Overview

This hands-on lab will show you how to create a two-process VOIP Windows Universal Application using the WebRTC library.

Objectives

This lab will show you how to create a prototype Windows Runtime application that uses WebRTC to establish audio and video calls. This application will be structured in a two-process model where all VOIP related logic is done in a background process. The application will show an example of the correct way to do VOIP calls in WinRT.

System requirements

You must have the following to complete this lab:

- Microsoft Windows 10
- Microsoft Visual Studio 2015 Update 2
- A PC
- Two Windows 10 devices with a camera and microphone

Setup

You must perform the following steps to prepare your computer for this lab:

- Install Microsoft Windows 10.
- Install Visual Studio 2015 Update 2.

Prerequisites

These labs are targeted to developers who can write and deploy a Universal Windows Application and are familiar with Visual Studio 2015 and C#.

Exercises

This Hands-on lab includes the following exercises.

1. Client and Server With Peer Lists and Peer to Peer Signaling
2. Call, Answer, and Hang Up Without WebRTC
3. Using WebRTC For Audio Calls
4. Video Capture and Background Rendering

Estimated time to complete this lab: 8 to 16 hours.

Exercise 1: Client and Server with Peer Lists and Peer to Peer Signaling

Task 1 – Opening the ChatterBox lab base solution

Open the ChatterBox lab solution in Visual Studio. The solution file is located under **ChatterBox-Base-1\ChatterBox.sln**. The lab solution contains the fundamentals we'll need to build on to a call state machine, and support audio and video.

There are several projects in the solution.

- **BackgroundHost** is a small shell of an executable that serves as a singleton service host for all background tasks.
- **ChatterBox** contains all of the primary code, including the UI, view models, and code to communicate to the background process through **AppService**.
- **Chatterbox.Background** contains the **AppService** endpoint to communicate with the foreground, the call handling and signaling logic, the signaling server, loads the WebRTC library.
- **ChatterBox.Client.WebRTCSwapChainPanel** is a class derived from **SwapChainPanel** and adds support for data binding the swap chain handle.
- **ChatterBoxClient.Universal.BackgroundRenderer** is a C++ project which contains most of the code necessary to render video from the background process to the UI in the foreground process.
- **ChatterBox.Server** is a .NET console application that serves as a signaling relay between peers and provides peer discovery. The server should be accessible by both peers.
- **ChatterBox.Communication** is a wrapper for **ChatterBox.Communication.Shared** allowing it to be used in ChatterBox. Both projects are needed because **ChatterBox.Communication.Shared** is used by the server, but is not a WinRT application.

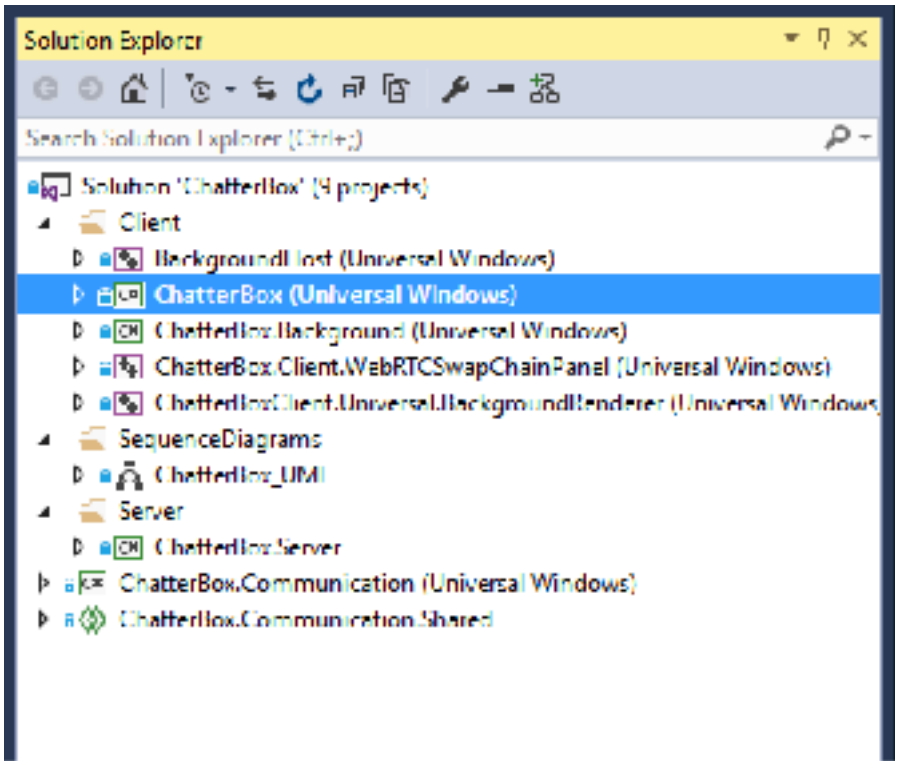

Figure 1

A listing of the projects contained in the ChatterBox solution.

Task 2 – Updating the Manifest

A few changes need to be made to the manifest to support WebRTC as a background process. First create a singleton service entry for the out of process host for all background tasks. Right-click on the **Package.appxmanifest**, and click **View Code**.

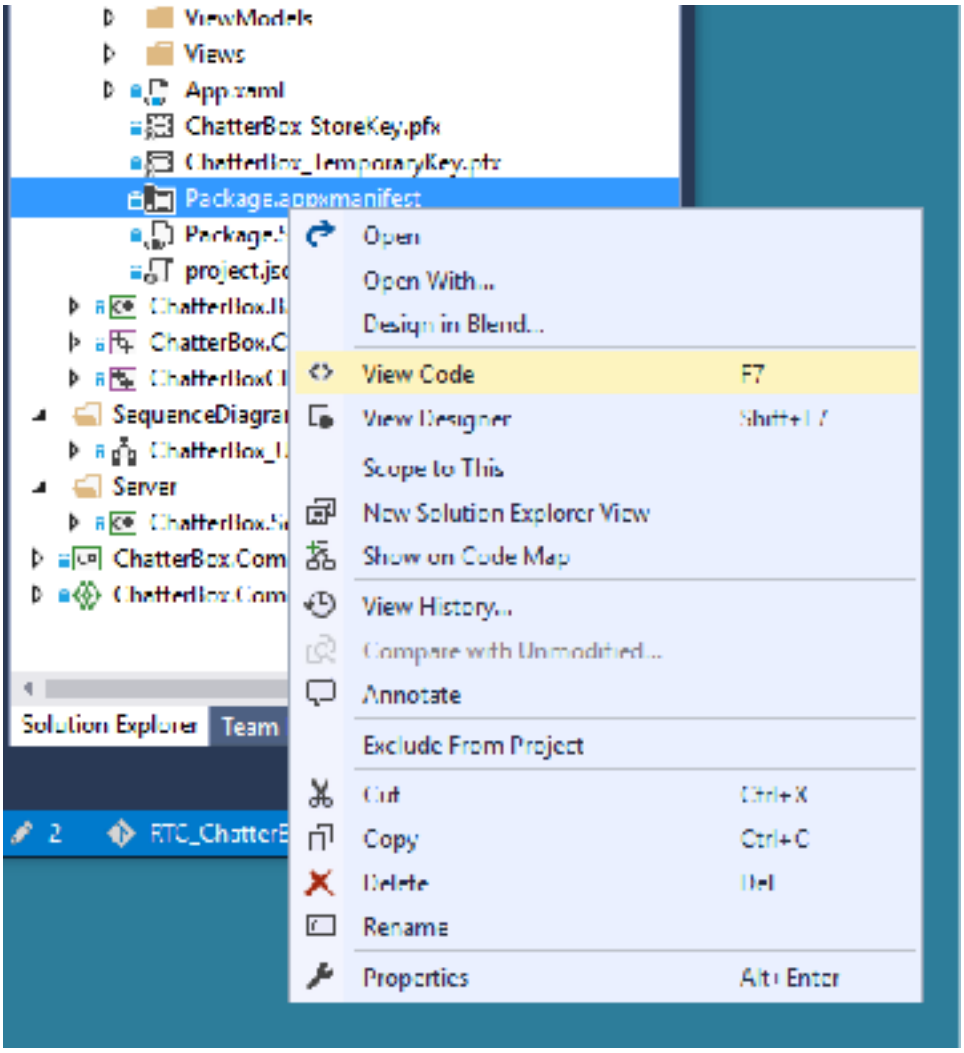

Figure 2
*Right-click and select **View Code** to open a file for editing.*

1. Add the following code to **Package.appxmanifest**.

XML

```
</Applications>

  <Extensions>

    <Extension Category="windows.activatableClass.outOfProcessServer">

      <OutOfProcessServer ServerName="BackgroundHost">

        <Path>BackgroundHost.exe</Path>

        <Instancing>singleInstance</Instancing>

        <ActivatableClass
ActivatableClassId="ChatterBox.BackgroundHost.Dummy" />
```

```
        </OutOfProcessServer>
      </Extension>
    </Extensions>
  </Package>
```

2. The manifest also needs to support communication over sockets and grant access to the audio and video hardware. Close the code view of Package.appxmanifest, and double-click the Package.appxmanifest in the Solution Explorer window.
3. In the **Capabilities** section, enable **Internet (Client)**, **Internet (Client & Server)**, **Microphone**, **Private Networks (Client & Server)**, and **Webcam**.

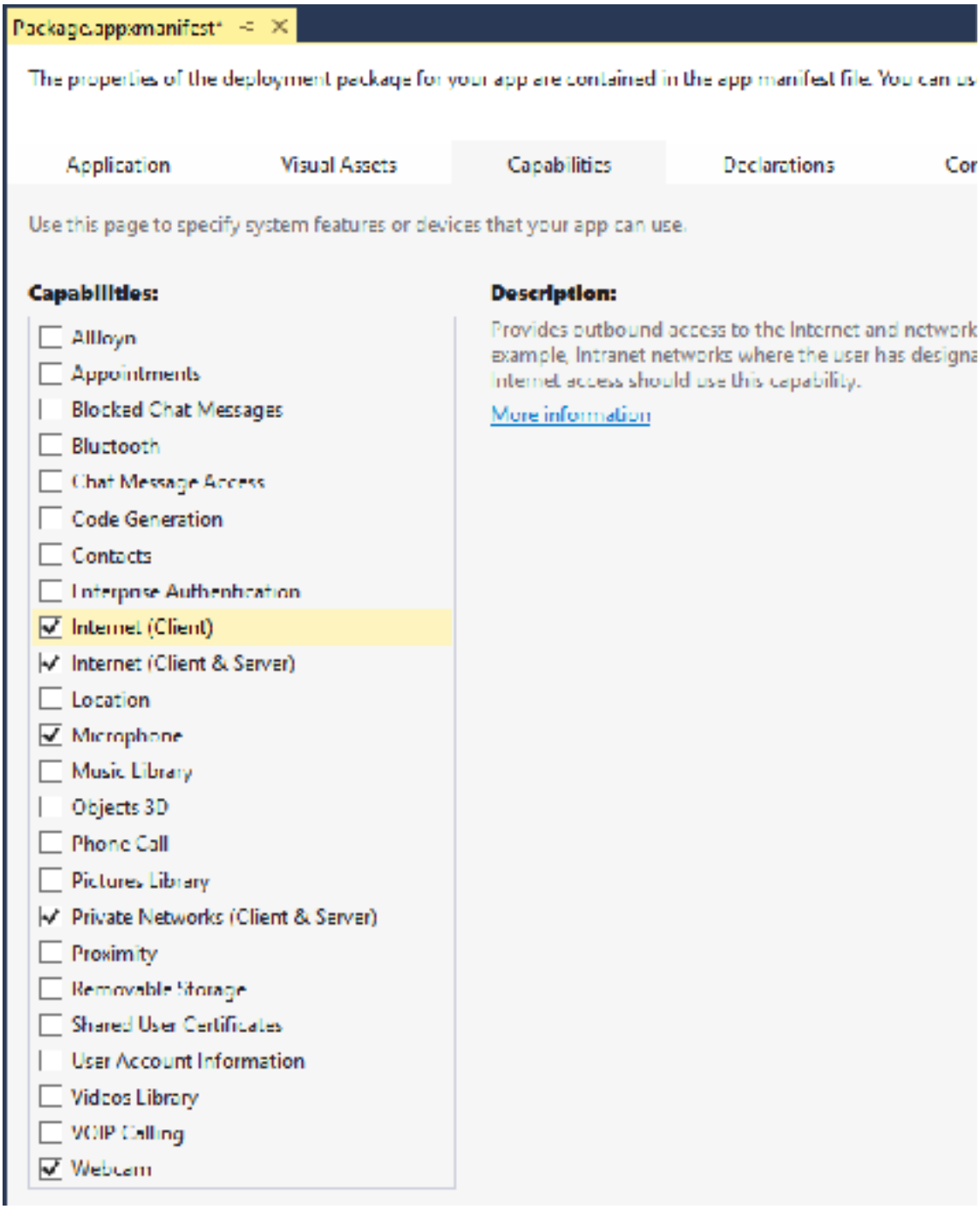

Figure 3
Enable the necessary capabilities for the lab.

Task 3 – Expose an App Service from the background process

1. In Package.appxmanifest, under the **Declarations** section, select **App Service** from the **Available Declarations** menu, and click **Add**.
2. In the **Name** field, type **ForegroundAppServiceTask.**
3. In the **ServerName** field, type **BackgroundHost.**

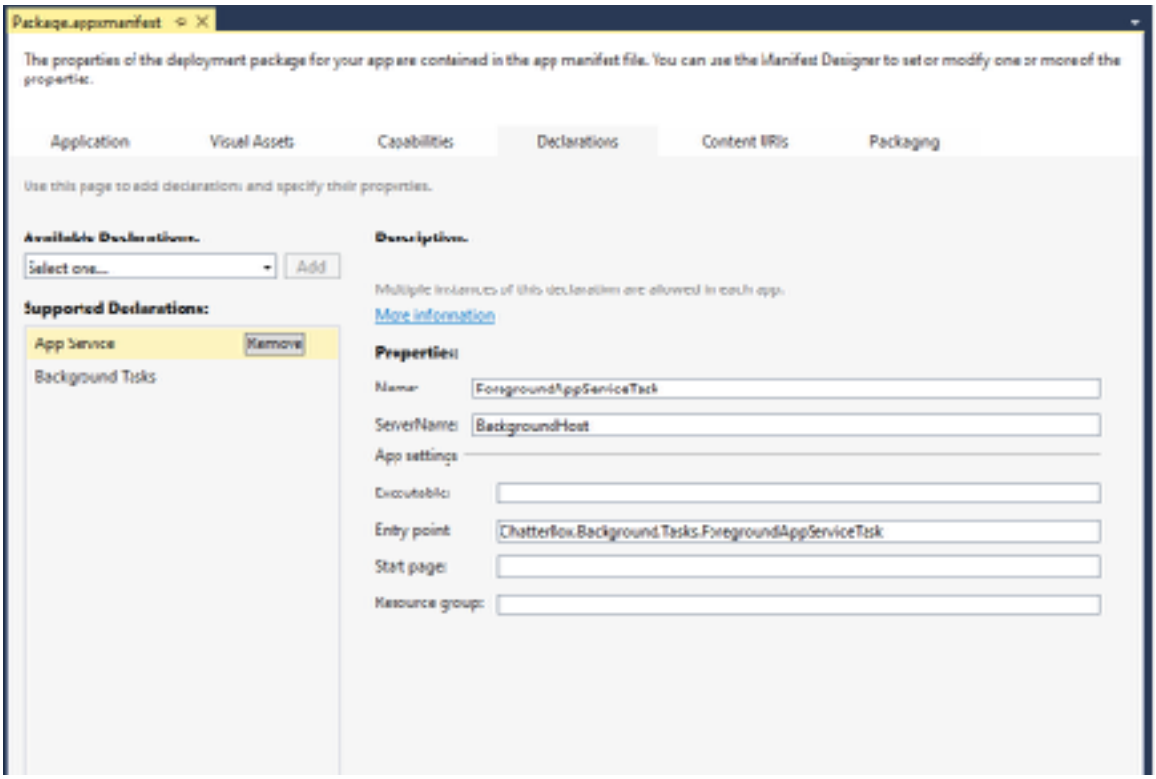

Figure 4

Set the Name, ServerName, and Entry point needed for the lab.

4. In the **Entry point** field, type **ChatterBox.Background.Tasks.ForegroundAppServiceTask**
5. In **ForegroundAppServiceTask.cs** you need to implement a deferral so that the task doesn't automatically close when returning from the `Run()` function and save a reference in the Hub instance.

C#

```
public sealed class ForegroundAppServiceTask : IBackgroundTask
{
    private BackgroundTaskDeferral _deferral;

    public void Run(IBackgroundTaskInstance taskInstance)
    {
        try
        {
            var triggerDetail = (AppServiceTriggerDetails)
taskInstance.TriggerDetails;
            // Keep a deferral to prevent the task from terminating.
            _deferral = taskInstance.GetDeferral();
```

```
            // Save this connection in the hub.  It will be used for
bidirectional communication.
            Hub.Instance.ForegroundConnection =
triggerDetail.AppServiceConnection;
            Hub.Instance.ForegroundTask = this;

            taskInstance.Canceled += (s, e) => Close();
            triggerDetail.AppServiceConnection.ServiceClosed += (s, e)
=> Close();
        }
        catch (Exception e)
        {
            _deferral?.Complete();
            throw;
        }
    }

}
```

6. Following `Run()`, implement a `Close()` function in the **ForegroundAppServiceTask** to release the deferral when the task is cancelled or the app service connection is closed when the foreground process is terminated.

C#

```
public sealed class ForegroundAppServiceTask : IBackgroundTask
{
    private BackgroundTaskDeferral _deferral;

    public void Run(IBackgroundTaskInstance taskInstance)
    {
    ...
    }
```

```
        private void Close()
        {
            Hub.Instance.ForegroundTask = null;
            Hub.Instance.ForegroundConnection = null;
            _deferral?.Complete();
        }
    }
```

Note: In Hub.cs, the setter for ForegroundConnection registers to App Service Requests. The function that handles the requests is already implemented further below in the file. The function will be revisited in the next task when we add a new API interface.

Task 4 - Exposing an API from the background process

Since WebRTC calls will be run inside a background process, you must expose an API layer so that it can be called from the foreground process, including the UI. This allows the UI to start an outgoing call, answer an incoming one, or hang up an active call.

The interface **ICallChannel** declares the functions that will be exposed to the foreground application, and its implementation **CallChannel** implements a state machine designed to manage the various incoming commands against the different possible call states.

1. In **Hub.cs**, add code to handle the case when an App Service request targets the call channel.

```
C#
    /// This Handles requests from the foreground by invoking the requested
    methods on a handler object
    private void HandleForegroundRequest(
        AppServiceConnection sender,
        AppServiceRequestReceivedEventArgs args)
    {
        var deferral = args.GetDeferral();
        try
        {
            //Identify the channel
            var channel = args.Request.Message.Single().Key;
```

```
            //Retrieve the message (format: <Method> <Argument - can be
    null and is serialized as JSON>)
            var message = args.Request.Message.Single().Value.ToString();


            //Invoke the requested method on the handler based on the
    channel type
            if (channel == nameof(ISignalingSocketChannel))
            {
                AppServiceChannelHelper.HandleRequest(args.Request,
    SignalingSocketService, message);
            }


            if (channel == nameof(IClientChannel))
            {
                AppServiceChannelHelper.HandleRequest(args.Request,
    SignalingClient, message);
            }


            if (channel == nameof(ICallChannel))
            {
               AppServiceChannelHelper.HandleRequest(args.Request,
    CallChannel, message);
            }
        }
        finally
        {
            deferral.Complete();
        }
    }
```

2. The `AppServiceChannelHelper` will use reflection to invoke the right function on the right type of call, and asynchronously reply with any return values. The only instance of CallChannel is kept in the hub.

Since the application requests are JSON serialized, which uses reflection, we need to add code to **Default.rd.xml**, and provide the namespace of any DTO types that will be sent over the application service.

In the Solution Explorer, expand **ChatterBox (Universal Windows)**, expand **Properties**, and double-click **Default.rd.xml**.

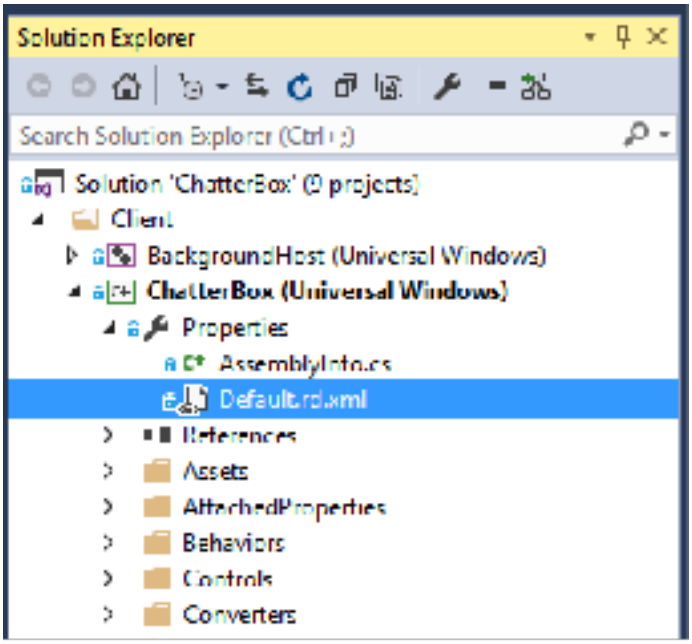

Figure 5
The location of Default.rd.xml in the ChatterBox solution.

3. Add the following code to **Default.rd.xml**.

XML

```
<Directives xmlns="http://schemas.microsoft.com/netfx/2013/01/
metadata">

  <Application>

    <!--

      An Assembly element with Name="*Application*" applies to all
assemblies in

      the application package. The asterisks are not wildcards.

    -->

    <Assembly Name="*Application*" Dynamic="Required All"
Serialize="Required All" />
```

```
    <Namespace Name="ChatterBox.Background.Signaling.Dto" Dynamic="All"
Serialize="All" Browse="All" DataContractJsonSerializer="All"
Activate="All" />

    <Namespace Name="ChatterBox.Background.AppService.Dto"
Dynamic="All" Serialize="All" Browse="All"
DataContractJsonSerializer="All" Activate="All" />

    <Namespace Name="ChatterBox.Background.AppService" Dynamic="All"
Serialize="All" Browse="All" DataContractJsonSerializer="All"
Activate="All" />

    <Namespace Name="ChatterBox.Communication.Messages.Interfaces"
Dynamic="All" Serialize="All" Browse="All"
DataContractJsonSerializer="All" Activate="All" />

    <Namespace Name="ChatterBox.Communication.Messages.Peers"
Dynamic="All" Serialize="All" Browse="All"
DataContractJsonSerializer="All" Activate="All" />

    <Namespace Name="ChatterBox.Communication.Messages.Registration"
Dynamic="All" Serialize="All" Browse="All"
DataContractJsonSerializer="All" Activate="All" />

    <Namespace Name="ChatterBox.Communication.Messages.Standard"
Dynamic="All" Serialize="All" Browse="All"
DataContractJsonSerializer="All" Activate="All" />

    <Namespace Name="ChatterBox.Communication.Contracts" Dynamic="All"
Serialize="All" Browse="All" DataContractJsonSerializer="All"
Activate="All" />


    <Type Name="ChatterBox.Communication.Messages.Relay.RelayMessage"
Dynamic="All" Serialize="All" Browse="All"
DataContractJsonSerializer="All" Activate="All" />

    <Type Name="ChatterBox.Background.SignalingSocketService"
Dynamic="All" Browse="All" />

    <Type Name="ChatterBox.Background.Signaling.SignalingClient"
Dynamic="All" Browse="All" />

    <Type Name="ChatterBox.Background.ForegroundClient" Dynamic="All" /
>

    <Type Name="ChatterBox.Background.Hub" Dynamic="Required All" />

    <Type Name="Windows.Graphics.Display.DisplayOrientations"
Dynamic="All" Serialize="All" Browse="All"
DataContractJsonSerializer="All" Activate="All" />
```

```
    </Application>
</Directives>
```

Task 5 - Add a proxy in the foreground for ICallChannel

In the foreground project, the `HubClient` class implements `ICallChannel`. This class will proxy all calls to the ICallChannel interface to the background process via the app service connection.

An example of this can be seen in the following code.

C#

```
public IAsyncAction CallAsync(OutgoingCallRequest request)
{
    return
InvokeHubChannelAsync<ICallChannel>(request).AsTask().AsAsyncAction();
}
```

In **App.xaml.cs**, register `HubClient` as the implementer of `ICallChannel` using Unity. You must do this in the `OnLaunched()` function.

C#

```
Container.RegisterInstance<ISignalingSocketChannel>(Container.Resolve<H
ubClient>(),
  new ContainerControlledLifetimeManager());
Container.RegisterInstance<IClientChannel>(Container.Resolve<HubClient>
(),
  new ContainerControlledLifetimeManager());
Container.RegisterInstance<ICallChannel>(Container.Resolve<HubClient>()
,
  new ContainerControlledLifetimeManager());
Container.RegisterType<ISocketConnection, SocketConnection>(new
ContainerControlledLifetimeManager());
Container.RegisterType<MainViewModel>(new
ContainerControlledLifetimeManager());
Container.RegisterType<SettingsViewModel>(new
ContainerControlledLifetimeManager());
```

Task 6 - Add a connection to the server

This solution contains a server that does peer discovery and message relays between peers. The application needs to be able to receive incoming calls regardless of whether the

application is running or not. To support that, once a TCP connection is established with the server, the socket is handed off to the socket brokering capabilities of WinRT.

A partially implemented signaling task already exists in the solution. When packets are received, Windows instantiates an instance of the **SignalingTask** in the background process. If no background process is running, then a process will be started automatically.

You must specify the process that will host this task, which cannot be done in the manifest visual editor. Open **Package.appxmanifest** by right clicking on the file name in the solution explorer and choosing **View Code.**

1. Add the following code to the manifest as a background task supporting system events.

XML

```
<Extensions>

  <uap:Extension Category="windows.appService"
EntryPoint="ChatterBox.Background.Tasks.ForegroundAppServiceTask">

    <uap:AppService Name="ForegroundAppServiceTask"
ServerName="BackgroundHost" />

  </uap:Extension>

  <Extension Category="windows.backgroundTasks"
EntryPoint="ChatterBox.Background.Tasks.SignalingTask">

    <BackgroundTasks ServerName="BackgroundHost">

      <Task Type="systemEvent" />

    </BackgroundTasks>

  </Extension>

  <Extension Category="windows.backgroundTasks"
EntryPoint="ChatterBox.Background.Tasks.SessionConnectedTask">

    <BackgroundTasks ServerName="BackgroundHost">

      <Task Type="general" />

      <Task Type="systemEvent" />

    </BackgroundTasks>

  </Extension>

</Extensions>
```

The signaling is done in the background process, so you now have `SignalingSocketChannel` acting as an interface to implement server connecting and disconnecting. The signaling channel, like the call channel, also has a proxy in HubClient.cs.

2. Add the `OwnerId` as the `TaskId` of the SignalingTask within the `ConnectToSignalingServerAsync` implementation in **HubClient.cs**.

C#

```
public IAsyncOperation<ConnectionStatus>
ConnectToSignalingServerAsync(ConnectionOwner connectionOwner)

{

    return InvokeHubChannelAsync<ISignalingSocketChannel,
ConnectionStatus>(new ConnectionOwner

    {

        OwnerId =
_taskHelper.GetTask(nameof(SignalingTask)).TaskId.ToString()

    });

}
```

ConnectionOwner contains the Global Unique Identifier, or **guid**, of the SignalingTask registered in the manifest. The guid is necessary to associate the socket to the signaling task as ownership is transferred to Windows.

3. In **SignalingSocketChannel.cs**, create the socket, enable socket brokering, and transfer the ownership of the socket to the operating system.

C#

```
public IAsyncOperation<ConnectionStatus>
ConnectToSignalingServerAsync(ConnectionOwner connectionOwner)

{

    return Task.Run(async () =>

    {

        try

        {

            SignaledPeerData.Reset();

            SignalingStatus.Reset();


            var socket = new StreamSocket();


socket.EnableTransferOwnership(Guid.Parse(connectionOwner.OwnerId),

                SocketActivityConnectedStandbyAction.Wake);


            var connectCancellationTokenSource = new
CancellationTokenSource(2000);

            var connectAsync = socket.ConnectAsync(new
HostName(SignalingSettings.SignalingServerHost),
```

```
                    SignalingSettings.SignalingServerPort,
SocketProtectionLevel.PlainSocket);

                var connectTask =
connectAsync.AsTask(connectCancellationTokenSource.Token);

                await connectTask;



socket.TransferOwnership(SignalingSocketOperation.SignalingSocketId);

                return new ConnectionStatus

                {

                    IsConnected = true

                };

            }

            catch (Exception exception)

            {

                Debug.WriteLine("Failed to connect to signalling server:
ex: " + exception.Message);

                return new ConnectionStatus

                {

                    IsConnected = false

                };

            }

        }).AsAsyncOperation();

    }
```

Now the socket is controlled by the operating system. When packets come over the socket, Windows starts an instance of SignalingTask. The task takes back control of the socket, reads the packets, gives control back to the operating system, and handles the request that was read. SignalingTask must also take a deferral to prevent the operating system from terminating the task before it's completed.

4. In **SignalingTask.cs**, add the following code to read from the socket.

C#

```
public async void Run(IBackgroundTaskInstance taskInstance)

{

    using (new
BackgroundTaskDeferralWrapper(taskInstance.GetDeferral()))
```

```
        {
            try
            {
                var details = (SocketActivityTriggerDetails)
taskInstance.TriggerDetails;
                switch (details.Reason)
                {
                    case SocketActivityTriggerReason.SocketActivity:

                        string request = null;
                        using (var socketOperation =
Hub.Instance.SignalingSocketChannel.SocketOperation)
                        {
                            if (socketOperation.Socket != null)
                            {
                                var socket = socketOperation.Socket;

                                const uint length = 65536;
                                var readBuf = new Buffer(length);

                                var readOp =
socket.InputStream.ReadAsync(readBuf, length,
                                                        InputStreamOptions.Partial);
                        // This delay is to limit how long we wait for
reading.
                        // StreamSocket has no ability to peek to see if
there's
                       // data waiting to be read, so you must limit it
this way.
                                for (var i = 0; i < 100 && readOp.Status ==
AsyncStatus.Started; ++i)
                                {
                                    await Task.Delay(10);
```

```
                                }
                                await socket.CancelIOAsync();


                                try
                                {
                                    var localBuffer = await readOp;
                                    var dataReader =
DataReader.FromBuffer(localBuffer);
                                    request =
dataReader.ReadString(dataReader.UnconsumedBufferLength);
                                }
                                catch (Exception ex)
                                {
                                    Debug.WriteLine($"ReadAsync exception:
{ex}");
                                }
                            }
                        }
                        if (request != null)
                        {
                            await
Hub.Instance.SignalingClient.HandleRequest(request);
                        }
                        break;
                    case SocketActivityTriggerReason.KeepAliveTimerExpired:
                        await
Hub.Instance.SignalingClient.ClientHeartBeatAsync();
                        break;
                    case SocketActivityTriggerReason.SocketClosed:
                        await
Hub.Instance.SignalingClient.ServerConnectionErrorAsync();
                        //
ToastNotificationService.ShowToastNotification("Disconnected.");
```

```
                        break;
                }
            }
            catch (Exception exception)
            {
                await
Hub.Instance.SignalingClient.ServerConnectionErrorAsync();
            }
        }
    }
```

The helper class SignalingSocketOperation takes and returns control of the socket and implements IDisposable. If the application needs to send packets over the socket, it can use this class to take ownership of the socket.

Task 7 - Handle server messages

The `SignalingClient` class receives the incoming server messages. You must handle the requests by extracting the function names and parameters from the server message and use reflection to invoke the matching function.

Add the following code to **SignalingClient.cs**.

C#

```
public IAsyncAction HandleRequest(string request)
{
    return Task.Run(async () =>
    {
        List<string> requests;
        // Large requests can be split into several packets.
        // We use a file to buffer requests until we can match
        // an Environment.NewLine indicating the end of a request.
        var fileExists = await BufferFileExists();
        if (fileExists)
        {
            var bufferFile = await GetBufferFile();
            await FileIO.AppendTextAsync(bufferFile, request);
```

```
            requests = (await
FileIO.ReadLinesAsync(bufferFile)).ToList();
            await bufferFile.DeleteAsync();
        }
        else
        {
            requests = request.Split(new[] {Environment.NewLine},
                StringSplitOptions.RemoveEmptyEntries).ToList();
        }

        for (var i = 0; i < requests.Count; i++)
        {
            // ServerChannelInvoker will invoke the function on "this"
which matches
            // the request.
            // If the result is asynchronous, await it.
            var invoked =
ServerChannelInvoker.ProcessRequest(requests[i]);
            if (i != requests.Count - 1) continue;
            if (invoked.Invoked)
            {
                var invokeResult = invoked.Result;
                var asyncAction = invokeResult as IAsyncAction;
                if (asyncAction != null)
                {
                    await asyncAction;
                }
                continue;
            }
            // Invocation failed, probably because the request string
is partial.
            // Put it back in the buffer file to wait for more packets.
```

```
                var bufferFile = await GetBufferFile();
                await FileIO.AppendTextAsync(bufferFile, requests[i]);
            }
        }).AsAsyncAction();
    }
```

ServerHeartBeatAsync() is periodically invoked by the server to monitor peer connectivity, OnPeerListAsync() and OnPeerPresenceAsync() inform the server of peer presence, and ServerRelayAsync() is invoked when a peer sends a message via the server relay.

Task 8 – Build and run

Once you build and run your application, it will be able to connect to the signaling server and see a list of the connected peers.

1. Start the server by finding ChatterBox.Server.exe and launching it from Windows Explorer.
2. Start ChatterBox and access the settings page by clicking the gear icon.
3. Enter the IP address of the server in the **Signaling Server Host** field.
4. Click **Save.**
5. Click **Reconnect** and your device will connect to the signaling server.

Repeat the process with a second device to connect it to the signaling server. Each instance of the application should be able to see the other one in the Chatter Members list.

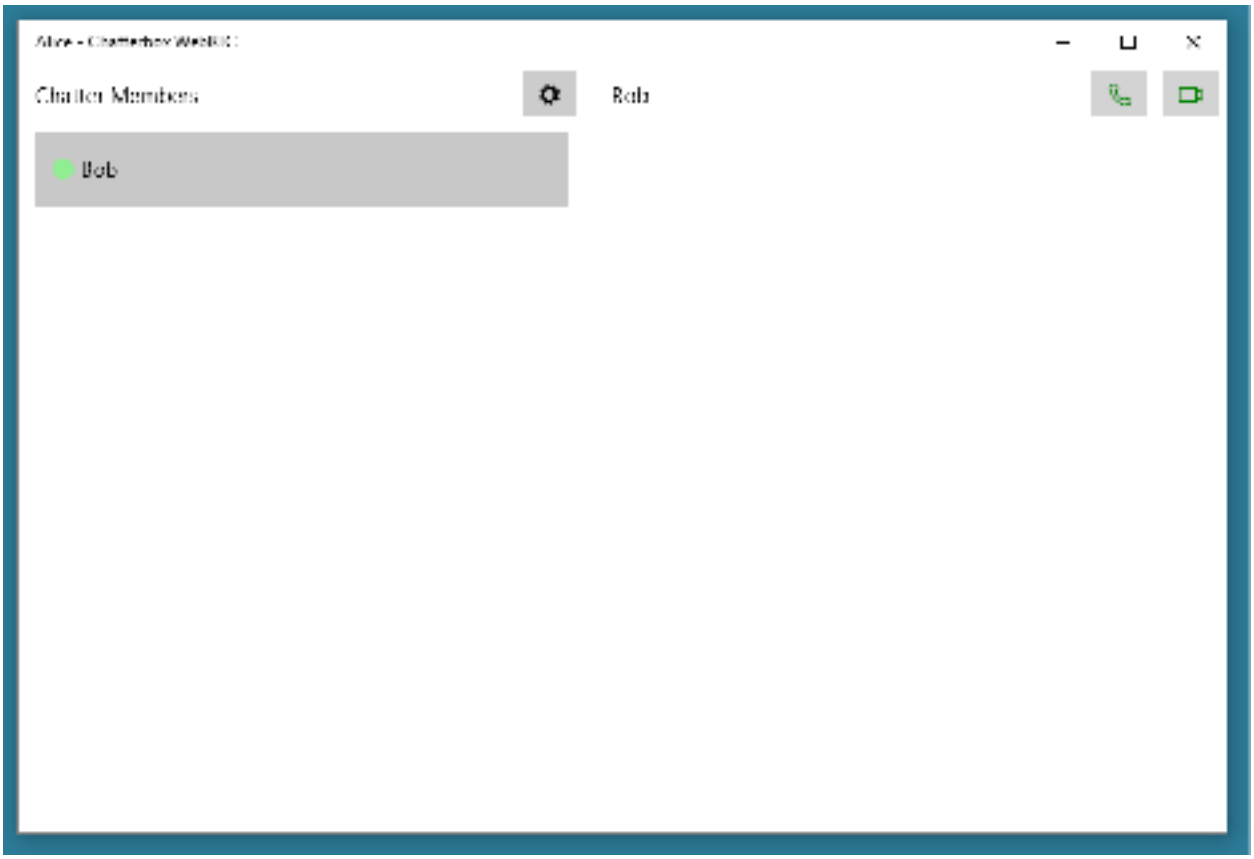

Figure 6
The Chatterbox application running on a Windows 10 device.

Exercise 2: Call, answer, and hang up without WebRTC

Task 1 – Implement starting a call in the state machine

A state machine tracks the state of calls and handles requests coming from the UI as well as the remote peer. The states are able to transition to other states based on the requests sent to the state machine. In this task, you will implement the flow of a relayed call message between two peers.

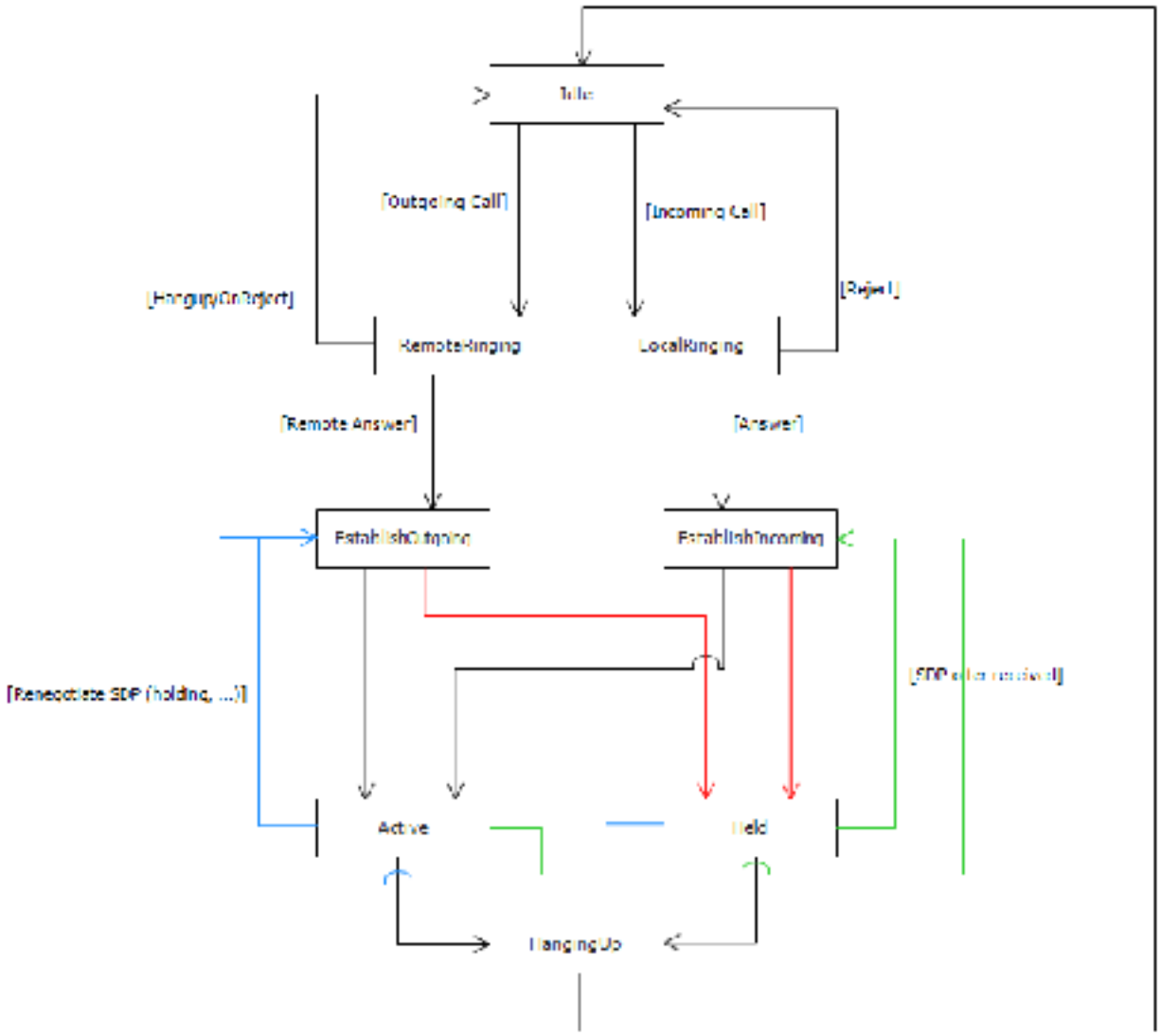

Figure 7
For sequence diagrams detailing the establishing of a call or client-server communication, please refer to the sequence diagrams in the lab solution.

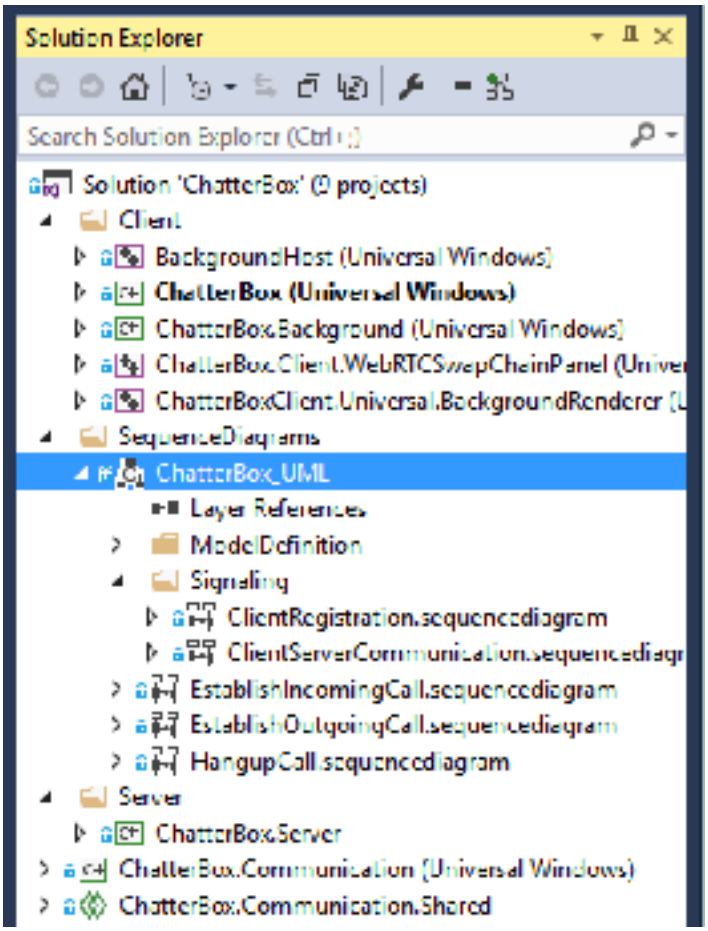

Figure 8
The location of the sequence diagrams in the ChatterBox solution.
.

1. Now you will implement the flow of a relayed call message between peers. In **idle.cs**, when initiating a call, add the following code to switch the state from Idle to RemoteRinging.

C#

```
public override async Task CallAsync(OutgoingCallRequest request)
{
    var remoteRingingState = new RemoteRinging(request);
    await Context.SwitchState(remoteRingingState);
}
```

2. In **RemoteRinging.cs**, add the following code to send a **Call** message to the remote peer.

C#

```
public override async Task OnEnteringStateAsync()
{
```

```
        Debug.Assert(Context.PeerConnection == null);


        Context.PeerId = _request.PeerUserId;


        var payload = JsonConvert.SerializeObject(_request);
        Context.SendToPeer(RelayMessageTags.Call, payload);


        _callTimeout = new Timer(CallTimeoutCallback, null, 30000,
    Timeout.Infinite);


        Context.CallType = _request.VideoEnabled ? CallType.AudioVideo :
    CallType.Audio;
    }
```

3. The `Relay()` function on the Hub is used to send a message to the peer. A RelayMessage is created with a tag indicating the action, as well as include an optional JSON payload. You must implement `SendToPeer()` in **CallContext.cs**.

C#

```
public void SendToPeer(string tag, string payload)
{
    if (PeerId == null)
        return;
    _hub.Relay(new RelayMessage
    {
        FromUserId = RegistrationSettings.UserId,
        ToUserId = PeerId,
        Tag = tag,
        Payload = payload
    });
}
```

4. The JSON payload contains tags that determine which behavior should be used by the receiving client, which can be **Call**, **CallAnswer**, **CallHangup**, and

CallReject. Each of these tags tie into the appropriate function on the ICallChannel.
Add a case for handling the **Call** tag in **SignalingClient.cs**.

C#

```
public IAsyncAction ServerRelayAsync(RelayMessage message)
{
    return Task.Run(async () =>
    {
        await ClientConfirmationAsync(Confirmation.For(message));

        // Handle call tags
        if (message.Tag == RelayMessageTags.Call)
        {
            await _callChannel.OnIncomingCallAsync(message);
        }
        else if (message.Tag == RelayMessageTags.CallAnswer)
        {
            await _callChannel.OnOutgoingCallAcceptedAsync(message);
        }
        else if (message.Tag == RelayMessageTags.CallReject)
        {
            await _callChannel.OnOutgoingCallRejectedAsync(message);
        }
        else if (message.Tag == RelayMessageTags.CallHangup)
        {
            await _callChannel.OnRemoteHangupAsync(message);
        }
    }).AsAsyncAction();
}
```

5. In **Idle.cs**, you must add a transition to switch state when receiving an incoming call.

C#

```
public override async Task IncomingCallAsync(RelayMessage message)
{
    var localRingingState = new LocalRinging(message);
    await Context.SwitchState(localRingingState);
}
```

Task 2 – Start and Stop VoipTask

Background processes in WinRT have very limited access to memory, processor, and network resources by default. In order to support a VOIP call running as a background task, you must have a special background task active during a call.

1. Add an entry for the VOIP task in **Package.appxmanifest**.

Note: The ServerName, in this case, is the BackgroundHost single instance server.

XML

```
<Extension Category="windows.backgroundTasks"
EntryPoint="ChatterBox.Background.Tasks.SignalingTask">
  <BackgroundTasks ServerName="BackgroundHost">
    <Task Type="systemEvent" />
  </BackgroundTasks>
</Extension>
<Extension Category="windows.backgroundTasks"
EntryPoint="ChatterBox.Background.Tasks.VoipTask">
  <BackgroundTasks ServerName="BackgroundHost">
    <uap:Task Type="phoneCall" />
  </BackgroundTasks>
</Extension>
<Extension Category="windows.backgroundTasks"
EntryPoint="ChatterBox.Background.Tasks.SessionConnectedTask">
  <BackgroundTasks ServerName="BackgroundHost">
    <Task Type="general" />
    <Task Type="systemEvent" />
  </BackgroundTasks>
</Extension>
```

2. Double-click on **Package.appxmanifest**.
3. From the **Capabilities** tab, enable **VOIP Calling**.

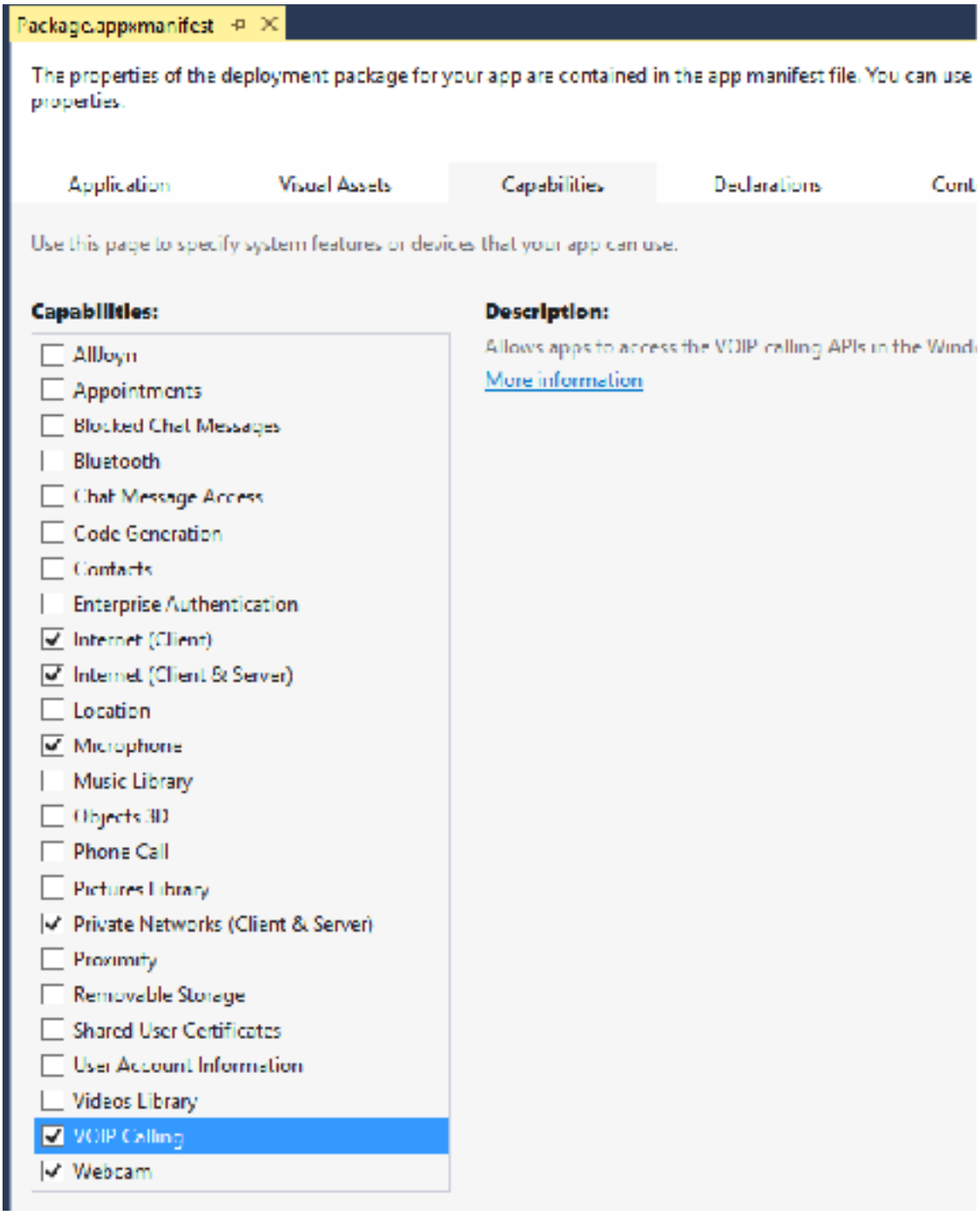

Figure 9

Enable VOIP calling on the Capabilities tab.

4. In addition to adding the VOIP task, additional flags need to be added directly to the manifest file. Add the following code to **Package.appxmanifest**.

XML

```
<Extensions>

  <uap:Extension Category="windows.protocol">
```

```
    <uap:Protocol Name="ms-voip-video" />
  </uap:Extension>
  <uap:Extension Category="windows.protocol">
    <uap:Protocol Name="ms-voip-call" />
  </uap:Extension>
  <uap:Extension Category="windows.voipCall" />
  <uap:Extension Category="windows.appService"
EntryPoint="ChatterBox.Background.Tasks.ForegroundAppServiceTask">
    <uap:AppService Name="ForegroundAppServiceTask"
ServerName="BackgroundHost" />
  </uap:Extension>
```

5. You must now implement VoipTask, which will keep a deferral that remains active for the duration of the call. When the call ends, the deferral is completed. Add the following code to **VoipTask.cs**.

C#

```
public sealed class VoipTask : IBackgroundTask
{
    private BackgroundTaskDeferral _deferral;

    public void Run(IBackgroundTaskInstance taskInstance)
    {
        try
        {
            if (Hub.Instance.VoipTaskInstance != null)
            {
                Debug.WriteLine("VoipTask already started.");
                return;
            }

            _deferral = taskInstance.GetDeferral();
            Hub.Instance.VoipTaskInstance = this;
            Debug.WriteLine($"{DateTime.Now} VoipTask started.");
```

```
            taskInstance.Canceled += (s, e) => CloseVoipTask();
        }
        catch (Exception e)
        {
            _deferral?.Complete();
            throw;
        }
    }

    public void CloseVoipTask()
    {
        Debug.WriteLine($"{DateTime.Now} VoipTask closed.");
        Hub.Instance.VoipTaskInstance = null;
        _deferral?.Complete();
        _deferral = null;
    }
}
```

6. A VOIP task must be started when a call is being established. It will occur when leaving the Idle state to transition to the IncomingCall or OutgoingCall state. This can be seen in the Idle.OnLeavingStateAsync() function. The VoipHelper class manages the lifetime of the voip task, and voip coordinator, which is covered in Task 3.
   In **VoipHelper.cs**, implement the StartVoipTask function.

C#

```
public async Task StartVoipTask()
{
    // Make sure there isn't already a voip task active and the
contract is available.
    if (Hub.Instance.VoipTaskInstance == null &&

ApiInformation.IsApiContractPresent("Windows.ApplicationModel.Calls.Cal
lsVoipContract", 1))
    {
        var vcc = VoipCallCoordinator.GetDefault();
```

```
        var voipEntryPoint = typeof (VoipTask).FullName;
        try
        {
            var status = await
vcc.ReserveCallResourcesAsync(voipEntryPoint);
            Debug.WriteLine($"ReserveCallResourcesAsync
{voipEntryPoint} result -> {status}");
        }
        catch (Exception ex)
        {
            const int rtcTaskAlreadyRunningErrorCode = -2147024713;
            if (ex.HResult == rtcTaskAlreadyRunningErrorCode)
            {
                Debug.WriteLine("VoipTask already running");
            }
            else
            {
                Debug.WriteLine($"ReserveCallResourcesAsync error ->
{ex.HResult} : {ex.Message}");
                throw;
            }
        }
    }
}
```

7. After the call is completed, the application must return to the idle call state and stop the voip task. Add the StopVoip function to VoipHelper.

C#

```
public void StopVoip()
{
    Hub.Instance.VoipTaskInstance?.CloseVoipTask();
}
```

Task 3 – Use the VoipCallCoordinator

In addition to the voip task, you must also create an instance of VoipPhoneCall to notify Windows of the call state and to support the call related events Windows can trigger.

1. In the **VoipHelper** class, add the functions that use VoipCallCoordinator to notify Windows of an incoming or outgoing call. If the application's UI is visible and the call is answered using the UI, then you should not use VoipCallCoordinator to request a new incoming call. This is because there is no way to mark an incoming call other than answering it directly via the OS prompt. As a workaround, you will use StartOutgoingCall instead.

C#

```
public async Task StartIncomingCallAsync(RelayMessage message)

{

        if (message == null) throw new
ArgumentNullException(nameof(message));



        // Check if the foreground UI is visible.

        // If it is then we don't trigger an incoming call because the
call

        // can be answered from the UI and VoipCallCoordinator doesn't
handle

        // this case.

        // As a workaround, when SetCallActive() is called when
answered from the UI

        // we create an instance of an outgoing call.

        var foregroundIsVisible = false;

        var state = await
Hub.Instance.ForegroundClient.GetForegroundStateAsync();

        if (state != null) foregroundIsVisible =
state.IsForegroundVisible;

        if (!foregroundIsVisible)

        {

            var voipCallCoordinator = VoipCallCoordinator.GetDefault();

            _voipCall = voipCallCoordinator.RequestNewIncomingCall(

                message.FromUserId, message.FromName,

                message.FromName,

                null, "ChatterBox", null, "", null,
```

```
                VoipPhoneCallMedia.Audio,
                new TimeSpan(0, 1, 20));

            SubscribeToVoipCallEvents();
        }
    }

    public void StartOutgoingCall(string userId, bool videoEnabled)
    {
        var capabilities = VoipPhoneCallMedia.Audio;
        if (videoEnabled)
        {
            capabilities |= VoipPhoneCallMedia.Video;
        }
        var voipCallCoordinator = VoipCallCoordinator.GetDefault();
        _voipCall = voipCallCoordinator.RequestNewOutgoingCall(
            userId, userId,
            "ChatterBox",
            capabilities);

        if (_voipCall == null) return;
        SubscribeToVoipCallEvents();
        // Immediately set the call as active.
        _voipCall.NotifyCallActive();
    }
```

2. Register VoipCallCoordinator events on the VoipPhoneCall, and forward those events to the call state machine.

C#

```
    private void SubscribeToVoipCallEvents()
    {
        if (_voipCall != null)
```

```
            {
                _voipCall.AnswerRequested += Call_AnswerRequested;
                _voipCall.EndRequested += Call_EndRequested;
                _voipCall.HoldRequested += Call_HoldRequested;
                _voipCall.RejectRequested += Call_RejectRequested;
                _voipCall.ResumeRequested += Call_ResumeRequested;
            }
        }

        private async void Call_AnswerRequested(VoipPhoneCall sender,
        CallAnswerEventArgs args)
        {
            _voipCall.NotifyCallActive();
            await Hub.Instance.CallChannel.AnswerAsync();
        }

        private async void Call_EndRequested(VoipPhoneCall sender,
        CallStateChangeEventArgs args)
        {
            await Hub.Instance.CallChannel.HangupAsync();
        }

        private async void Call_HoldRequested(VoipPhoneCall sender,
        CallStateChangeEventArgs args)
        {
            await Hub.Instance.CallChannel.HoldAsync();
        }

        private async void Call_RejectRequested(VoipPhoneCall sender,
        CallRejectEventArgs args)
        {
            await Hub.Instance.CallChannel.RejectAsync(new IncomingCallReject
```

```
        {
            Reason = "Rejected"
        });
    }

    private async void Call_ResumeRequested(VoipPhoneCall sender,
    CallStateChangeEventArgs args)
    {
        await Hub.Instance.CallChannel.ResumeAsync();
    }
```

3. You must also notify Windows when stopping a call through the StopVoip() function.

C#

```
    public void StopVoip()
    {
        if (_voipCall != null)
        {
            _voipCall.NotifyCallEnded();
            _voipCall = null;
        }

        Hub.Instance.VoipTaskInstance?.CloseVoipTask();
    }
```

Task 4 - Build and Run the application

The application will now allow the peers to establish a call session. If the UI is visible, an incoming call will appear there. If the UI is not visible, Windows will provide a prompt to let you select whether to answer or reject a call. At this point, no WebRTC audio or video media will be exchanged by the peers.

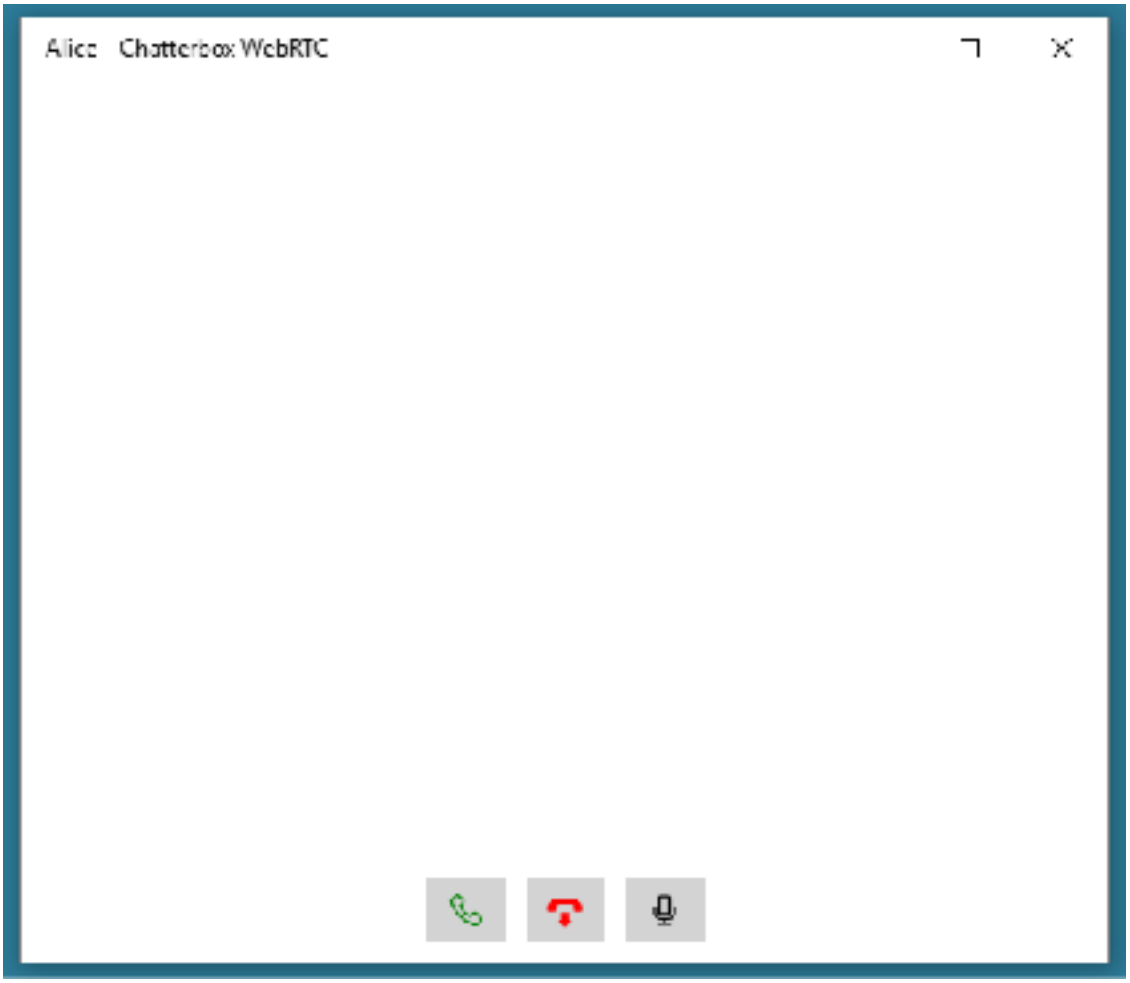

Figure 10
The Chatterbox WebRTC window showing a blank video screen when the application UI is visible.

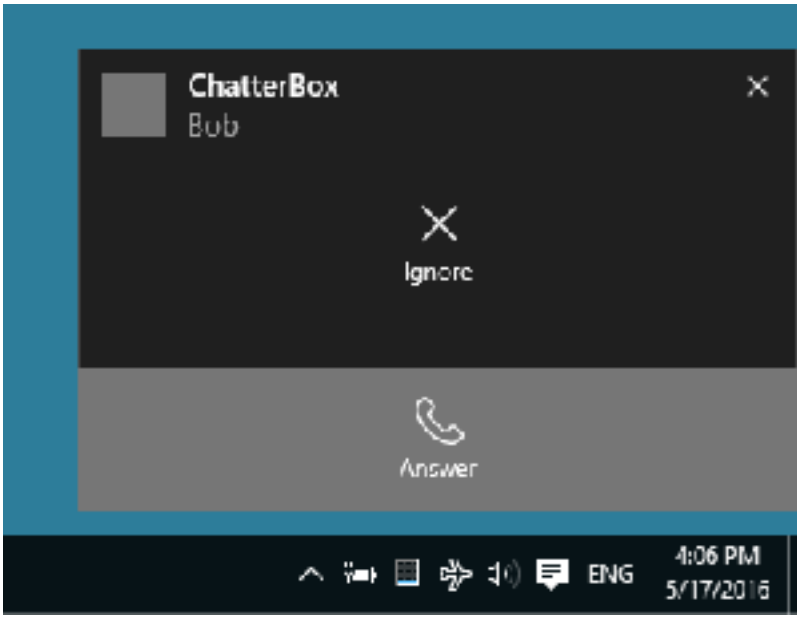

Figure 11
The ChatterBox application showing an incoming call from Bob when the application UI is not visible.

Exercise 3 – Using WebRTC for audio calls

In this exercise, you will use the WebRTC library and add support for establishing an audio call with a peer.

Task 1 – Note reference to WebRTC NuGet Package

Right-click on the **Chatterbox.Background** project in the **Solution Explorer** and choose **Manage NuGet Packages.**

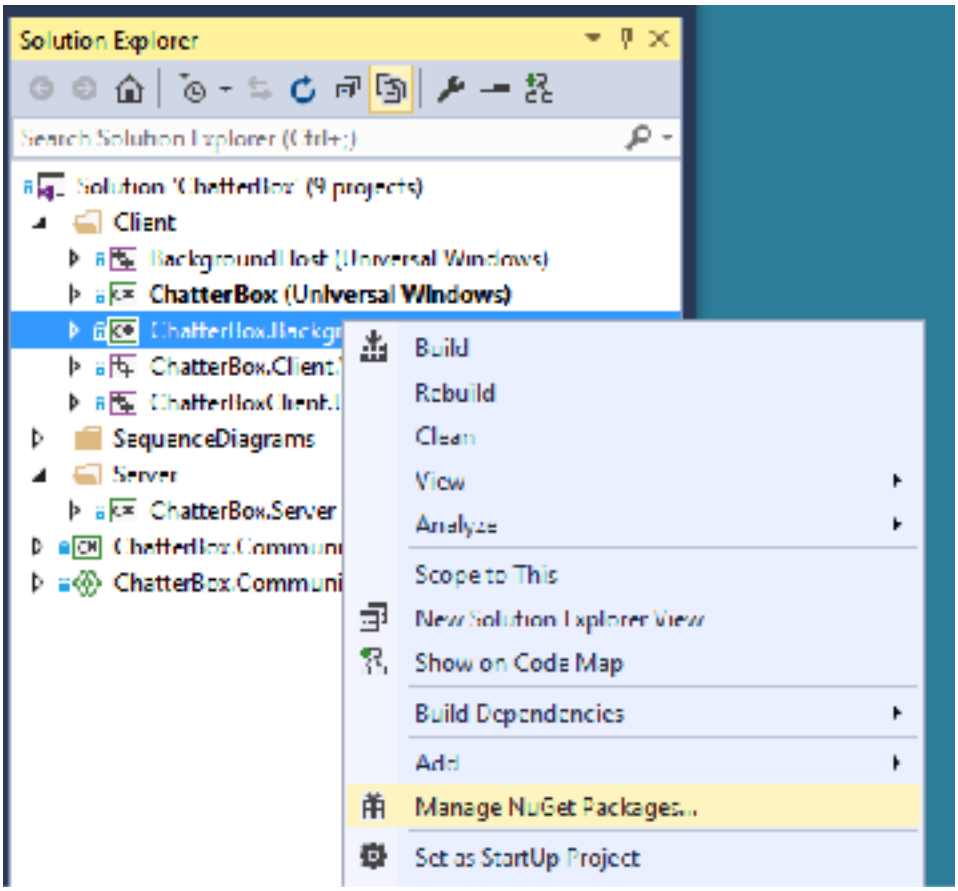

Figure 12
Right-click Chatterbox.Background and choose Manage NuGet Packages.

You will see that the WebRTC NuGet Package is already installed. This package wraps a native implementation of Google's WebRTC library, which is exposed as a WinRT library.

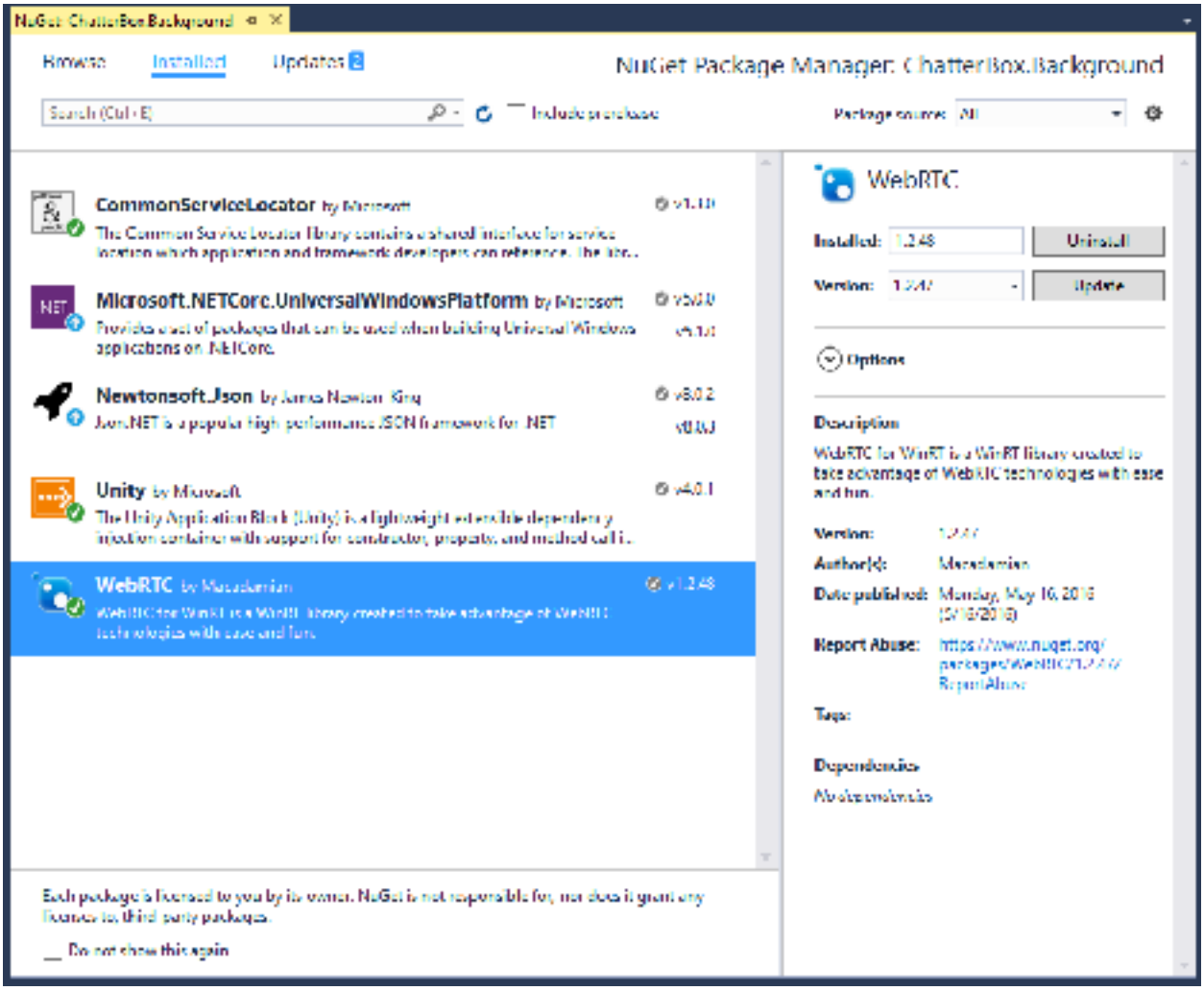

Figure 13
WebRTC is already included as an installed package.

Note: WebRTC is written in C++. You must select to use an ARM, x64, or x86 CPU configuration to proceed with the exercise.

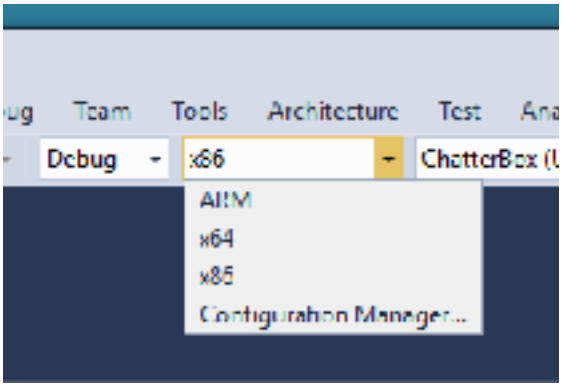

Figure 14
Select ARM, x64, or x86 from the CPU configuration drop-down select box.

Task 2 – Initialize WebRTC

WebRTC should only be initialized at the start of a call. However, when the application's foreground and background processes are not running, the background process will still periodically be started by the OS to handle the server heartbeats over the brokered socket. WebRTC shouldn't be initialized every time the background process is started (for every heartbeat), so we must make sure it's only initialized at the same time as the VOIP task is started when leaving the idle state.

Initialize WebRTC and create the media object in **RtcManager.cs**. The media object is used to initialize the audio/video capture.

C#

```
public void EnsureRtcIsInitialized()
{       lock (_lock)
        {
            if (!_rtcIsInitialized)
            {
                // On Windows 10, we don't need to use the
CoreDispatcher.
                // Pass null to initialize.
                WebRTC.Initialize(null);
                // Cache the media object for later use.
                Media = WebRTCMedia.CreateMedia();
                _rtcIsInitialized = true;



                WebRTCMedia.SetDisplayOrientation(_displayOrientation);



                // Uncomment the following line to enable WebRTC
logging.
                // Logs are:
                //  - Saved to local storage. Log folder location can
be
                // obtained using WebRTC.LogFolder()
```

```
                    //  - Sent over network if client is connected to TCP
port
                    // 47003
                    // WebRTC.EnableLogging(LogLevel.LOGLVL_INFO);
                }
            }
        }
```

Task 3 – Configuring RTCPeerConnection, GetUserMedia, and SDP

To be able to do NAT traversal and have peers residing on different network to be able to establish a call, you need to create a RTCPeerConncetion with ICE servers settings.

Then GetUserMedia() is called to enable the audio stream over the peer connection.

Since the support for call holding will be done at a later step, you need to handle the case where a peer connection already exists. In this case, the media needs to be reinstated.

To negotiate capabilities, both peers must exchange a text-based **SDP offer** and **answer** in order to fully establish the WebRTC connection. One peer must send the SDP offer which is used by the other peer to generate a SDP answer.

At the end of the OnEnteringStateAsync() function, add code to create and send an SDP offer.

Note: You also need to remove the transition to the Active state when sending an SDP offer, as this will be done when the SDP answer is received from the peer. The code you must remove appears in strikethrough.

1. Make the following changes to **EstablishOutgoing.cs**.

```
C#
public override async Task OnEnteringStateAsync()
{
    if (Context.VoipHelper.HasCall())
    {
        Context.VoipHelper.SetCallActive(Context.PeerId,
Context.IsVideoEnabled);
    }
    else
    {
        Context.VoipHelper.StartOutgoingCall(Context.PeerId,
Context.IsVideoEnabled);
    }
```

```
    // If PeerConnection is not null, then this is an SDP
renegotiation.
    if (Context.PeerConnection == null)
    {
        var config = new RTCConfiguration
        {
            IceServers = new List<RTCIceServer>
            {
                new RTCIceServer
                {
                    Url = "stun:stun.l.google.com:19302"
                },
                new RTCIceServer
                {
                    Url = "stun:stun1.l.google.com:19302"
                },
                new RTCIceServer
                {
                    Url = "stun:stun2.l.google.com:19302"
                },
                new RTCIceServer
                {
                    Url = "stun:stun3.l.google.com:19302"
                },
                new RTCIceServer
                {
                    Url = "stun:stun4.l.google.com:19302"
                },
                // Turn server in case networks are too restrictive to
allow peer-to-peer.
                new RTCIceServer
```

```
                {
                    Url = "turn:40.76.194.255:3478",
                    Username = "testrtc",
                    Credential = "rtc123"
                }
            }
        };
        Context.PeerConnection = new RTCPeerConnection(config);
    }

    // Even for just a renegotiation, it's easier to just teardown the
media capture and start over.
    if (Context.LocalStream != null)
    {
        Context.PeerConnection.RemoveStream(Context.LocalStream);
    }
    if (Context.RemoteStream != null)
    {
        Context.PeerConnection.RemoveStream(Context.RemoteStream);
    }
    foreach (var stream in Context.PeerConnection.GetLocalStreams())
    {
        Debug.WriteLine("Streams remaining.");
    }

    Context.LocalStream?.Stop();
    Context.LocalStream = null;
    Context.RemoteStream?.Stop();
    Context.RemoteStream = null;
    Context.ResetRenderers();
```

```
    Context.LocalStream = await
RtcManager.Instance.Media.GetUserMedia(new RTCMediaStreamConstraints
    {
        audioEnabled = true
    });
    Context.PeerConnection.AddStream(Context.LocalStream);
    ~~await Context.SwitchState(new Active());~~

    var sdpOffer = await Context.PeerConnection.CreateOffer();
    await Context.PeerConnection.SetLocalDescription(sdpOffer);

    Context.SendToPeer(RelayMessageTags.SdpOffer, sdpOffer.Sdp);
}
```

2. When the SDP answer is received, it must pass it to WebRTC and then transition to the active call state. Add the following code to **EstablishOutgoing.cs**.

C#

```
public override async Task OnSdpAnswerAsync(RelayMessage message)
{
    await Context.PeerConnection.SetRemoteDescription(new
RTCSessionDescription(RTCSdpType.Answer, message.Payload));
    await Context.SwitchState(new Active());
}
```

Note: Before calling `GetUserMedia()`, device selection (microphone, speaker and camera) can be done using the functions `SelectAudioCaptureDevice()`, `SelectAudioPlayoutDevice()`, and `SelectVideoDevice()`. In addition, codec selection and prioritization can be done by modifying the SDP message before calling `SetLocalDescription()`.

3. When receiving the SDP offer, you want to create a peer connection, call GetUserMedia(), generate an SDP answer, send the SDP answer to the peer and finally transition to the Active state. However, as with sending the SDP offer, we must remove the transition to the active state when entering the IncomingCall state. Make the following changes to **EstablishIncoming.cs**.

C#

```
public override async Task OnEnteringStateAsync()
```

```
{
    Context.VoipHelper.SetCallActive(Context.PeerId,
Context.IsVideoEnabled);
    // Remove the change to Active state and wait for the SDP.
    await Context.SwitchState(new Active());
}
```

4. Then implement the SDP offer handling in the same file.

C#

```
public override async Task OnSdpOfferAsync(RelayMessage message)
{
    // If PeerConnection is not null, then this is an SDP
renegotiation.
    if (Context.PeerConnection == null)
    {
        var config = new RTCConfiguration
        {
            IceServers = new List<RTCIceServer>
            {
                new RTCIceServer
                {
                    Url = "stun:stun.l.google.com:19302"
                },
                new RTCIceServer
                {
                    Url = "stun:stun1.l.google.com:19302"
                },
                new RTCIceServer
                {
                    Url = "stun:stun2.l.google.com:19302"
                },
```

```
                new RTCIceServer
                {
                    Url = "stun:stun3.l.google.com:19302"
                },
                new RTCIceServer
                {
                    Url = "stun:stun4.l.google.com:19302"
                },
                // Turn server in case networks are too restrictive to
allow peer-to-peer.
                new RTCIceServer
                {
                    Url = "turn:40.76.194.255:3478",
                    Username = "testrtc",
                    Credential = "rtc123"
                }
            }
        };
        Context.PeerConnection = new RTCPeerConnection(config);
    }


     // Even for just a renegotiation, it's easier to just teardown the
media capture and start over.
    if (Context.LocalStream != null)
    {
        Context.PeerConnection.RemoveStream(Context.LocalStream);
    }
    Context.LocalStream?.Stop();
    Context.LocalStream = null;
    Context.RemoteStream?.Stop();
    Context.RemoteStream = null;
```

```
        Context.ResetRenderers();


        Context.LocalStream = await
    RtcManager.Instance.Media.GetUserMedia(new RTCMediaStreamConstraints
        {
            audioEnabled = true
        });
        Context.PeerConnection.AddStream(Context.LocalStream);


        await Context.PeerConnection.SetRemoteDescription(new
    RTCSessionDescription(RTCSdpType.Offer, message.Payload));
        var sdpAnswer = await Context.PeerConnection.CreateAnswer();
        await Context.PeerConnection.SetLocalDescription(sdpAnswer);
        Context.SendToPeer(RelayMessageTags.SdpAnswer, sdpAnswer.Sdp);
        await Context.SwitchState(new Active());
    }
```

Task 4 - Exchanging ICE Candidates

In order to establish a peer-to-peer connection through NATs, firewalls, and proxies, WebRTC uses STUN and TURN servers to discover external IP addresses. The ICE candidates for the peers need to be exchanged through messages relayed over the signaling server.

You must register to the WebRTC notification that indicates that a new ICE candidate has been discovered. Then, save them to a queue to send them in batches to the server to avoid overloading the server.

1. In **CallContext.cs**, add the event registration in the PeerConnection function, add the following code.

```
C#
public RTCPeerConnection PeerConnection
{
    get { return _peerConnection; }
    set
    {
        _peerConnection = value;
        if (_peerConnection != null)
        {
```

```
                // Register to the events from the peer connection.
                // We'll forward them to the state.
                _peerConnection.OnIceCandidate += evt =>
                {
                    if (evt.Candidate != null)
                    {
                        var task = QueueIceCandidate(evt.Candidate);
                    }
                };

                 _peerConnection.OnAddStream += evt =>
                {
                    if (evt.Stream != null)
                    {
                        Task.Run(async () => { await WithState(async st =>
await st.OnAddStreamAsync(evt.Stream)); });
                    }
                };
            }
        }
    }
```

2. ICE candidate discovery begins during the SDP offer and answer exchange, and can take some time, many states can receive or send them. You can avoid duplication of candidates by adding the following code to **BaseCallState.cs**.

C#

```
    /// <summary>
    /// Ice candidates can come during several states.
    /// Anytime candidates come in, they need to be added
    /// to the peerconnection.
    /// </summary>
    /// <param name="message"></param>
    /// <returns></returns>
```

```
public virtual async Task AddRemoteIceCandidateAsync(RelayMessage
message)
{
    if (Context.PeerConnection == null)
        return;
    try
    {
        var candidates =

(DtoIceCandidates)JsonConvert.DeserializeObject(message.Payload,
typeof(DtoIceCandidates));
        foreach (var candidate in candidates.Candidates)
        {
            await
Context.PeerConnection.AddIceCandidate(candidate.FromDto());
        }
    }
    catch (Exception)
    {
        if (Debugger.IsAttached)
        {
            throw;
        }
    }
}


public virtual async Task SendLocalIceCandidatesAsync(RTCIceCandidate[]
candidates)
{
    Context.SendToPeer(RelayMessageTags.IceCandidate,
JsonConvert.SerializeObject(candidates.ToDto()));
}
```

Task 5 – Build and run the application

The application can now establish audio calls between two peers, and can connect to ICE servers on the Internet. The ICE servers allow the call to continue, even if the peers are not on the same network.

Exercise 4: Video Capture and Background Rendering

In this exercise you will enable video capture and render both the local and remote video streams in the application. WebRTC will run as a background process, and so you will use SwapChainPanel instead of MediaElement. A SwapChainPanel allows us to render within a background process on a surface held by the foreground process.

Task 1 – Change GetUserMedia() to support video

The `OutgoingCallRequest` class already has a property that specifies whether the call has video.

1. Save the video property in **RemoteRinging.cs** when entering that state.

C#

```
public override async Task OnEnteringStateAsync()

{

    Debug.Assert(Context.PeerConnection == null);



    Context.PeerId = _request.PeerUserId;

    Context.IsVideoEnabled = _request.VideoEnabled;
```

2. Save the video property in **LocalRinging.cs** when entering that state.

C#

```
public override async Task OnEnteringStateAsync()

{

    Debug.Assert(Context.PeerConnection == null);

    Context.PeerId = _message.FromUserId;

    Context.PeerName = _message.FromName;

    Context.IsVideoEnabled = _callRequest.VideoEnabled;
```

3. In **EstablishOutgoing.cs** and **EstablishIncoming.cs**, use the video property when calling the GetUserMedia function to support a video call.

C#

```
Context.LocalStream = await RtcManager.Instance.Media.GetUserMedia(new
RTCMediaStreamConstraints

{
```

```
        videoEnabled = Context.IsVideoEnabled,
        audioEnabled = true
    });
```

Note: Enabling video capture in the stream adds video information to the SDP offers and answers.

Task 2 – Rendering Video

Rendering video from a background process into SwapChainPanel is fairly complex and is best broken down into four major components.

1. Registering a scheme handler to allow passing an IMediaSource to an IMFMediaEngine.
2. Creating an instance of IMFMediaEngine and passing it the IMediaSource.
3. Creating an IMediaSource from the WebRTC MediaStream to setup the renderer.
4. Acquiring a swap chain handle from IMFMediaEngine and passing it to the SwapChainPanel.

Registering a scheme handler to allow passing an IMediaSource to an IMFMediaEngine

Creating an IMFMediaEngine and passing it an IMediaSource uses a complex method. The media engine doesn't take the media source directly, but instead takes a video source URL.

1. Register a scheme handler in **Renderer.cpp** by adding the following code.

```
C++
void Renderer::SetupSchemeHandler()
{
    using Windows::Foundation::ActivateInstance;
    // Create a media extension manager.  It's used to register a
scheme handler.
    HRESULT hr =
ActivateInstance(HStringReference(RuntimeClass_Windows_Media_MediaExten
sionManager).Get(), &_mediaExtensionManager);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"Failed to
create media extension manager"));
    }
    // Create an IMap container.  It maps a source URL with an
IMediaSource so it can be retrieved by the scheme handler.
    ComPtr<IMap<HSTRING, IInspectable*>> props;
```

```
    hr =
ActivateInstance(HStringReference(RuntimeClass_Windows_Foundation_Colle
ctions_PropertySet).Get(), &props);

    if (FAILED(hr))

    {

        throw ref new COMException(hr, ref new String(L"Failed to
create collection property set"));

    }

    // Register the scheme handler.  It takes the IMap container so it
can be passed to the scheme

    // handler when its invoked with a given source URL.

    // The SchemeHandler will extract the IMediaSource from the map.

    ComPtr<IPropertySet> propSet;

    props.As(&propSet);

    HStringReference
clsid(L"ChatterBoxClient.Universal.BackgroundRenderer.SchemeHandler");

    HStringReference scheme(L"webrtc:");

    hr = _mediaExtensionManager-
>RegisterSchemeHandlerWithSettings(clsid.Get(), scheme.Get(),
propSet.Get());

    if (FAILED(hr))

    {

        throw ref new COMException(hr, ref new String(L"Failed to to
register scheme handler"));

    }

    _extensionManagerProperties = props;

}
```

2. Map a source URL back to the media source by adding the following code to **SchemeHandler.cpp**.

C++

```
IFACEMETHODIMP SchemeHandler::BeginCreateObject(

    _In_ LPCWSTR pwszURL,

    _In_ DWORD dwFlags,

    _In_ IPropertyStore *pProps,
```

```
        _COM_Outptr_opt_  IUnknown **ppIUnknownCancelCookie,
        _In_ IMFAsyncCallback *pCallback,
        _In_ IUnknown *punkState)
    {
        if (ppIUnknownCancelCookie != nullptr)
        {
            *ppIUnknownCancelCookie = nullptr;
        }



        if ((pwszURL == nullptr) || (pCallback == nullptr))
        {
            return E_INVALIDARG;
        }



        if ((dwFlags & MF_RESOLUTION_MEDIASOURCE) == 0)
        {
            return E_INVALIDARG;
        }
        // Cast the IPropertySet to an IMap and extract the media source
    that
        ComPtr<ABI::Windows::Foundation::Collections::IMap<HSTRING,
    IInspectable*>> propMap;
        HRESULT hr = _extensionManagerProperties.As(&propMap);
        if (FAILED(hr))
        {
            return hr;
        }
        ComPtr<IInspectable> frameSourceInspectable;
```

```
    hr = propMap->Lookup(HStringReference(pwszURL).Get(),
&frameSourceInspectable);
    if (FAILED(hr))
    {
        unsigned int size;
        hr = propMap->get_Size(&size);
        if (size == 0)
        {
            return E_FAIL;
        }
        return hr;
    }
    propMap->Clear();



    // Asynchronously return the media source.
    ComPtr<IMFAsyncResult> result;
    hr = MFCreateAsyncResult(frameSourceInspectable.Get(), pCallback,
punkState, &result);
    if (FAILED(hr))
    {
        return hr;
    }
    hr = pCallback->Invoke(result.Get());
    if (FAILED(hr))
    {
        return hr;
    }
    return result->GetStatus();
}
```

```
IFACEMETHODIMP SchemeHandler::EndCreateObject(
    _In_ IMFAsyncResult *pResult,
    _Out_  MF_OBJECT_TYPE *pObjectType,
    _Out_  IUnknown **ppObject)
{
    if ((pResult == nullptr) || (pObjectType == nullptr) || (ppObject
== nullptr))
    {
        return E_INVALIDARG;
    }
    *pObjectType = MF_OBJECT_INVALID;
    *ppObject = nullptr;
    HRESULT hr = pResult->GetStatus();
    if (FAILED(hr))
    {
        return hr;
    }
    ComPtr<IUnknown> source;
    hr = pResult->GetObject(&source);
    if (FAILED(hr))
    {
        return hr;
    }
    *ppObject = source.Get();
    (*ppObject)->AddRef();
    *pObjectType = MF_OBJECT_MEDIASOURCE;
    return S_OK;
}
```

3. Add the scheme handler as an extension in **Package.appxmanifest**.

```
XML
```

```
    <Extensions>
        <Extension Category="windows.activatableClass.outOfProcessServer">
          <OutOfProcessServer ServerName="BackgroundHost">
            <Path>BackgroundHost.exe</Path>
            <Instancing>singleInstance</Instancing>
            <ActivatableClass
    ActivatableClassId="ChatterBox.BackgroundHost.Dummy" />
          </OutOfProcessServer>
        </Extension>
        <Extension Category="windows.activatableClass.inProcessServer">
          <InProcessServer>
            <Path>ChatterBoxClient.Universal.BackgroundRenderer.dll</Path>
            <ActivatableClass
    ActivatableClassId="ChatterBoxClient.Universal.BackgroundRenderer.Schem
    eHandler" ThreadingModel="both" />
          </InProcessServer>
        </Extension>
      </Extensions>
```

Creating an instance of IMFMediaEngine and passing it the IMediaSource

To create the media engine, you must first create a DirectX device. You can try using a hardware implementation first, but create a fallback to software if it fails.

1. Add the following code to **Renderer.cpp**.

```
    C++
    void Renderer::CreateDXDevice()
    {
      static const D3D_FEATURE_LEVEL levels[] =
      {
          D3D_FEATURE_LEVEL_11_1,
          D3D_FEATURE_LEVEL_11_0,
          D3D_FEATURE_LEVEL_10_1,
          D3D_FEATURE_LEVEL_10_0,
          D3D_FEATURE_LEVEL_9_1,
          D3D_FEATURE_LEVEL_9_2,
```

```
      D3D_FEATURE_LEVEL_9_3
  };

  D3D_FEATURE_LEVEL featureLevel;
  HRESULT hr = S_OK;

  // First attempt to use hardware device.
  hr = D3D11CreateDevice(nullptr, D3D_DRIVER_TYPE_HARDWARE, nullptr,
    D3D11_CREATE_DEVICE_VIDEO_SUPPORT,
    levels, ARRAYSIZE(levels), D3D11_SDK_VERSION, &_device,
&featureLevel,
    &_dx11DeviceContext);

  if (SUCCEEDED(hr))
  {
    ComPtr<ID3D10Multithread> multithread;
    hr = _device.Get()->QueryInterface(IID_PPV_ARGS(&multithread));
    if (FAILED(hr))
    {
      throw ref new COMException(hr, ref new String(L"Failed to set
device to multithreaded"));
    }
    multithread->SetMultithreadProtected(TRUE);
  }
  else // Fallback to software implementation
  {
    hr = D3D11CreateDevice(nullptr, D3D_DRIVER_TYPE_WARP, nullptr,
      D3D11_CREATE_DEVICE_VIDEO_SUPPORT, levels, ARRAYSIZE(levels),
      D3D11_SDK_VERSION, &_device, &featureLevel, &_dx11DeviceContext);
    if (FAILED(hr))
    {
```

```
            throw ref new COMException(hr, ref new String(L"Failed to create
a DX device"));
        }
      }
    }
```

2. Creating the media engine requires creating a device manager for the selected device and sending it to the device manager factory, along with additional settings.

C++

```
void Renderer::SetupDirectX()
{
    HRESULT hr = MFStartup(MF_VERSION);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"MFStartup
failed"));
    }
    CreateDXDevice();
    UINT resetToken;
    // Create a device manager
    Microsoft::WRL::ComPtr<IMFDXGIDeviceManager> dxGIManager;
    hr = MFCreateDXGIDeviceManager(&resetToken, &dxGIManager);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new
String(L"MFCreateDXGIDeviceManager failed"));
    }
    // Pass is the DirectX device created above.
    hr = dxGIManager->ResetDevice(_device.Get(), resetToken);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"ResetDevice
failed"));
```

```
    }
    // These attributes will be passed to the media engine created
below.
    ComPtr<IMFAttributes> attributes;
    hr = MFCreateAttributes(&attributes, 3);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new
String(L"MFCreateAttributes failed"));
    }
    // Pass it the device manager which contains the DirectX device.
    hr = attributes->SetUnknown(MF_MEDIA_ENGINE_DXGI_MANAGER,
(IUnknown*)dxGIManager.Get());
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"Failed to set
the DXGI manager"));
    }
    // Set a callback to receive media engine events.
    ComPtr<MediaEngineNotify> notify;
    notify = Make<MediaEngineNotify>();
    notify->SetCallback(this);
    hr = attributes->SetUnknown(MF_MEDIA_ENGINE_CALLBACK,
(IUnknown*)notify.Get());
    if (FAILED(hr))
    {
      throw ref new COMException(hr, ref new String(L"attributes-
>SetUnknown(MF_MEDIA_ENGINE_CALLBACK, (IUnknown*)notify.Get())
failed"));
    }

    // Set output video format.
```

```
    hr = attributes->SetUINT32(MF_MEDIA_ENGINE_VIDEO_OUTPUT_FORMAT,
DXGI_FORMAT_NV12);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"attributes-
>SetUINT32(MF_MEDIA_ENGINE_VIDEO_OUTPUT_FORMAT, DXGI_FORMAT_NV12)
failed"));
    }
    // Create the media engine.
    ComPtr<IMFMediaEngineClassFactory> factory;
    hr = CoCreateInstance(CLSID_MFMediaEngineClassFactory, nullptr,
CLSCTX_ALL, IID_PPV_ARGS(&factory));
    if (FAILED(hr))
    {
      throw ref new COMException(hr, ref new String(L"Failed to create
media engine class factory"));
    }
    hr = factory->CreateInstance(
        MF_MEDIA_ENGINE_REAL_TIME_MODE |
MF_MEDIA_ENGINE_WAITFORSTABLE_STATE,
        attributes.Get(), &_mediaEngine);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"Failed to
create media engine"));
    }

    // Query the IMFMediaEngineEx interface.
    // It contains additional functions used throughout the code.
    hr = _mediaEngine.As(&_mediaEngineEx);
    if (FAILED(hr))
    {
```

```
        throw ref new COMException(hr, ref new String(L"Failed to
create media engineex"));
    }

    // This call allows us to get a swap chain HANDLE to pass to the
UI.
    hr = _mediaEngineEx->EnableWindowlessSwapchainMode(TRUE);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"Failed to
enable Windowsless swapchain mode"));
    }
    _mediaEngineEx->SetRealTimeMode(TRUE);
    // Skylake video adapter has an issue with scaling. Using mirror
mode for this device is a workaround.
    if (TestForSkylakeDisplayAdapter())
    {
        OutputDebugString(L"Skylake display adapter detected, switching
to mirror mode\n");
        _mediaEngineEx->EnableHorizontalMirrorMode(TRUE);
    }
}
```

3. Passing the IMediaSource to the media engine requires you to create a random GUID URL, inserting it in the scheme handler map, and then setting the source URL on the media engine.

C++

```
void Renderer::SetupRenderer(uint32 foregroundProcessId,
Windows::Media::Core::IMediaSource^ streamSource,
    Windows::Foundation::Size videoControlSize)
{
    OutputDebugString(L"Renderer::SetupRenderer\n");
    _renderControlSize = videoControlSize;
    _streamSource = streamSource;
    _foregroundProcessId = foregroundProcessId;
```

```
    SetupSchemeHandler();
    SetupDirectX();
    boolean replaced;
    auto streamInspect = reinterpret_cast<IInspectable*>(streamSource);
    // Create a random URL that we'll use to map to the media source.
    std::wstring url(L"webrtc://");
    GUID result;
    HRESULT hr = CoCreateGuid(&result);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"Failed to
create a GUID"));
    }
    Guid gd(result);
    url += gd.ToString()->Data();
    // Insert the url and the media source in the map.
    hr = _extensionManagerProperties-
>Insert(HStringReference(url.c_str()).Get(), streamInspect, &replaced);
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"Failed to
insert a media stream into media properties"));
    }
    // Set the source URL on the media engine.
    // The scheme handler will find the media source for the given URL
and
    // return it to the media engine.
    BSTR sourceBSTR;
    sourceBSTR = SysAllocString(url.c_str());
    hr = _mediaEngine->SetSource(sourceBSTR);
    SysFreeString(sourceBSTR);
    if (FAILED(hr))
```

```
    {
        throw ref new COMException(hr, ref new String(L"Failed to set
media source"));
    }
    // Finally, trigger a load on the media engine.
    hr = _mediaEngine->Load();
    if (FAILED(hr))
    {
        throw ref new COMException(hr, ref new String(L"Failed load
media from source"));
    }
}
```

4. Once the media engine has loaded the media source, it triggers certain events. When it reports that it's ready to start playing, it will call `Play()` on the media engine. When the call format changes, you will request a swap chain handle and send it to the foreground.

C++

```
void Renderer::OnMediaEngineEvent(uint32 meEvent, uintptr_t param1,
uint32 param2)
{
  HANDLE swapChainHandle;
  switch ((DWORD)meEvent)
  {
  case MF_MEDIA_ENGINE_EVENT_ERROR:
    // Throw media engine errors so we catch them in the debugger.
    throw ref new COMException((HRESULT)param2, ref new String(L"Failed
OnMediaEngineEvent"));
    break;
  case MF_MEDIA_ENGINE_EVENT_FORMATCHANGE:
    // When the format changes, get a new swap chain handle and
    // send it to the foreground process.
    ReleaseStaleSwapChainHandleWhenExpired();
    CheckForegroundProcessId();
```

```
    if ((SUCCEEDED(_mediaEngineEx-
>GetVideoSwapchainHandle(&swapChainHandle))) &&
      (swapChainHandle != nullptr) && (swapChainHandle !=
INVALID_HANDLE_VALUE))
    {
      SendSwapChainHandle(swapChainHandle);
    }
    break;
  case MF_MEDIA_ENGINE_EVENT_CANPLAY:
    // Start playing automatically.
    _mediaEngine->Play();
    break;
  case MF_MEDIA_ENGINE_EVENT_TIMEUPDATE:
    // Various timed checks and cleanups.
    ReleaseStaleSwapChainHandleWhenExpired();
    CheckForegroundProcessId();
    break;
  }
}
```

5. Sending the swap chain handle to the foreground process requires the handle to be duplicated in a way that allows the process to access it. The logic for this is contained in **RemoteHandle.cpp,** but the `RenderFormatUpdate` must be triggered in **Renderer.cpp**. `RenderFormatUpdate` must be triggered with both the duplicated swap chain handle and the video dimensions. Add the following code to **Renderer.cpp**.

C++

```
void Renderer::SendSwapChainHandle(HANDLE swapChain)
{
  // Update the remote swap chain handle.
  if (swapChain != INVALID_HANDLE_VALUE)
  {
    _swapChainHandle.DetachMove(_staleSwapChainHandle);
    _staleHandleTimestamp = GetTickCount64();
```

```
        _swapChainHandle.AssignHandle(swapChain, _foregroundProcessId);
      }
      // Along with the handle, we also send the dimensions of the video.
      DWORD width;
      DWORD height;
      _mediaEngine->GetNativeVideoSize(&width, &height);

      if (_swapChainHandle.GetRemoteHandle() != INVALID_HANDLE_VALUE)
      {
        RenderFormatUpdate((int64)_swapChainHandle.GetRemoteHandle(),
    width, height, _foregroundProcessId);
      }
      // Save the video dimensions and recalculate the scaling/cropping.
      Windows::Foundation::Size size;
      size.Width = (float)width;
      size.Height = (float)height;
      EnterCriticalSection(&_lock);
      _videoSize = size;
      AsyncRecalculateScale();
      LeaveCriticalSection(&_lock);
    }
```

Creating an IMediaSource from the WebRTC MediaStream to set up the renderer.

Two instances of the renderers can now be added to the CallContext, along with variables to store the remote and local video control sizes. By the end of this lab, you'll use both the control sizes and the video sizes to calculate proper scaling and cropping so the videos fill the area used by the SwapChainPanel controls.

1. Add the following code to CallContext.cs.

C#

```
public MediaStream LocalStream
{
    get { return _localStream; }
    set
    {
```

```
            _localStream = value;
            ApplyMicrophoneConfig();
            ApplyVideoConfig();
        }
    }
    public Size LocalVideoControlSize { get; set; }
    public Renderer LocalVideoRenderer { get; private set; }

    public MediaStream RemoteStream { get; set; }
    public Size RemoteVideoControlSize { get; set; }
    public Renderer RemoteVideoRenderer { get; private set; }

    public void ResetRenderers()
    {
        if (LocalVideoRenderer != null)
        {
            LocalVideoRenderer.Teardown();
            LocalVideoRenderer = null;
        }
        if (RemoteVideoRenderer != null)
        {
            RemoteVideoRenderer.Teardown();
            RemoteVideoRenderer = null;
        }

        LocalVideoRenderer = new Renderer();
        RemoteVideoRenderer = new Renderer();

        GC.Collect();
    }
```

2. Notifications regarding control size and foreground process ID changes must be passed to the renderers. These will be used later to set up rendering, scaling, and cropping of the video stream. Add the following code to **CallChannel.cs**.

C#

```
public IAsyncAction OnLocalControlSizeAsync(VideoControlSize size)
{
    return Context.WithContextAction(cx =>
    {
        cx.LocalVideoControlSize = size.Size;
        cx.LocalVideoRenderer.SetRenderControlSize(size.Size);
    }).AsAsyncAction();
}

 public IAsyncAction OnRemoteControlSizeAsync(VideoControlSize size)
{
    return Context.WithContextAction(cx =>
    {
        cx.RemoteVideoControlSize = size.Size;
        cx.RemoteVideoRenderer.SetRenderControlSize(size.Size);
    }).AsAsyncAction();
}

public IAsyncAction SetForegroundProcessIdAsync(uint processId)
{
    return Context.WithContextAction(cx =>
    {
        Context.ForegroundProcessId = processId;

Context.LocalVideoRenderer?.UpdateForegroundProcessId(processId);

Context.RemoteVideoRenderer?.UpdateForegroundProcessId(processId);
    }).AsAsyncAction();
```

```
}
```

3. After calling `GetUserMedia()`, create an `IMediaSource` for the local video stream and set up the renderer in **EstablishIncoming.cs** and **EstablishOutgoing.cs**.

C#

```
Context.LocalStream = await RtcManager.Instance.Media.GetUserMedia(new
RTCMediaStreamConstraints

{

    videoEnabled = Context.IsVideoEnabled,

    audioEnabled = true

});

Context.PeerConnection.AddStream(Context.LocalStream);


// Setup the rendering of the local capture.

var tracks = Context.LocalStream.GetVideoTracks();

if (tracks.Count > 0)

{

    var source = RtcManager.Instance.Media.CreateMediaSource(tracks[0],
CallContext.LocalMediaStreamId);


Context.LocalVideoRenderer.SetupRenderer(Context.ForegroundProcessId,
source, Context.LocalVideoControlSize);

}
```

4. The application must also do this when receiving the remote video stream. In **EstablishIncoming.cs**, **EstablishOutgoing.cs**, and **Active.cs**, add the following code.

C#

```
internal override async Task OnAddStreamAsync(MediaStream stream)

{

    Context.RemoteStream = stream;

    var tracks = stream.GetVideoTracks();

    if (tracks.Count > 0)

    {
```

```
            var source =
RtcManager.Instance.Media.CreateMediaSource(tracks[0],
CallContext.PeerMediaStreamId);

Context.RemoteVideoRenderer.SetupRenderer(Context.ForegroundProcessId,
source, Context.RemoteVideoControlSize);
        }
    }
```

Acquiring a swap chain handle from IMFMediaEngine and passing it to the SwapChainPanel

1. In **CallContext.cs**, register for render format events on the renderers. These events carry the handle that must be passed to the swap chain panel.

C#

```
public void ResetRenderers()
{
    if (LocalVideoRenderer != null)
    {
        LocalVideoRenderer.Teardown();
        LocalVideoRenderer = null;
    }
    if (RemoteVideoRenderer != null)
    {
        RemoteVideoRenderer.Teardown();
        RemoteVideoRenderer = null;
    }

    LocalVideoRenderer = new Renderer();
    RemoteVideoRenderer = new Renderer();

    LocalVideoRenderer.RenderFormatUpdate +=
LocalVideoRenderer_RenderFormatUpdate;
    RemoteVideoRenderer.RenderFormatUpdate +=
RemoteVideoRenderer_RenderFormatUpdate;
    GC.Collect();
```

```
}
```

2. When the event is received, the application must call `OnUpdateFrameFormat()` in the foreground process through the hub.

C#

```
private void LocalVideoRenderer_RenderFormatUpdate(long
swapChainHandle, uint width, uint height, uint foregroundProcessId)
{
    _hub.OnUpdateFrameFormat(
        new FrameFormat
        {
            IsLocal = true,
            SwapChainHandle = swapChainHandle,
            Width = width,
            Height = height,
            ForegroundProcessId = foregroundProcessId
        });
}

private void RemoteVideoRenderer_RenderFormatUpdate(long
swapChainHandle, uint width, uint height, uint foregroundProcessId)
{
    _hub.OnUpdateFrameFormat(
        new FrameFormat
        {
            IsLocal = false,
            SwapChainHandle = swapChainHandle,
            Width = width,
            Height = height,
            ForegroundProcessId = foregroundProcessId
        });
}
```

3. Implement a function that allows the UI to directly query the frame format.

C#

```
internal FrameFormat GetFrameFormat(bool local)
{
    var videoRenderer = local ? LocalVideoRenderer :
RemoteVideoRenderer;
    if(videoRenderer != null && videoRenderer.IsInitialized)
    {
        Int64 swapChainHandle = 0;
        UInt32 width = 0, height = 0;
        UInt32 foregroundProcessId = 0;
        if(videoRenderer.GetRenderFormat(out swapChainHandle, out
width, out height, out foregroundProcessId))
        {
            return new FrameFormat
            {
                IsLocal = local,
                SwapChainHandle = swapChainHandle,
                Width = width,
                Height = height,
                ForegroundProcessId = foregroundProcessId
            };
        }
    }
    return null;
}
```

4. In the foreground, update the values which are data bound to the swap chain panel. The solution uses a special class derived from `SwapChainPanel` which adds data binding capabilities for the handle. Add the following code to **ConversationViewModel.cs**.

C#

```
private void OnFrameFormatUpdate(FrameFormat obj)
{
    if (CallState == CallState.Idle)
```

```
        {
            return;
        }

        if (obj.ForegroundProcessId != Renderer.GetProcessId())
        {
            // Ignore this update because it's for an old foreground
    process
            return;
        }

        if (obj.IsLocal)
        {
            LocalSwapChainPanelHandle = obj.SwapChainHandle;
        }
        else
        {
            RemoteSwapChainPanelHandle = obj.SwapChainHandle;
        }
    }
```

At this point, the whole video rendering stack is implemented. Setting up the local or remote renderers in the background results in the video being rendered to the swap chain panels on the foreground.

Task 3 – Video scaling and cropping

Rendering video from the background process into a swap chain panel in the foreground process requires us to do scaling and cropping manual on the media engine. The algorithm takes the size of the swap chain panel and the size of the video frames. It scales, centers, and crops the video frame so it fills in the swap chain panel. Whenever the size of the swap chain panel or the size of the video changes, this function is invoked. Add the following code to **renderer.cpp**.

C++

```
    void Renderer::RecalculateScale(Windows::Foundation::Size
    renderControlSize,
        Windows::Foundation::Size videoSize)
    {
```

```
        if ((renderControlSize.Width <= 0.0f) || (renderControlSize.Height
    <= 0.0f) ||
            (videoSize.Width <= 0.0f) || (videoSize.Height <= 0.0f))
        {
            return;
        }
        if (_mediaEngineEx == nullptr)
        {
            return;
        }
        double videoAspect = double(videoSize.Width) /
    double(videoSize.Height);
        double renderControlAspect = double(renderControlSize.Width) /
    double(renderControlSize.Height);
        // Scale to fill the swap chain panel.
        double scalingFactor = (videoAspect > renderControlAspect) ?
            (double(renderControlSize.Height) / double(videoSize.Height)) :
            (double(renderControlSize.Width) / double(videoSize.Width));
        if (scalingFactor == 0.0)
        {
            scalingFactor = 0.0001;
        }
        int scaledRenderControlWidth =
    int(double(renderControlSize.Width) / scalingFactor);
        int scaledRenderControlHeight =
    int(double(renderControlSize.Height) / scalingFactor);
        // Amount to crop from the width and height.
        int cropX = max((int)videoSize.Width - scaledRenderControlWidth,
    0);
        int cropY = max((int)videoSize.Height - scaledRenderControlHeight,
    0);
        // Convert crop to percentage and divide by 2 to share the crop
    equally
```

```
        // on both edges.  This centers the final image.
        float cropFrX = float((double(cropX) / double(videoSize.Width)) /
    2.0);
        float cropFrY = float((double(cropY) / double(videoSize.Height)) /
    2.0);
        // The final crop/scale rectangle with values between 0.0 and 1.0.
        MFVideoNormalizedRect rect = MFVideoNormalizedRect { cropFrX,
    cropFrY, 1.0f - cropFrX, 1.0f - cropFrY };
        RECT r = { 0, 0, (LONG)renderControlSize.Width ,
    (LONG)renderControlSize.Height };
        MFARGB borderColour = { 0, 0, 0, 0xFF };
        _mediaEngineEx->UpdateVideoStream(&rect, &r, &borderColour);
    }
```

Task 4 - Suspending and resuming the app

When the application is suspended and there is an active video call, we have to release access to the camera. WebRTC allows us to temporarily suspend a video track, which releases camera access, turns off the camera, and sends black frames to the peer instead. Resuming a video track turns the camera back on and resumes sending a video feed from the camera.

1. When suspending the app, instruct the background process to suspend video capture and detach both swap chain panels from their handle. You'll also hang up any call that is in the process of connecting. Add the following code to **App.xaml.cs**.

C#

```
private async void OnSuspending(object sender, SuspendingEventArgs e)
{
    Debug.WriteLine($"{DateTime.Now.ToString("HH:mm:ss.ffff")}
App.OnSuspending");
    var deferral = e.SuspendingOperation.GetDeferral();


    var client = Container.Resolve<HubClient>();


    // Suspend video capture and rendering in the background.
    await client.SuspendCallVideoAsync();


    // Disconnect the rendering on the UI.
```

```
        // We do it here instead of waiting for the background
        // to notify us because we're about to be suspended.
        await client.OnUpdateFrameFormatAsync(new FrameFormat
        {
            IsLocal = true,
            Width = 0,
            Height = 0,
            SwapChainHandle = 0,
            ForegroundProcessId = 0
        });
        await client.OnUpdateFrameFormatAsync(new FrameFormat
        {
            IsLocal = false,
            Width = 0,
            Height = 0,
            SwapChainHandle = 0,
            ForegroundProcessId = 0
        });

        // Hangup pending calls
        var callStatus = await client.GetCallStatusAsync();
        if (callStatus != null &&
            (callStatus.State ==
    Background.AppService.Dto.CallState.LocalRinging ||
             callStatus.State ==
    Background.AppService.Dto.CallState.RemoteRinging))
        {
            await client.HangupAsync();
        }

        deferral.Complete();
```

```
}
```

2. In BaseCallState.cs, implement a function to suspend video by disabling all video tracks in the local stream.

C#

```
internal async Task SuspendCallVideoAsync()
{
    // Detach any renderers.
    Context.ResetRenderers();
    // Don't send RenderFormatUpdate here. The UI is suspending
    // and may not get the message.
    if (Context.LocalStream != null)
    {
        foreach (var track in Context.LocalStream.GetVideoTracks())
        {
            track.Suspended = true;
        }
    }
}
```

3. To resume a call, you need to pass the new foreground process ID to the background process, then resume the video. The process ID needs to be updated in order for the background process to be able to duplicate the swap chain handle. In **App.xaml.cs**, add the following code at the end of `Resume()`.

C#

```
    /* If the call was hung-up while we were suspended, we need to
update the UI */
    if (contactView.SelectedConversation != null)
    {
        if (contactView.SelectedConversation.CallState !=
CallState.Idle)
        {
            // By calling Initialize, we force to get the call state
from the background
            await contactView.SelectedConversation.InitializeAsync();
        }
```

```
        }

        var client = Container.Resolve<HubClient>();
        if (client.IsConnected)
        {
            await
    client.SetForegroundProcessIdAsync(WebRTCSwapChainPanel.CurrentProcessI
    d);
            await client.ResumeCallVideoAsync();
        }

        Window.Current.Activate();
    }
```

4. Resume the video capture and re-initialize both local and remote renderers by adding the following code to **BaseCallState.cs**.

C#

```
    internal async Task ResumeCallVideoAsync()
    {
        if (Context.LocalVideoRenderer.IsInitialized &&
            Context.RemoteVideoRenderer.IsInitialized)
        {
            return;
        }

        Context.ResetRenderers();

        // Setup remote before local as it's more important.
        if (Context.RemoteStream != null)
        {
            var tracks = Context.RemoteStream.GetVideoTracks();
            if (tracks.Count > 0)
            {
```

```
            var source =
RtcManager.Instance.Media.CreateMediaSource(tracks[0],
CallContext.PeerMediaStreamId);

Context.RemoteVideoRenderer.SetupRenderer(Context.ForegroundProcessId,
source, Context.RemoteVideoControlSize);
        }
        else
        {
            System.Diagnostics.Debug.WriteLine("Failed to resume remote
video, no video track");
        }
    }
    if (Context.LocalStream != null)
    {
        var tracks = Context.LocalStream.GetVideoTracks();
        foreach (var track in tracks)
        {
            track.Suspended = false;
        }

        if (tracks.Count > 0)
        {
            var source =
RtcManager.Instance.Media.CreateMediaSource(tracks[0],
CallContext.LocalMediaStreamId);

Context.LocalVideoRenderer.SetupRenderer(Context.ForegroundProcessId,
source, Context.LocalVideoControlSize);
        }
        else
        {
            System.Diagnostics.Debug.WriteLine("Failed to resume local
video, no video track");
```

```
            }
        }
    }
```

Task 5 – Putting a call on hold

When the application has an active call, another call can come in and interrupt the WebRTC call. When the user answers the second call, the VOIP call coordinator will send a hold event. The application needs to handle that event by releasing access to the camera and microphone.

1. In **Active.cs**, add the following code to handle a hold request. This will transition the application to the EstablishOutgoing state, and trigger a SDP renegotiation.

C#

```
public override async Task HoldAsync()
{
    // Re-negotiate the SDP.
    var state = new EstablishOutgoing(true);
    await Context.SwitchState(state);
}
```

2. When holding, we remove all media from the stream and create a new SDP offer. When receiving the SDP answer, the call will transition to a **Held** state. Add the following code to **EstablishOutgoing.cs**.

C#

```
public override async Task OnEnteringStateAsync()
{
    if (_hold)
    {
        Context.VoipHelper.SetCallHeld();
    }
    else if (Context.VoipHelper.HasCall())
    {
        Context.VoipHelper.SetCallActive(Context.PeerId,
Context.IsVideoEnabled);
    }
    else
    {
```

```
            Context.VoipHelper.StartOutgoingCall(Context.PeerId,
    Context.IsVideoEnabled);
        }

        // If PeerConnection is not null, then this is an SDP
    renegotiation.
        if (Context.PeerConnection == null)
        {
            var config = new RTCConfiguration
            {
                IceServers = new List<RTCIceServer>
                {
                    new RTCIceServer
                    {
                        Url = "stun:stun.l.google.com:19302"
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun1.l.google.com:19302"
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun2.l.google.com:19302"
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun3.l.google.com:19302"
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun4.l.google.com:19302"
```

```
                    },
                    // Turn server in case networks are too restrictive to
    allow peer-to-peer.
                    new RTCIceServer
                    {
                        Url = "turn:40.76.194.255:3478",
                        Username = "testrtc",
                        Credential = "rtc123"
                    }
                }
            };
            Context.PeerConnection = new RTCPeerConnection(config);
        }

        // Even for just a renegotiation, it's easier to just teardown the
    media capture and start over.
        if (Context.LocalStream != null)
        {
            Context.PeerConnection.RemoveStream(Context.LocalStream);
        }
        if (Context.RemoteStream != null)
        {
            Context.PeerConnection.RemoveStream(Context.RemoteStream);
        }
        foreach (var stream in Context.PeerConnection.GetLocalStreams())
        {
            Debug.WriteLine("Streams remaining.");
        }

        Context.LocalStream?.Stop();
        Context.LocalStream = null;
```

```
        Context.RemoteStream?.Stop();
        Context.RemoteStream = null;
        Context.ResetRenderers();

        if (!_hold)
        {
            Context.LocalStream = await
    RtcManager.Instance.Media.GetUserMedia(new RTCMediaStreamConstraints
            {
                videoEnabled = Context.IsVideoEnabled,
                audioEnabled = true
            });
            Context.PeerConnection.AddStream(Context.LocalStream);

            // Setup the rendering of the local capture.
            var tracks = Context.LocalStream.GetVideoTracks();
            if (tracks.Count > 0)
            {
                var source =
    RtcManager.Instance.Media.CreateMediaSource(tracks[0],
    CallContext.LocalMediaStreamId);

    Context.LocalVideoRenderer.SetupRenderer(Context.ForegroundProcessId,
    source, Context.LocalVideoControlSize);
            }
        }

        var sdpOffer = await Context.PeerConnection.CreateOffer();
        await Context.PeerConnection.SetLocalDescription(sdpOffer);

        Context.SendToPeer(RelayMessageTags.SdpOffer, sdpOffer.Sdp);
    }
```

```
public override async Task OnSdpAnswerAsync(RelayMessage message)
{
    await Context.PeerConnection.SetRemoteDescription(new
RTCSessionDescription(RTCSdpType.Answer, message.Payload));

    if (SdpUtils.IsHold(message.Payload))
    {
        await Context.SwitchState(new Held());
    }
    else
    {
        await Context.SwitchState(new Active());
    }

}
```

3. In **Active.cs**, add the following code to transition the call to EstablishIncoming state when receiving an SDP offer.

C#

```
public override async Task OnSdpOfferAsync(RelayMessage message)
{
    // Re-negotiate the SDP.
    var state = new EstablishIncoming();
    await Context.SwitchState(state);
    // Hand-off the SDP to the new state.
    await state.OnSdpOfferAsync(message);
}
```

4. In **EstablishIncoming.cs**, the application needs to read whether the peer is sending media through the SDP offer. If no media is being sent, the application will assume a hold request is being sent. If this is the case remove all media from the local stream, and send back an SDP answer to the peer, and transition to a Held state.

C#

```
public override async Task OnEnteringStateAsync()
{
    Context.VoipHelper.SetCallActive(Context.PeerId,
Context.IsVideoEnabled);
    // We wait for the SDP.
}

public override async Task OnSdpOfferAsync(RelayMessage message)
{
    bool isHold = SdpUtils.IsHold(message.Payload);
    if (isHold)
    {
        Context.VoipHelper.SetCallHeld();
    }
    else
    {
        Context.VoipHelper.SetCallActive(Context.PeerId,
Context.IsVideoEnabled);
    }

    // If PeerConnection is not null, then this is an SDP
renegotiation.
    if (Context.PeerConnection == null)
    {
        var config = new RTCConfiguration
        {
            IceServers = new List<RTCIceServer>
            {
                new RTCIceServer
                {
                    Url = "stun:stun.l.google.com:19302"
```

```
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun1.l.google.com:19302"
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun2.l.google.com:19302"
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun3.l.google.com:19302"
                    },
                    new RTCIceServer
                    {
                        Url = "stun:stun4.l.google.com:19302"
                    },
                    // Turn server in case networks are too restrictive to
allow peer-to-peer.
                    new RTCIceServer
                    {
                        Url = "turn:40.76.194.255:3478",
                        Username = "testrtc",
                        Credential = "rtc123"
                    }
                }
            };
            Context.PeerConnection = new RTCPeerConnection(config);
        }
```

```
        // Even for just a renegotiation, it's easier to just teardown the
    media capture and start over.
        if (Context.LocalStream != null)
        {
            Context.PeerConnection.RemoveStream(Context.LocalStream);
        }
        Context.LocalStream?.Stop();
        Context.LocalStream = null;
        Context.RemoteStream?.Stop();
        Context.RemoteStream = null;
        Context.ResetRenderers();

        if (!isHold)
        {
            Context.LocalStream = await
    RtcManager.Instance.Media.GetUserMedia(new RTCMediaStreamConstraints
            {
                videoEnabled = Context.IsVideoEnabled,
                audioEnabled = true
            });
            Context.PeerConnection.AddStream(Context.LocalStream);

            var tracks = Context.LocalStream.GetVideoTracks();
            if (tracks.Count > 0)
            {
                var source =
    RtcManager.Instance.Media.CreateMediaSource(tracks[0],
    CallContext.LocalMediaStreamId);

    Context.LocalVideoRenderer.SetupRenderer(Context.ForegroundProcessId,
    source, Context.LocalVideoControlSize);
            }
```

```
        }

        await Context.PeerConnection.SetRemoteDescription(new
    RTCSessionDescription(RTCSdpType.Offer, message.Payload));
        var sdpAnswer = await Context.PeerConnection.CreateAnswer();
        await Context.PeerConnection.SetLocalDescription(sdpAnswer);
        Context.SendToPeer(RelayMessageTags.SdpAnswer, sdpAnswer.Sdp);
        if (isHold)
        {
            await Context.SwitchState(new Held());
        }
        else
        {
            await Context.SwitchState(new Active());
        }
    }
```

5. Resuming from a held state involves transitioning the call back through EstablishOutgoing/EstablishIncoming to get access to the camera and microphone to resume the call. In **Held.cs**, add the following code to allow a call to resume either locally or in response to an incoming SDP offer.

C#

```
    public override async Task OnSdpOfferAsync(RelayMessage message)
    {
        // Re-negotiate the SDP.
        var state = new EstablishIncoming();
        await Context.SwitchState(state);
        // Hand-off the SDP to the new state.
        await state.OnSdpOfferAsync(message);
    }

    public override async Task ResumeAsync()
    {
```

```
            // Re-negotiate the SDP.
            var state = new EstablishOutgoing(false);
            await Context.SwitchState(state);
        }
```

Task 6 - Build and run the application

The application can now establish video calls. Click the video icon to start a video call, or the phone icon for an audio only call.

Summary

In this lab, you have learned how to create a two-process Windows Universal VOIP application by using the WebRTC. Additional features are part of the full size ChatterBox sample application.