How to Build Music Sync

MusicSync is a music streaming application designed to play one track on many Windows 10 machines simultaneously. There are three components in this solution. The first is the Server, or Publisher, called MusicServer. The second is the Speaker, or Subscriber called VirtualSpeaker. The final component is a runtime component which contains the Subscriber logic. This final component will be referenced in both the Server and Speaker projects as both will act as a Subscriber.  
  
The MusicServer application contains both a Publisher and a Subscriber. It is a Subscriber to it’s Publisher in order to play music on the local host device.  
The VirtualSpeaker application only contains a Subscriber

Getting Started  
In order to build and test this project you will need some experience with C#, and XAML. You are required to use Windows 10 and Visual Studio 2015 (Community Edition or greater) when building this app.

Create a new Universal Application

1. Open Visual Studio 2015 and select **File | New | Project** (CTRL + SHIFT + N)
2. This will open a “New Project” window. Navigate to Templates | Visual C# and
3. Choose Blank App (Universal Apps) and Name your application MusicServer

Machine generated alternative text:
New Project 
D Recent 
Installed 
Templates 
Visual 
Windows 
Universal 
D Windows 8 
Classic Desktop 
Web 
Android 
Cloud 
.NET Framework 4.5.2 
• Sort by. Default 
Search Installed Tel 
Type: Visual C# 
A project for a single-page Universal 
Windows Platform app that has no 
predefined controls or layout. 
Show telemetry in the Windows 
Dev Center 
Installs the Application Insights SDK 
to send usage telemetry to the 
Windows Dev Center. 
Learn more 
Privacy statement 
Browse... 
Create directory for solution 
Add to source control 
Blank App (Universal Windows) Visual 
Class Library (Universal Wind... 
Windows Runtime Compone... 
Visual C# 
Visual C# 
Visual C# 
Unit Test App (Universal Wind... Visual C# 
Coded UI Test Project (Windo... 
Click here to go online and find templates. 
D Online 
Location: 
Solution name: 
MusicServer 
C:1Work 
MusicServer 

1. This application is going to use the local network to send and receive UDP packets, so Open the Package.appxmanifest in the Solution Explorer and go to the Capabilities tab, and check Private Networks (Client & Server) and Internet (Client & Server):

Machine generated alternative text:
Capabilities: 
All Joyn 
Blocked Chat Messages 
Bluetooth 
Chat Message Access 
Code Generation 
Enterprise Authentication 
Internet (Client) 
Internet (Client & Server) 
Location 
Microphone 
Music Library 
Objects 3D 
Phone Call 
Pictures Library 
Private Networks (Client & Server) 
Proximity 

1. Build and Run the new Application (F5), you should see a blank window shown on the screen. Close the Application.

# Exercise 1 – Setting up the View and Opening a Media File:

The view for the server is going to be a log window with the list of attached virtual speakers, and some control buttons:

* Select File
* Send, and
* Play/Stop.

A ListView will present the list of speakers and their status which is the only thing to update in this interface. This lab will use the Debug Output window of Visual Studio so make sure it is visible while debugging.

1. In solution explorer, select the MainPage.xaml file. Add the following XAML inside the Grid element. This XAML contains a title and a StackPanel that will display the running devices IPAddress. This will be the IPAddress to which the Virtual speakers will need to connect.

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}">

<StackPanel Orientation="Vertical" HorizontalAlignment="Center">

<!-- Title -->

<TextBlock x:Name="TitleTextBlock" Text="Music Server" FontSize="30" HorizontalAlignment="Center" Margin="4"/>

<!-- IP Address of the Host -->

<StackPanel x:Name="IPAddressStackPanel" Orientation="Horizontal" HorizontalAlignment="Center">

<TextBlock x:Name="IPAddressTextBlock" Text="Your IP Address: " HorizontalAlignment="Left" Margin="4"/>

<TextBlock x:Name="HostIPAddressTextBlock" Text="" HorizontalAlignment="Right" Margin="4"/>

</StackPanel>

</StackPanel>

</Grid>

1. Underneath the IPAddressStackPanel add another two StackPanels. The first one will provide a way for a media file to be selected. The second StackPanel will have a Send Button and an interchanging Play/Stop Button depending on the current state of the media playback.

<!-- Media Selection Panel -->

<StackPanel x:Name="MediaFileSelectionStackPanel" Orientation="Horizontal" Margin="4" HorizontalAlignment="Center">

<Button x:Name="SelectMediaFileButton" Content="Select media file" HorizontalAlignment="Right" Click="SelectButton\_Click" IsEnabled="False"/>

</StackPanel>

<!-- Media Controls Panel -->

<StackPanel x:Name="ControlStackPanel" Orientation="Horizontal">

<Button x:Name="SendButton" Content="Send" Margin="4" Click="SendButton\_Click" IsEnabled="False" Width="120"/>

<Button x:Name="PlayButton" Grid.Row="1" Content="Play" Margin="4" Click="PlayButton\_Click" IsEnabled="False" Width="120"/>

</StackPanel>

1. Add a ListView to display the speakers and associated statuses.

<!-- Message Log ListView -->

<ListView x:Name="MessageLogListView" />

1. In MainPage.xaml.cs add the below code to respond to the click events and outline the tasks to complete for each event. The project should now build.

namespace MusicServer

{

public sealed partial class MainPage : Page

{

public MainPage()

{

this.InitializeComponent();

this.Loaded += MainPage\_Loaded;

}

private void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

// TODO: initialize publisher and load in virtual speakers.

}

private void SelectButton\_Click(object sender, RoutedEventArgs e)

{

// TODO: open FileOpenPicker

}

private void SendButton\_Click(object sender, RoutedEventArgs e)

{

// TODO: send the media file to remote speakers

}

private void PlayButton\_Click(object sender, RoutedEventArgs e)

{

// TODO: send the Stop or Play commands to virtual speakers

}

}

}

1. Create a new method called GetThisIPAddress in the MainPage.xaml.cs file. This will return the IPV4 address of the machine. You will need to include the Windows.Networking.Connectivity namespace.   
   Call this new method from the Loaded event handler in MainPage.xaml.cs and set the HostIPAddressTextBlock. While in the Loaded method also set the SelectMediaFileButton to be enabled.

using Windows.Networking.Connectivity;

private void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

//get the current IP address of this machine and display it

HostIPAddressTextBlock.Text = GetThisIPAddress();

SelectMediaFileButton.IsEnabled = true;

// TODO: initialize publisher and load in virtual speakers.

}

private string GetThisIPAddress()

{

string lastHostName = "";

var hosts = NetworkInformation.GetHostNames();

foreach (var host in hosts)

{

// The last host name is always this computer.

if (host.Type == Windows.Networking.HostNameType.Ipv4)

{

lastHostName = host.DisplayName;

}

}

return lastHostName;

}

1. The Select button should open a FileOpenPicker interface. The results of the user choice will give you the audio file to be streamed to virtual speakers. The audio file will be referenced as a StorageFile, an object with many reading and streaming capabilities.  
   Add a private variable in the MainPage class for this StorageFile object. You will need to add the Windows.Storage namespace:

using Windows.Storage;

public sealed partial class MainPage : Page

{

private StorageFile \_mediaFile;

1. In the SelectButton\_Click method, create and open a FileOpenPicker, and set the result to the mediaFile. Update the UI to reflect the chosen filename. This method will await the PickSingleFileAsync method, so update the method definition to be async. If a media file is chosen then enable the send button.

using Windows.Storage.Pickers;

using System.Diagnostics;

private async void SelectButton\_Click(object sender, RoutedEventArgs e)

{

// load filepicker for user to select a media file

FileOpenPicker filePicker = new FileOpenPicker();

filePicker.SuggestedStartLocation = PickerLocationId.MusicLibrary;

filePicker.FileTypeFilter.Add(".mp4");

filePicker.FileTypeFilter.Add(".MOV");

filePicker.FileTypeFilter.Add(".mp3");

filePicker.FileTypeFilter.Add(".wav");

filePicker.ViewMode = PickerViewMode.Thumbnail;

\_mediaFile = await filePicker.PickSingleFileAsync();

if (\_mediaFile != null)

{

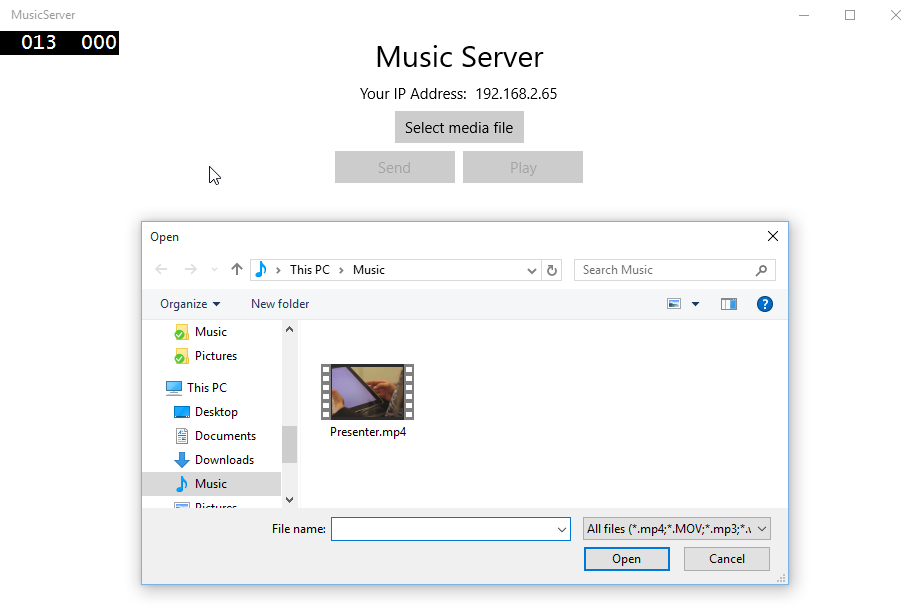
Debug.WriteLine(\_mediaFile.DisplayName);

SendButton.IsEnabled = true;

}

}

1. Build and run the application (F5). Click the Select button and choose a compatible audio file (try to choose one under 10MB to make debugging easy). The file name will appear in the text field at the top of the page:



# Exercise 2 – Creating a Publisher, Speaker and begin Listening:

You need an object responsible for managing connections with virtual speakers, and sending the commands to those clients. This object is called the Publisher.

1. Create a new class at the root level of the MusicServer project and call it Publisher.cs. Right click on the MusicServer solution, and select **Add** | **Class**

Machine generated alternative text:
Add New Item 
Installed 
Visual 
Code 
General 
D Web 
XAML 
D Online 
MusicServer 
Publisher.cs 
Sort by. Default 
XAML View 
Blank Page 
Content Dialog 
Resource Dictionary 
Templated Control 
User Control 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Type: Visual C# 
An empty class definition 
Click here to go online and find templates. 

1. The publisher will listen on a socket (which will be port 21121 on every adapter for the machine running this application) for any connection requests. The socket will have a maximum packet size of 10,000 bytes (which can be changed later depending on the connection quality). Put these public variables in the new Publisher class and change the class definition to make the Publisher class public. You will need to add a new namespace to use the StreamSocketListener

using Windows.Networking.Sockets;

public class Publisher

{

// this is a the port number in a released app this should be configurable

private const string SERVICE\_NAME = "21121";

// this max packet size is used in case of a poor internet connection in order to get the packet in reasonable size transferred

private const int MAX\_PACKET\_SIZE = 10000;

private StreamSocketListener \_listener;

1. Open the socket in a new method called Initialize(). Within this method the socket listener will register for the ConnectionRecieved event, which occurs when a connection request is made by a client on that socket. Also create the Listener\_ConnectionRecieved method to handle this event (leave this method empty for now):

using System.Diagnostics;

public async Task<bool> Initialize()

{

bool success = true;

try

{

//set up a listener socket for speakers to connect with

\_listener = new StreamSocketListener();

\_listener.Control.KeepAlive = true;

\_listener.Control.QualityOfService = SocketQualityOfService.LowLatency;

\_listener.ConnectionReceived += Listener\_ConnectionReceived;

await \_listener.BindServiceNameAsync(SERVICE\_NAME);

}

catch (Exception ex)

{

Debug.WriteLine("Error: " + ex);

success = false;

}

return success;

}

private void Listener\_ConnectionReceived(  
 StreamSocketListener sender,   
 StreamSocketListenerConnectionReceivedEventArgs args)  
{  
 // TODO: add connected client to list of speakers

}

1. In the MainPage.xaml.cs, create a new private variable of type **Publisher** called \_*publisher* and call Initialize on the \_*publisher* in the MainPage\_Loaded method (The loaded event needs to be updated to async in order to await the initialize call). Move the call to enable the SelectMediaFileButton inside a conditional check (if statement) on the publisher initialize method.

public sealed partial class MainPage : Page

{

private StorageFile \_mediaFile;

private Publisher \_publisher;

public MainPage()

{

this.InitializeComponent();

this.Loaded += MainPage\_Loaded;

}

private async void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

//get the current IP address of this machine and display it

HostIPAddressTextBlock.Text = GetThisIPAddress();

//set up as a publisher and bind the UI to the list of virtual speakers

\_publisher = new Publisher();

bool init = await \_publisher.Initialize();

if (init)

{

//allow a new media file to be selected

SelectMediaFileButton.IsEnabled = true;

}

else

{

Debug.WriteLine("Error: Publisher failed to initialize");

}

}

1. Build and Run the application to check that the listener is initializing correctly. An easy way to check is to see if the Select Media button is enabled. If there are errors then pay attention to the Debug Output window in Visual Studio.
2. You may have noticed the TODO comments for adding speakers, the speakers are the virtual speakers which are added through the listener when their connection is received. Make a new Speaker class now in a new class file called Speaker, create it at the root level of MusicServer:

Machine generated alternative text:
Add New Item 
Installed 
Visual 
Code 
General 
D Web 
XAML 
D Online 
MusicServer 
Speaker.cs 
Sort by. Default 
XAML View 
Blank Page 
Content Dialog 
Resource Dictionary 
Templated Control 
User Control 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Type: Visual C# 
An empty class definition 
Click here to go online and find templates. 

1. The Speaker object has the following properties: a Name, an Address (IP), a Status, and a StreamSocket. It will also need to override the ToString method.

Change the class definition to be public.

using Windows.Networking.Sockets;

namespace MusicServer

{

public class Speaker

{

public string Name { get; set; }

public string Address { get; set; }

public string Status { get; set; }

public StreamSocket Socket { get; set; }

public override string ToString()

{

return String.Format("{0}:{1} {2}", Name, Address, Status);

}

}

}

1. The publisher will contain the list of attached speakers. This list need to be in an observable collection as the speaker list will be reflected in the user interface, which can use a databinding if the collection is observable. Put the speakers property at the top of the Publisher class under your current private variables:

using System.Collections.ObjectModel;

public class Publisher

{

// this is a the port number in a released app this should be configurable

private const string SERVICE\_NAME = "21121";

// this max packet size is used in case of a poor internet connection in order to get the packet in reasonable size transferred

private const int MAX\_PACKET\_SIZE = 10000;

private StreamSocketListener \_listener;

//the collection of virtual speakers

// observable so that the UI can update when it changes

private ObservableCollection<Speaker> \_speakers;

public ObservableCollection<Speaker> Speakers

{

get

{

if (\_speakers == null)

{

\_speakers = new ObservableCollection<Speaker>();

}

return \_speakers;

}

}

1. You now have a list of speakers that is guaranteed to be initialized. In the MainPage class set the MessageLog.ItemsSource to the collection of Speakers in the Publisher.

private async void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

//get the current IP address of this machine and display it

HostIPAddressTextBlock.Text = GetThisIPAddress();

//set up as a publisher and bind the UI to the list of virtual speakers

\_publisher = new Publisher();

MessageLogListView.ItemsSource = \_publisher.Speakers;

bool init = await \_publisher.Initialize();

if (init)

{

//allow a new media file to be selected

SelectMediaFileButton.IsEnabled = true;

}

else

{

Debug.WriteLine("Error: Publisher failed to initialize");

}

}

Now the speakers in the publisher are set up and the publisher is listening, you are ready to connect to the publisher in the next exercise.

# Exercise 3 – Creating a Subscriber and Connecting:

This application is designed to serve many virtual speakers attached over a network. These other speakers will subscribe to the host, or Publisher application running on a different machine. The Publisher application should also play music itself. The most logical way to do this is for the Publisher application to also have a Subscriber, and subscribe to "itself" as a speaker. This is also good for testing and debugging because it only requires a single machine to get the expected output of the application.

1. Create a new class file in the root MusicServer project now and call it Subscriber.cs:

Machine generated alternative text:
Add New Item 
Installed 
Visual 
Code 
General 
D Web 
XAML 
D Online 
MusicServer 
Subscriber.cs 
Sort by. Default 
XAML View 
Blank Page 
Content Dialog 
Resource Dictionary 
Templated Control 
User Control 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Visual C# 
Type: Visual C# 
An empty class definition 
Click here to go online and find templates. 

1. Change the class definition of the Subscriber class to be public. Add two new private variables, which will be used later to receive and play the media stream sent from the server. Initialise these member variables in the constructor.

using Windows.Storage.Streams;

namespace MusicServer

{

public class Subscriber

{

// two streams in memory.

// One stream is used to receive the file

// the other is used to play the file

private static IRandomAccessStream \_incomingStream;

private static IRandomAccessStream \_playingStream;

public Subscriber()

{

//the two streams actually point to the same underlying data

\_incomingStream = new InMemoryRandomAccessStream();

\_playingStream = \_incomingStream.CloneStream();

}

1. Add a method that will allow the Subscriber to connect to the Publisher. In the Subscriber class, create a new private variable that will be the socket that is used to establish a connection.

Create a new async method called ConnectAsync. This method will need two parameters. The first will be the host to which the Subscriber will connect, and the second parameter is the name of the subscriber. This method will create a new StreamSocket, attempt a connection and output "Connected" if successful.

using Windows.Networking.Sockets;  
using Windows.Networking;

using System.Diagnostics;

namespace MusicSubscriber

{

public sealed class Subscriber

{

// two streams in memory.

// One stream is used to receive the file

// the other is used to play the file

private static IRandomAccessStream \_incomingStream;

private static IRandomAccessStream \_playingStream;

private StreamSocket \_socket;

public Subscriber()

{

//the two streams actually point to the same underlying data

\_incomingStream = new InMemoryRandomAccessStream();

\_playingStream = \_incomingStream.CloneStream();

}

public async void ConnectAsync(string host, string name)

{

HostName hostName;

try

{

hostName = new HostName(host);

}

catch (ArgumentException)

{

Debug.WriteLine("Error: Invalid host name {0}.", host);

return;

}

\_socket = new StreamSocket();

\_socket.Control.KeepAlive = true;

\_socket.Control.QualityOfService = SocketQualityOfService.LowLatency;

//hard coded port - can be user-specified but to keep this sample simple it is 21121

await \_socket.ConnectAsync(hostName, "21121");

Debug.WriteLine("Connected");

}

}

1. In MainPage.xaml.cs create a new private variable called subscriber. Then update the MainPage\_Loaded method to replace the //TODO: load in local virtual speaker with the following code:

public sealed partial class MainPage : Page

{

private StorageFile \_mediaFile;

private Publisher \_publisher;

private Subscriber \_subscriber;

public MainPage()

{

this.InitializeComponent();

this.Loaded += MainPage\_Loaded;

}

private async void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

//get the current IP address of this machine and display it

HostIPAddressTextBlock.Text = GetThisIPAddress();

//set up as a publisher and bind the UI to the list of virtual speakers

\_publisher = new Publisher();

MessageLogListView.ItemsSource = \_publisher.Speakers;

bool init = await \_publisher.Initialize();

if (init)

{

//allow a new media file to be selected

SelectMediaFileButton.IsEnabled = true;

// create a virtual speaker for this local machine to listen to the music we send

\_subscriber = new Subscriber();

//hardcoded to localhost and name for local speaker

\_subscriber.ConnectAsync("localhost", "Host Speaker");

}

else

{

Debug.WriteLine("Error: Publisher failed to initialize");

}

}

1. Build and Run the Application (F5). As there is no interface to visually see if the Subscriber has successfully connected, monitor the output windows and look out for the "Connected" message. This will appear when the await call has processed from the Subscriber.
2. To send the subscriber name to the Publisher on connection, update the Connect method inside Subscriber.cs to write the subscriber name to the sockets output stream. To do this first create a new private variable of type DataWriter. Then update the Connect method to write the name to the output stream.

private DataWriter \_writer;

public async void Connect(string host, string name)

{

HostName hostName;

try

{

hostName = new HostName(host);

}

catch (ArgumentException)

{

Debug.WriteLine("Error: Invalid host name {0}.", host);

return;

}

\_socket = new StreamSocket();

\_socket.Control.KeepAlive = true;

\_socket.Control.QualityOfService = SocketQualityOfService.LowLatency;

//hard coded port - can be user-specified but to keep this sample simple it is 21121

await socket.ConnectAsync(hostName, "21121");

Debug.WriteLine("Connected");

//first message to send is the name of this virtual speaker

\_writer = new DataWriter(\_socket.OutputStream);

\_writer.WriteUInt32(\_writer.MeasureString(name));

\_writer.WriteString(name);

try

{

await \_writer.StoreAsync();

Debug.WriteLine("{0} registered successfully.", name);

}

catch (Exception exception)

{

Debug.WriteLine("Send failed with error: " + exception.Message);

// If this is an unknown status it means that the error if fatal and retry will likely fail.

if (SocketError.GetStatus(exception.HResult) == SocketErrorStatus.Unknown)

{

throw;

}

}

}

1. While the Subscriber has now successfully connected to the Publisher, the Publisher is currently not doing anything with the connection message it receives. Implement handling this connection received message.

In Publisher.cs edit the Listener\_ConnectionReceived method, set the method definition to be async and replace the //TODO with the code below.

This method will now try and read the Input stream and create a new speaker from the subscriber name and add the speaker to the collection of speakers. As the interface has a List which is data-bound to the list of speakers, you should see the new speaker added to the list.

using Windows.Storage.Streams;

using Windows.UI.Core;

private async void Listener\_ConnectionReceived(StreamSocketListener sender, StreamSocketListenerConnectionReceivedEventArgs args)

{

Debug.WriteLine("Connection Received on Port {0}", sender.Information.LocalPort);

StreamSocket streamSocket = args.Socket;

if (streamSocket != null)

{

DataReader reader = new DataReader(streamSocket.InputStream);

try

{

// Read first 4 bytes (length of the subsequent string).

uint sizeFieldCount = await reader.LoadAsync(sizeof(uint));

if (sizeFieldCount != sizeof(uint))

{

// The underlying socket was closed before we were able to read the whole data.

return;

}

// Read the length of the 'packet'.

uint length = reader.ReadUInt32();

uint actualLength = await reader.LoadAsync(length);

if (length != actualLength)

{

// The underlying socket was closed before we were able to read the whole data.

return;

}

string name = reader.ReadString(actualLength);

Speaker speaker = new Speaker()

{

Name = name,

Address = streamSocket.Information.RemoteAddress.DisplayName,

Status = "Connected",

Socket = streamSocket

};

await Windows.ApplicationModel.Core.CoreApplication.MainView.CoreWindow.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,

() =>

{

Speakers.Add(speaker);

});

reader.DetachStream();

Debug.WriteLine("New speaker added " + name);

}

catch (Exception e)

{

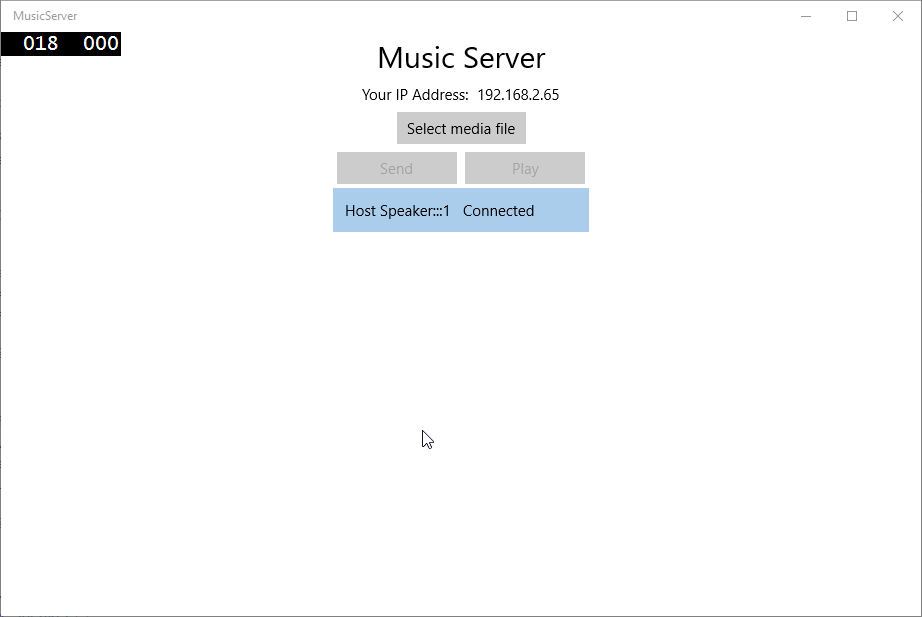
Debug.WriteLine("Error in connection received: " + e);

}

}

}

1. Build and run the application (F5), when the Publisher creates its own speaker this will now appear in the list of speakers. Any additional speakers in the future will also appear here.



# Exercise 4 – Sending a File Stream through a Stream Socket:

A connection has been successfully made between a publisher and a subscriber. Now you need to send a stream of music to all the speakers.

1. In MainPage.xaml.cs implement the SendButton\_Click method. Replace the //TODO with the following code that will read a stream from a media file to a local byte array.

using Windows.Storage.Streams;

private async void SendButton\_Click(object sender, RoutedEventArgs e)

{

// read the file and send it via the publisher

using (IRandomAccessStream stream = await \_mediaFile.OpenReadAsync())

{

byte[] fileBytes = new byte[stream.Size];

using (DataReader reader = new DataReader(stream))

{

await reader.LoadAsync((uint)stream.Size);

reader.ReadBytes(fileBytes);

}

//TODO: call send on the publisher

}

}

1. The fileBytes array needs to be sent to all speakers. As the Publisher class has a collection of speakers it should be responsible to send the fileBytes. It is also useful to know what type of data is being sent. An enum called MessageType will be used to determine the data type sent. In the Solution Explorer right click the project and click **Add | Class**. Name the new class *MessageType.cs*. Then change the class definition to be public enum. This enum will provide information on what the Publisher is sending to the Subscriber.

public enum MessageType

{

Unknown,

Message,

Media,

Play,

Stop,

Ready

}

1. Create a new internal async method in Publisher.cs called Send. This method will loop through the speakers and send the fileBytes in many packets. As there are many packets being sent, firstly a header packet should be sent to notify the speaker about the file about to be sent, this is the place to indicate the message is a media file. This way the Subscriber can implement a method to continuously listen until it has received all packets.

public async void Send(byte[] fileBytes)

{

try

{

if (Speakers != null && Speakers.Count > 0 && fileBytes != null)

{

//iterate through the speakers and send out the media file to each speaker

foreach (Speaker speaker in Speakers)

{

StreamSocket socket = speaker.Socket;

if (socket != null)

{

IOutputStream outStream = socket.OutputStream;

using (DataWriter dataWriter = new DataWriter(outStream))

{

//write header bytes to indicate to the subscriber

//information about the file to be sent

dataWriter.WriteInt16((short)MessageType.Media);

dataWriter.WriteInt32(fileBytes.Length);

await dataWriter.StoreAsync();

//start from 0 and increase by packet size

int partNumber = 0;

int sourceIndex = 0;

int bytesToWrite = fileBytes.Length;

while (bytesToWrite > 0)

{

dataWriter.WriteInt32(partNumber);

int packetSize = bytesToWrite;

if (packetSize > MAX\_PACKET\_SIZE)

{

packetSize = MAX\_PACKET\_SIZE;

}

byte[] fragmentedPixels = new byte[packetSize];

Array.Copy(fileBytes, sourceIndex, fragmentedPixels, 0, packetSize);

dataWriter.WriteBytes(fragmentedPixels);

Debug.WriteLine("sent byte packet length " + packetSize);

await dataWriter.StoreAsync();

sourceIndex += packetSize;

bytesToWrite -= packetSize;

partNumber++;

Debug.WriteLine("sent total bytes " + (fileBytes.Length - bytesToWrite));

}

//Finally DetachStream

dataWriter.DetachStream();

}

}

}

//TODO: check the endpoints have received the packets

}

}

catch (Exception ex)

{

Debug.WriteLine(ex.ToString());

}

}

1. In the MainPage.xaml.cs file call \_publisher.Send in the SendButton\_Click method.

private async void SendButton\_Click(object sender, RoutedEventArgs e)

{

byte[] fileBytes = null;

using (var stream = await mediaFile.OpenReadAsync())

{

fileBytes = new byte[stream.Size];

using (DataReader reader = new DataReader(stream))

{

await reader.LoadAsync((uint)stream.Size);

reader.ReadBytes(fileBytes);

}

\_publisher.Send(fileBytes);

}

}

1. The Subscriber needs to handle receiving the file stream. Update the Connect method in the Subscriber.cs to handle the MessageTypes it cares about. A switch statement will be used to handle the MessageTypes of Media, Play and Stop. Add the code at the bottom of the current Connect method in Subscriber.cs

public async void ConnectAsync(string host, string name)

{

// … previous code above

MessageType currentMessageType = MessageType.Unknown;  
 // then wait for the audio to be sent to us   
 DataReader reader = new DataReader(\_socket.InputStream);  
   
 while (true)  
 {  
 uint x = await reader.LoadAsync(sizeof(Int16));  
 Int16 t = reader.ReadInt16();  
 currentMessageType = (MessageType)t;  
   
 switch (currentMessageType)  
 {  
 case MessageType.Media:  
 await ReadMediaFileAsync(reader);  
 break;  
 case MessageType.Play:  
 {  
 //TODO: implement media play back when Publisher sends play message  
 }  
 break;  
 case MessageType.Stop:  
 {  
 //TODO: implement media stop when Publisher sends stop message  
 }  
 break;  
 default:  
 break;  
 }  
 currentMessageType = MessageType.Unknown;  
 }  
 }

You may notice the Media type is the only type implemented as of yet. The implementation requires a method that hasn't been created so create a new method for this and leave it blank for now.

using System.Threading.Tasks;

private async Task ReadMediaFileAsync(DataReader reader)  
 {   
 }

1. The ReadMediaFile method will continue to listen and read the network stream directly to the incoming stream while there are still bytes of the file to read. This loop will continue to run until all packets have been received. Create two new private class variables that will hold the max packet size and the total bytes read

// this max packet size is used in case of a poor internet connection in order to get the packet in reasonable size transferred

private const uint MAX\_PACKET\_SIZE = 10000;  
private uint \_totalBytesRead = 0;

private async Task ReadMediaFileAsync(DataReader reader)

{

//a media file will always start with an int32 containing the file length

await reader.LoadAsync(sizeof(int));

int messageLength = reader.ReadInt32();

Debug.WriteLine("Message Length " + messageLength);

\_totalBytesRead = 0;

uint bytesRead = 0;

IBuffer readBuffer = new Windows.Storage.Streams.Buffer(MAX\_PACKET\_SIZE);

// read as many blocks as are in the incoming stream - this prevents blocks getting dropped

do

{

await reader.LoadAsync(sizeof(int));

int partNumber = reader.ReadInt32();

Debug.WriteLine("Part " + partNumber);

readBuffer = await \_socket.InputStream.ReadAsync(readBuffer, MAX\_PACKET\_SIZE,

InputStreamOptions.Partial);

bytesRead = readBuffer.Length;

Debug.WriteLine("Bytes read " + bytesRead);

if (bytesRead > 0)

{

\_incomingStream.WriteAsync(readBuffer).GetResults();

\_totalBytesRead += bytesRead;

}

Debug.WriteLine("Total bytes read: " + \_totalBytesRead);

}

while (\_totalBytesRead < messageLength);

Debug.WriteLine("Incoming stream length " + \_incomingStream.Size);

}

1. Once the fileBytes have all been read, the Subscriber can tell the Publisher it has finished receiving the file. At the bottom of the ReadMediaFiles add in the following. This will send a MessageType of **Ready** back to the Publisher.

private async Task ReadMediaFile(DataReader reader)

{

// …

if (\_totalBytesRead >= messageLength)

{

if (\_writer == null)

{

\_writer = new DataWriter(\_socket.OutputStream);

}

\_writer.WriteUInt16((UInt16)MessageType.Ready);

await \_writer.StoreAsync();

messageLength = 0;

}

}

1. The Publisher should listen for the **Ready** message from the Subscriber. When the Publisher receives this message it should update the Subscribers status to ready. Replace the *//TODO: check the endpoints have received the packets* with the following code:

internal async void Send(byte[] fileBytes)  
 {  
 try  
 {  
 if (null != Speakers && Speakers.Count > 0 && null != fileBytes)  
 {  
 //… previous code above

//check the speakers have all received the file

foreach (Speaker speaker in Speakers)

{

StreamSocket socket = speaker.Socket;

if (socket != null)

{

//wait for the 'I got it' message

DataReader reader = new DataReader(socket.InputStream);

uint x = await reader.LoadAsync(sizeof(short));

MessageType t = (MessageType)reader.ReadInt16();

if (MessageType.Ready == t)

{

await Windows.ApplicationModel.Core.CoreApplication.MainView.CoreWindow.Dispatcher.RunAsync(CoreDispatcherPriority.Normal,

() =>

{

Speakers.Remove(speaker);

speaker.Status = "Ready";

Speakers.Add(speaker);

});

}

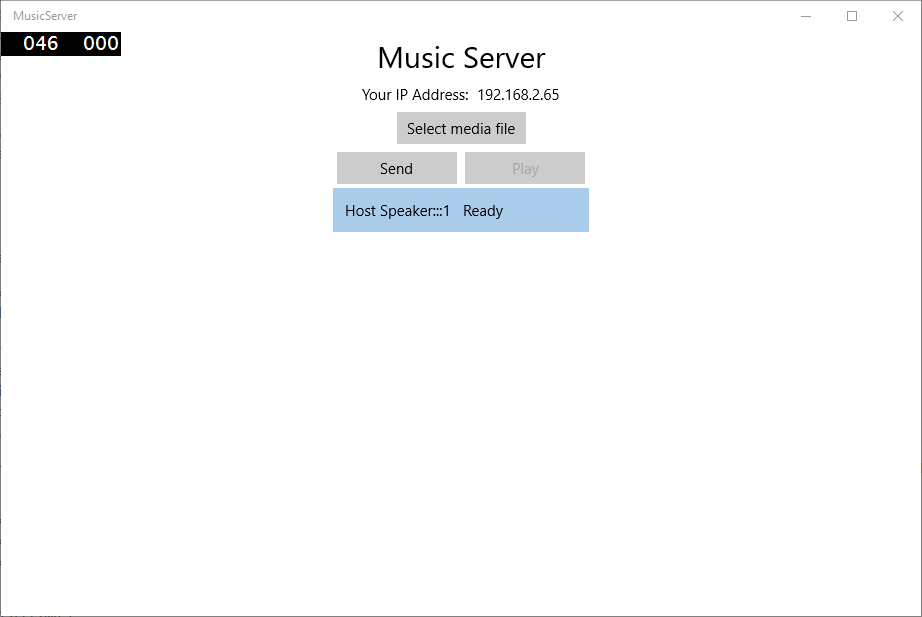
reader.DetachStream();

}

}

}  
 }  
 catch (Exception ex)  
 {  
 Debug.WriteLine(ex.ToString());  
 }  
 }

1. Build and Run the application. Select a media file and press send. In the output you will see that the subscriber is reading the bytes it has been sent. When it has completed receiving the file the Subscribers status will be updated to ready.



# Exercise 5 – Sending Play/Stop Commands:

The file stream has been successfully sent and received and now you are ready to implement the play and stop controls.

1. In MainPage.xaml.cs add the play button functionality first. For this example the play button and stop button will be used as the same button. You can implement separate buttons if you want. Add a new private variable to determine if the media file is playing. Then replace the //TODO in PlayButton\_Click method with the following:

private bool \_playing;

private void PlayButton\_Click(object sender, RoutedEventArgs e)

{

//toggle the state between play and stop

if (!\_playing)

{

//TODO: call publisher play method

PlayButton.Content = "Stop";

}

else

{

//TODO: call publisher stop method

PlayButton.Content = "Play";

}

\_playing = !\_playing;

}

1. Add a state toggle method that will either be passed a Play or Stop message type. Add this toggle method to the Publisher class. This method will iterate through all the Speakers and send a message to each one.

public async void ToggleMediaState(MessageType type)

{

try

{

if (Speakers != null && Speakers.Count > 0)

{

foreach (Speaker speaker in Speakers)

{

StreamSocket socket = speaker.Socket;

if (socket != null)

{

IOutputStream outStream = socket.OutputStream;

using (DataWriter dataWriter = new DataWriter(outStream))

{

dataWriter.WriteInt16((Int16)type);

await dataWriter.StoreAsync();

dataWriter.DetachStream();

}

}

}

}

}

catch (Exception ex)

{

Debug.WriteLine("Error playing: " + ex);

}

}

1. In the MainPage class call the ToggleMediaState method (from Publisher) in the PlayButton\_Click method. For this sample the play and stop button are going to be the same button, so the ToggleMediaState method needs to be called twice in the PlayButton\_Click method, once to play and once to stop. The message type needs to be the opposite of the Content string being set.

private void PlayButton\_Click(object sender, RoutedEventArgs e)

{

//toggle the state between play and stop

if (!\_playing)

{

\_publisher.ToggleMediaState(MessageType.Play);

PlayButton.Content = "Stop";

}

else

{

\_publisher.ToggleMediaState(MessageType.Stop);

PlayButton.Content = "Play";

}

\_playing = !\_playing;

}

1. For the play button to be finished you must update the Connect method of the Subscriber to handle Play messages. This method will get the current media player, set its source and call play. In the switch statement, within the Play case, replace the //TODO with the following code. You will need the Windows.Media.Playback namespace in order to access the Media Player:

using Windows.Media.Playback;

public async void Connect(string host, string name)

{

// …

switch (currentMessageType)

{

case MessageType.Media:

await ReadMediaFile(reader);

break;

case MessageType.Play:

{

MediaPlayer mediaPlayer = BackgroundMediaPlayer.Current;

if (mediaPlayer != null)

{

if (mediaPlayer.CurrentState != MediaPlayerState.Playing)

{

mediaPlayer.SetStreamSource(\_playingStream);

mediaPlayer.Play();

Debug.WriteLine("Player playing. TotalDuration = " +

mediaPlayer.NaturalDuration.Minutes + ':' + mediaPlayer.NaturalDuration.Seconds);

}

}

}

break;

case MessageType.Stop:

{

//TODO: implement media stop when Publisher sends stop message

}

break;

default:

break;

}

currentMessageType = MessageType.Unknown;

}

}

1. In the Connect method of the Subscriber class, add code to handle receiving a Stop message. Update the switch statement in the connect method and replace the //TODO when the case is MessageType.Stop with the following:

public async void Connect(string host, string name)

{

// …

switch (currentMessageType)

{

case MessageType.Media:

await ReadMediaFile(reader);

break;

case MessageType.Play:

{

MediaPlayer mediaPlayer = BackgroundMediaPlayer.Current;

if (mediaPlayer != null)

{

if (mediaPlayer.CurrentState != MediaPlayerState.Playing)

{

mediaPlayer.SetStreamSource(\_playingStream);

mediaPlayer.Play();

Debug.WriteLine("Player playing. TotalDuration = " +

mediaPlayer.NaturalDuration.Minutes + ':' + mediaPlayer.NaturalDuration.Seconds);

}

}

}

break;

case MessageType.Stop:

{

MediaPlayer mediaPlayer = BackgroundMediaPlayer.Current;

if (mediaPlayer != null)

{

if (mediaPlayer.CurrentState == MediaPlayerState.Playing)

{

mediaPlayer.Pause();

Debug.WriteLine("Player paused");

}

}

}

break;

default:

break;

}

currentMessageType = MessageType.Unknown;

}

}

1. The functionality is now implemented for Play and Stop. However, you may have noticed earlier that the play button is disabled by default. This is because play should only be called when all virtual speakers have a status of ready. This way it is known if they have all received the entire media stream and are ready to play. In order to check if all speakers are ready, create a new method in the Publisher class called CheckSpeakersAreReady. This will loop through the list of speakers and check if all the speakers are ready, and if they are ready, then the Play button can be enabled.

public bool CheckSpeakersAreReady()

{

bool speakersReady = true;

foreach (Speaker speaker in Speakers)

{

if (!Equals(speaker.Status, "Ready"))

{

speakersReady = false;

break;

}

}

return speakersReady;

}

1. In the MainPage class code file, at the bottom of the Loaded method register for CollectionChanged event on the publisher.Speakers collection. Then implement the CollectionChanged handler method

private async void MainPage\_LoadedAsync(object sender, RoutedEventArgs e)

{

//get the current IP address of this machine and display it

HostIPAddressTextBlock.Text = GetThisIPAddress();

//set up as a publisher and bind the UI to the list of virtual speakers

\_publisher = new Publisher();

MessageLogListView.ItemsSource = \_publisher.Speakers;

bool init = await \_publisher.Initialize();

if (init)

{

//allow a new media file to be selected

SelectMediaFileButton.IsEnabled = true;

// create a virtual speaker for this local machine to listen to the music we send

\_subscriber = new Subscriber();

//hardcoded to localhost and name for local speaker

\_subscriber.ConnectAsync("localhost", "Host Speaker");

}

else

{

Debug.WriteLine("Error: Publisher failed to initialize");

}

// listen for changes in the collection of virtual speakers

\_publisher.Speakers.CollectionChanged += Speakers\_CollectionChanged;

}

private void Speakers\_CollectionChanged(object sender, System.Collections.Specialized.NotifyCollectionChangedEventArgs e)

{

// if the virtual speakers all have received the media file and are ready to play

if (\_publisher.CheckSpeakersAreReady())

{

//Allow media file to be played if all speakers are ready

PlayButton.IsEnabled = true;

}

else

{

PlayButton.IsEnabled = false;

}

}

1. Build and run the application (F5). Select a media file and click send. When the speaker is Ready press play. The sound will now begin playing and the play button will update to stop. Pressing stop will now stop the sound playing.

# Exercise 6 –Virtual Speaker and Subscriber Runtime Component

The MusicServer Project has a Subscriber built in to play the music on the local host machine. In order to play music on other machines this Subscriber needs to be included in another project that can be run on the other machines. Creating a new application called Virtual Speaker allows you to run the Server on one machine and the Virtual Speaker application on other machines. This way machines can play back the media file at the same time.

Create a new Universal Windows Platform Application

1. In the same solution as the MusicServer project select **File | New | Project** (CTRL + SHIFT + N)
2. This will open a “New Project” window. Navigate to Templates | Visual C# and choose Blank App (Universal Apps) and Name your application VirtualSpeaker
3. This application is going to use the local network to send and receive UDP packets, so Open the Package.appxmanifest in the Solution Explorer and go to the Capabilities tab, and check Private Networks (Client & Server) and Internet (Client & Server):

Machine generated alternative text:
Capabilities: 
All Joyn 
Blocked Chat Messages 
Bluetooth 
Chat Message Access 
Code Generation 
Enterprise Authentication 
Internet (Client) 
Internet (Client & Server) 
Location 
Microphone 
Music Library 
Objects 3D 
Phone Call 
Pictures Library 
Private Networks (Client & Server) 
Proximity 

1. Build and Run the new Application (F5), you should see a blank window shown on the screen. Close the Application.
2. In MainPage.xaml copy this xaml to setup the interface of the Virtual Speaker. The structure needed for the Virtual speaker is very simple. The elements needed are a Title, IPAddress and an area to connect to the MusicServer.

<Grid Background="{ThemeResource ApplicationPageBackgroundThemeBrush}" Margin="8">

<StackPanel Orientation="Vertical" >

<!-- Title -->

<TextBlock x:Name="TitleTextBlock" Text="Virtual Speaker" FontSize="28" HorizontalAlignment="Center" Margin="4"/>

<!-- IP Address of the Host -->

<StackPanel x:Name="IPAddressStackPanel" Orientation="Horizontal" HorizontalAlignment="Center" >

<TextBlock x:Name="IPAddressTextBlock" Text="Your IP Address: " Margin="4"/>

<TextBlock x:Name="HostIPAddressTextBlock" Margin="4"/>

</StackPanel>

<!-- Area to connect to localhost or a Client IP Address -->

<StackPanel x:Name="ConnectStackPanel" Orientation="Vertical">

<TextBox x:Name="HostNameTextBox" PlaceholderText="Server IP Address" Width="160" Margin="4"/>

<TextBox x:Name="SpeakerNameTextBox" PlaceholderText="Speaker Name" Width="160" Margin="4"/>

<Button x:Name="ConnectButton" Content="Connect" Click="ConnectButton\_Click" Width="160" Margin="4" HorizontalAlignment="Center"/>

</StackPanel>

</StackPanel>

</Grid>

1. In the MainPage.xaml.cs code behind file create a new method to handle the ConnectButton click event

private void ConnectButton\_Click(object sender, RoutedEventArgs e)

{

string host = HostNameTextBox.Text;

string name = SpeakerNameTextBox.Text;

}

1. Retrieve the IPAddress of this device for display in the interface using the same code as the MusicServer project. Create a new method called GetThisIPAddress and then register a handler for the loaded event in the constructor. In the loaded event handler assign the returned IP address to the IPAddress TextBlock. You will need the following namespace

using Windows.Networking.Connectivity;

public sealed partial class MainPage : Page

{

public MainPage()

{

this.InitializeComponent();

this.Loaded += MainPage\_Loaded;

}

private void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

HostIPAddressTextBlock.Text = GetThisIPAddress();

}

private void ConnectButton\_Click(object sender, RoutedEventArgs e)

{

string host = HostNameTextBox.Text;

string name = SpeakerNameTextBox.Text;

}

private string GetThisIPAddress()

{

string lastHostName = "";

var hosts = NetworkInformation.GetHostNames();

foreach (var host in hosts)

{

// The last host name is always this computer.

if (host.Type == Windows.Networking.HostNameType.Ipv4)

{

lastHostName = host.DisplayName;

}

}

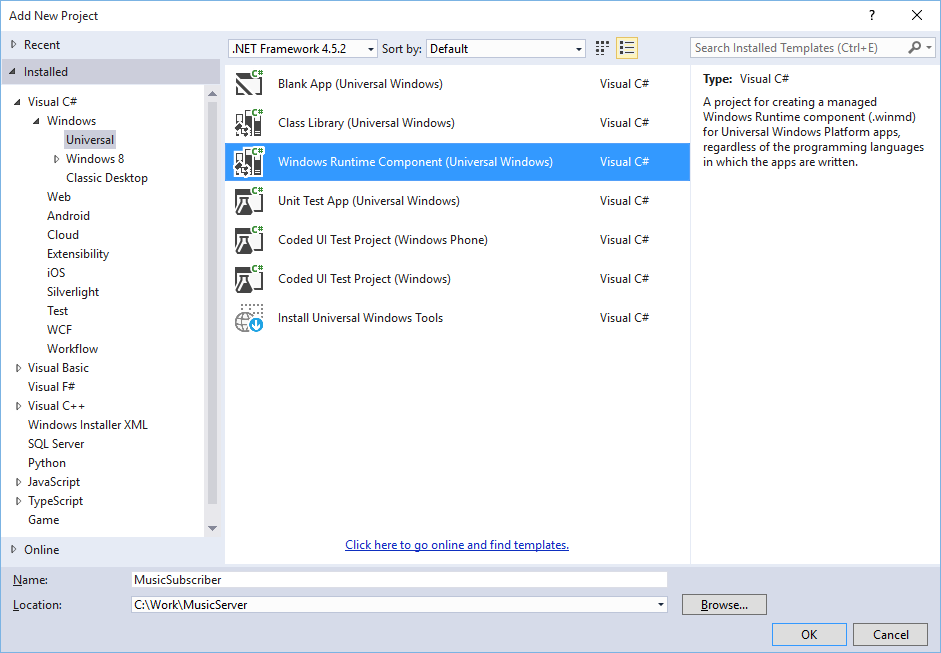
return lastHostName;

}

}

The VirtualSpeaker application will only act as a speaker, so only the Subscriber class is required. As the Subscriber class has been created in the Server project already you should remove this and include it in a runtime component. By doing this the component will be available to both the server and any other virtual speakers.

1. In the same solution as the MusicServer project select **File | New | Project** (CTRL + SHIFT + N)
2. This will open a “New Project” window. Navigate to Templates | Visual C# and choose Windows Runtime Component (Universal Apps) and Name your application MusicSubscriber.



1. In the new MusicSubscriber runtime component create a new class called *Subscriber.cs.* Change the class definition to be public sealed. Inside the class copy in everything that is inside the Subscriber.cs class in the MusicServer project. Your class in the runtime component should now be the same code but be in a difference namespace. You will need the following namespaces when copying across the subscriber class to the runtime component subscriber class.

using System.Diagnostics;

using Windows.Media.Playback;

using Windows.Networking;

using Windows.Networking.Sockets;

using Windows.Storage.Streams;

1. Subscriber.cs has been ported across to the runtime component. Another class you will need to port over is the MessageType enum list. In MusicSubscriber create a new class called *MessageType.cs*. Change the class to look as below

namespace MusicSubscriber

{

public enum MessageType

{

Unknown,

Message,

Media,

Play,

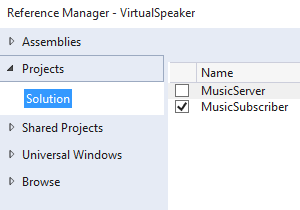
Stop,

Ready

}

}

1. The MusicSubscriber runtime component now has all the classes it needs. To use this runtime component in other projects you will need to add a reference to it. In the VirtualSpeaker project right click on **References | Add Reference…** | **Projects dropdown   
   | Solution.** Tick the MusicSubscriber



1. In the MainPage.xaml.cs code file for the VirtualSpeaker project create a new private variable for the subscriber. You will need to add a reference to the MusicSubscriber. Then update the MainPage loaded method to create a new instance of the Subscriber, and in ConnectButton\_Click call subscriber.Connect:

using MusicSubscriber;

public sealed partial class MainPage : Page

{

private Subscriber \_subscriber;

public MainPage()

{

this.InitializeComponent();

this.Loaded += MainPage\_Loaded;

}

private void MainPage\_Loaded(object sender, RoutedEventArgs e)

{

HostIPAddressTextBlock.Text = GetThisIPAddress();

\_subscriber = new Subscriber();

}

private void ConnectButton\_Click(object sender, RoutedEventArgs e)

{

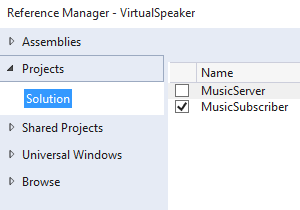
string host = HostNameTextBox.Text;

string name = SpeakerNameTextBox.Text;

\_subscriber.ConnectAsync(host, name);

}

1. The Subscriber and MessageType classes in MusicServer can be removed as the MusicServer can reference the same code in the MusicSubscriber component.   
   Delete the Subscriber.cs class and the MessageType.cs class from the MusicServer project. Add a reference to the MusicSubscriber. In the MusicServer project right click on **References | Add Reference…** | **Projects dropdown   
   | Solution.** Tick the MusicSubscriber



1. Add a reference to the MusicSubscriber namespace in both the MainPage.xaml.cs and the Publisher.cs.
2. The VirtualSpeaker Project is now complete. Build and Run the application (F5).   
   Run the MusicServer application on a different machine on the same local network.   
   In the VirtualSpeaker application that is running, type in the IPAddress of the MusicServer, and a name for the speaker, then click connect. The name of the VirtualSpeaker should appear in the MusicServer list of speakers. On the MusicServer select a media file and click send. When the Virtual Speaker has received the file it will return a Ready message.   
   Click Play on the MusicServer and the media file will be playing locally and on the Virtual Speaker.

In this project you have learned how to send files between machines. You will have discovered how to create a publisher and subscriber model architecture, where multiple subscribers can connect to the same server on a local area network. You have also understood how to play media files in the background of your application.

To extend this application you could explore the following scenarios:

1. Create a playlist of media files in the music server and send them to the virtual speakers. As the first file is received by the speakers it can start to be played while the next file is being sent.
2. Currently the media is played or stopped. Can you work out how to pause the media being played and then continue playing at the point it was paused?
3. The music is not accurately synchronized between virtual speakers. In order to create a synchronized playback you would need to use a timer and create a heartbeat to keep the speakers synchronized with the server. For playlists with more than one song this is going to be important to resolve.