WBOR STREAMING APP:

A DOCUMENTATION OF THE NARRATIVE

Connor Smith

MOBILE COMPUTING

**Introduction**

Originally I planned on having an app that would serve the following functions: grab the web stream, grab the current playlist, and perhaps implement a DJ tool to allow DJs to chart songs right on their iPhone, iPad, or iPod touch. Unfortunately, writing the code to handle the stream proved to be a much greater task than expected and ultimately took over the bulk of my project.

**The difficulties of streaming**

When I went out to start writing the code for streaming, it wasn't immediately clear how I could do this. At first I tried using the AVAudioPlayer class, but it became apparent that it wouldn't handle m3u streams. I saw that there was another, seemingly older, approach to handling streams, using the AudioQueue class. This code looked terribly daunting and I was still somewhat set on the idea that there was a method that would handle all the network stuff for me and just grab the mp3's from the m3u URL. I sought out help from Ben Johnson (class of 2011). He checked with one of his colleague developers at Raizlabs. It turned out that I would indeed have to use the AudioQueue class, which is the hardest API in iOS to understand because it is simultaneously very complex and scarcely documented (not a good combo). But, I liked the idea of a challenge so I went ahead and started making attempts and grasping how a select few other people used it. I made very little actual progress for a long time. I mostly was trying to figure out what types of methods these people were calling, what the methods were doing, and when/how to invoke them.

On the Friday before it was due, I came to you in a state of near panic. I had been making progress on a very high level both working through AudioQueue examples as well as setting up the framework for my apps tab bar layout and MVC paradigm. After we talked I decided to try to "reverse engineer" the audio streamer example from Matt Gallagher. Meanwhile I started to build my own streaming model from scratch with the limited, but steadily increasing, knowledge of handling network streams I was gaining. After really stripping down the example app, I found the AudioQueue stuff to be....almost straightforward, but not quite. Basically what I do in my app is I call the start method, which performs two primary tasks. It initializes the audio session so we can start playing audio, and then calls openstream, a method that reads a stream from the url I have sent in from my FirstViewController. Next comes the Queue code. Essentially here I am creating buffers, filling them up with mp3's parsed from the stream, sending them to the AudioQueue object, and playing them from the AudioQueue session while the buffers refill with more data. This process repeats indefinitely. If I wanted to implement the ability to track to earlier recorded parts of the stream, I would have to constantly write the data in the buffers to an audio file. My main reason for now implementing this was my concern that the file could become very large and my app would be very bloated.

**Other Issues I Ran Into**

**Callback Functions:** One thing that really perplexed me about AudioQueue was the use of callback functions. It seemed I had to write some C functions, whose name (but not parameters) I would pass as a parameter in some of my calls to lower level Core Audio type functions. My understanding is that the functions to which I passed these callbacks as a reference to would call the callback functions on their own terms with their own parameters. There was no way around using these callbacks that I didn't QUITE get because they are taken as required parameters in a few system-like calls I need to make.

**Handler methods:** The "handler" methods were another thing that went over my head at first. Basically these methods handle different states of action from the buffers. They reset them when they (the buffers) are full, and make all of the calls with the callback functions as parameters. They keep the program perpetually running based on the state of the buffers. If the buffers are empty, let's refill them. If we have audio packets from the network, let's put them into buffers appropriately.

**Stopping the stream:** You would think this would be pretty simple. False. My early attempts to stop the queue resulted in a very temporary moment of silence but the stream would continue running after that (even when I told the AudioQueue to stop). What I ended up having to do was stop the queue, then manually clear all of the buffers, then dispose of the audioqueue (which is supposed to clear the buffers for me, but that didn't seem to work on its own). So I finally got the audio to stop, but I noticed in my debugger that a few of my methods were still being called perpetually. I think this had to do with the way my handler methods and callback functions were interacting on their own. I wasn't satisfied with an app that still is taking up extra CPU time when it's not supposed to be doing anything, so I baked a state-like check into the system in a few key locations that basically only runs if the system is not supposed to be stopping.

**The Stream Works! But Now What?**

**JSON:** So now that I had a stream that could start and stop and a simple UI to handle this, I set out to start grabbing other data from the website. Luckily for me, WBOR does not include any (but maybe some) metadata in its stream. Instead I had to make calls to the website to get the playlist data. Unfortunately, the playlist section of the website is down currently due to a bug that Harrison Chapman (the prior webmaster) and I need to fix. I was however, able to grab a JSON file that contained the current song as well as some other (mostly useless) information. New to iOS5 is a JSON parser that can easily read JSON data and toss it into an objective-c data type. The JSON file I grabbed from WBOR contains a single dictionary so I but all of the data in an NSDictionary and then Grabbed the string from a specific key in the dictionary and cut it to the appropriate length.

**Future Work:** I am still planning on implementing the ability to chart songs, although this will be in a separate DJ tool app. Having this in the same app as the streamer would be a conflict of interest for the user as well as for WBOR management. My current plan is to gather data from the user on a form in a view. I will but this data into an NSDictionary and then convert it into a JSON object. I will then send this JSON object to the wbor.org server. In order to handle the server-side stuff and actually make the site accept the data and use it, I will need to write backend API in the website code. I'm thinking about how I can create a spot either on the website or as a tab item on the app that shows where all the users listening to WBOR app are. The app would send the user's location (if the user agrees obviously) to the website and then receive the aggregate data of location coordinates. I also will improve the .xib views based on reaction from the public and management when I release the app this spring.

**Conclusion**

Despite the fact that my app could not be completed in time with full functionality, I think the functionality it does serve as a great tool for a mobile device. I have programmed a mobile device to continually pickup a network stream and read it into a playable form. Before you had to be within the very limited broadcast range to hear it on radio (about 15 minute drive in any direction from Bowdoin, varying with the weather) or at a computer with WiFi. Now you can listen to WBOR wherever you go using 3G or WiFi! I found myself wanting this ability most when I'm driving at home or anywhere that isn't Bowdoin. Although it has no utility in employing fancy touch gestures, it makes use of one of the key advantages of any mobile device: mobility. Equally important, it has laid the groundwork for me to implement more mobile features in the future, as mentioned earlier.