Sending the Audio Data

To choose and play a song from iTunes library, `MPMediaPickerController` can be used. From the picker's delegate method, we will get an array of `MPMediaItems` i.e. `mediaPicker:didPickMediaItems:`.

An `MPMediaItem` has many properties we can look at for song title or author, but we're only interested in the `MPMediaItemPropertyAssetURL` property. We use that to create an `AVURLAsset` from which we can read the file data by using `AVAssetReader` and `AVAssetReaderTrackOutput`.

```
NSURL *url = [myMediaItem valueForProperty:MPMediaItemPropertyAssetURL];
AVURLAsset *asset = [AVURLAsset URLAssetWithURL:url options:nil];
AVAssetReader *assetReader = [AVAssetReader assetReaderWithAsset:asset
error:nil];
AVAssetReaderTrackOutput *assetOutput = [AVAssetReaderTrackOutput
assetReaderTrackOutputWithTrack:asset.tracks[0] outputSettings:nil];

[self.assetReader addOutput:self.assetOutput];
[self.assetReader startReading];
```

Here, we create the `AVURLAsset` from the media item. Then we use it to create an `AVAssetReader` and `AVAssetReaderTrackOutput`. Finally, we add the output to the reader and start reading. The method `startReading` will only open the reader and make it ready for later when we request data from it.

Next, we'll open our `NSOutputStream` and send the reader output data to it when its delegate method is invoked with the event `NSStreamEventHasSpaceAvailable`.

```
CMSampleBufferRef sampleBuffer = [assetOutput copyNextSampleBuffer];

CMBlockBufferRef blockBuffer;
AudioBufferList audioBufferList;

CMSampleBufferGetAudioBufferListWithRetainedBlockBuffer(sampleBuffer, NULL,
&audioBufferList, sizeof(AudioBufferList), NULL, NULL,
kCMSampleBufferFlag_AudioBufferList_Assure16ByteAlignment, &blockBuffer);

for (NSUInteger i = 0; i < audioBufferList.mNumberBuffers; i++) {
  AudioBuffer audioBuffer = audioBufferList.mBuffers[i];
  [audioStream writeData:audioBuffer.mData
maxLength:audioBuffer.mDataByteSize];
}

CFRelease(blockBuffer);
CFRelease(sampleBuffer);
```

First, we get the sample buffer from the reader output. Then we call the function `CMSampleBufferGetAudioBufferListWithRetainedBlockBuffer` to get a list of audio buffers. Finally, we write each audio buffer to the output stream.

We're now streaming a song from our iTunes library. Now, let's look at how to receive this stream and play the audio.

The Data Stream

Since we are using Multipeer Connectivity, the `NSInputStream` is already made for us. First, we need to start the stream to receive the data.

```
// Start receiving data
inputStream.delegate = self;
[inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
forMode:NSDefaultRunLoopMode];
[inputStream open];
```

Here, we set the delegate to the class so we can handle the stream events. Next, we tell the stream to run in the current run loop, which could be on a separate thread, and to use the default run loop mode. It is important to use the default mode or our delegate methods will not be called. Finally, open the stream to start receiving data.

Our class should conform to the `NSStreamDelegate` protocol so we can handle events from the `NSInputStream`.

```
@interface MyCustomClass () <NSStreamDelegate>
//...
@end

@implementation MyCustomClass
//...

- (void)stream:(NSStream *)aStream handleEvent:(NSStreamEvent)eventCode
{
    if (eventCode == NSStreamEventHasBytesAvailable) {
        // handle incoming data
    } else if (eventCode == NSStreamEventEndEncountered) {
        // notify application that stream has ended
    } else if (eventCode == NSStreamEventErrorOccurred) {
        // notify application that stream has encountered and error
    }
}

//...
@end
```

Above, we use the delegate method to handle events received from the stream. When the stream ends or has an error, we should notify the application so it can decide what to do next. For now, we are only interested in the event that the stream has data for us to process. We need to take this data and pass it into the Audio File Stream Services.

The Stream Parser

It is an 'AudioFileStream' class that where encoded audio data can be pushed into and get back decoded audio data. AudioFileStream can be created as follows:

```
AudioFileStreamID audioFileStreamID;
AudioFileStreamOpen((__bridge void *)self, AudioFileStreamPropertyListener,
AudioFileStreamPacketsListener, 0, &audioFileStreamID);
```

The parser can be created by passing it a reference to the class, a property changed callback function and a packets received callback function. We need to create these callback functions to use the reference to the class

and call methods within the class.

```
void AudioFileStreamPropertyListener(void *inClientData, AudioFileStreamID
inAudioFileStreamID, AudioFileStreamPropertyID inPropertyID, UInt32
*ioFlags)
{
    MyCustomClass *myClass = (__bridge MyCustomClass *)inClientData;
    [myClass didChangeProperty:inPropertyID flags:ioFlags];
}

void AudioFileStreamPacketsListener(void *inClientData, UInt32
inNumberBytes, UInt32 inNumberPackets, const void *inInputData,
AudioStreamPacketDescription *inPacketDescriptions)
{
    MyCustomClass *myClass = (__bridge MyCustomClass *)inClientData;
    [myClass didReceivePackets:inInputData
packetDescriptions:inPacketDescriptions numberOfPackets:inNumberPackets
numberOfBytes:inNumberBytes];
}
```

Inside the `didChangeProperty:flags:` method, we are looking for the `kAudioFileStreamProperty_ReadyToProducePackets` property which tells us that all other properties have been set. Now we can retrieve the `AudioStreamBasicDescription` from the parser. The `AudioStreamBasicDescription` contains information about the audio such as sample rate, channels, and bytes per packet and is necessary for creating our audio queue.

```
AudioStreamBasicDescription basicDescription;
UInt32 basicDescriptionSize = sizeof(basicDescription);
AudioFileStreamGetProperty(audioFileStreamID,
kAudioFileStreamProperty_DataFormat, &basicDescriptionSize,
&basicDescription);
```

The other function for the packets received callback will return the decoded audio data that we will add to the audio queue buffers later.

Now, it's time to pass the encoded data into the parser from the stream's `NSStreamEventHasBytesAvailable` event.

```
uint8_t bytes[512];
UInt32 length = [audioStream readData:bytes maxLength:512];
AudioFileStreamParseBytes(audioFileStreamID, length, data, 0);
```

The file stream will continue to parse bytes until it has enough to decypher the type of file. At this point, it invokes its property changed callback with the property `kAudioFileStreamProperty_ReadyToProducePackets`. After this, it will invoke its packets received callback with nicely packaged packets of decoded audio data for us to use.

The Audio Queue

Create audio buffers, to fill them, and then enqueue them with An `AudioQueue` class. It gives us audio control like play, pause, and stop. Now, let's create the queue and its buffers.

```
AudioQueueRef audioQueue;
AudioQueueNewOutput(&basicDescription, AudioQueueOutputCallback, (__bridge
void *)self, NULL, NULL, 0, &audioQueue);
```

```
AudioQueueBufferRef audioQueueBuffer;
AudioQueueAllocateBuffer(audioQueue, 2048, &audioQueueBuffer);
```

To create the audio queue we need to pass the `AudioQueueNewOutput` function the `AudioStreamBasicDescription` we received from the parser, a callback function that is invoked when the system is done with a buffer and reference to the class. Next, we create one audio buffer using the `AudioQueueAllocateBuffer` function and give it the audio queue and the size of bytes it can hold.

Now, we wait for the parser to invoke its packets received callback. Then, we fill an empty buffer with the packets. There are two possible formats for the data received from the parser, VBR or CBR. Variable Bitrate (VBR) means that the bit rate can change from packet to packet where as Constant Bitrate (CBR) means that it will be constant.

In the case of VBR, we can only fill the buffer with whole packets which contain many bytes. This means that the buffer may not fill up before we have to send it to the system. With CBR, we fill the buffer to the brim and then send it along.

CBR

```
AudioQueueBufferRef audioQueueBuffer = [self aFreeBuffer];
memcpy((char *)audioQueueBuffer->mAudioData, (const char *)data, length);
```

We also need some logic to make sure we're not overfilling the buffer and if we don't completely fill it, we should wait for more.

VBR

```
AudioQueueBufferRef audioQueueBuffer = [self aFreeBuffer];
memcpy((char *)audioQueueBuffer->mAudioData, (const char *)(data +
packetDescription.mStartOffset), packetDescription.mDataByteSize);
```

Here, we need to check if the packet will fit in the leftover space in the buffer. If it can't fit another packet of `mDataByteSize` then we will have to get another buffer. We also need to hold on to our packet descriptions for enqueueing.

When the buffer is full, we enqueue it to the system with `AudioQueueEnqueueBuffer`.

```
AudioQueueEnqueueBuffer(audioQueue, audioQueueBuffer,
numberOfPacketDescriptions, packetDescriptions);
```

Now we're ready to play audio. When all the buffers have been filled and enqueued, we can start playing sound with `AudioQueuePrime` and then `AudioQueueStart`.

```
AudioQueuePrime(audioQueue, 0, NULL);
AudioQueueStart(audioQueue, NULL);
```

`AudioQueueStart` allows us to pass a second parameter, instead of `NULL`, that represents a time to start playing. We will ignore this for now but it could be useful if audio synchronization is needed.

The End

Those are the basics for streaming audio through Multipeer Connectivity. At the end of this adventure I created a open source library that brings together everything described here in a more organized and structured fashion. If you'd like more detail, the complete code and examples are on GitHub at tonyd256/TDAudioStreamer.