# Party Mobile iOS Application Design Report

Anthony Olivares and Zarah Gutierrez

Computer Engineering Department

Under the advisement of Dr. Hugh Smith

California Polytechnic State University, San Luis Obispo, United States

anolivar@calpoly.edu and zgutierr@calpoly.edu

# Table of Contents

# Abstract

The Party Mobile Application project was created in order to give users the power of a surround sound system at their fingertips. Following the ideas of apps such as AmpMe and SoundSeeder this application aims to synchronize audio between phones in order to play music. Just like surround speakers, these phones would play music throughout a room, car, or house for the enjoyment of users. Party Mobile also aims to give people the ability to share their music with the rest of the world with its playlist upload functionality. Any user can upload a playlist of their music to the application's server, and then play it in a session for a group of people. Additionally, all playlists uploaded to the server are public so users can also listen to playlists compiled by other people. With its various functionalities, the Party Mobile Application aims to give users the ability to share, listen, and enjoy the music with their friends anywhere in the world.

The application accomplishes this goal through two primary means. The first is utilizing Bluetooth as the primary method of communication between devices. All phones in a session connect to each other using Bluetooth and send each other the necessary data to play music synchronized. Alternatively, the application can also use wifi to send data to phones; however, this method is situational based on the security settings of whatever network your phone is on. In addition, to this, the application also uses an Amazon AWS server to store all of the playlists created by users. Utilizing the Swift coding language Party Mobile is able to interact with the AWS server directly, and download songs and playlists to user phones. These two parts of the application, when put together, create the experience that Party Mobile was created to give its users. In the rest of this report, readers can learn about the details behind the apps two main features as well as the development cycle of the application.

# Introduction

## Goals and Objectives

PartyMobile aims to design and implement a mobile application that allows users to share and listen to music synchronously. This application will allow a small group of individuals in the same room to play and listen to a song at the same time. It will also enable users to create public listening "sessions" in which a single user can share and listen to music with a larger group composed of any user around the world. With this application, users will be able to create and upload and share music playlists from their device's local storage and download playlists other users have made.

## Deliverables

Upon completion, the project should provide a music sharing and listening service through a fully-functional mobile application. The mobile application will be compatible with iOS and/or Android devices and available to the public through a mobile application store(s).

## Outcomes

Currently, the system is a fully functional iOS application that allows users within a close rage to listen to music synchronously; its major features include uploading songs, creating song playlists, and starting and/or joining private sessions of small, select group of users within close proximity to each other. The application is able to synchronize music between multiple users simultaneously within a few milliseconds apart, performing better than initially expected. But, due to copyright infringement concerns, we did not implement the public "sessions" that allows any user around the world to join; the concept of sharing music to any user is similar to broadcasting music, which requires proper licensing. Implementing public "sessions" would

have presented different challenges because it requires different communication technologies. WiFi and Bluetooth communication are limited by distance; devices that are located out of reach require a backend server that handles all inter-device communication, which requires more resources and time than the current implementation.

Users can upload songs from their local storage and create playlists to be used during listening sessions.All application users can access and upload to the same repository of songs and playlists, which needs to be altered in the future in order for private “sessions” to be considered as fair use. Users should only have access to songs and playlists that they own and songs that they listen to via private sessions.We were not able to implement a user-based authentication system that restricts users’ access to their own set of playlists and songs since the main focus of the project was to synchronized music sharing.

# Background

## Similar Systems

In the early stages of this project research was done on existing products that were similar to the product that this team wanted to create. This was done to avoid copyright issues as well as prove that our project was feasible with real world examples. The first we discovered was AmpMe [5]. This mobile application allows users to sync the audio of their phones or laptops and play songs together. This app was essentially what we wanted to create so the team knew that our app could not be put on the Apple’s application store. However, we agreed to continue making the app so that we could learn how to utilize bluetooth as a means of communication as well as gain experience working with Amazon’s AWS server.

## Bluetooth Research

Bluetooth as a communication protocol relies on broadcasted services as a means to communicate data across devices [6]. One of the primary reasons that bluetooth was considered,

and eventually implemented was because it provided uniform structure for data transmission that is defined by the Attribute Protocol (ATT) and the Generic Attribute Profile (GATT) built on top of ATT [6]. In typical setups one device acts as the slave broadcasting a specific service while the master searches for the service and eventually connects to the device. It can then utilize various frameworks to access the characteristics inside the broadcasted service to obtain relevant data. This one-to-one connection is how devices such as airpods work with apple phones and laptops. Bluetooth also has the capability to handle communication between multiple devices; this type of setup is called a piconet, which is the setup this project will be creating between connected devices [7]. When multiple piconets become connected to each other through bridge devices that can communicate on both the setup becomes a scatternet [7].

In order to utilize Bluetooth for IOS development it is easiest to utilize one of Swift's built in frameworks. This project initially utilized the Core Bluetooth framework because it focused solely on bluetooth data communication and did not try to incorporate wifi [4]. It also allows developers to create their own custom services and characteristic, which as described above is how data is communicated through Bluetooth. However, this framework also requires developers to ensure that their services adhere to the GATT standard, and due to a bug that will be discussed in a later section this framework was removed from the application.

In its stead MultipeerConnectivity was implemented [8]. This framework removes the low level service creation of Core Bluetooth and only requires developers to specify a unique ID and session for each device. It then utilizes built in delegate functions to connect devices, send, and receive data. This simplified the creation of this app communication protocol so that we only needed to set up the connection, and then could send whatever data we wanted, and the framework would handle sending the data to the target device.

## Amazon Web Services Research

Amazon Web Services provides computing platforms to individuals, companines, and the government. Our system utilizes some of its major services, such as Amazon Simple Storage System (S3) and Amazon Cognito. Amazon S3 is a storage service that stores any type of object

that are arranged into “buckets.” Objects stored in an S3 bucket is identified with a unique key. Storing songs and song information is an integral part of this system, which required a stable, simple file system. Amazon S3 provides a viable storage system solution that can easily organize and store songs uploaded using this application. It also allows our system to scale easily when more users use this application. On the same note, we used Amazon Cognito, which controls user authentication and permissions for mobile applications, because it allows our system to easily integrate more users in the future while maintaining proper use access rights by handling all user authentication workload. Amazon Cognito ensures only users with the proper permissions are able to access information from the application’s S3 bucket.

## Copyright Issues

Since this project’s main feature involves sharing copyrighted material, it is important to consider copyright laws during the application’s design and implementation process. The application must be designed such that the distribution of copyrighted material is considered as fair use.

According to U.S. laws pertaining to copyrights, playing music in a private setting with only a few people, such as at home or in a car, using a “single receiving apparatus” does not require a music license [1]. The project’s private listening “sessions” fall under this category because copyrighted material will be shared to a limited number of users that are within close proximity. Another important aspect of US copyright laws related to this project is that music cannot be “stored in a system or network… for a longer period than is reasonably necessary to facilitate the transmissions for which it was made [2].” This means that users that do not own the song must only have the song in their phone during the duration of the listening session. Lastly, music broadcasters must be licensed by the Federal Communications Commission, by a cable system or satellite carrier to broadcast music [3]. This section is important when designing the system’s “public sessions,” because the “public sessions” are very similar to broadcasting music to anyone around the world.

# Specifications

## Problems Encountered

### Unstable Connection Between Devices

In order to achieve the synchronized audio output that this project was created for; bluetooth was initially chosen as the means to transmit data between phones. This was favored over wifi because early on the team found a well documented framework in Swift known as Core Bluetooth [4] that would allow for customized services to be made on each phone and broadcasted to all devices that knew to listen for that service. Additionally, this allowed us to filter and scan for devices to connect to based on this service's unique id. It was these discoveries that led to the initial implementation of bluetooth over wifi although this was later changed.

Due to issues with maintaining a stable connection between phones using the CoreBluetooth framework the team considered transitioning into using the MultipeerConnectivity Framework in Swift. After a meeting with Professor Scovil to discuss the benefits of this possible transition, and in the hopes that this choice would be more efficient than continuing debug efforts the team decided to use the new framework. The MultipeerConnectivity Framework gave the app the flexibility to determine whether it should utilize bluetooth or wifi without the need for user intervention. As long as the team specified a "session" for each phone and implemented the necessary delegate functions in the code to tell the app how to handle new connections the framework would automatically connect devices. Additionally, the framework provides the necessary delegate functions to send data just as the CoreBluetooth Framework did. This framework also does not require designers to specify their own unique service that must follow the GATT standard. This is done for the designer, and all that is required is that each device have a unique identifier and session.

After the new framework implementation the phones were able to hold a stable connection and development of the application continued. The main cause of the bug with the CoreBluetooth Framework was never discovered, but after initial testing the team had some

hypotheses. We were able to eliminate the possibility of user error by removing all forms of input after a connection is attempted between devices. Furthermore, we were able to see in the debug terminal that the phones were initially successful in connecting, and would disconnect after a certain amount of time passed. At no time was any input given to the application nor were any errors being registered by the built in delegate functions of the framework. At this stage the team decided to transition to the new framework, but it is believed that perhaps the custom service developed for the app did not correctly conform to the GATT standard so the phones were rejecting the connection. Another, hypothesis was that the phones were mistakenly believing that they were both a master and slave at the same time, and thus severing the connection. However, no errors were raised by Swift, or the framework's delegate functions to support this hypothesis.

## Downloading Song Data Out of Order:

During the development of this application the team discovered an issue where the first song of a playlist would not be the first one to finish downloading, and thus would cause an error to occur. This edge case would happen when the first song was significantly longer than others in the playlist. The app would only wait for any song in the playlist to be done downloading before playing because it was assumed that since the information for the first song in the playlist was sent before all the others that song would finish downloading first. However, when this was discovered to be untrue the team had to alter the application's source code so that the DJ only sent the url for the first song, waited for each listener to tell them that the song was ready, and then send the urls for all the other songs in the playlist. This proved more difficult than first believed because all functions in swift are asynchronous so making this slight change required new variables be created, and the use of Swift's NotificationCenter class to allow the main thread of the application to execute functions out of its scope.

## Unable to Create Consecutive Sessions:

During the final stages of testing the team discovered that when users attempted to create a new session, after successfully creating one previously, our application would crash. This is

because in the source code the objects used to represent the bluetooth connection and the audio player were seen as null in the second session despite having worked in the previous one. It was discovered that this is because when we used the NotificationCenter class in our code we mistakenly thought that it would be able to pass data across classes. However, through immense troubleshooting we discovered that this class is only effectively at passing data to local functions within a class and not to another. In order to fix this bug we restructured the bluetooth and audio classes so that all function calls related to those objects were contained in their classes so that we did not need to use the NotificationCenter class.

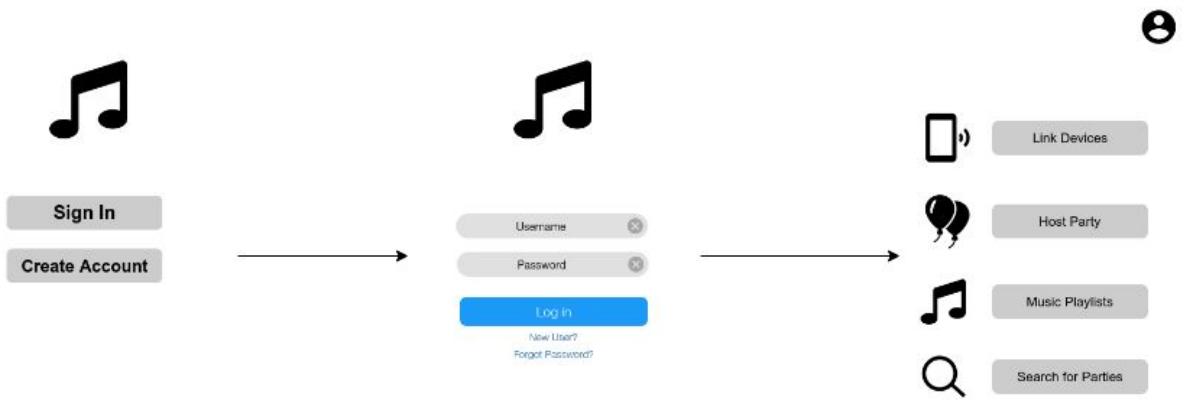


Figure 1: Main Menu Storyboard

This system has three main features, uploading song playlists, hosting a listening session, and joining a listening session (Figure 1). Users are able to upload songs from their local storage, which is an important requirement because users are using this application to share their music to others.

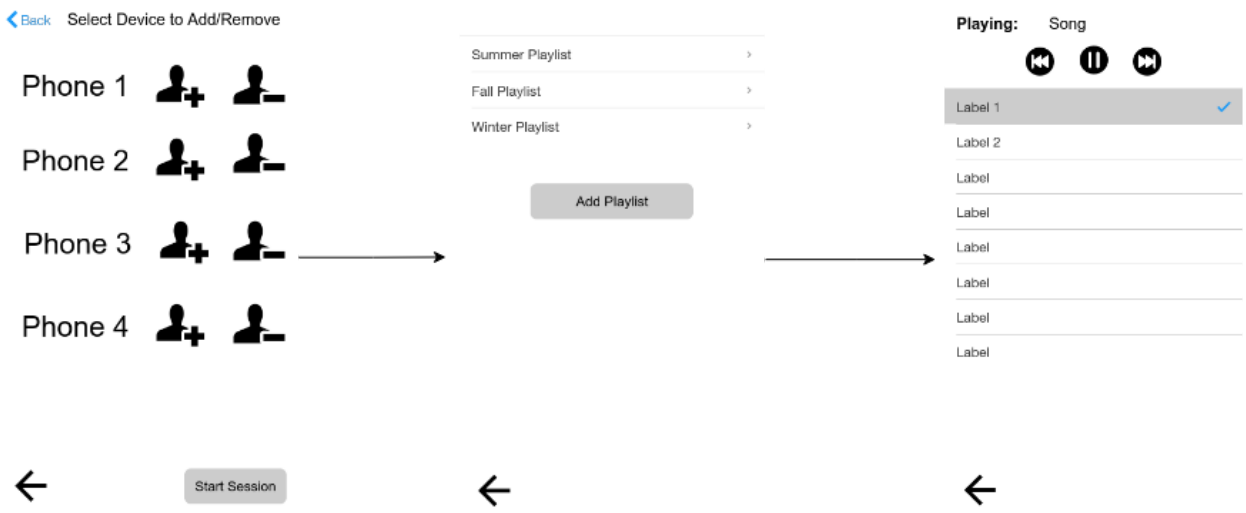


Figure 2: “Private Session” Storyboard

A “Private Session” allows a central user (also called the DJ) to start a session of four users. The screen in figure 2 displays icons that allows the DJ to scan for all users within a cross proximity. This is the core functionality of the system. Users can use this feature as a way to share music to multiple people while using headphones or earphones.

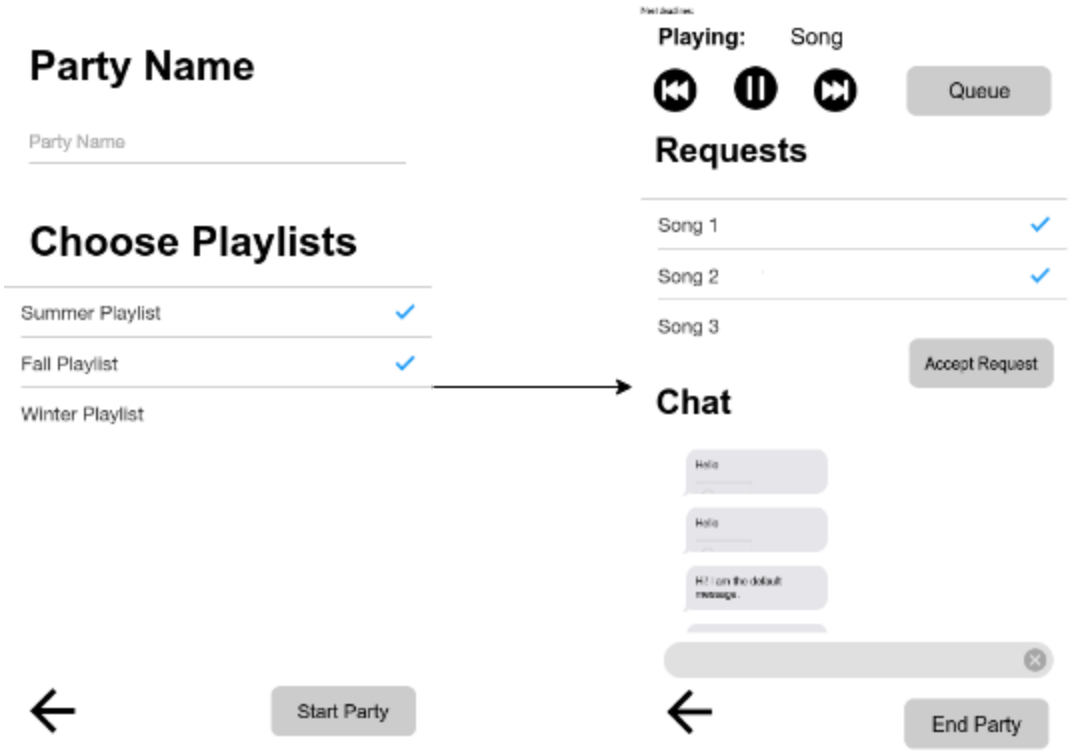


Figure 3: “Public Session” Storyboard

When setting up “Party Sessions”, the DJ can choose a playlist or multiple playlists to broadcast to multiple users. As seen in figure 3, the Party Session features a media controller that allows the DJ to control the playlist, simulating the same experience of a DJ. It also features a queue that shows the next few songs; this allows the central user to anticipate the next songs and gain more control over the order of the playlist. Although listeners do not have control over the playlist during “Party Sessions,” they are allowed to request songs during the sessions, which allows interactions between the DJ and the listeners.

# Design

## Top Concepts

In order to develop the best viable solution, the team conceptualized and compared multiple systems. The three top concepts are a “React Native Mobile Application”, an “Android

Mobile Application", and an "iOS Mobile Application." All of these concepts consider the specifications listed in this report; it's important for these systems to allow communication between multiple devices. Conceptualizing these systems involved  research in similar systems, possible users, and considering our skillset.

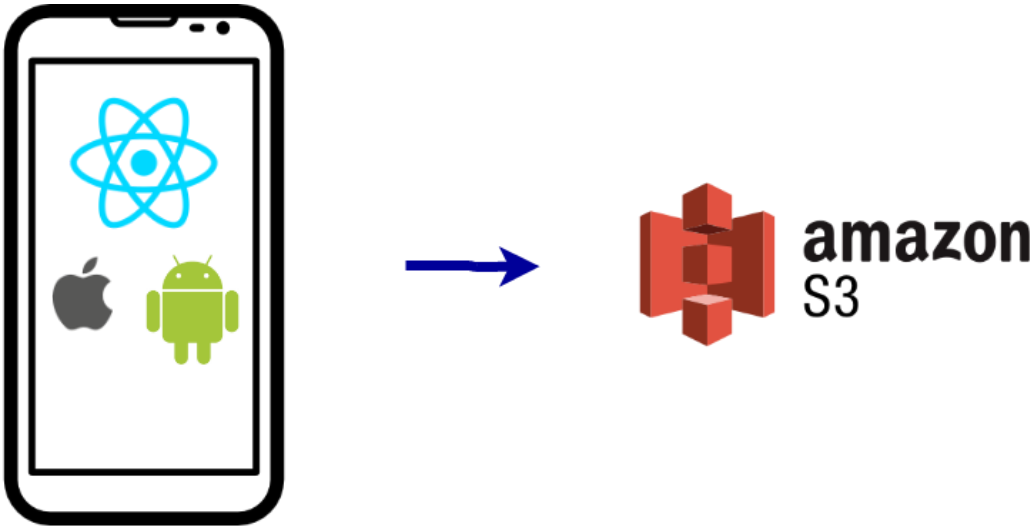


Figure 4: "React Native Mobile Application" Design

The "React Native Mobile Application," depicted in figure 4 is a Javascript based mobile application that uses Amazon Web Services' Simple Storage System for the system's remote storage. This system is cross platform, which means both Android and iOS users are able to download and use the system, but this limits the system to WiFi based device communication. As of now, cross platform bluetooth connection is not possible. The "React Native Mobile Application" would have been a viable solution because it allows most smartphones users to use the PartyMobile application, but it limits device communication methods to only WiFi, and the school's WiFi doesn't allow unauthorized applications to use it as a means of communication.

To address the school WiFi communication issue, the team conceptualized two other designs that allows both bluetooth and and WiFi connection: the "Android Mobile Application"

and “iOS Mobile Application.” A major drawback of both designs are not crossplatform; therefore, this systems will only be available to a select group of users.

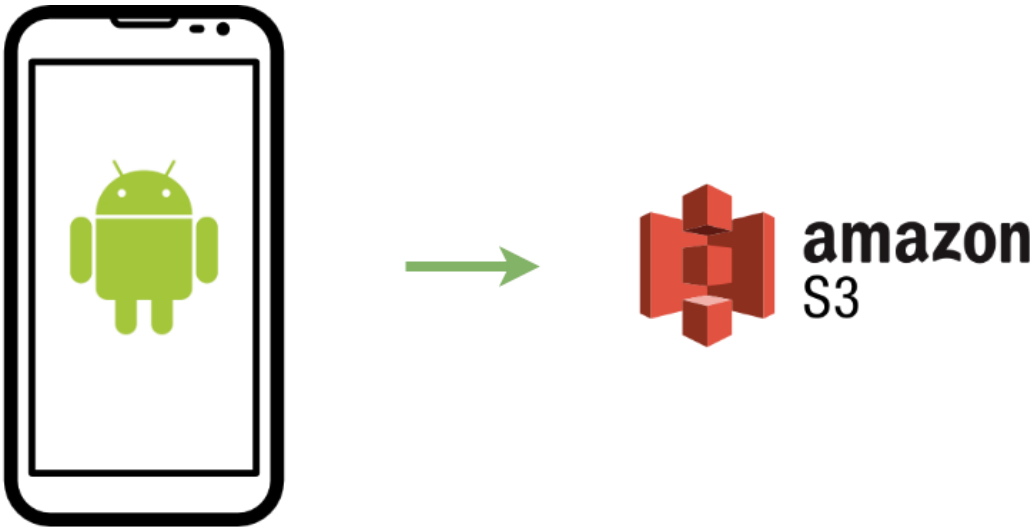


Figure 5: “Android Mobile Application” Design

The “Android Mobile Application” depicted in figure 5 is a Java based mobile application that also uses Amazon Web Services’ Simple Storage System for the system’s remote storage. This system is only compatible with Android operating systems but unlike the previous concept, it allows for both WiFi and bluetooth connection. Although the team is most familiar with working with Java and Android Studios, they did not proceed with this solution because neither of the team members owned an Android device to use for testing.

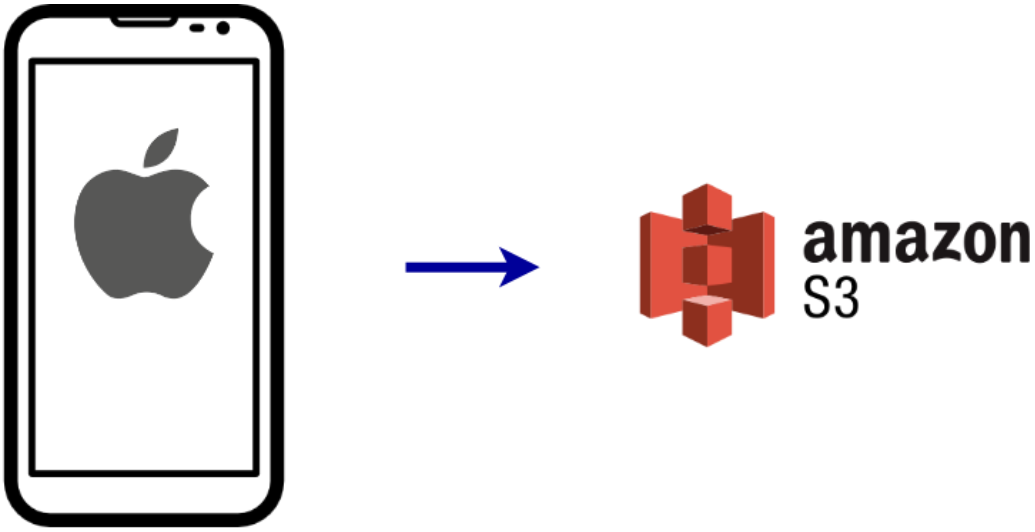


Figure 6: “iOS Mobile Application” Design

## Final Design

The final system is an iOS application that uses Amazon Web Services’ Amazon Simple Storage System (AWS S3) as the system’s database and the Apple’s MultipeerConnectivity Framework for inter-device communication (figure 6). We chose the “iOS Mobile Application” because it allows us to use both Bluetooth and WiFi, and the project team members have iOS devices to test it with. As seen in figure 7, the system allows a maximum of seven listeners and one main user who controls a listening session. All listeners have a bidirectional connection with the main user but do not have any communication with the other listeners.

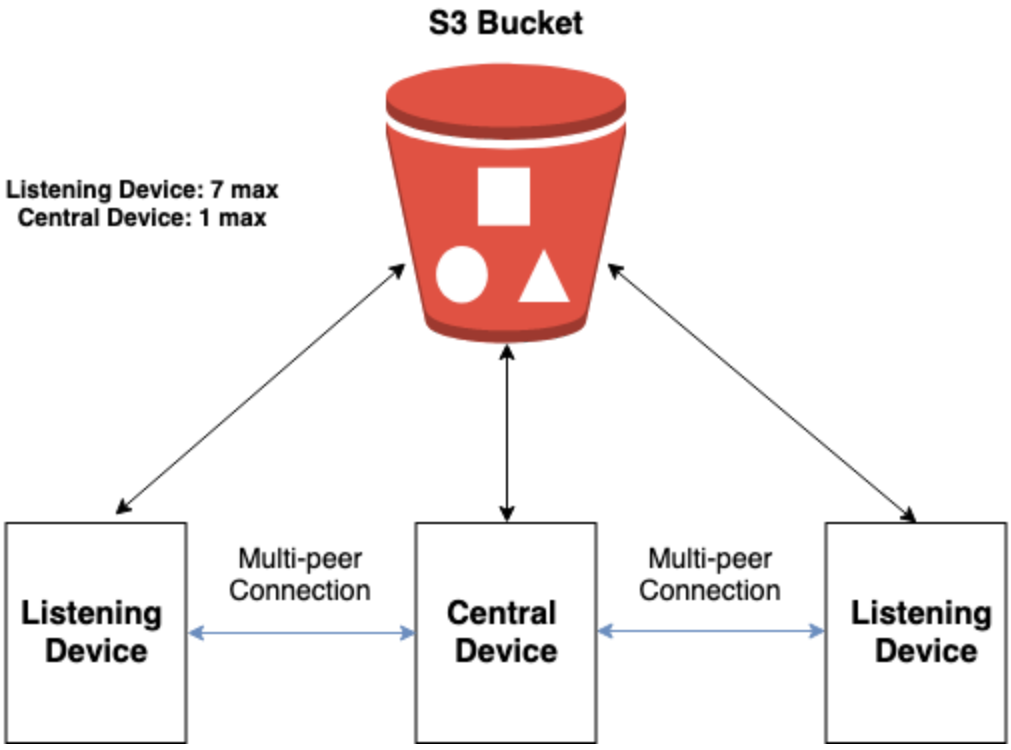


Figure 7: System Architecture

The system uses an AWS S3 bucket as its remote storage location for the users' songs. When prompted, devices download a given song from the system's AWS S3 bucket. We chose an AWS S3 bucket due to its simple design in which it stores objects with a unique associated key and AWS offers AWS Cognito, which can authenticate and regulate access to songs. Its ability to be arranged as a simple file system and online graphical user interface convinced us to choose it over another online storage system.

This application utilizes the MultipeerConnectivity (MC) Framework to establish a session between devices and allow the DJ and listener(s) to send song information between each other as well as interrupts to control the audio (Figure 7). As mentioned earlier, the MC Framework allows us to view device communication in a higher level; we didn't have to worry about choosing Bluetooth or WiFi, or adhering to GATT standards. It also provides useful

delegate functions that facilitate device communication that the CoreBluetooth library didn't provide.

# Implementation

As seen in figure 9, the AWS S3 bucket configured for this application is modeled after common file systems; each user is represented by a unique directory and the user's songs are stored under his/her directory. This allows a separation of property and other users unable to download another user's song. AWS S3 is this system's main form of storage because this doesn't require a complex storage structure; it only needs a simple remote file system that allows easy uploading and downloading of files.

Users are allowed to create song playlists that they wish to stream. These playlists are organized using lists of song titles, and these lists are saved to json files that are stored under the user's directory in the system's AWS S3 bucket. All of the user's playlist keys are stored in a standardized json file, "playlist.json" (Figure 9) .This folder structure is simple but effective for song and playlist access; the application simply downloads the appropriate json files that contain the keys of playlists and songs and access these files using their unique keys.

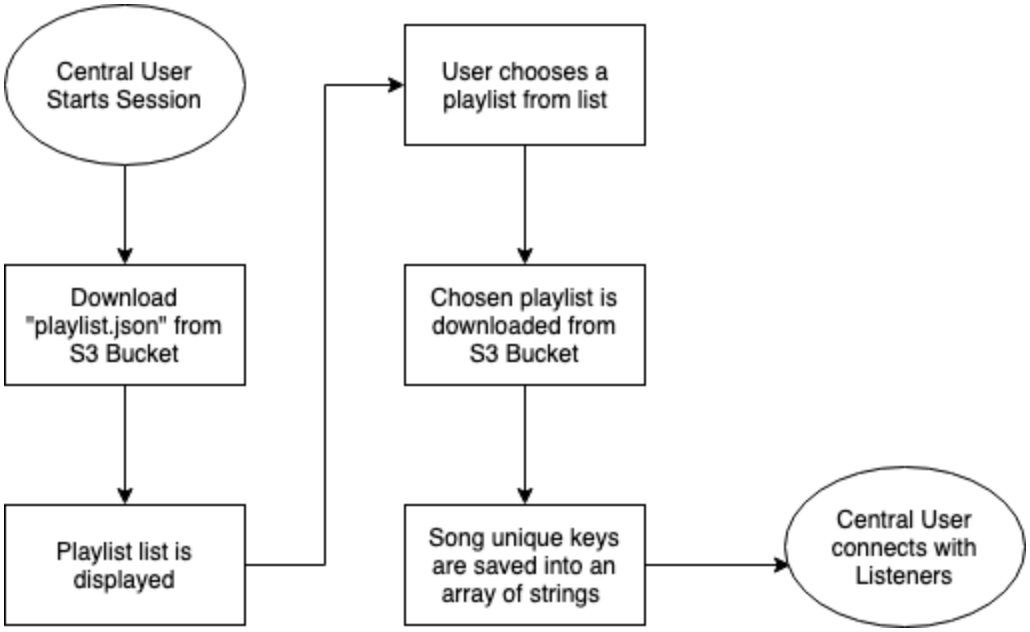


Figure 8: Song and Playlist Access Flowchart

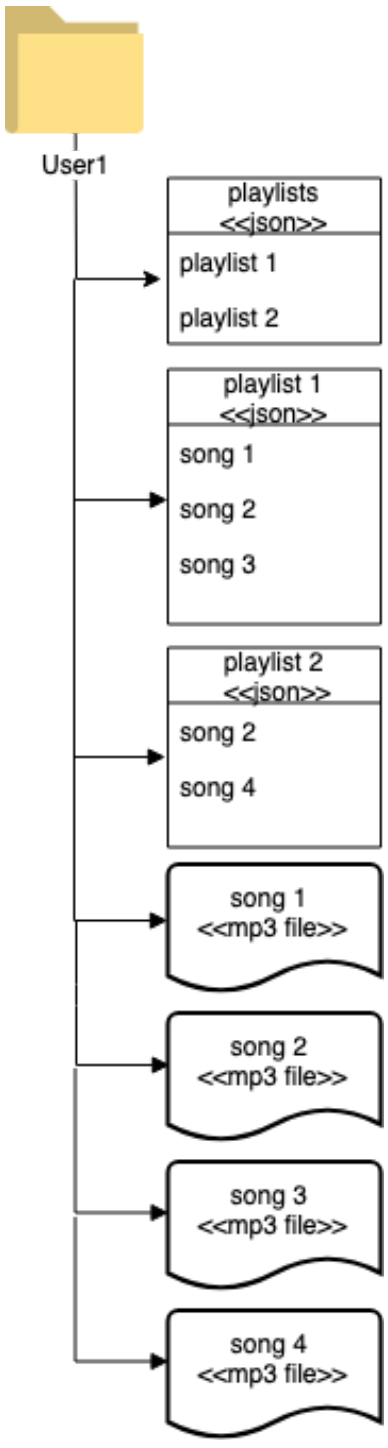


Figure 9: AWS S3 Bucket Structure

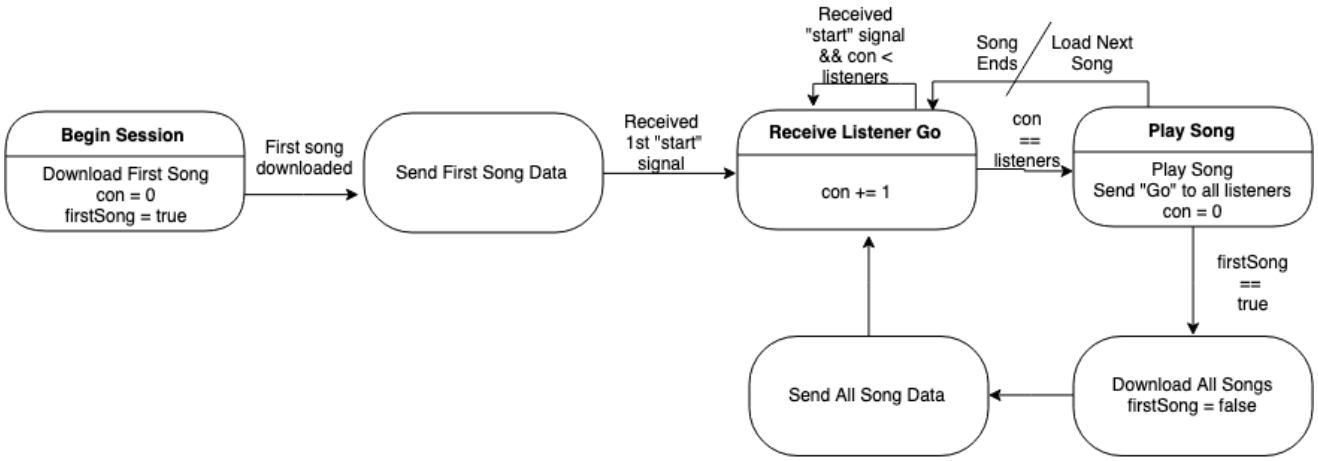


Figure 10: Central User State Diagram

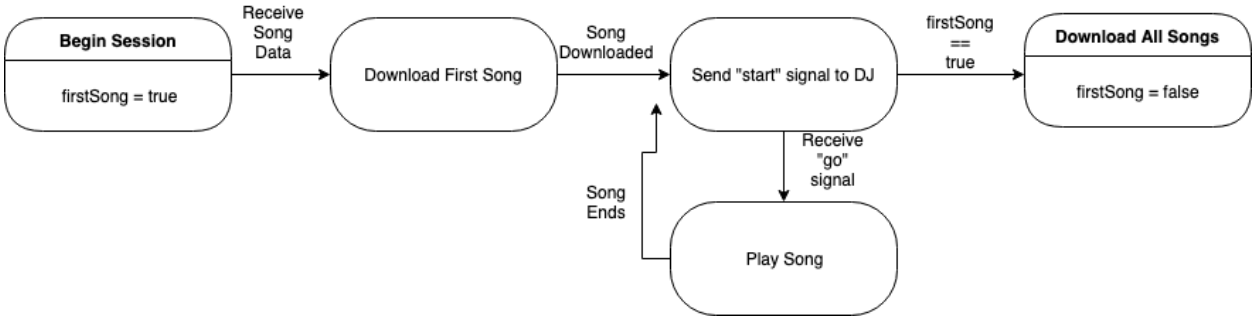


Figure 11: Listener State Diagram

The beginning of a session is the most crucial part of the whole session; it determines the level of synchronization between devices. At the beginning of a session, as seen in figure 10, the central user downloads the first song, and once the download is completed, the central user sends the first song data. This process ensures that the first song is in fact the first song that plays, because we attempted to send all of the song data at once and songs played out of order since songs download at different rates. As the first song plays, the central user and the listeners

(Figure 11 and Figure 10*)* download all of the other songs in the background. This will allow a smooth transition from song to song.

Synchronization between the central user and all listeners is an integral part of this system, and in order to provide optimal synchronization, throughout a listening session, there are multiple checks sent from listeners to the central user, vice versa. For every song, the listeners send a "start" signal when they are ready to begin the song, and when the central user receives all of the listener's start signal, the central user sends a "go" signal to all listeners and plays the song. When the listeners receive the central user's "go" signal, they will play the next song. This process will repeat for every song to ensure that users are as synchronized as possible.

# Future Work

At its current state, this system is not at its full potential. There are multiple features that are yet to be implemented, such as handling inter-device communication when the application goes in the background. Currently, when a user receives a call, opens another application, or any other task that places PartyMobile in the background, the user disconnects from the Multipeer Communication session. Since this is a common use case, this system must handle it without interrupting the user's experience. To address this problem, the central user needs to maintain a copy of peers connected to it. When the application is no longer in the foreground, for most states, a song is playing, which does not explicitly need inter-device communication because the song continues to play and does not need to send any data packets . Once the central user's application returns to the foreground, it will iterate through all of its peers and attempt to reconnect, but if it remains in the background longer than a span of the current song playing, the session is ended (Figure 13) . Listeners are notified that the session ended. On the other hand, if a listener goes to the background, the central user's number of connected devices is decremented so the central user doesn't need to wait for the disconnected user's "start" signal for the next song. Once the application returns to the foreground and the current song has not ended, the listener simply connects to the session, but if the song already ended, the listener cannot

reconnect back to the session (Figure 12) .

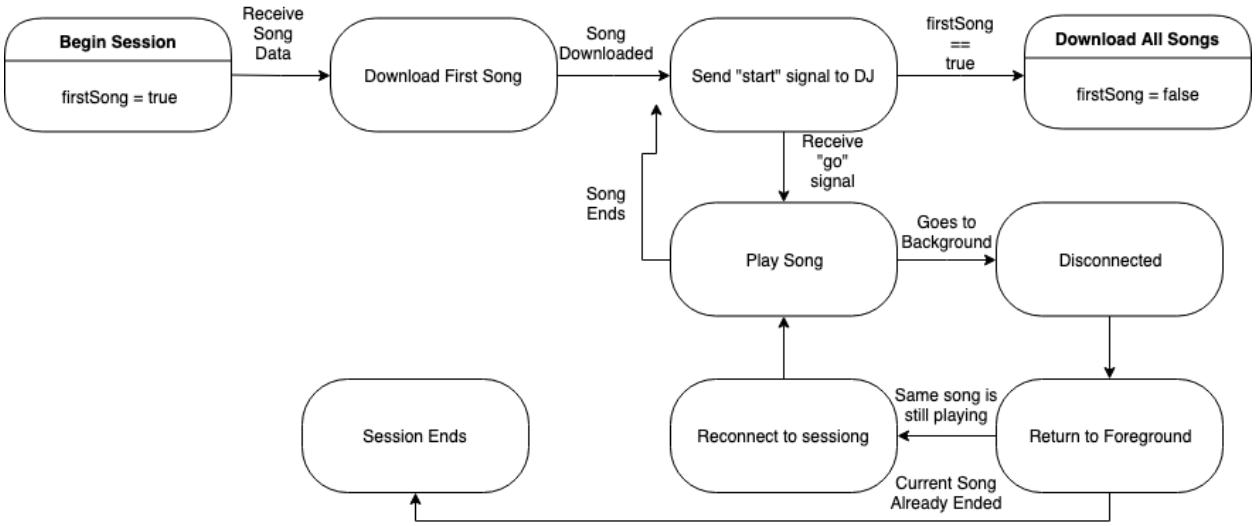


Figure 12: Future Listener State Diagram

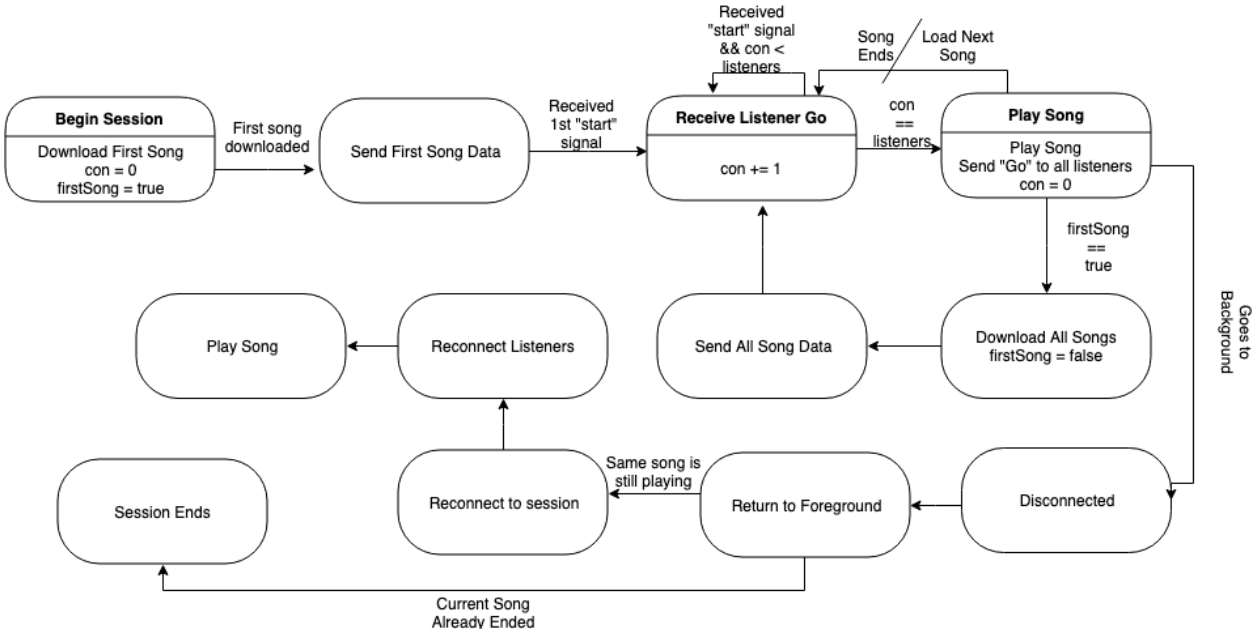


Figure 13: Future Central User State Diagram

If given more time, another feature that we would have implemented is the ability to create a user account. Each user will have the ability to create a username used when discovering devices  and a user will only have access to playlists that they uploaded and those that are shared to them via private listening sessions. Integrating user accounts involves using Amazon Cognito to store user information and handle user authentication that will allow only authorized users to access playlist and adding new screens that allows users to signup, log in, and edit user information. This feature enforces the idea of fair use since only the owner of the song will be able to share music to a select group of users.

Since streaming music is a major aspect of mobile music sharing, integrating music streaming services such as Spotify, Pandora, and Amazon Music are a great addition to the system. This involves learning and using their respective APIs and obtaining the proper rights and licenses to use these services and adding the feature to use a third-party playlist during listening sessions.

# References

[1] 17 U.S.C. §110(5)(A).

[2] 17 U.S.C. §110(11)(B).

[3] 17 U.S.C. §110(5)(B).

[4] https://developer.apple.com/documentation/corebluetooth

[5] https://www.ampme.com/

[6] https://www.bluetooth.com/specifications/gatt/

[7]https://www.tutorialspoint.com/wireless_communication/wireless_communication_bluetooth.htm

[8]https://developer.apple.com/documentation/multipeerconnectivity