windows Store apps on the same machine? You can make network calls to the loopback (127.0.0.1) if enabled in Visual Studio in the properties page for the project, but for actual deployed Windows Store apps, this is forbidden. To be very clear: Windows Store apps aren't allowed to open network connections to the same machine, even if it happens to work in Visual Studio. For that reason, I won't include examples for that scenario here.

If you'd rather go the NETMF route to, you know, control robots and blow things up, evict neighbors, and the like, I have some source code posted on my personal site as well as on the official .NET team blog.

18.1 Chat app viewmodel

When working with socket communication, a meaningful app can get complex quite quickly. So rather than tackle the entire app all at once, we'll start with the chat portion and refactor after that before adding more functionality in the next chapters.

A chat app is good for learning peer-to-peer connectivity. The message structure is simple, and the UI interaction patterns have already been explored elsewhere in this book. It'll look a little sparse at first—a good portion of the UI will be blank other than the chat column over on the left, as shown in the first figure in this chapter.

As I mentioned, this app will follow the MVVM pattern introduced earlier in this book. We'll again use Laurent Bugnion's MVVM Light toolkit in this app, because it's a huge time-saver when you want to use commanding and strongly typed property change notifications.

Figure 18.2 shows how the viewmodel fits into the picture. For this round, all the functionality is inside the viewmodel itself, because you want to focus on learning socket communication.

> **NOTE** For those of you more experienced with MVVM, sitting on the edge of your seats screaming at this book because the sockets code is implemented directly in the viewmodel, I hear you. You know I wouldn't do that to you and just leave it out there. By the end of this chapter, you'll have all that sockets code factored out into its own service class. In the meantime, I recommend getting one of those little desktop Zen gardens.

All of the UI interaction for the chat functionality uses binding to communicate with the viewmodel. In fact, because of the naming convention for the viewmodel (it

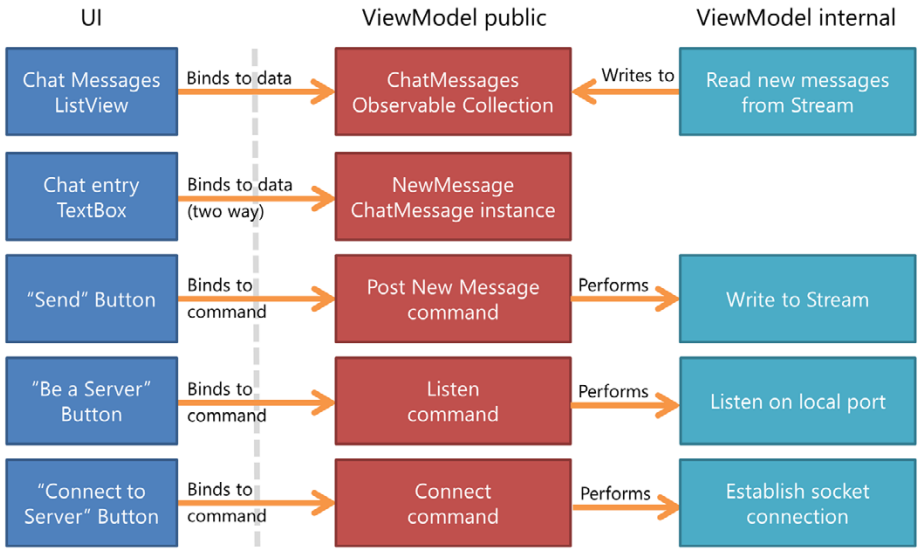

Figure 18.2 The UI is bound to the `ViewModel` using commands for buttons and also using one-way and two-way data binding for input and the messages list. Currently, all of the socket communication is also inside the `ViewModel`, but that will change before the end of this chapter.

matches the page name "MainPage" with "MainViewModel"), you're able to use the MVVM Light viewmodel locator implementation to automatically wire up the UI. The result is code-behind with no lines of code other than those from the stock template. An empty code-behind file is generally not the ultimate or most important goal, but when it comes about as the result of good app structure, you'll find you can better design the UI and better test the app functionality.

To start, create a new project named SocketApp, using the MVVM Light XAML/C# template for Windows Store apps. If you're unfamiliar with this template, please refer to chapter 9 on MVVM and controls.

The main tasks we'll look at in this section are building out the skeleton of the `MainViewModel` class and creating the `ChatMessage` model. The `MainViewModel` will have several placeholder methods, which you'll complete later in this chapter.

18.1.1 The MainViewModel class

The `MainViewModel` class is what the UI uses to communicate with the rest of the app. It's where the command instances are located and where the bindable properties are surfaced to the UI. Open up the `MainViewModel` class source file in the ViewModel folder and replace its contents with what's shown in this listing.

Listing 18.1 The skeleton `MainViewModel`

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using GalaSoft.MvvmLight.Threading;
```

```
using SocketApp.Model;
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;

namespace SocketApp.ViewModel
{
  public class MainViewModel : ViewModelBase
  {
    private const string PortOrService = "5150";

    private StreamSocket _socket;
    private StreamSocketListener _listener;

    public MainViewModel()
    {
      ChatMessages = new ObservableCollection<ChatMessage>();
      ConnectionStatus = "Not connected.";
      ServerAddress = "pete-surface64";

      CreateNewMessage();

      PostNewMessageCommand = new RelayCommand(
        () => PostNewMessage(), () => CanPostNewMessage());

      ListenCommand = new RelayCommand(
        () => Listen(), () => CanListen());

      ConnectCommand = new RelayCommand(
        () => Connect(),() => CanConnect());
    }

    public ObservableCollection<ChatMessage> ChatMessages { get; set; }

    private ChatMessage _newMessage;
    public ChatMessage NewMessage
    {
      get { return _newMessage; }
      set { Set<ChatMessage>(() => NewMessage, ref _newMessage, value); }
    }

    public RelayCommand PostNewMessageCommand { get; private set; }

    private void CreateNewMessage()
    {
      if (NewMessage != null)
        NewMessage.PropertyChanged -= NewMessage_PropertyChanged;
```

Socket server port number

Input/output socket

Socket listener

Server name (change this)

Chat messages

Message entry

Create empty message

```
      NewMessage = new ChatMessage();
      NewMessage.PropertyChanged += NewMessage_PropertyChanged;
    }

    void NewMessage_PropertyChanged(object sender,
         PropertyChangedEventArgs e)
    {
      if (e.PropertyName == "Message")
        PostNewMessageCommand.RaiseCanExecuteChanged();
    }

    public async void PostNewMessage() { }
    public bool CanPostNewMessage() { return true; }

    private string _serverAddress;
    public string ServerAddress
    {
      get { return _serverAddress; }
      set { Set<string>(() => ServerAddress, ref _serverAddress, value); }
    }

    private string _connectionStatus;
    public string ConnectionStatus
    {
      get { return _connectionStatus; }
      set { Set<string>(() => ConnectionStatus,
                ref _connectionStatus, value); }
    }

    public RelayCommand ConnectCommand { get; private set; }
    public async void Connect() { }
    public bool CanConnect() { return true; }

    public RelayCommand ListenCommand { get; private set; }
    public async void Listen() { }
    public bool CanListen() { return true; }
  }
}
```

Post chat message

Connect to server

Be a server

This single viewmodel has all the endpoints required for wiring up the UI. Several of the methods are placeholders (such as the code to be a server or connect to one) and will be implemented throughout this section.

The viewmodel has a hardcoded server name. In my case, the server is my Microsoft Surface. Replace that with the machine name or IP address of the machine you intend to connect to. The port number is also up to you, as long as you pick something that's out of the restricted range of well-known ports. If you have firewall issues on your network, you'll find that changing it to port 80 will work, as long as the server machine isn't running a web server of any sort.

The message list and message entry functionality works just as you've seen in the previous chat example in this book (chapter 9 on MVVM and controls), so I won't go into detail on that pattern here.

18.1.2 ChatMessage model class

The viewmodel includes a couple references to the `ChatMessage` class, using the following listing as the body of a class named `ChatMessage` in the Model folder.

Listing 18.2 The `ChatMessage` model class

```
using GalaSoft.MvvmLight;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SocketApp.Model
{
  public class ChatMessage : ObservableObject
  {
    private string _message;
    public string Message                                  <-- Message text
    {
      get { return _message; }
      set { Set<string>(() => Message, ref _message, value); }
    }
  }
}
```

The `ChatMessage` class contains a single property in this initial version: the text of the message. Using only this single property will help you keep the sockets messaging format simple and understandable for this first version.

The viewmodel and model are two of the most importance pieces for this app. The viewmodel, in particular, is where all the sockets action will happen. Now that you have everything that the UI binds to, it's time to create the UI itself.

18.2 The user interface

The app you create in this chapter will be the chat portion of a larger peer-to-peer app. Because you want as much space as possible for app content, there will be no title portion. In addition, because there's only a single page in this app, there's no need for a navigation button at the top left.

The UI will make use of data binding to the viewmodel both for the list of chat messages as well as for the entry of the new chat message. It will also use data binding to update the connection status in the app bar and command binding for the three buttons.

Figure 18.3 shows what the UI will look like once you've completed the app.

In this section, you'll build out the XAML user interface for this app. You'll start with a simple skeleton and then add in the styles and resources used for the buttons and text. From there, you'll create an app bar with two buttons and a couple of `TextBlock` elements. The app bar will be sticky and visible, so you don't have to manually

Figure 18.3 A cropped view of the UI, showing the main elements plus the app bar

show it. Next, you'll create the `ListView` for the chat messages and the controls that allow for message entry. We'll wrap up this section with a very sparse set of visual states for the different page view states.

18.2.1 XAML skeleton

This app has only a single page: MainPage.xaml. You're not going to do anything specific to support portrait and snapped views in this version, but snapped view will "just work." Crack open MainPage.xaml and replace its contents with what you see in the next listing. This will serve as the starting structure for the interface.

Listing 18.3 MainPage.xaml skeleton

```
<common:LayoutAwarePage x:Class="SocketApp.MainPage"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
  xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
  xmlns:common="using:SocketApp.Common"
  xmlns:ignore="http://www.ignore.com"
  mc:Ignorable="d ignore"
  d:DesignHeight="768"
  d:DesignWidth="1366"
  DataContext="{Binding Main, Source={StaticResource Locator}}">

  <!-- Styles and resources go here -->

  <!-- App Bar goes here -->
</common:LayoutAwarePage>
```

ViewModelLocator

Styles and resources

App bar

```
  <Grid Background="{StaticResource ApplicationPageBackgroundThemeBrush}">

    <!-- Content goes here -->                    <-- Content

    <!-- Visual States Go Here -->                <-- Visual states
  </Grid>
</common:LayoutAwarePage>
```

For this chapter, I make use of the `ViewModelLocator` provided in MVVM Light. You can see its reference as the data context for this page.

The comments in this listing are placeholders for content you'll add in the next several listings.

18.2.2 Styles and resources

The page includes styles for the buttons as well as colors that will be used for text and other elements. These are all included in the resources section of the page.

The first of those is the styles and resources local to this page. The following listing contains the XAML to place at that spot.

Listing 18.4 MainPage.xaml styles and resources

```
<Page.Resources>
  <ResourceDictionary>
    <ResourceDictionary.MergedDictionaries>
      <ResourceDictionary Source="Skins/MainSkin.xaml" />
    </ResourceDictionary.MergedDictionaries>

    <SolidColorBrush x:Key="AccentBrush" Color="#FF220088" />
    <SolidColorBrush x:Key="HighlightBrush" Color="#FF5500FF" />

    <Style x:Key="WorldAppBarButtonStyle" TargetType="ButtonBase"
           BasedOn="{StaticResource AppBarButtonStyle}">
      <Setter Property="AutomationProperties.AutomationId"
              Value="WorldAppBarButton" />
      <Setter Property="AutomationProperties.Name"
              Value="World" />
      <Setter Property="Content"
              Value="&#xE128;" />
    </Style>

    <Style x:Key="RemoteAppBarButtonStyle" TargetType="ButtonBase"
           BasedOn="{StaticResource AppBarButtonStyle}">
      <Setter Property="AutomationProperties.AutomationId"
              Value="RemoteAppBarButton" />
      <Setter Property="AutomationProperties.Name" Value="Remote" />
      <Setter Property="Content" Value="&#xE148;" />
    </Style>
  </ResourceDictionary>
</Page.Resources>
```

Server app bar button

Connect app bar button

The styles used in the resources section of the page are for the buttons. The two app bar button styles are copied directly from the commented-out, template-provided styles in app.xaml. The colors will be used later in this chapter and in the next.

Figure 18.4 The app bar buttons for the chat app

18.2.3 App bar buttons

This app contains only two app bar buttons, shown in figure 18.4. The first is used when you want to connect to the app running as a server on another machine. The second is used when you want the machine itself to be the server. On any given machine, you'll choose only one of these options, not both. But you don't do any validation of that or enable/disable the buttons (via the commands). That's something you could easily add in if you'd like—it would be handled 100% in the viewmodel.

The app bar also includes some `TextBlock` elements on the left, one of which you'll use to display the connection status.

The app bar buttons, the status text, and the app bar that contains them are all shown in the following listing. Place this markup in the page section reserved for the app bar.

Listing 18.5 MainPage.xaml app bar

```
<Page.BottomAppBar>
  <AppBar IsSticky="True" IsOpen="True">
    <Grid>
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>

      <StackPanel Orientation="Vertical" Grid.Column="0"
                  HorizontalAlignment="Left"
                  VerticalAlignment="Center">
        <TextBlock x:Name="ConnectionStatus" FontSize="20"
                   TextWrapping="Wrap"
                   Text="{Binding ConnectionStatus}" />
        <TextBlock x:Name="IPAddressDisplay" />
      </StackPanel>

      <StackPanel Orientation="Horizontal"
                  HorizontalAlignment="Right"
                  Grid.Column="1">
        <Button x:Name="ConnectToServer"
                Command="{Binding ConnectCommand}"
                Style="{StaticResource RemoteAppBarButtonStyle}"
                AutomationProperties.Name="Connect to server" />
        <Button x:Name="BeAServer"
                Command="{Binding ListenCommand}"
                Style="{StaticResource WorldAppBarButtonStyle}"
                AutomationProperties.Name="Be a server"/>
```

Connection status

Buttons on right

Connect app bar button

Server app bar button

```
      </StackPanel>
    </Grid>
  </AppBar>
</Page.BottomAppBar>
```

Each of the buttons uses command binding to communicate with the viewmodel. Note also that the styles used are the ones you added to the resources section earlier.

18.2.4 Chat app content

Finally, we get to the action part of the app: the chat messages list and entry UI. For the chat app, the content consists of the `ListView` containing the chat messages, a `TextBox` to type the message, and a button to send the message. The markup to place in the "content" placeholder on the page is shown here.

Listing 18.6 MainPage.xaml content

```
<Grid Margin="0,0,0,110">
  <Grid.ColumnDefinitions>
    <ColumnDefinition Width="320" />
    <ColumnDefinition Width="*" />
    <ColumnDefinition Width="Auto" />
  </Grid.ColumnDefinitions>

  <Grid Grid.Column="0"
        Margin="10,10,10,0">
    <Grid.RowDefinitions>
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
    </Grid.RowDefinitions>

    <ListView Grid.Row="0" Margin="0,0,0,10"
              ItemsSource="{Binding ChatMessages}">
      <ListView.ItemTemplate>
        <DataTemplate>
          <Grid Margin="0,5,0,5">
            <TextBlock FontSize="20" TextWrapping="Wrap"
                       Text="{Binding Message}" />
          </Grid>
        </DataTemplate>
      </ListView.ItemTemplate>
    </ListView>

    <Grid Grid.Row="1">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="*" />
        <ColumnDefinition Width="Auto" />
      </Grid.ColumnDefinitions>

      <TextBox x:Name="ChatEntry" MaxLength="512"
               Text="{Binding NewMessage.Message, Mode=TwoWay}"
               TextWrapping="Wrap" FontSize="14"
               Grid.Column="0" Height="60"
               HorizontalAlignment="Stretch" />
```

List of chat messages

Chat entry

Send message

```
        <Button Content="Send" Command="{Binding PostNewMessageCommand}"
                Margin="5,0,0,0" FontSize="12"
                VerticalAlignment="Stretch"
                Grid.Column="1" />

    </Grid>
  </Grid>
</Grid>
```

This listing shows how binding is used for the button commands. One interesting note here: The `MainViewModel` code for the `CanPostNewMessage` function always returns true. This is because there's currently no easy way to update the command for each letter typed in the `TextBox`. In other XAML-based UI, you'd set the `UpdateSourceTrigger` to `PropertyChanged`, but that's not yet available in WinRT XAML. You may have seen this in chapter 9 where you had to tab off the chat field in order to enable the button (or click the button twice).

The final bit of markup is for the visual states. As I mentioned earlier, you're not doing anything special to handle the different orientations and states, but the UI is simple enough that it works as is. The following listing has the XAML to place in the visual states section.

Listing 18.7 MainPage.xaml Visual States

```
<VisualStateManager.VisualStateGroups>
  <VisualStateGroup x:Name="ApplicationViewStates">
    <VisualState x:Name="FullScreenLandscape" />
    <VisualState x:Name="Filled" />
    <VisualState x:Name="FullScreenPortrait" />
    <VisualState x:Name="Snapped" />
  </VisualStateGroup>
</VisualStateManager.VisualStateGroups>
```

View states

The UI is in place, and most of the rest of the structure of the app is there. But the `MainViewModel` is full of placeholders for things like connecting to a server and listening for new connections.

Note that one thing you didn't have to do is open up the code-behind. Everything in this app (for at least the chat functionality) will be handled via binding to view-model data items and commands.

Nothing in the UI should be surprising to you, because it's all stuff we've covered in previous chapters, simply reapplied here. Binding of data and commands is an especially important concept to learn. One of the commands will be to enable the app as a socket server, so let's cover that next.

18.3 Listening for connections

Sockets require that at least one endpoint be set up to listen for new connections ahead of time. In a client/server-based solution, like the browser and a web server, the server is what listens. Listening simply establishes the initial communication; after a

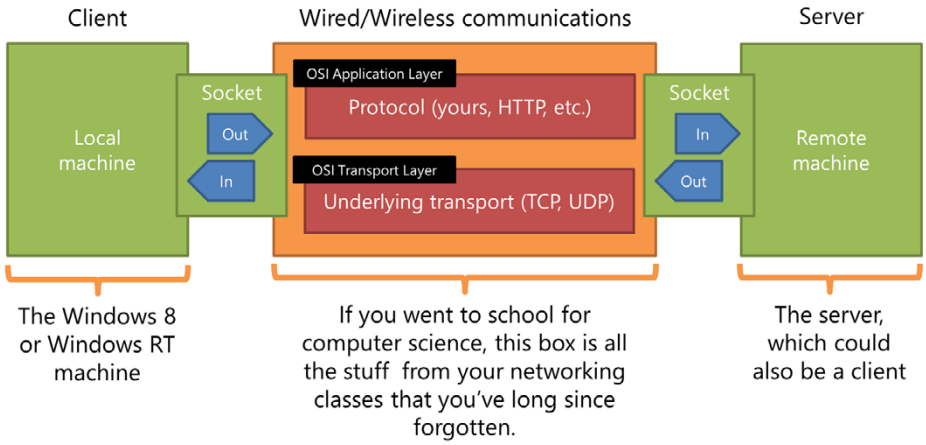

Figure 18.5 Sockets are the inputs and outputs for each machine. Common protocols like HTTP use sockets underneath. When using socket communication directly, you control the protocol.

connection is established between two endpoints, both sides participate equally in the conversation. It's this listening action that makes something a server.

Figure 18.5 shows how sockets and the endpoints fit into the communications.

Sockets are identified by a service name or, more commonly, a port number. Taking a web server, for example, the server-side socket is typically port 80 (or 443 for secure sockets). The local socket is almost always dynamically allocated from the higher range. So, to make your app a server, you need to listen on the specific socket number, and clients need to know that's the socket number to connect to.

Because the communication between the two endpoints is simply the sockets with whatever protocol or messaging you layer on top of it, the endpoints don't need to be built from the same source code. In fact, in the case of a web server, you have two completely different pieces of software with the browser on the client and the web server on the server. I've also used sockets to control a .NET Micro Framework robot from a Windows 8 app; again, completely different software. For this app, we'll make the app support both the client and server roles for a truly peer-to-peer experience. For any given instance of the app on a machine, only one role is active at a time.

Commercial chat apps typically involve a completely separate server to enable tracking who are online or offline at any given time and what their addresses are. Clients connect to that server initially and may then have direct peer-to-peer connections afterward or simply route everything through the server.

In our app, the chat function will use TCP sockets to enable communication between two Windows 8 PCs but in a purely peer-to-peer way: One PC will establish itself as the server and the other will be able to connect to it. Figure 18.6 shows the flow of the connection.

For simplicity in this example, the IP address (or host name) for the server app will be hardcoded. You'll change that approach later in this chapter when you build out

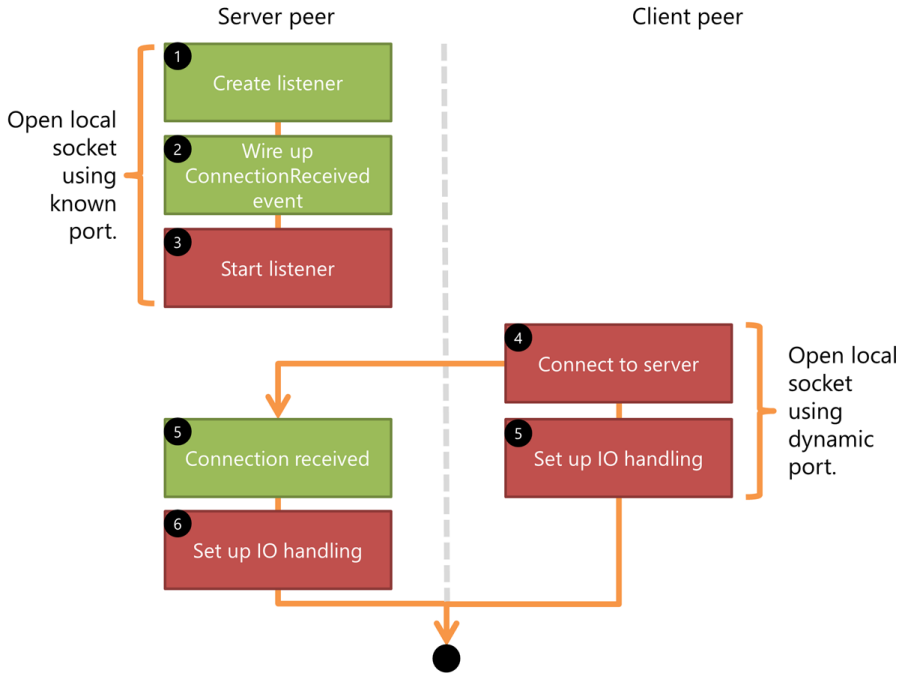

Figure 18.6 Chat app connection flow. Step 5 happens logically simultaneously on each machine. Once the connection is established, the peers are on equal footing. The remote port on the server is a known port number. In most cases, you don't control the local port number from code. You can, but typically you'll let the system dynamically allocate that. This is how browser connections work, for example.

the larger app. It's also important to note that once the connection is established, the peers are truly peers—the listener is no longer required because the communication is handled by reading from and writing to streams. In fact, as you'll see shortly, the IO handling code on each machine is identical.

The `MainViewModel` code to listen for a connection is shown here.

Listing 18.8 `MainViewModel` additions for the socket server

```
public async void Listen()
{
  _listener = new StreamSocketListener();
  _listener.ConnectionReceived += OnConnectionReceived;

  await _listener.BindServiceNameAsync(PortOrService);

  var hostNames = NetworkInformation.GetHostNames();

  ConnectionStatus = "Waiting for connection on: ";

  int i = 0;
```

Create listener

Wire up connection handler

Listen on port

```
    foreach (HostName name in hostNames)
    {
      if (i > 0)
        ConnectionStatus += " and ";

      ConnectionStatus += name.DisplayName;
      i++;
    }
  }

  void OnConnectionReceived(StreamSocketListener sender,
              StreamSocketListenerConnectionReceivedEventArgs args)
  {
    DispatcherHelper.CheckBeginInvokeOnUI(() =>
    {
      ConnectionStatus = "Connection received from " +
        args.Socket.Information.RemoteHostName.DisplayName;

      _socket = args.Socket;

      SetUpInputHandling();
    });
  }
```

Display names for this server

Execute on UI thread

Display connection source

Cache open socket

Set up input handling

This code establishes the running app as a socket server. It first creates a `StreamSocketListener` instance. This class, through the `BindServiceNameAsync` function, is what handles opening up a port and routing communication from that port to your app. When a connection is received on that port, the `ConnectionReceived` event fires.

The code in this method also uses the `DispatcherHelper` class, provided as part of MVVM Light. The `CheckBeginInvokeOnUI` will dispatch the function to the UI if and only if it isn't already running on the UI thread (or, more correctly, has access to the UI thread). If the code is running on the UI thread already, it'll simply execute the code. This method is helpful because the connection received event doesn't fire on the UI thread, but the code (setting the `ConnectionStatus`, for example) requires access to that thread.

Once inside the `ConnectionReceived` handler, all you really need is the socket. The socket passed in is what provides your read/write communications pathway with the remote machine. The code then calls `SetUpInputHandling`, the body of which is shown next.

Listing 18.9 Handing incoming messages in the `MainViewModel` class

```
private DataWriter _writer;
private DataReader _reader;

private void SetUpInputHandling()
{
  _writer = new DataWriter(_socket.OutputStream);
  _reader = new DataReader(_socket.InputStream);

  var t = Task.Factory.StartNew(async () =>
  {
```

Create output writer

Create input reader

Start background thread

```
        _reader.InputStreamOptions = InputStreamOptions.Partial;

        while (true)
        {
          var count = await _reader.LoadAsync(512);

          var message = _reader.ReadString(count);

          DispatcherHelper.CheckBeginInvokeOnUI(() =>
          {
            var chatMessage = new ChatMessage();
            chatMessage.Message = message;

            ChatMessages.Add(chatMessage);
          });
        }
      });
    }
```

Return with partial read

Read up to 512 characters

Read string

Add message on UI thread

The `SetUpInputHandling` function first creates the reader and writer for the streams. The `OutputStream` property is where you write data to send it to the other machine. The `InputStream` is where messages received on this machine show up.

The function is named the way it is because its primary responsibility is to spin up a background thread that monitors in the incoming stream of data.

The background thread, started using the `StartNew` method of the task factory, is an endless loop that waits on the `LoadAsync` method of the `DataReader` class. The `DataReader` (and `DataWriter`) class isn't required for handling IO on the stream, but it certainly makes the task a lot easier by providing high-level methods that understand how to read fundamental types like `int`, `string`, `double` and more.

Finally, in the appx manifest you'll see the Internet Client capability set by default (this was discussed in the previous chapter). For this solution to work, you'll need to also add the Internet (Client and Server) capability. For this app to work on a local network, across machines, you'll need to set the Private Networks (Client & Server)

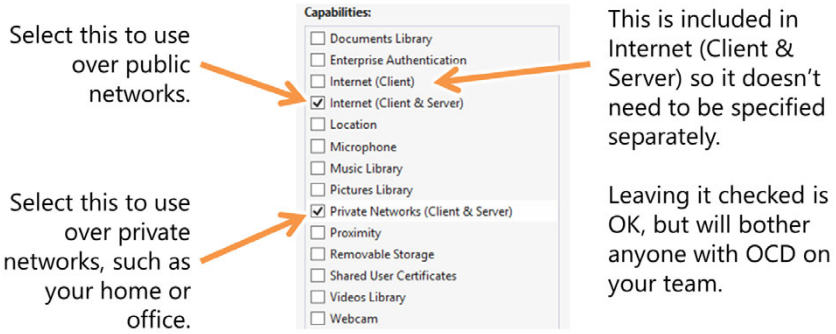

Figure 18.7 To use this app, you'll need to request server permissions, not just client permissions. Additionally, if this app is used on a private network (this is a domain network or any network with sharing turned on), the Private Networks capability must be selected. Removing the Internet (Client) capability is optional.

capability. Although not necessary, I recommend unchecking the Internet Client capability. Figure 18.7 shows the set of capabilities to request for this app.

With these changes made to the manifest, the server peer is now set up and waiting for connections from the client peer.

18.4 Connecting to the server and sending data

Connecting to an existing server is much simpler than waiting for a connection from a client. Or, at least it's quite a bit simpler in WinRT than it is in Silverlight. I recall the client sockets code for connecting in Silverlight was pages long due primarily to the callback-based async approach. I'm happy this model was greatly simplified for Windows Store apps, because connecting via sockets is an essential task for so many games and communications apps.

Connecting to an existing server is an easy task, primarily handled with a single line of code. Figure 18.8 shows the basics of the workflow, including the additional steps to send data to the server peer.

In this section we'll look at how to open a socket connection to another machine. Once the connection is established, you'll learn how to use the `DataWriter` to write to the socket's input stream.

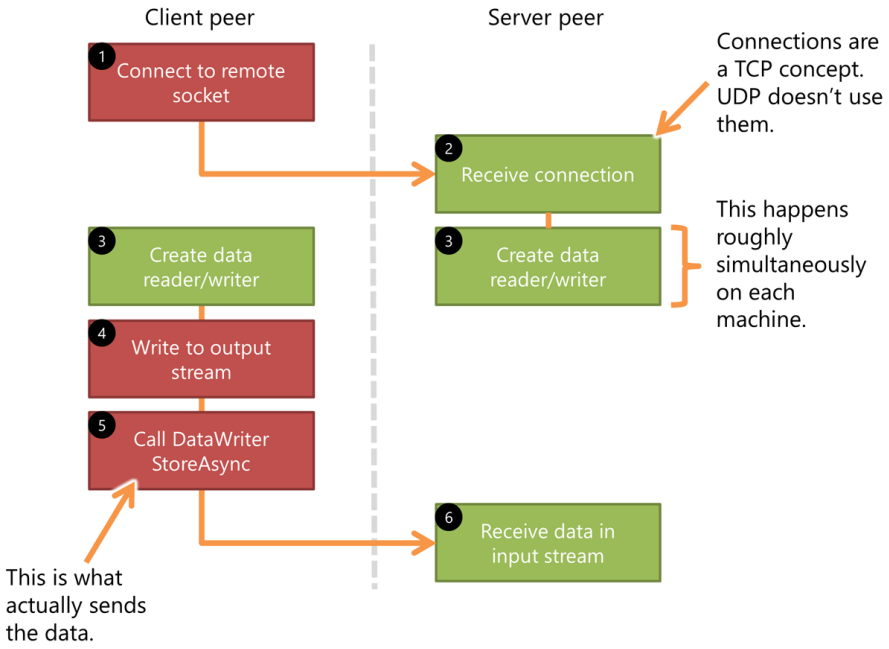

Figure 18.8 Data sending workflow. First the client connects to the server, and then (in the case of TCP sockets) both sides create the read/write data streams. The client then writes to the output stream, but the message isn't sent until you call the `StoreAsync` method of the `DataWriter`.

18.4.1 Connecting to an endpoint

In socket communication, the two sides of the conversation are equals. Although it may be convenient to refer to one as the client and the other as the server, they are technically just endpoints. That said, someone has to be listening at first, and someone else has to connect to them. They aren't equals until that connection is established. Think of it like the telephone: Someone had to call, and someone had to be listening for the phone to ring, but once the chatting starts, it doesn't matter who called whom (well, unless the call is an argument about how "you never call." No, I'm not bitter.)

The next listing shows the connect code to place in the `MainViewModel` class.

Listing 18.10 `MainViewModel` code to connect to an existing server

```
public async void Connect()
{
  var hostName = new HostName(ServerAddress);

  _socket = new StreamSocket();

  await _socket.ConnectAsync(hostName, PortOrService);

  SetUpInputHandling();

  ConnectionStatus = "Connected to server at " + ServerAddress + ".";
}
```

Create socket

Connect to server

Set up input handling

This code first creates a `HostName` instance using the `ServerAddress`. `HostName` is a flexible class that can use an IP address, local name, or any other DNS-recognized endpoint name.

Once the `HostName` is created, the code creates a socket. This differs from the server code in that the socket isn't provided to the code in an event handler here; instead you explicitly create the socket and then call `ConnectAsync` passing in the host name and the port number.

After creating the connection, the code calls the same `SetUpInputHandling` code that the server peer code calls. At this point, the two machines are equals and will communicate solely via the streams associated with the sockets.

18.4.2 Sending data

A chat app must be able to send new messages. This is accomplished by writing the message data to the stream. How you structure your message at this point is critical, because you need to know how to parse it on the receiving end, where the data is just raw bytes.

For our first example, we'll keep it simple and post only the string message, as shown in the following listing.

Listing 18.11 `MainViewModel` code to post a new message

```
public async void PostNewMessage()
{
  ChatMessages.Add(NewMessage);

  _writer.WriteString(NewMessage.Message);

  await _writer.StoreAsync();

  CreateNewMessage();
}
```

Add local message

Add message text to stream

Send the message

This code first adds the message to the local collection. This is so you can see your own messages on the message timeline. It then writes the message string to the socket using the `DataWriter` created in the `SetUpInputHandling` method. Writing to the stream isn't enough to send the message, however. To do that, you need to call the `StoreAsync` method of the `DataWriter`. That naming may seem a little odd, but the `DataWriter` isn't something specific to sockets—it could work with file streams where the naming makes a bit more sense.

Finally, the `CreateNewMessage` method creates a new message for the UI to bind to by "newing" one up and assigning `NewMessage` to that new instance.

Run the app on the two different machines. On the server machine, hit the "be a server" button. On the client machine, hit the button to connect to the server. Now, enter a message in either chat `TextBox`. You should see the message echoed locally and (quite quickly) reflected on the screen on the other machine. You've just implemented something that can be the basis for most any multiuser network app or multiplayer game. Sure, the networking graph gets more complex when you add additional players, but the basics of communication are the same.

The code does look a bit sloppy in the `MainViewModel`, though. It's not my style to shove everything in there, but it does make it simpler to learn. Now that you understand how the code works, let's clean it up. I simply couldn't live with myself if we didn't.

18.5 Refactoring for better structure and flexibility

So that you could focus on learning how to use sockets and not worry about architecture, all of the socket communication in the app is currently inside the `MainViewModel`. This made for the least amount of mental overhead when learning the relatively complex topic of socket communications.

As you know from previous chapters, I prefer to factor that type of "guts" code out into separate services classes. This adds a little bit of complexity to the app architecture but provides for a nice clean structure and the ability to expand our approach to include other communications mechanisms. In our case, it's going to provide the ability to support an additional type of communications transport layer, as well as additional messages we'll use in this chapter and the next.

In this new architecture, the viewmodel's public interface will remain the same, and so the UI will remain the same as well (with just one addition). The socket com-

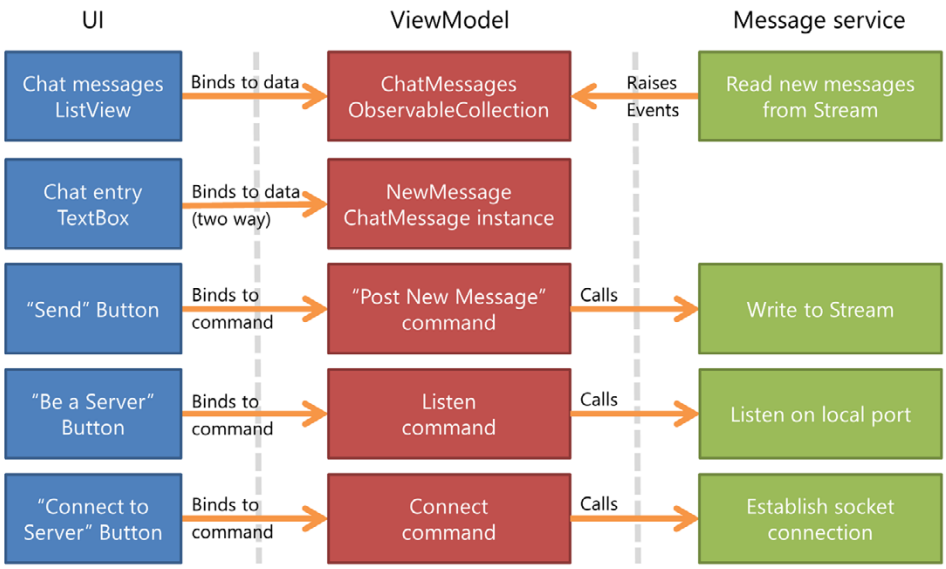

Figure 18.9 The refactored app architecture. Except for the addition of player information we'll cover shortly, the public viewmodel interface remains the same, so no changes are required at the UI level.

munication will be factored out into a separate service class. Figure 18.9 shows the changed architecture.

At this level, the architecture looks almost identical to our original. The real difference is the addition of a new class that contains all the sockets code originally in the viewmodel.

In order to support using other socket communications types, such as UDP, you'll also extract an interface that's common across any communications mechanism. Finally, you'll add a few little details that weren't in the initial version, such as providing the Windows username as part of the message information. This will require refactoring and updating the model objects and the viewmodel. To support displaying the name, you'll also make a small update to the main page XAML.

18.5.1 The updated ChatMessage class

The previous version of the `ChatMessage` class had only a single property: the message text. You now want to support the name of the person who sent the message. The name (and potentially other properties in the future) is encapsulated in the `Player` class and surfaced through the `Player` property of the `ChatMessage` class, as shown in the following listing.

Listing 18.12 The updated `ChatMessage` class with new `Player` property

```
using GalaSoft.MvvmLight;
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SocketApp.Model
{
  public class ChatMessage : ObservableObject
  {
    private Player _player;
    public Player Player                                    <-- New Player property
    {
      get { return _player; }
      set { Set<Player>(() => Player, ref _player, value); }
    }

    private string _message;
    public string Message
    {
      get { return _message; }
      set { Set<string>(() => Message, ref _message, value); }
    }
  }
}
```

The `Player` class is, like the `ChatMessage` class, a class in the Model folder. Unlike `ChatMessage`, `Player` doesn't inherit from `ObservableObject` because you don't expect to make changes to the object after it is created. The next listing has the new class source.

Listing 18.13 The new `Player` class in the Model folder

```
using System;
using System.Linq;

namespace SocketApp.Model
{
  public class Player
  {
    public string Name { get; set; }                        <-- Player name
  }
}
```

The `ChatMessage` class now exposes a `Player` property, which itself is a `Player` instance with a `Name` property. In order to make use of this new property, you'll need to make a small update to the `DataTemplate` in the `ListView` on MainPage.xaml. Replace the entire `ListView` with the markup from this listing.

Listing 18.14 The updated chat message `ListView`

```
<ListView Grid.Row="0" Margin="0,0,0,10"
          ItemsSource="{Binding ChatMessages}">
  <ListView.ItemTemplate>
    <DataTemplate>
      <Grid Margin="0,5,0,5">
```

```
        <Grid.RowDefinitions>
          <RowDefinition Height="Auto" />
          <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>

        <TextBlock FontSize="24" Grid.Row="0" Margin="2"
                   Foreground="{StaticResource HighlightBrush}"
                   Text="{Binding Player.Name}"
                   TextWrapping="Wrap" />
        <TextBlock FontSize="20" Grid.Row="1" Margin="20,0,0,10"
                   Text="{Binding Message}" TextWrapping="Wrap" />
      </Grid>
    </DataTemplate>
  </ListView.ItemTemplate>
</ListView>
```

Two rows in the grid

Player name in top row

With that, the UI and model object changes are complete. The next step is to create the interface for the messaging service.

18.5.2 The IMessageService interface

One of the goals of this refactor is to make it easier to swap out different networking implementations. A classic approach for this is to define a common interface and use only that interface from the code. Although you won't use it here, using an interface also opens up the ability to use Dependency Injection (DI) and locator patterns to automatically wire up concrete types. You'll take a simpler approach and create the concrete type in the constructor of the viewmodel.

First, add a new folder named Services in the root of your project. This folder will contain the interface as well as the concrete classes that implement it. As was the case in previous chapters, *Services* in this context means a class that provides functionality to other classes in the app.

Next, add a new interface named `IMessageService`. You could simply create a class and then replace the contents, or you can use the Interface project template shown in figure 18.10.

The `IMessageService` interface exposes the functions that will be used by the viewmodel, specifically, connection and disconnection, listening for new connections,

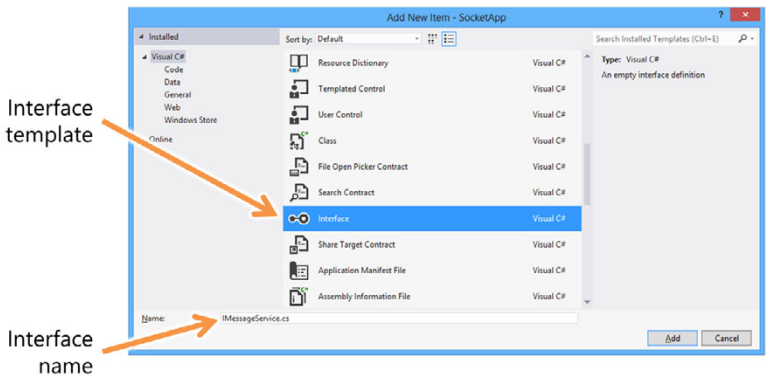

Figure 18.10 Into the Services folder, add a new interface named `IMessageService`.

and sending a message, and then events for the various communications from the service back to the viewmodel. The following listing has the interface source.

Listing 18.15 The `IMessageService` interface

```
using SocketApp.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using Windows.Networking;

namespace SocketApp.Services
{

  // this space intentionally left blank

  public interface IMessageService
  {
    void Connect(Player me, string remoteHostName);
    void Disconnect();
    void Listen(Player me);

    void SendChatMessage(ChatMessage message);

    IReadOnlyList<HostName> GetHostNames();

    event EventHandler<ChatMessageReceivedEventArgs>
                                          ChatMessageReceived;
    event EventHandler<ConnectionReceivedEventArgs> ConnectionReceived;
    event EventHandler<PlayerJoinedEventArgs> PlayerJoined;
    event EventHandler<PlayerExitedEventArgs> PlayerExited;
  }
}
```

Supporting types will go here

Open or close connection

Events

In addition to the expected connection management functions, there are several new event handlers. Two of the event handlers are there to tell you when players join and when they exit. For this chapter, you'll only work with player joining, because player exiting will also require handling things like the app going into suspension—a topic for another chapter.

The event handlers require a number of supporting types, as shown in the following listing. Place this source code in the same file in the spot designated in listing 18.15.

Listing 18.16 Supporting types for the `IMessageService` interface

```
public enum WireMessageType
{
  ChatMessage,
  PlayerJoin,
  PlayerLeave
}

public class ChatMessageReceivedEventArgs : EventArgs
{
  public ChatMessage Message { get; private set; }
```

Message type

```
  public ChatMessageReceivedEventArgs(ChatMessage message)
  {
    Message = message;
  }
}

public class ConnectionReceivedEventArgs : EventArgs
{
  public Player Player { get; private set; }
  public HostName HostName { get; private set; }

  public ConnectionReceivedEventArgs(Player player, HostName hostName)
  {
    Player = player;
    HostName = hostName;
  }
}

public class PlayerJoinedEventArgs : EventArgs                    <-- Player joined
{
  public Player Player { get; private set; }

  public PlayerJoinedEventArgs(Player player)
  {
    Player = player;
  }
}

public class PlayerExitedEventArgs : EventArgs                    <-- Player exited
{
  public Player Player { get; private set; }

  public PlayerExitedEventArgs(Player player)
  {
    Player = player;
  }
}
```

The `WireMessageType` enum is the interesting part of this listing. It's used as the first element of the message going over the wire (or over the air) to tell the code how the rest of the bytes are to be processed. In this app, you'll support three types of messages, the formats of which are shown in table 18.1.

Table 18.1 The three message types supported in the chat app

Type	Description	Format and byte positions
`ChatMessage`	A free-form chat message sent user to user	0-3: Int32 : `WireMessageType ChatMessage` 4-7: Int32 : String (not byte) length of chat message 8-?: string : The chat message
`PlayerJoin`	Notification that a player has joined the conversation	0-3: Int32 : `WireMessageType PlayerJoin` 4-7: Int32 : String (not byte) length of player name 8-?: string : The player's name
`PlayerLeave`	Notification that a player has left the conversation (not used in this chapter)	0-3: Int32 : `WireMessageType PlayerLeave`

Establishing a solid and flexible messaging pattern for your apps is an important step to implementing communications. In the earlier version in this chapter, I sent only a single string, the chat message, because that was really easy to do. You can see from this table that adding additional message types or additional fields for existing message types requires some real thought.

Now that you understand the message structure, let's implement the class that makes it all happen: the `TcpStreamMessageService` class.

18.5.3 The TcpStreamMessageService class

The `TcpStreamMessageService` class is the meat of communications infrastructure in this app. It is to this class that you refactored most of the functionality that was previously in the `MainViewModel`. In addition, because of the new message types and the tracking of player identity, this version is more complex than what was in the `MainViewModel` previously.

Start with creating a new class named `TcpStreamMessageService` in the Services folder. Replace the contents of that file with the following code.

Listing 18.17 Overall skeleton of the `TcpStreamMessageService` class

```
using GalaSoft.MvvmLight.Threading;
using SocketApp.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;

namespace SocketApp.Services
{
  public class TcpStreamMessageService : IMessageService
  {
    private class PlayerConnection
    {
      public Player LocalPlayer { get; set; }
      public Player RemotePlayer { get; set; }
      public DataWriter Writer { get; set; }
      public DataReader Reader { get; set; }
      public StreamSocket Socket { get; set; }
    }

    public event EventHandler<ChatMessageReceivedEventArgs>
                                        ChatMessageReceived;
    public event EventHandler<ConnectionReceivedEventArgs>
                                        ConnectionReceived;
    public event EventHandler<PlayerJoinedEventArgs> PlayerJoined;
    public event EventHandler<PlayerExitedEventArgs> PlayerExited;
```

Implement interface

Connection information class

Events

```
    private const string PortOrService = "5150";
    private const int MaxMessageSize = 1024;

    private StreamSocketListener _listener;
    private PlayerConnection _connection;

    public async void Connect(Player me, string remoteHostName){}
    private void CloseConnection() {}
    public void Disconnect() {}

    public async void Listen(Player me) {}

    private void ProcessIncomingMessages() { }

    private async void SendPlayerJoinMessage() {}
    private async void SendPlayerLeaveMessage() {}
    public async void SendChatMessage(ChatMessage message) {}

    public IReadOnlyList<HostName> GetHostNames()
    {
      return NetworkInformation.GetHostNames();
    }
  }
}
```

Connection information

Connection management

Listen as server

Parse and process messages

Send messages

This class has all the functions to implement the `IMessageService` interface, as well as some private functions to provide better code structure.

To make it easier for you to expand on this class to support more players, I've encapsulated the connection information into a class-scoped private class named `PlayerConnection`. You keep only an instance of that private class, but to support multiple players, you may want to keep a dictionary or collection of connections.

OPENING AND CLOSING THE CONNECTION

As you suspected, there's more to this class than what's in this first listing, starting with the connection management code shown in the next listing. Add this to the same `TcpStreamMessageService` class.

Listing 18.18 Connection management

```
public async void Connect(Player me, string remoteHostName)
{
  var hostName = new HostName(remoteHostName);

  var pc = new PlayerConnection();
  pc.Socket = new StreamSocket();

  pc.LocalPlayer = me;
  pc.RemotePlayer = null;

  await pc.Socket.ConnectAsync(hostName, PortOrService);

  pc.Reader = new DataReader(pc.Socket.InputStream);
  pc.Writer = new DataWriter(pc.Socket.OutputStream);
```

Set Player information

Connect

Create readers/writers

```
  _connection = pc;

  ProcessIncomingMessages();

  SendPlayerJoinMessage();
}


private void CloseConnection()
{
  if (_connection != null)
  {
    if (_connection.Writer != null)
      _connection.Writer.Dispose();

    if (_connection.Reader != null)
      _connection.Reader.Dispose();

    if (_connection.Socket != null)
      _connection.Socket.Dispose();

    _connection.LocalPlayer = null;
    _connection.RemotePlayer = null;

    _connection = null;
  }
}

public void Disconnect()
{
  SendPlayerLeaveMessage();
  CloseConnection();
}
```

Start read thread

Send introduction message

Close connection

Disconnect

Note that this class doesn't implement `IDisposable`. But because it's keeping instances of classes that do implement `IDisposable`, the containing class should as well.

The code to connect to an existing peer server is similar to what you had in the original version. The main addition here is the sending of an introduction message to the other machine.

LISTENING AS A SERVER

The server listening code is also very similar to the previous version, as shown in the following listing.

Listing 18.19 Listening as a server

```
public async void Listen(Player me)
{
  _listener = new StreamSocketListener();
  _listener.ConnectionReceived += (s, e) =>
    {
```

Create listener

Wire up received handler

```
        var remoteHost = e.Socket.Information.RemoteHostName;

        var pc = new PlayerConnection();

        pc.Socket = e.Socket;
        pc.Reader = new DataReader(pc.Socket.InputStream);
        pc.Writer = new DataWriter(pc.Socket.OutputStream);
        pc.LocalPlayer = me;
        pc.RemotePlayer = null;

        _connection = pc;

        ProcessIncomingMessages();
        SendPlayerJoinMessage();
      };
    await _listener.BindServiceNameAsync(PortOrService);
}
```

Create reader/writer

Send introduction message

Start read thread

Listen on port

Rather than have a separate `ConnectionReceived` event handler as you did in the first version, you put the event handler into an inline lambda expression inside the `Listen` method. Functionally it's almost identical to the original version, with the same addition of sending an introduction message.

SENDING MESSAGES

So, what about that introduction message? This is where you start seeing some new features in this implementation. The next listing includes the functions to send all the supported message types.

Listing 18.20 Sending messages

```
private async void SendPlayerJoinMessage()
{
  if (_connection.Writer != null)
  {
    string playerName = "(unknown)";
    if (_connection.LocalPlayer != null)
      playerName = _connection.LocalPlayer.Name;

    _connection.Writer.WriteInt32((Int32)WireMessageType.PlayerJoin);
    _connection.Writer.WriteInt32((Int32)playerName.Length);
    _connection.Writer.WriteString(playerName);

    await _connection.Writer.StoreAsync();
  }
}

private async void SendPlayerLeaveMessage()
{
  if (_connection.Writer != null)
  {
    _connection.Writer.WriteInt32((Int32)WireMessageType.PlayerLeave);

    await _connection.Writer.StoreAsync();
  }
}
```

Send introduction message

Send goodbye message

```
public async void SendChatMessage(ChatMessage message)
{
  if (_connection.Writer != null)
  {
    _connection.Writer.WriteInt32((Int32)WireMessageType.ChatMessage);
    _connection.Writer.WriteInt32(message.Message.Length);
    _connection.Writer.WriteString(message.Message);

    await _connection.Writer.StoreAsync();
  }
}
```

Send a chat message

Any message that includes a string (or other variable-length data) must also send a count along with the data. This is so the receiving code can properly parse the contents of the message. You could also use C-style string terminators (\0 or 0x00), but I prefer Pascal-style count prefixes (also used by COM and .NET binary serialization). Figure 18.11 shows a comparison of the two string styles.

I went with the Pascal-style approach. Neither is perfect, and each has its own security concerns, especially when it comes to denial-of-service attacks. For example, a missing terminator in a C-style string can be a mess, while an erroneous high length in a Pascal-style string can cause your app to wait forever. The introduction message is important because it's what the endpoints use to share the name of the person connecting. This is done a single time instead of with each message in order to cut down on the amount of data sent across the wire. You could easily extend this to send the bytes of their profile image along with the name, something you definitely wouldn't want to send with every message.

You can now see why the `DataWriter` can be such a huge help. Instead of having to break types down into arrays of bytes, the writer can natively write strings as well as integer types and many others. This makes the code quite a bit simpler to write and understand.

> **TIP** Whenever possible, using the `DataWriter` instead of working with the streams directly will save you a lot of code and time. The `DataWriter` (and related `DataReader`) include methods for writing and reading common types of data without requiring you to worry about the individual bytes the data breaks down into.

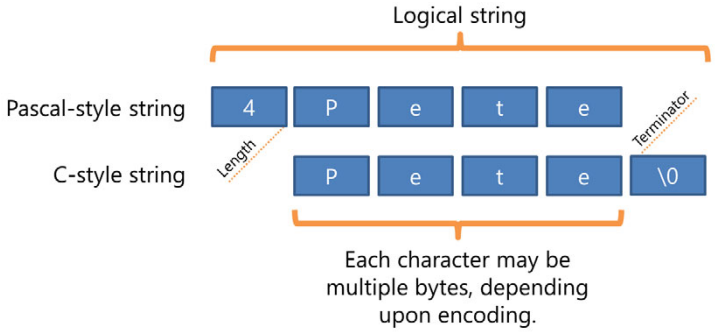

Figure 18.11 The two string styles you may want to consider for your communications.

Other than the additional fields for the length and message type, the process to write and send a message is identical to what you had in the original version: write the data, and then call `StoreAsync`.

PROCESSING MESSAGES

Because of the new message formats, however, reading the messages is more involved. The next listing includes the code for `ProcessIncomingMessages`, which handles all that ugliness.

Listing 18.21 Processing messages

```
private void ProcessIncomingMessages()
{
  var t = Task.Factory.StartNew(async () =>                    <- Start background thread
  {
    _connection.Reader.InputStreamOptions =                    <- Return with partial read
                  InputStreamOptions.Partial;

    while (true)
    {
      var count = await _connection.Reader.LoadAsync(MaxMessageSize);   <- Read data

      var messageType = (WireMessageType)_connection.Reader.ReadInt32();   <- Get message type

      switch (messageType)
      {
        case WireMessageType.PlayerLeave:                      <- Player left
          if (PlayerExited != null)
            PlayerExited(this, new
                  PlayerExitedEventArgs(_connection.RemotePlayer));
          break;

        case WireMessageType.PlayerJoin:                       <- Player joined
          if (PlayerJoined != null)
          {
            var nameLength = _connection.Reader.ReadInt32();
            var name = _connection.Reader.ReadString((uint)nameLength);

            var remotePlayer = new Player();
            remotePlayer.Name = name;

            _connection.RemotePlayer = remotePlayer;

            PlayerJoined(this, new PlayerJoinedEventArgs(remotePlayer));
          }
          break;

        case WireMessageType.ChatMessage:                      <- Chat message
          var msgLength = _connection.Reader.ReadInt32();
          var text = _connection.Reader.ReadString((uint)msgLength);

          var msg = new ChatMessage();
          msg.Message = text;
          msg.Player = _connection.RemotePlayer;
```

```
            if (ChatMessageReceived != null)
              ChatMessageReceived(this, new
                          ChatMessageReceivedEventArgs(msg));
            break;
        }
      }
    });
}
```

As in the previous version, this code spins up a background thread that reads from the stream. The `LoadAsync` method is an async call but acts like a blocking call. The execution will halt at that line until there's data to be read.

The `InputStreamOptions.Partial` setting was used in the original version of this code as well. Without this setting, the `LoadAsync` method will return only when it has read `MaxMessageSize` bytes. You instead want it to return when it has read a complete message, regardless of how small it is.

The parsing is the reverse of the writing code. Note how the code reads the string length (in the case of the "player joined" and "chat message" types) and then uses that to tell the reader how many characters to read from the buffer and treat as the string contents.

Message framing

TCP streaming sockets operate on streams of data, not packets of data. Because this is a chat app, where people have to type at a keyboard (or screen), the chances of getting more than one message at once are minimal. The code I've provided here treats it as though you're working with packets of data, even though this isn't technically correct.

But if you're in a situation where you could potentially get a lot more than a single message, you need to loop through all of the unconsumed buffer (using the `UnconsumedBufferLength` property) and read as much as possible, potentially even waiting on more bytes because of partial messages.

In short, with TCP streamed sockets, a single send does not necessarily result in a single receive. TCP isn't doing anything to preserve your message boundaries or frames. You have to do that yourself.

How you go about doing this depends on the size and structure of your messages. I had to do similar processing with MIDI serial communications in the .NET Micro Framework, and trust me, it considerably complicates the parsing code.

Because this class must communicate back with the `MainViewModel`, each of the blocks of code raises an event. Note that because this is executing on a background thread, the events will not be fired on the UI thread. This will be important to know when you get to the `MainViewModel` code.

Finally, you need to update the `MainViewModel` to use this new code. As you can imagine, this will require quite a few changes, because you've essentially gutted it and replaced all of its functionality with this new service.

18.5.4 Updated MainViewModel

The public interface (with the exception of the new `Player` information in the `ChatMessage`) for the `MainViewModel` is the same as what you started with. But there are enough small changes to the code to make me decide to simply provide you with the full source code for the viewmodel, broken across several listings.

INITIAL STRUCTURE

The first of these listings contains the skeleton of the viewmodel as well as the constructor.

Listing 18.22 Overall `MainViewModel` structure

```
using GalaSoft.MvvmLight;
using GalaSoft.MvvmLight.Command;
using GalaSoft.MvvmLight.Threading;
using SocketApp.Model;
using SocketApp.Services;
using System;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;
using Windows.System.UserProfile;

namespace SocketApp.ViewModel
{
  public class MainViewModel : ViewModelBase
  {
    private IMessageService _messageService;

    public MainViewModel()
    {
      _messageService = new TcpStreamMessageService();

      _messageService.ChatMessageReceived +=
                                OnServiceChatMessageReceived;
      _messageService.ConnectionReceived += OnServiceConnectionReceived;
      _messageService.PlayerExited += OnServicePlayerExited;
      _messageService.PlayerJoined += OnServicePlayerJoined;

      ChatMessages = new ObservableCollection<ChatMessage>();
      ConnectionStatus = "Not connected.";
      ServerAddress = "pete-surface64";

      PostNewMessageCommand = new RelayCommand(
        () => PostNewMessage(),
        () => CanPostNewMessage());

      ListenCommand = new RelayCommand(
        () => Listen(),
        () => CanListen());
```

Service variable

Create service

New event handlers

```
      ConnectCommand = new RelayCommand(
        () => Connect(),
        () => CanConnect());

      InitializePlayer();
      CreateNewMessage();
    }

    private Player _player;
    public Player Player
    {
      get { return _player; }
      set { Set<Player>(() => Player, ref _player, value); }
    }

    private async void InitializePlayer()
    {
      _player = new Player();
      _player.Name = await UserInformation.GetDisplayNameAsync();
    }
  }
}
```

Initialize player

Player

Get Windows username

Notice how in the constructor you create a concrete instance of the `TcpStreamMessageService` class but assign it to a variable of type `IMessageService`. By referring only to the interface in code, you'll need to change only this single line of code in the constructor when it comes time to swap out socket implementations.

The other interesting part of this listing is the code to get the username. The `UserInformation` class has a number of async functions that you can use to get the username, display name, the user's image, and much more.

> **TIP** The `UserInformation` class in the `Windows.System.UserProfile` namespace contains information about the logged-in user. You can use this to get the name, username, and Windows profile picture, among other things.

CONNECTING TO THE SERVER

The next listing has the new code to connect to an existing server. Because of the new service class, the calls here are reduced to just a few lines of code.

Listing 18.23 Connecting as a client

```
private string _serverAddress;
public string ServerAddress
{
  get { return _serverAddress; }
  set { Set<string>(() => ServerAddress, ref _serverAddress, value); }
}

private string _connectionStatus;
public string ConnectionStatus
{
  get { return _connectionStatus; }
  set { Set<string>(() => ConnectionStatus, ref _connectionStatus, value);}
}
```

```
public RelayCommand ConnectCommand { get; private set;}

public void Connect()
{
    _messageService.Connect(Player, ServerAddress);
    ConnectionStatus = "Connected to server at " + ServerAddress + ".";
}

public bool CanConnect()
{
    return true;
}
```

Connect using service

The `Connect` method no longer includes all the code to call out to the sockets functions. Instead, it simply calls the `Connect` function of the message service.

LISTENING FOR CONNECTIONS

Listening for a new connection has been similarly pared down. Here's the code.

Listing 18.24 Listening as a server

```
public RelayCommand ListenCommand { get; private set; }

public void Listen()
{
  _messageService.Listen(Player);

  ConnectionStatus = "Waiting for connection on: ";

  int i = 0;

  foreach (HostName name in _messageService.GetHostNames())
  {
    if (i > 0)
      ConnectionStatus += " and ";

    ConnectionStatus += name.DisplayName;
    i++;
  }
}

public bool CanListen() { return true; }
```

Listen using service

SENDING CHAT MESSAGES

Next, you have the functionality to send chat messages, shown in the following listing. Much of the viewmodel code around chat messages is there just to provide interaction with the UI. The posting of the messages themselves happens in the message service.

Listing 18.25 Sending chat messages

```
public ObservableCollection<ChatMessage> ChatMessages { get; set; }

private ChatMessage _newMessage;
public ChatMessage NewMessage
{
```

```
    get { return _newMessage; }
    set { Set<ChatMessage>(() => NewMessage, ref _newMessage, value); }
}

public RelayCommand PostNewMessageCommand { get; private set; }

private void CreateNewMessage()
{
    if (NewMessage != null)
        NewMessage.PropertyChanged -= NewMessage_PropertyChanged;

    NewMessage = new ChatMessage();
    NewMessage.Player = Player;
    NewMessage.PropertyChanged += NewMessage_PropertyChanged;
}

void NewMessage_PropertyChanged(object sender, PropertyChangedEventArgs e)
{
    if (e.PropertyName == "Message")
        PostNewMessageCommand.RaiseCanExecuteChanged();
}

public async void PostNewMessage()
{
    ChatMessages.Add(NewMessage);
    _messageService.SendChatMessage(NewMessage);

    CreateNewMessage();
}

public bool CanPostNewMessage() { return true; }
```

Assign player info

Send message using service

EVENT HANDLERS

Finally, you have the event handlers. Because the message service exposes a number of events for communicating back to the viewmodel, this is an essential part of the communication. Also, recall how I mentioned that the events aren't coming back on the UI thread: This is handled using the `DispatcherHelper` in each of the event handlers shown in the next listing.

Listing 18.26 Event handlers

```
void OnServicePlayerJoined(object sender, PlayerJoinedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    ConnectionStatus = "Just joined: " +
            e.Player.Name;
  });
}

void OnServicePlayerExited(object sender, PlayerExitedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
```

Player joined

Player exited

```
    ConnectionStatus = "Just exited: " +
            e.Player.Name;
  });
}

void OnServiceConnectionReceived(object sender,                    <-- New connection
                                 ConnectionReceivedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    ConnectionStatus = "Connection received from " +
            e.HostName.DisplayName;
  });
}

void OnServiceChatMessageReceived(object sender,                   <-- New chat message
                                  ChatMessageReceivedEventArgs e)
{
  DispatcherHelper.CheckBeginInvokeOnUI(() =>
  {
    ChatMessages.Add(e.Message);
  });
}
```

Each of the event handlers uses the `DispatcherHelper` to ensure the code has access to the UI thread. These handlers are the only communication back to the viewmodel from the message service, so there are handlers for each type of message. If you extend the number of messages your service supports, you'll want to provide appropriate events for those messages as well.

Run the app now, just as you did before. When you connect, you'll see the acknowledgment of the player joining. Once you send a message, you'll notice the additional field in the new data template in the `ListView`.

Those were nice little enhancements thrown in to spice it up a little. The real reason for going through the trouble to refactor and use interfaces is so you can try UDP sockets with very little additional effort.

18.6 Trying out UDP sockets

So far, you've used TCP streaming sockets in this app. Streaming sockets are appropriate for many tasks, but they aren't the absolute lightest weight approach. TCP sockets have additional overhead (both in time and in bytes transferred) for guaranteeing delivery and ordering of messages. In some cases, especially a game sending many real-time updates, it's more critical to be light and fast than it is to worry about getting every single message across the wire.

For those times when you want something a little faster and a little lighter, you have User Datagram Protocol (UDP). UDP is a transport layer protocol like TCP. But unlike TCP, there's no guarantee of delivery or message ordering. If you consider what that means, you can see where something like an audio stream might be better transported over TCP, whereas something like a heartbeat signal or your location in a game at that second might be better transported over UDP. Those aren't hard-and-fast rules,

of course. Most audio- and video-streaming protocols build over UDP with all the ordering and other tasks handled higher up in the application layer.

If you want more information on the differences between the two protocols, Wikipedia has two great pages full of details of the headers, reliability, and much more:

- http://en.wikipedia.org/wiki/User_Datagram_Protocol
- http://en.wikipedia.org/wiki/Transmission_Control_Protocol

Choosing a transport protocol isn't something you should take lightly if performance and reliability of communications are important to you. But, with appropriate abstraction in your app, you can avoid locking yourself into any single protocol early in app development.

In this section, you'll build a UDP version of the messaging service for this app. You'll make sure it has the same interface as the TCP version so that either protocol may be easily swapped in or out.

18.6.1 Creating the UdpMessageService class

The UDP implementation of the message service will be in its own class file, just like the TCP version. In the Services folder, add a new class file named `UdpMessageService`, and paste into it the following code.

Listing 18.27 The skeleton for the `UdpMessageService` class

```
using GalaSoft.MvvmLight.Threading;
using SocketApp.Model;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Windows.Networking;
using Windows.Networking.Connectivity;
using Windows.Networking.Sockets;
using Windows.Storage.Streams;

namespace SocketApp.Services
{
  public class UdpMessageService : IMessageService
  {
    private class PlayerConnection
    {
      public Player LocalPlayer { get; set; }
      public Player RemotePlayer { get; set; }
      public DataWriter Writer { get; set; }
      public DatagramSocket Socket { get; set; }
    }

    public event EventHandler<ChatMessageReceivedEventArgs>
                 ChatMessageReceived;
    public event EventHandler<ConnectionReceivedEventArgs>
                 ConnectionReceived;
```

Note no DataReader

Identical to TcpStreamMessageService code

```
    public event EventHandler<PlayerJoinedEventArgs> PlayerJoined;
    public event EventHandler<PlayerExitedEventArgs> PlayerExited;

    private const string PortOrService = "5150";
    private const int MaxMessageSize = 1024;
    private PlayerConnection _connection;

    public async void Connect(Player me, string remoteHostName) {}
    private void CloseConnection() {}
    public void Disconnect() {}

    public async void Listen(Player me) {}

    private async void SendPlayerJoinMessage() {}
    private async void SendPlayerLeaveMessage() {}
    public async void SendChatMessage(ChatMessage message) {}
    public IReadOnlyList<HostName> GetHostNames() {}
  }
}
```

Identical to TcpStreamMessageService code

Copy full from TcpStreamMessageService code

The `SendPlayerJoinMessage`, `SendPlayerLeaveMessage`, `SendChatMessage`, and `GetHostNames` functions are all identical to those in the `TcpStreamMessageService` class, so simply copy them over into this file.

This version of the `PlayerConnection` private class also omits the `DataReader` instance. You'll see why shortly.

18.6.2 Listening for connections

Or, more correctly stated, "binding to a service." The connectivity doesn't work quite the same way it did with TCP streaming sockets. For example, connections aren't persistent and aren't two-way by default. Instead, each side must both connect and listen if it wishes to communicate two-way.

Listing 18.28 Listening for connections

```
public async void Listen(Player me)
{
  _connection = new PlayerConnection();

  BindLocalSocket();
}
private async void BindLocalSocket()
{
  _connection.Socket = new DatagramSocket();
  _connection.Socket.MessageReceived += OnSocketMessageReceived;
  await _connection.Socket.BindServiceNameAsync(PortOrService);
}
```

MessageReceived

Bind

To listen for a connection, you need to create the `DatagramSocket` instance and then wire up the `MessageReceived` event handler. There's no polling loop as you had with TCP, just a discrete message handler. Finally, you bind the socket using `BindServiceNameAsync` or one of the equivalent methods. Doing this establishes a socket you can read from.

18.6.3 Connecting to another machine

The biggest difference between UDP and the TCP code you built earlier is that UDP doesn't have the concept of a persistent connection established before sending data. It just sends data to a specified endpoint. UDP is truly "fire and forget." The `ConnectAsync` function is merely a convenience that handles binding behind the scenes. You could simply bind the port and send messages or even get an output stream with the specified endpoint and start writing.

The following listing establishes the connection to the remote machine, mainly by setting up the data writer and calling the `ConnectAsync` function to handle the binding.

Listing 18.29 Connecting to another machine

```
public async void Connect(Player me, string remoteHostName)
{
  _connection = new PlayerConnection();
  _connection.LocalPlayer = me;

  BindLocalSocket();

  ConnectToRemoteSocket(new HostName(remoteHostName), PortOrService);
}

private async void ConnectToRemoteSocket(HostName hostName,
          string portOrService)
{
  await _connection.Socket.ConnectAsync(hostName, portOrService);
  _connection.Writer = new DataWriter(_connection.Socket.OutputStream);

  SendPlayerJoinMessage();
}

private void CloseConnection()
{
  if (_connection != null)
  {
    if (_connection.Writer != null)
      _connection.Writer.Dispose();

    if (_connection.Socket != null)
      _connection.Socket.Dispose();

    _connection.LocalPlayer = null;
    _connection.RemotePlayer = null;

    _connection = null;
  }
}

public void Disconnect()
{
  SendPlayerLeaveMessage();

  CloseConnection();
}
```

Required to listen for replies

Connect to remote

Set up writer

Send introduction message

Cleanup

Say "good-bye" and clean up

This listing shows more of the two-way dance you need to do when using UDP sockets. In order to be able to send messages, you need to connect. In order to be able to receive messages, you need to bind.

18.6.4 Receiving and parsing messages

Receiving messages is done via an event handler. The event fires only when messages are received. For that reason, there's no need for a loop or the `LoadAsync` method you used with TCP.

The first part of listing 18.30 establishes a writeable connection to the remote socket. This is required because you don't know who is connecting to you up front, and UDP sockets don't offer the convenience of a negotiated connection and the established read/write streams. Instead, if you want to talk to the person who contacted you, you must connect to them.

Listing 18.30 Receiving and parsing messages

```
async void OnSocketMessageReceived(DatagramSocket sender,
                        DatagramSocketMessageReceivedEventArgs args)
{
  if (_connection.Writer == null)
    ConnectToRemoteSocket(args.RemoteAddress, args.RemotePort);     <-- Connect back

  var reader = args.GetDataReader();
  var messageType = (WireMessageType)reader.ReadInt32();

  switch (messageType)
  {
    case WireMessageType.PlayerLeave:
      if (PlayerExited != null)
        PlayerExited(this,
            new PlayerExitedEventArgs(_connection.RemotePlayer));
    break;

    case WireMessageType.PlayerJoin:
      if (PlayerJoined != null)
      {
        var nameLength = reader.ReadInt32();
        var name = reader.ReadString((uint)nameLength);

        var remotePlayer = new Player();
        remotePlayer.Name = name;

        _connection.RemotePlayer = remotePlayer;

        PlayerJoined(this, new PlayerJoinedEventArgs(remotePlayer));
      }
      break;

    case WireMessageType.ChatMessage:
      var msgLength = reader.ReadInt32();
      var text = reader.ReadString((uint)msgLength);
```

```
      var msg = new ChatMessage();
      msg.Message = text;
      msg.Player = _connection.RemotePlayer;

      if (ChatMessageReceived != null)
        ChatMessageReceived(this, new ChatMessageReceivedEventArgs(msg));
      break;    }
  }
```

The `switch` statement that parses the messages is almost identical to the TCP version, except for having to use the `args.GetDataReader` method to get the data reader. Other than that, the method is considerably cleaner because of the lack of looping and data loading. When working with UDP sockets, there exists no persistent stream, so the data reader is provided anew each time data is received.

Finally, crack open the `MainViewModel` and change the constructor so it instantiates the new class:

```
//_messageService = new TcpStreamMessageService();
_messageService = new UdpMessageService();
```

Everything else in the `MainViewModel` class stays the same. If you want to switch back to TCP, simply change this one statement.

If you run the app now, you should see approximately the same thing you saw with the TCP version. My "introduction" message handling isn't identical, but you should be able to send messages back and forth without issue.

For two-way communication, I personally find TCP sockets easier to work with. For very quick one-way communication, however, it's hard to beat UDP sockets. If you get some good core patterns working with UDP, it can be quite efficient to work with. But remember, if you want built-in reliability, you want TCP.

WebSockets

Windows Store apps can also use WebSockets. Despite the name, the protocol doesn't operate over HTTP; only the initial negotiation does. Communication with WebSockets is done over commonly open ports, like port 80, so they are very firewall friendly.

There are two types of WebSockets available in WinRT: the `MessageWebSocket` and the `StreamWebSocket`. The `MessageWebSocket` helps by handling message framing for you but is otherwise conceptually similar to the UDP (Datagram) socket approach. Similarly, the `StreamWebSocket` is conceptually similar to the TCP stream socket we covered earlier.

You typically won't use WebSockets for peer-to-peer communication, because the technology requires a web server for the initial handshake. Instead, you need to set up a proper server (or find one) that's running WebSocket services. For those reasons, I don't go into detail on them here.

For more information, see http://bit.ly/WinRTWebSockets.

18.7 Summary

Socket communication is something that many business developers never run across in their own apps. But in the games, communications, and social network app worlds, it's extremely popular. Sockets provide low-level access without the overhead of a protocol such as HTTP. For that reason, sockets are lightweight and very fast, but for the same reasons, they generally take more effort to use.

In this chapter you built an MVVM app that used both TCP sockets and UDP sockets to communicate between two Windows 8 machines. I was absolutely tickled the first time I realized that a Windows 8 app can be a socket server. That's unusual in a sandboxed environment but enables so many cool scenarios.

TCP sockets are good for reliable communication between two endpoints. I also find them a bit easier to work with compared to UDP when you need to have bidirectional communication. TCP will guarantee delivery and order of data, at the cost of some additional frame overhead and potentially some speed loss.

UDP sockets are good when you want something extremely fast and don't care if it ever gets there. Where TCP is more like FedEx, UDP is more like throwing the package in your cousin's pickup truck and asking him to drop it off. Sure, it'll probably work, and yes, it was a lot cheaper (and maybe even faster), but reliability is definitely a concern.

In the next chapter, we'll expand on the app developed here and update the UI so it works more like a game. We'll add in player graphics, and you'll also learn a bit about Blend and user controls.

WINDOWS/.NET

Windows Store App Development

Pete Brown

The Windows Store provides an amazing array of productivity tools, games, and other apps directly to the millions of customers already using Windows 8.x or Surface. Windows Store apps boast new features like touch and pen input, standardized app-to-app communication, and tight integration with the web. And, you can build Windows Store apps using the tools you already know: C# and XAML.

Windows Store App Development introduces the Windows 8.x app model to readers familiar with traditional desktop development. You'll explore dozens of carefully crafted examples as you master Windows features, the Windows Runtime, and the best practices of app design. Along the way, you'll pick up tips for deploying apps, including selling through the Windows Store.

What's Inside

- Designing, creating, and selling Windows Store apps
- Developing touch and sensor-centric apps
- Working C# examples, from feature-level techniques to complete app design
- Making apps that talk to each other
- Mixing in C++ for even more features

This book requires some knowledge of C#. No experience with Windows 8 is needed.

Pete Brown is a Developer Evangelist at Microsoft and author of *Silverlight 4 in Action* and *Silverlight 5 in Action*.

"Informative, fun, and easy to read."
—Todd Miranda
NxtDimension Solutions

"Broad coverage of all aspects of W8 XAML development."
—Roland Civet, iSolutions For You!

"Pete is a consistently great author, and once again he nails his subject."
—Gordon Mackie Openfeatured Ltd.

"Your roadmap to modern Windows design."
—Patrick Toohey
Mettler-Toledo Hi-Speed

"Much less a book than a must-have tool for efficient and quality app development."
—Dave Campbell, WynApse

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit manning.com/WindowsStoreAppDevelopment

ISBN 13: 978-1-617290-94-7
ISBN 10: 1-617290-94-7
54999
9 781617 290947

MANNING $49.99 / Can $52.99 [INCLUDING eBOOK]