Universitat Oberta de Catalunya

uoc.edu

UOC Network Media Player

Ramon Haro Marqués

Grau de tecnologies de la telecomunicació

Sistemes de comunicació

Raúl Parada Medina

Carlos Monzo Sanchez

01/2019

FITXA DEL TREBALL FINAL

Títol del treball:	*UOC Network Media Player*
Nom de l'autor:	*Ramon Haro Marqués*
Nom del consultor/a:	*Raul Parada Medina*
Nom del PRA:	*Carlos Monzo Sancez*
Data de lliurament (mm/aaaa):	*01/2019*
Titulació o programa:	*Grau en tecnologies de la telecomunicació*
Àrea del Treball Final:	*Sistemes de comunicació*
Idioma del treball:	*Anglès*
Paraules clau	*iOS, Network, Audio, Video, Sharing*

Resum del Treball (màxim 250 paraules):

Avui en dia gairebé tothom disposa de dispositius mòbils en el quals, es guarden imatges, vídeos i àudio. A més, aquest dispositius tenen la capacitat de reproduir contingut audiovisual des-de plataformes com, d'entre altres, són YouTube o Spotify.

Es pot afirmar que, amb aquests dispositius, la manera com es consumeix productes audiovisuals avui en dia, tant al carrer com a la llar, ha canviat i antics sistemes de reproducció com el vinil i el reproductor de CD han sigut substituïts per aquests nous dispositius.

A més de tot això, els sistemes de reproducció estant, cada vegada més, tendint a tenir connectivitat sense fils. D'aquesta manera, es pot reproduir el contingut d'un dispositiu mòbil a un altaveu sense necessitat de cap tipus de cable. D'entre altres, el més utilitzat avui en dia és el Bluetooth encara que nous protocols estant agafant força. Un d'ells es el que es coneix vulgarment

com ‘Wi-Fi Speaker’ o altaveu a través del Wi-Fi i que fa servir el protocol UPnP.

Aviat ens donem compte que el Bluetooth té limitacions a la hora de connectivitat: necessita ser emparellat, número limitat de dispositius connectats simultàniament, etc.

Aquest projecte demostrarà com fent servir el protocol UPnP un dispositiu mòbil pot seleccionar contingut audiovisual tan sigui de la memòria interna com d’altres dispositius connectats a una mateixa xarxa i reproduir-la localment o enviar-los a altres dispositius o altaveus com, per exemple, un altre dispositiu mòbil, una Smart TV or els coneguts altaveus Sonos.

Abstract (in English, 250 words or less):

Nowadays, almost everyone has mobile devices in which images, videos and audio are saved. In addition, these devices have the ability to reproduce audio-visual content from platforms such as, among others, YouTube or Spotify.

It can be said that, with these devices, the way in which audio-visual products are consumed today, both in the street and at home, has changed and old reproduction systems such as the vinyl and the CD player have been replaced by these new devices.

In addition to all this, reproduction systems are increasingly tending to have wireless connectivity. In this way, the content of a mobile device can be played on a speakerphone without the need for any type of cable. Among others, the most commonly used today is Bluetooth, although new protocols are taking on force. One of them is what is commonly known as ‘Wi-Fi Speaker’ or speaker via Wi-Fi and uses the UPnP protocol.

We soon realize that Bluetooth has limitations in connectivity: it needs to be matched, a limited number of devices connected simultaneously, etc.

This project will demonstrate how using the UPnP protocol a mobile device can select audio-visual content, either from the internal memory or from other devices connected to the same network and play it locally or send it to other devices or speakers such as, another mobile device, a smart TV or the well-known Sonos speakers.

Index

1. Introduction 8

1.1 Context and project motivation 8

1.2 Project goals 11

1.3 Approach and method 12

1.4 Project scheduling 13

1.5 Brief summary of the product obtained 14

1.6 Brief description of all the other chapters of the memory 15

2. State of the art 18

2.1 Smartphones, tablets, wearables and TV Boxes 18

2.1.1 iOS devices 18

2.1.2 Android devices 22

2.2 Streaming Services 23

2.2.1 Spotify 24

2.2.2 Netflix 24

2.2.3 BBC iPlayer 25

2.2.4 YouTube 25

2.3 Wireless media receptors 26

2.3.1 Smart TV 26

2.3.2 Sonos Speakers 26

2.3.3 Amazon Alexa 27

3. The UPnP protocol 28

3.1 Overview 28

3.1.1 UPnP architecture 29

3.2 Network configuration 30

3.2.1 Addressing 30

3.2.2 Discovery 30

3.2.3 Description 31

3.2.4 Control 32

3.2.5 Event notification 33

3.2.6 Presentation 34

3.3 UPnP AV architecture and services 35

3.3.1 UPnP AV architecture 35

3.3.2 MediaServer services 37

3.3.3 MediaRenderer services 38

4. OpenHome library 41

4.1 Overview 41

4.2 OpenHome list of GitHub projects 41

4.3 Services 43

4.4 Compiling the library 45

4.4.1 Steps for compiling for different structures 46

4.4.2 Fat library creation 49

4.4.3 Composing the OhNet folder 50

5. iOS APP Project 51

5.1 XCode 51

5.2 Mac and iOS programming languages 51

5.3 First steps for developing the APP project 53

5.3.1 Creating projects and files 54

5.3.2 Setting up an Xcode project for working with C libraries 57

5.3.3 Dependency manger in iOS 61

5.4 Implementing the library in Swift 64

5.4.1 Proxy implementation and delegation 64

5.4.2 Provider types and lists of actions for services 67

6. App flow and test 69

6.1 Verifying the provider implementation and inspecting proxies 69

6.2 App flow and functionalities 71

6.2.1 Initialisation and default devices selection 71

6.2.2 Main UIViewControllers 72

7. Conclusions 77

8. Bibliography 79

List of figures

Figure 1 Internet and mobile users in 2018 8
Figure 2: Sonos device 10
Figure 3: Smart TV 10
Figure 4: Amazon echo 10
Figure 5: Google Chromecast 10
Figure 6 Project scheduling using a Gantt diagram 13
Figure 7 iPhone 1 18
Figure 8 Collection of iOS devices 19
Figure 9 iPhone completing a transaction with Apple pay 20
Figure 10 Collection of Apple watches 20
Figure 11 Apple TV with Siri voice control remote 21
Figure 12 HomePod speaker 21
Figure 13 Android Pixel 3 and iPhone X 22
Figure 14 UPnP discovery example 30
Figure 15 UPnP device description process 31
Figure 16 UPnP device description XML example 32
Figure 17 UPnP control point - device interaction 33
Figure 18 UPnP GENA example 34
Figure 19 UPnP presentation example 34
Figure 20 UPnP AV device interaction model 36
Figure 21 UPnP AV general device architecture 37
Figure 22 OhNet Github clone process 42
Figure 23 OhNet project cloned to the computer 42
Figure 24 OhNet servce.xml file 43
Figure 25 OhNet XML services folder 43

Figure 26 OhNet regeneration terminal process 46
Figure 27 OhNet build folder generated 46
Figure 28 OhNet compilation for iOs-armv7 debug and release using the terminal 46
Figure 29 Build folder after first compilation 47
Figure 30 Multi compile command process 47
Figure 31 Final compilation bundles 47
Figure 32 OhNet lib folder for arm64 bundle 48
Figure 33 Include folder for arm64 bundle 48
Figure 34 OhNet terminal command to check the .a file is libc++ 49
Figure 35 OhNet terminal command to generate all the fat libraries 49
Figure 36 Include folder from OhNet 50
Figure 37 OhNetBuild folder completed 50
Figure 38 Xcode iOS project type choices 54
Figure 39 Xcode project basic configuration information 55
Figure 40 Xcode project and targets view 55
Figure 41 Xcode general target configuration 56
Figure 42 Website to generate the app icons 56
Figure 43 Xcode app icon setup 57
Figure 44 Xcode dummy objective-C creation 57
Figure 45 Xcode bridging header creation request 58
Figure 46 Xcode prove that the bridging header is created 58
Figure 47 OhNet generated folder copied into the Xcode project's root folder . 58
Figure 48 .a imports to Xcode 58
Figure 49 Xcode search path configuration for release 59
Figure 50 Xcode header search path configuration 59

Figure 51 Xcode c++ linker flag configuration 60
Figure 52 Xcode standard CLANG C++ library configuration 60
Figure 53 Xcode bridging header declaring the OhNet headers to import 60
Figure 54 Carthage cartfile creation 62
Figure 55 Carthage dependency declaration 62
Figure 56 Carthage dependencies installation process 63
Figure 57 Carthage downloaded dependencies 63
Figure 58 Carthage frameworks imported to an Xcode target 63
Figure 59 OhNet connection manager Constructor and De-constructor 64
Figure 60 Synchronous action example 64
Figure 61 OhNet Connection manager asynchronous begin action 65
Figure 62 OhNet Connection manager asynchronous end action 65
Figure 63 Swift begin action 65
Figure 64 Swift end action 66
Figure 65 Function for setup de notification 66
Figure 66 Function to be called after the notification is triggered for obtaining the updated value 66
Figure 67 Swift implementation for OhNet notification 66
Figure 68 Swift Connection Manager proxy protocol 67
Figure 69 Swift Connection Manager proxy protocol variable 67
Figure 70 SourceProtocol and SinkProtocol available 68
Figure 71 Swift computed variable for source protocol info property 68
Figure 72 Cydia Impactor interface 69
Figure 73 Network device inspection 70
Figure 74 List of actions and properties for AVTransport service 70
Figure 75 Invoking play action for AVTransport 71

Figure 76 Server selection 72
Figure 77 Renderer selection 72
Figure 78 Library view controller 73
Figure 79 Library view controller with servers dropdown 73
Figure 80 Popup play menu 74
Figure 81 Popup play menu with renderers dropdown view 74
Figure 82 Play queue UOCApp_iOS-27f4 74
Figure 83 Play queue UOCApp_iOS-27f4 with renderers dropdown view 74
Figure 84 Play queue 192.168.0.26 - Sonos Play:1 74
Figure 85 Now playing UOCApp_iOS-27f4 75
Figure 86 Now playing 192.168.0.26 - Sonos Play:1 75

1. Introduction

1.1 Context and project motivation

Nowadays, we live in a world where telecoms have hugely grown during the last decade. Most of this growth is due to the mobile and the Internet era. According to the latest publication from '*We are social*' and '*Hootsuite*', I can say that worldwide in 2018 (McDonald, 2018).

- The number of Internet users is 4.021 billion, up 7 per cent year-on-year.
- The number of social media users is 3.196 billion, up 13 per cent year-on-year.
- The number of mobile phone users is 5.135 billion, up 7 per cent year-on-year.

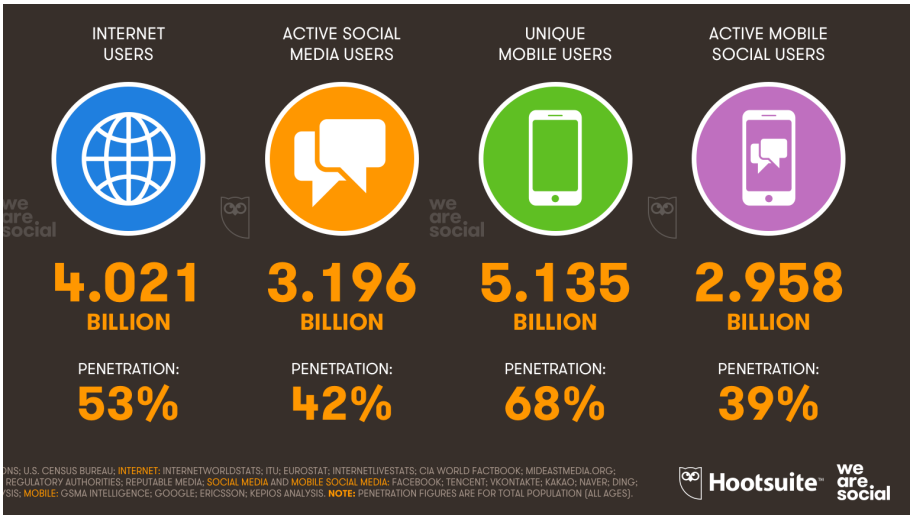

Figure 1 Internet and mobile users in 2018[1]

On the other hand, this growth comes with devices with high computational power, high definition display, high quality image and video capture, high internal storage memory and high Internet speed connection. If we have a look closer to the Internet speed increase in recent years, we will be able to identify another success. That is indeed, media streaming services such as *YouTube*, *Netflix* and *Spotify* and IPTV services among others.

Furthermore, in the last decade connectivity between devices has followed the tendency of wireless.

So far, I have identified that half of the population worldwide has a smartphone, goes online for consuming audio-visual products, stores its personal videos, images and audio in a single device and they have the need to share content.

After I have identified the new needs, I can come to the problem this project is intended to tackle. People has media and they want to share or stream it onto another device. Imagine that a group of people are meeting after their summer holidays and they want to show pictures and videos taken with their mobile phones to share their experiences. How do they achieve that? Let's see a couple of options and explore the negatives of these.

- **Device base**
    - The media content is on the physical device and it goes from hand to hand. In this case, you need to verbally repeat the description of the photo every time it changes from one hand to another.
    - Lack of control over what is being viewed and the possibility of showing private or intimate content.
- **A Slideshow on a DVD or USB**
    - Wasting time preparing the content and the music.
    - Not all televisions have a DVD player or USB built in.
    - Usage of non-reusable storage (DVD).
    - Limited time on each image.
- **Cable extension to the television**
    - Although a simple option, consideration for cables compatible for both Android and iOS is required.
    - A lightning to HDMI connector for iOS is expensive.
    - Cables have length and space limitations.
- **Bluetooth player**
    - Does not support video
- **Apple TV or Android box**
    - Screen mirroring isn't possible through an Android box.
    - An Apple TV box doesn't have USB connection.
- **Cloud based app**
    - Login needed for each different user.

I have explored one possible scenario of sharing photos and videos so now let's imagine that you are at home and you want to listen music and stream it into one or many rooms. How can we solve this problem today?

- **Install speakers and wires into the different bedrooms.**
    - Expensive and demands the need to hide the cables.
    - If reallocation of a speaker is needed, more installation and corrections need to be made.
- **Bluetooth speakers**
    - Not many devices are available which support many speakers at the same time.
    - These are expensive.
    - If supported, the number of devices is reduced due to its broadcast specifications.

In addition to these problems, I need to consider an extra problem that concerns Apple users. If you have an iOS device, you might have experienced how tedious it can be getting media both in and out from a device to another non-Apple device.

At this point, I am going to suggest the platform that can and will replace completely the way audio-visual content is shared between different devices in the future. The platform is called *UPnP* (Universal Plug and Play) and the library that handles the media services is called '*OpenHome'*. You may not have heard about it, but it has been around for a while and I am sure that you might know some products that implement this protocol.

Figure 2: Sonos device[2]

Figure 3: Smart TV[3]

Figure 4: Amazon echo[4]

Figure 5: Google Chromecast[5]

UOC Network Media Player is an iOS application which will implement the UPnP services needed for sharing media content, will be compatible at least with Sonos devices, Smart TV's and Android devices in the same network and it will be able to get content from any device that implements the media server and share it to another device that implements the media renderer through its services. In the same way, any device will be able to share with the iOS app. It sounds confusing but I am going to outline how it is done.

I have chosen to develop an iOS application because there are not many options available on the market, and as mentioned above sharing content from Apple to non-Apple devices can be frustrating.

1.2 Project goals

Having given a brief project introduction describing the current context and needs which inspired this project, the goals below show what this project intends to achieve:

- An explanation of the history of *UPnP* protocol and its services.
- An explanation of the *OpenHome* library that handles the UPnP protocol and its services.
- An exploration of the *OpenHome* library and its compilation for iOS Devices. (The library will be compiled as a C library. Others will be shown but not compiled.)
- An outline of the compiled library and the creation of a *fat library* (a library with multiple architectures) for supporting these architectures: ARMV7, ARM64, x64, x86.
- An Xcode project creation and setup for working with C libraries.
- A brief explanation of Apple's programming languages and why Swift is used.
- An implementation for the iOS *OpenHome* API .
- An implementation for the app's User Interface (UI).
- Create an IPA (app file installation) from the Xcode project for sideloading (install from outside the Apple store). In addition, and depending on the results, the app may be also updated to the Apple iOS app store.

1.3 Approach and method

One year ago, I started working as an iOS developer at Convert Technologies[6] which is based in Newbury in the United Kingdom. The company has a product called 'Plato' which is a high-resolution UPnP box that uses an Android device as well as a screen for controlling the box. The UPnP implementation at that time was developed using Android only, however the company wanted to develop an iOS app for interacting with the box with the same capabilities that the android device had.

The existing iOS project was coded using Objective-C with an inefficient structure; an inadequate model; notification abuse; abuse of segues and other issues. The first project I took on as an employee of Convert Technologies was a reimplementation of the user interface. Within this project I faced many struggles due to re-using the existing services, their lack of scalability and I soon realised that the API which implemented the C library needed to be rewritten.

I was able to complete a basic reimplementation of the user interface before moving onto a completely different project within the company, however, six months ago I returned to working on the iOS UPnP app. At that moment I essentially started the app from scratch, reinventing the library compilation due to its use of libstdc++. I was yet to manage several challenges during this time, including being the sole iOS developer within the company, having limited knowledge about *OpenHome*, dealing with poor or non-existing documentation for iOS as well as many others. Although I was faced with these challenges the UPnP and *OpenHome* gradually became an obsession and something I feel very passionate about.

Gathering all the knowledge I have gained in the last six months I will begin by re-writing the app from scratch, compiling the library, setting up the Xcode, and building the API structure and proxy services. At this point the providers remain unknown to me.

Today the internet is full of forums, blogs and videos with huge amounts of self-learning content on the widest range of topics and I intend to utilise this feature available to me when undertaking this project. References will be included to demonstrate how I have achieved something or come to a particular method if

need be. A huge aim for this project is to share knowledge and standards for other iOS developers and I feel this strategy will help to achieve this.

1.4 Project scheduling

Having set the goals for this project, each needs to be expanded and tasks assigned. Within this section I will outline the tasks and how they relate to each of my initial goals.

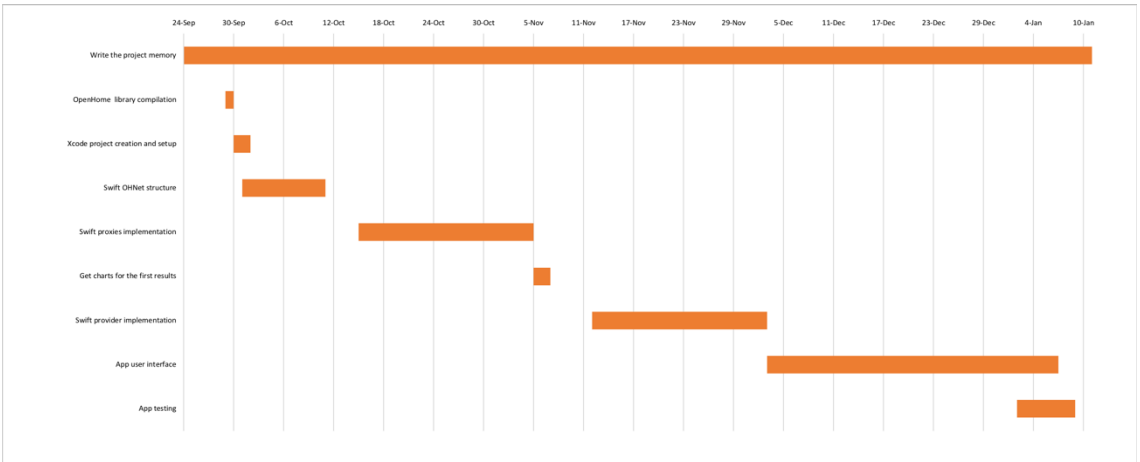

Figure 6 Project scheduling using a Gantt diagram

1. Write the project memory: This task will be ongoing for the whole project. All the documentation for the app development will be written as the app grows - 109 days.
2. OpenHome Library compilation: This task will include all the library compilation process for each structure and the creation of an iOS universal library or fat library - 1 day.
3. Xcode project creation and setup: This task will cover all the steps needed for creating an iOS app project in Xcode and will set up the project for including the library - 2 days.
4. Swift OhNet structure: This task will start writing in Swift code the API structure for implementing the library - 10 days.
5. Swift proxies implementation: This task will write all the proxy classes in Swift. I will describe more about proxies in the following sections - 21 days.
6. Swift provider implementation: This task will write all the provider classes in Swift. I will outline more about providers in the following sections - 21 days.

7. App UI (User Interface): Design for the app visual aspects and the user interactions - 35 days.
8. App testing: Perform tests for discovering bugs and crashes - 7 days.

1.5 Brief summary of the product obtained

For this project I set a number of goals to be achieved. Most of these goals were UPnP and *OpenHome* descriptions and Implementation because I thought it would be really important to have a clear beginner background about the technology used. One of the products I have developed through this project is a guide that:

- Explains the UPnP technology and history.
- Explains the *OpenHome* library (OhNet) that handles a UPnP communication between devices on the same network.
- Defines the OhNet GitHub repository, describes the key components and shows step by step how to build the OhNet library with libc++ and for multiple architectures.
- Shows step by step how to set up the Xcode environment for working with the C libraries.
- Identifies the Swift codding key components that will handle the proxies and the provider functionalities.

In addition to that, the other product I have developed in this project is a mobile application for iOS devices that can be found as a git repository in my Bitbucket account[7]. The reason I have my repositories in Bitbucket is because it gives me more functionalities for my free account such as private repositories or the possibility to update files over 100 Mb.

Furthermore, the repository contains a folder called ‘Releases’ that contains all the IPA files. This file will install the app in your iOS phone by sideloading using a program called ‘Cydia impactor[8]’. The app is also available for downloading on the Apple iOS app store. The app is available under the name ‘UOC Net MP’.

Having a closer look to the mobile application I can say that this:

- Contains the OhNet library that I consider another product.

- Implements the OhNet initialisation and manages the addition and communication of the services.
- Implements both the renderer and server services for proxies including:
    - AVTransport: UPnP renderer service.
    - RenderingControl: UPnP renderer service.
    - ConnectionManager: UPnP renderer and server service.
    - ContentDirectory: UPnP server service.
- Implements the renderer provider services including:
    - AVTransport: UPnP.
    - RenderingControl: UPnP.
    - Playlist: *OpenHome*.
    - Info: *OpenHome*.
    - Product: *OpenHome*.
    - Time: *OpenHome*.
    - Volume: *OpenHome*.
    - Radio: *OpenHome*.
    - ConnectionManager: UPnP.
- Provides a user interface (UI) that follows a demo flow. The first step for the demo is to select a *MediaServer* and a *MediaRenderer*. They can be changed during the app flow. After that, the app allows the user to browse content from devices and play them into other devices including the app itself. Furthermore, the app also offers the possibility to set volume, progress and play next or previous track.
- Commented and clean code.

1.6 Brief description of all the other chapters of the memory.

Finally, before finishing the introduction section, I will describe the following chapters and what they contain.

Firstly, in section 2 I will explain the state of the art outlining the current situation about people's behaviours and technologies within this project. In this section I will be speaking about:

- History of smartphone devices from their apparition in 2007 to the present moment (2018) and how much they have evolved.

- Streaming services where I will enumerate 4 of the most important by audio, movie/tv shows, online TV and all type content videos. Also, I will explain the evolution of the streaming services and how successful they are nowadays among people around the world.
- Wireless devices that, in fact, use the UPnP protocol. In here I hope you can see idea of having a mobile app that is able to interact with all of them.

Then, in section 3 I will explain the theory about the UPnP protocol. All the content for this section is a summary from all the UPnP documentation provided in their website which is around 300 pages long. Some parts of this content can be, of course, found online. The theory included contains:

- Quick overview defining the key components of the UPnP architecture.
- Network configuration that explains the steps required for a device to discover and interact with another device.
- Explanation of the UPnP AV sub architecture that, in fact, is the base of this project. Here I will define concepts related to the iOS app such as *MediaRenderer*/*MediaServer* and the media content that can be exchanged.
- Introduction to UPnP services which are the responsible to perform a set of actions between devices.

Furthermore, in section 4 I will start explaining how to make real all the theory from the previous section by introducing the OhNet library. Unlike the previous section, this one will be really practical, and I will:

- Compile step by step the library for iOS that will make all the UPnP technology possible.
- Speak about how services are defined and two different set of services; The UPnP and the *OpenHome* ones.

Finally, in section 5 I will explain Apple's developing history, tools and programming languages and I will explain in a practical manner:

- All the steps needed to create and setup an Xcode project and description of the main components for this type of projects.
- All the steps for importing the C OhNet library.

- Dependency manager and the third-party libraries I will use.
- Introduction of the proxy and provider terminology.
- Swift techniques used in the project for implementing the communication from app to service and service to another device service.

2. State of the art

So far, I have described the project and its necessities. I am now going to outline an overview about technologies and products that motivated this project. I will also discuss some other apps and products that you can find on the market already.

In this overview I will cover:

- Smartphone devices
- Streaming platforms
- Wireless media receptors

2.1 Smartphones, tablets, wearables and TV Boxes

2.1.1 iOS devices

In 2007, Steve Jobs introduced the iPhone and iPhone OS 1 alongside it. During the press conference, Jobs referred to the operating system as OS X because it shared a similar Unix core compared to the full-fledged desktop version of the operating system. When Apple launched the iPhone SDK one year later, the name changed to iPhone OS (Orf, 2016).

The first iPhone is one of the most important gadgets of all time. It took ideas from within the budding mobile industry and made them more people-friendly. It also created the basic 'Springboard' app—a grid of apps on a screen—that hasn't changed much in nine years.

Figure 7 iPhone 1[9]

iPhone OS introduced multi-touch and the general underpinnings of Apple's ideas for mobile computing, but the operating system's greatest triumph was selling the idea that an iPod, camera, phone, and internet machine could really be packed into one device that fits inside your pocket.

On July 11, 2008, Apple dramatically expanded the capabilities of its mobile operating system with iPhone OS 2. The new version added third-party apps (in what is now known as the App Store) and location services through the newly added GPS unit on the iPhone 3G. Apple also introduced its MobileMe cloud software, but the idea never quite took off.

For the next iOS releases, Apple introduced new features such as push notifications, facetime, iCloud, iMessage, maps and the revolutionary Siri. Furthermore, a bigger device called an iPad was introduced and the iOS was integrated for the new generation of iPods. These devices started to gain peoples acceptance and they already fitted into more and more people's lives, allowing them to perform a large number of actions and tasks. With the same device people could go on a trip using GPS, take pictures, share them either on social media or via messages, and make phone calls, as well as much more. Nowadays, that all sounds normal, but a new era had started.

Figure 8 Collection of iOS devices[10]

However, it was not until 2014 where, according to the new apple's CEO Tim Cook, after Steve Jobs' death, the biggest change to iOS occurred. The operating system was completely overhauled with a simpler design, flattering icons, and Helvetica font. Apps were given edge-to-edge designs and the operating system included a new parallax-scrolling home screen. Apple also introduced the frosted glass Control Centre, for quick access to options like flashlight, Bluetooth, and another new feature, AirDrop making a file share process seamless between

Apple devices. iPhone also introduced Touch ID and Apple pay bringing a new mind-blowing security concept into smartphone. With your fingerprint you could unlock your phone and authorise actions like paying in stores with the device's NFC chip.

Figure 9 iPhone completing a transaction with Apple pay[11]

Alongside this, Apple created the concept of having a wallet inside the device where people could store credit cards and tickets. So, in addition of going on a trip, people could store theatre and flight tickets amongst others, and pay for goods with the same device. However, the expansion of Apple pay has moved forward slowly, and still today only certain countries implement it.

In recent years, Apple released the Apple watch that brings even more portability to these daily routines and, despite its functionalities being much more reduced than the iPhones or iPads, it brings addons for controlling them. This offers the possibility to check emails and reply with your voice, make phone calls, play music and keep track of your exercise.

Figure 10 Collection of Apple watches[12]

Finally, Before I conclude with this brief introduction to iOS devices, I would like to discuss two more Apple devices. The first one is called Apple TV, it was released before the iWatch, however, it is a key part of this project.

Apple TV is a digital media player and micro console. It is a small network appliance and entertainment device that can receive digital data from specific sources and stream it to a capable television.

Apple TV is a HDMI-compliant source device. To use it for viewing it has to be connected to an enhanced-definition or high-definition widescreen television via a HDMI cable. The device has no integrated controls and can only be controlled externally, either by an Apple Remote or Siri Remote controlled device (with which it is sold) using its infrared/Bluetooth capability, by the Apple TV Remote app (downloadable from Apple iOS App Store) on iOS devices, such as the iPhone, iPod Touch, iPad, and Apple Watch, using its Wi-Fi capability, or by some third-party infrared remotes.

The second product and the newest type of product released by Apple is the HomePod which is a Wi-Fi high definition speaker. By using Air Play, iOS devices can share media content from one device to another and even play video into bigger screens with the Apple TV or play music by using the HomePod.

Figure 11 Apple TV with Siri voice control remote[13]

Figure 12 HomePod speaker[14]

Certainly, if you are an Apple user, you have an all integrated solution for sharing all types of media content. However, one of the big issues with Apple devices is its obsession for creating Apple products for Apple devices. The HomePod, despite its great sound quality had a very poor acceptance due to its incompatibility to play music from streaming services like Spotify or from Android devices which is the biggest platform competitor for these types of devices.

2.1.2 Android devices

So far, I have explained one of the biggest smartphone platforms on the market, the iOS, but there is another one today that offers more or less the same capabilities but with some differences regarding licensing and file managing. This is the Android platform which started releasing smartphone devices after the first iPhone was released.

Currently, despite a large number of legal battles at the beginning for copyright and patents (similar to the Windows ones), these two platforms are leading the smartphone market and they continue to keep releasing new features. When one platform releases something, the other then also implements it and the other way around too.

Figure 13 Android Pixel 3 and iPhone X[15]

Android is a mobile operating system developed by Google, based on a modified version of the Linux kernel and designed primarily for touchscreen mobile devices such as smartphones and tablets. In addition, Google also has Android TV, Android tablets and Android watches, each with a specialized user interface. Variants of Android are also used on game consoles, digital cameras, PCs and other electronics (Contributors, Android, 2018).

Android has its own official device model but since Android is an open source project, different companies can develop different smartphone models hence meaning Android devices have a larger number of devices sold.

I won't go any further into the Android platform, but I will enumerate the main differences between iOS and Android devices which refers to the project need.

- Android allows the usage of external memory such micro SD cards or OTG USB which make file transfers from / to the device easier.
- The file system is public and the user can create folders freely and organise public files with almost no restrictions.

2.2 Streaming Services

It is more than probable that if someone asked you less than 8 years ago what would you think about either watching television on your laptop or in the same device you used for making calls and some internet browsing, you would have said, 'there is no need, that you had a 32 inch television,' at least I did anyway. It turns out that nowadays one way or another, I consume more television by either using my 15-inch laptop or my 7-inch phone than I consume with my 48-inch television.

Furthermore, television is not the only thing that people consume much more today. The huge success for the laptop and the phone that has taken over the television is due to what are called *streaming platforms*. Theses platforms specialise in different types of content and they offer a large variety of music, video, live television, television catch-up and, as we will see when we speak about YouTube, the new television.

If someone asked me 8 years ago about these services, I would have said I hope they arrive soon so there is no need to spend hours at a blockbuster store choosing a DVD to watch. Despite this in their early years they were very different to how they are now, these platforms have gained a real importance in recent years and that is strongly related to the devices discussed in the previous chapter.

After introducing what and how important streaming platforms are, we will have a deeper look into the most important ones for each type I described before.

2.2.1 Spotify

Founded by Daniel Ek and Martin Lorentzon in 2006 in Stockholm, Sweden, Spotify was the first successful music streaming platform and the most important today.

The Spotify application was launched on 7 October 2008. While free accounts remained available by invitation only to manage the growth of the service, the launch opened paid subscriptions to everyone. At the same time, Spotify AB announced licensing deals with major music labels (Contributors, Spotify, 2018).

With Spotify people can have a free account with adverts or subscriptions with no adverts, as well as the possibility to download songs to listen to offline. With both accounts users can create and share playlists and see Facebook friends playlists. In the last year Spotify launched a videoclip service.

2.2.2 Netflix

For a video streaming platform, the most significant is Netflix. With its headquarters located in Los Gatos, California, Netflix is the world's leading internet entertainment service with 130 million memberships in over 190 countries, enjoying TV series, documentaries and feature films across a wide variety of genres and languages. Members can watch as much as they want, anytime, anywhere, on any internet-connected screen. Members can play, pause and resume watching, all without commercials or commitments (Contributors, Netflix, 2018).

Today Netflix is an online streaming company however, it started in 1998 as the first online DVD rental company. Users, including myself, because this is how I first used Netflix in 2011, received the orders by post and returned them when they finished watching it.

Despite the streaming service being first introduced in 2007, the content was poor and at least in the United Kingdom in 2012 it only had best-selling films and some national content.

Currently, Netflix has improved massively and it offers all the content by streaming only. It creates a large variety of what to watch suggestions and it

generates a large number of in-house TV shows and movies for a reasonable monthly subscription.

The downside however is that much film and tv based content is still missing and to use it you need to have a subscription as opposed to other video streaming platforms such as *Now TV* and *HBO*.

2.2.3 BBC iPlayer

In United Kingdom BBC iPlayer is the biggest online and catch-up television streaming platform. By using either the smartphone or TV box app or the website users can watch live content online or programs and shows aired in the past.

These streaming platforms depend on each country so there is not a worldwide company that have all the live tv and catch-up television. If you are in Barcelona '*TV3 a la carta'* would be the equivalent.

2.2.4 YouTube

I am not sure how aware were in February 2005 the PayPal employees that created the video streaming platform YouTube.

At the beginning it was not a platform but a video sharing website where users could upload, share and view video content. In summer 2006 the website already had grown up massively hosting more 65.000 new video uploads per day and with an average of 100 million views per day. It was also in 2006 when Google bought YouTube (Contributors, YouTube, 2018).

In a bit more than 10 years, it has radically changed the way how people consume video content achieving an average of 7 billion of hours of video watched a month. YouTube is currently the third most visited website and the second largest search engine in the Internet.

When people say that YouTube is the television is because it allows to user to create channels allowing them to share a huge variety of self-created video content with news, tutorials, reviews, among others.

2.3 Wireless media receptors

So far, we have seen how much the mobile technology and the online streaming technology has grown in less than 10 years. In fact, overall technology has grown in a large number of sectors and, in this section, I will speak about another technology that is a key component for this project.

Wireless media receptors are widely spread in our society and the most common technology for these devices is indeed the Bluetooth one. However, with the fast Internet and IOT (internet of the things) proliferation, other technologies have gained more popularity. This technology is known as either the Wi-Fi speakers or network speakers but, in addition to play media content over the same network, they also can get voice control orders and interact with other IOT appliances.

In this section we will see some products for the above type of devices and how they can play media content and stream it to other devices within the network.

2.3.1 Smart TV

It would be simpler if we start describing a device that you would know for sure. The term ‘smart’ in these devices is based on the capability to connect to the internet however, in many situations the term ‘smart’ is used for naming products or even things.

The smart TV contains applications and a market which offers the possibility to obtain apps. These apps are mostly catch-up and streaming services, despite being similar to the TV boxes we saw earlier they are not quite the same.

These devices are capable to stream content from the aerial, the HDMI, the USB, and others. They are also UPnP ready and ideal to share video content.

2.3.2 Sonos Speakers

Sonos is an American company founded in 2002 by John MacFarlane, Craig Shelburne and Tom Cullen based in Santa Barbara, California. Sonos is widely known for the smart speakers it develops and manufactures.

MacFarlane introduced a prototype at the 2004 Consumer Electronics Show, which was released in 2005 as a bundle called the Digital Music System. The company expanded upon the prototype and product design, adding mesh

networking with AES encryption to allow the speakers to play music simultaneously in multiple rooms. Between 2011 and 2014, the company released numerous speakers and added more services. They worked with Bruce Mau incorporating a rebrand of the company, which took effect in 2015. The company has partnered with other companies adding to their catalogue of services Heart Radio, Spotify, MOG, QQ Music, and Amazon Music (Contributors, Sonos, 2018).

They are also partnering with Amazon to enable Alexa to control Sonos speakers, intending to eventually work with every voice assistant on the market. By the end of 2018 Google Assistant will also be supported by Sonos in 2018.

Sonos devices are by far my favourite devices for listening to music, not only because of the sound quality but also for the simplicity on having devices in each room and being able to easily join them in groups.

2.3.3 Amazon Alexa

This device developed by the American company Amazon is now widely spread, mostly in English speaking countries such the USA or UK, China and some others. Other countries like Spain are still waiting for Alexa to understand Spanish voices.

Alexa is a network speaker that has less quality than the Sonos devices but what makes Alexa great is its capability of performing actions by using voice and without pressing any remote, by just saying: ‘Alexa, play that song’ in the vicinity of the device.

Furthermore, if you have a smart TV you can order Alexa to play content from a network device.

3. The UPnP protocol

This chapter explains the UPnP protocol, its architecture, how two devices in the same network can discover one another and, finally, I will explain the UPnP AV architecture that implements the services for *MediaRenderers* and a *MediaServer*s and what services the app will implement. This section is a selection of documentation that can be obtained from the UPnP website[16] (Contributors, UPnP (Universal Plug and Play), 2018).

3.1 Overview

Universal Plug and Play (UPnP) is a set of networking protocols that allows devices to seamlessly discover each other's presence on the network and establish functional network services for data sharing, communications, and entertainment. A Network device can be a personal computer, a printer, internet gateways, mobile phones, network attached storages (NAS) and Wi-Fi access points. (UPnP, 2008)

The UPnP technology was promoted by the UPnP Forum, a computer industry initiative to enable simple and robust connectivity to stand-alone devices and personal computers from many different vendors. The Forum consisted of over eight hundred vendors involved in everything from consumer electronics to network computing. Since 2016, all UPnP efforts are now managed by the Open Connectivity Foundation (OCF).

UPnP assumes that the network runs Internet Protocol (IP) and then leverages HTTP, SOAP and XML on top of IP, in order to provide device/service description, actions, data transfer and eventing. Device search requests and advertisements are supported by running HTTP on top of UDP (port 1900) using multicast (known as HTTPMU). Responses to search requests are also sent over UDP but are instead sent using unicast (known as HTTPU).

Conceptually, UPnP extends plug and play which is a technology for dynamically attaching devices directly to a computer to zero configuration networking for residential and SOHO wireless networks. UPnP devices are 'plug and play' in that, when connected to a network, they automatically establish working configurations with other devices.

3.1.1 UPnP architecture

UPnP technology defines an architecture for pervasive P2P (peer-to-peer) network connectivity of intelligent appliances, wireless devices, and PCs of all form factors. It is designed to bring easy-to-use, flexible, standard-based connectivity to ad-hoc or unmanaged networks whether in the home, in a small business, public spaces, or attached to the Internet. UPnP technology provides a distributed, open networking architecture that leverages TCP/IP and the Web technologies to enable seamless proximity networking in addition to control and data transfer among networked devices.

The UPnP Device Architecture (UDA) is more than just a simple extension of the plug and play peripheral model. It is designed to support zero-configuration, 'invisible' networking, and automatic discovery for a breadth of device categories from a wide range of vendors. This means a device can dynamically join a network, obtain an IP address, convey its capabilities, and learn about the presence and capabilities of other devices. Finally, a device can leave a network smoothly and automatically without leaving any unwanted state behind.

The technologies leveraged in the UPnP architecture include Internet protocols such as IP, TCP, UDP, HTTP, and XML. Like the Internet, contracts are based on wire protocols that are declarative, expressed in XML, and communicated via HTTP. Using internet protocols is a strong choice for UDA because of its proven ability to span different physical media, to enable real world multiple-vendor interoperation, and to achieve synergy with the Internet and many home and office intranets. The UPnP architecture has been explicitly designed to accommodate these environments. Further, via bridging, UDA accommodates media running non-IP protocols when cost, technology, or legacy prevents the media or devices attached to it from running IP.

What is 'universal' about UPnP technology? No device drivers; common protocols are used instead. UPnP networking is media independent. UPnP devices can be implemented using any programming language, and on any operating system. The UPnP architecture does not specify or constrain the design of an API for applications; OS vendors may create APIs that suit their customers' needs.

3.2 Network configuration

Now the UPnP protocol has been defined, it is time to describe how devices can be discovered in a network. The steps for this process are addressing, discovery, description, control, event notification and presentation.

3.2.1 Addressing

The foundation for UPnP networking is IP addressing. Each device must implement a DHCP client and search for a DHCP server when the device is first connected to the network. If no DHCP server is available, the device must assign itself an address. The process by which a UPnP device assigns itself an address is known within the UPnP Device Architecture as AutoIP.

3.2.2 Discovery

Once a device has established an IP address, the next step in UPnP networking is discovery. The UPnP discovery protocol is known as the Simple Service Discovery Protocol (SSDP).

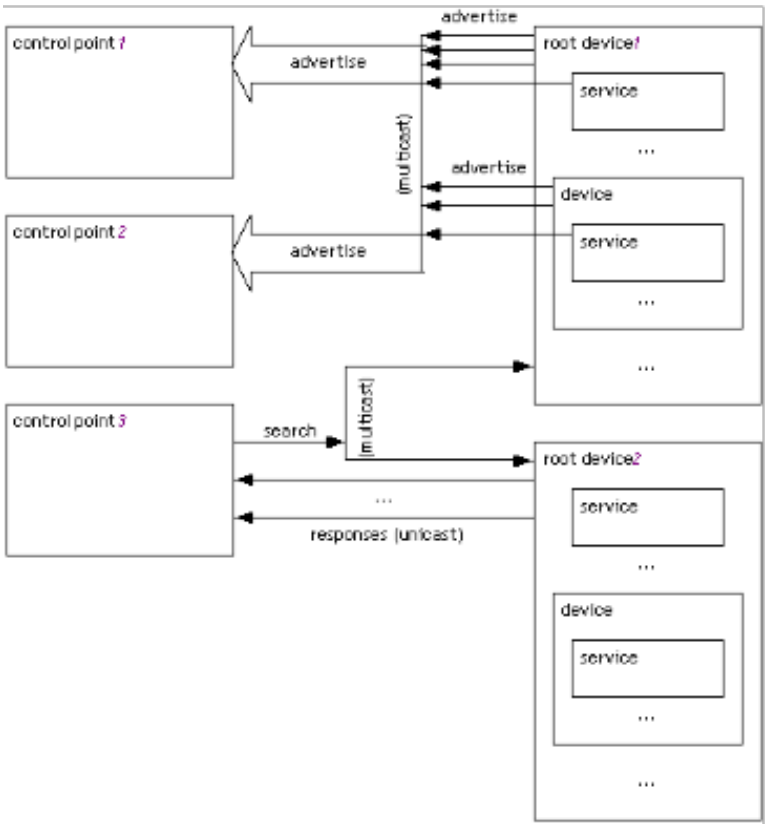

Figure 14 UPnP discovery example

As Shown above, when a device is added to the network, SSDP allows that device to advertise its services to control points on the network. This is achieved by sending SSDP live messages. When a control point is added to the network, SSDP allows that control point to actively search for devices of interest on the network or listen passively to the SSDP live messages of a device. The fundamental exchange is a discovery message containing a few essential specifics about the device or one of its services, for example, its type, identifier, and a pointer (network location) to more detailed information.

3.2.3 Description

After a control point has discovered a device, the control point still knows very little about the device. For the control point to learn more about the device and its capabilities, or to interact with the device, the control point must retrieve the device's description from the location (URL) provided by the device in the discovery message.

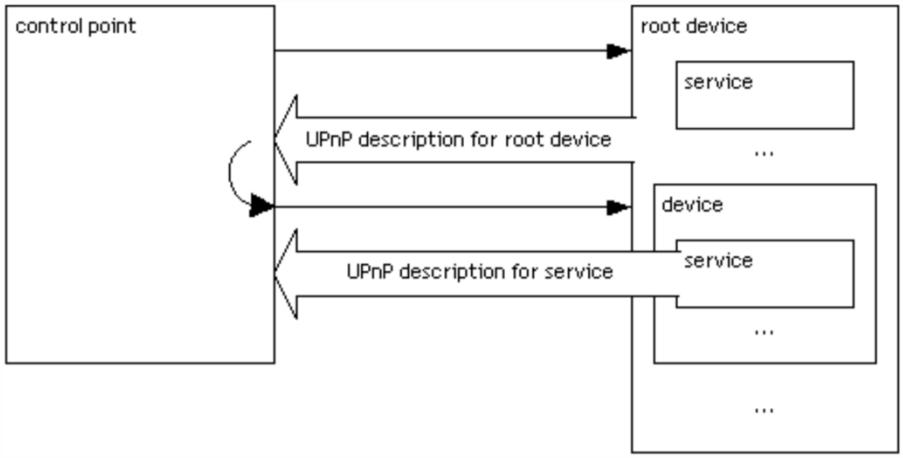

Figure 15 UPnP device description process

The UPnP Device Description is expressed in XML and includes vendor-specific manufacturer information like the model name and number, serial number, manufacturer name, (presentation) URLs to vendor-specific web sites, etc. The description also includes a list of any embedded services. For each service, the Device Description document lists the URLs for control, eventing and service description. Each service description includes a list of the commands, or *actions*, to which the service responds, and parameters, or *arguments*, for each action;

the description for a service also includes a list of variables; these variables model the state of the service at run time, and are described in terms of their data type, range, and event characteristics.

```
<?xml version="1.0"?>
<root xmlns="urn:schemas-upnp-org:device-1-0">
  <specVersion>
    <major>1</major>
    <minor>0</minor>
  </specVersion>
  <URLBase>base URL for all relative URLs</URLBase>
  <device>
    <deviceType>urn:schemas-upnp-org:device:deviceType:v</deviceType>
    <friendlyName>short user-friendly title</friendlyName>
    <manufacturer>manufacturer name</manufacturer>
    <manufacturerURL>URL to manufacturer site</manufacturerURL>
    <modelDescription>long user-friendly title</modelDescription>
    <modelName>model name</modelName>
    <modelNumber>model number</modelNumber>
    <modelURL>URL to model site</modelURL>
    <serialNumber>manufacturer's serial number</serialNumber>
    <UDN>uuid:UUID</UDN>
    <UPC>Universal Product Code</UPC>
    <iconList>
      <icon>
        <mimetype>image/format</mimetype>
        <width>horizontal pixels</width>
        <height>vertical pixels</height>
        <depth>color depth</depth>
        <url>URL to icon</url>
      </icon>
      XML to declare other icons, if any, go here
    </iconList>
    <serviceList>
      <service>
        <serviceType>urn:schemas-upnp-org:service:serviceType:v</serviceType>
        <serviceId>urn:upnp-org:serviceId:serviceID</serviceId>
        <SCPDURL>URL to service description</SCPDURL>
        <controlURL>URL for control</controlURL>
        <eventSubURL>URL for eventing</eventSubURL>
      </service>
      Declarations for other services defined by a UPnP Forum working committee (if any)
        go here
      Declarations for other services added by UPnP vendor (if any) go here
    </serviceList>
    <deviceList>
      Description of embedded devices defined by a UPnP Forum working committee (if any)
        go here
      Description of embedded devices added by UPnP vendor (if any) go here
    </deviceList>
    <presentationURL>URL for presentation</presentationURL>
  </device>
</root>
```

Figure 16 UPnP device description XML example

3.2.4 Control

Having retrieved a description of the device, the control point can send actions to a device's service. To do this, a control point sends a suitable control message to the control URL for the service (provided in the device description). Control messages are also expressed in XML using the Simple Object Access Protocol (SOAP). Much like function calls, the service returns any action-specific values in response to the control message. The effects of the action, if any, are

modelled by changes in the variables that describe the run-time state of the service.

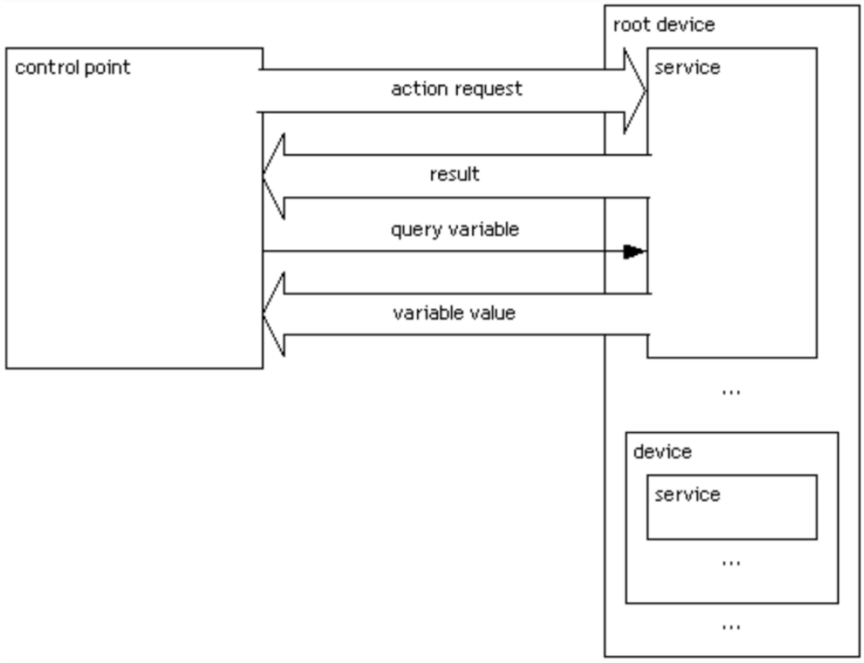

Figure 17 UPnP control point - device interaction

3.2.5 Event notification

Another capability of UPnP networking is event notification, or *eventing*. The event notification protocol defined in the UPnP Device Architecture is known as General Event Notification Architecture (GENA). A UPnP description for a service includes a list of actions the service responds to and a list of variables that model the state of the service at run time. The service publishes updates when these variables change, and a control point may subscribe to receive this information. The service publishes updates by sending event messages. Event messages contain the names of one or more state variables and the current value of those variables. These messages are also expressed in XML. A special initial event message is sent when a control point first subscribes; this event message contains the names and values for all *evented* variables and allows the subscriber to initialize its model of the state of the service. To support scenarios with multiple control points, eventing is designed to keep all control points equally informed about the effects of any action. Therefore, all subscribers are sent all event messages, subscribers receive event messages for all ‘evented’ variables

that have changed, and event messages are sent no matter why the state variable changed (either in response to a requested action or because the state the service is modelling changed).

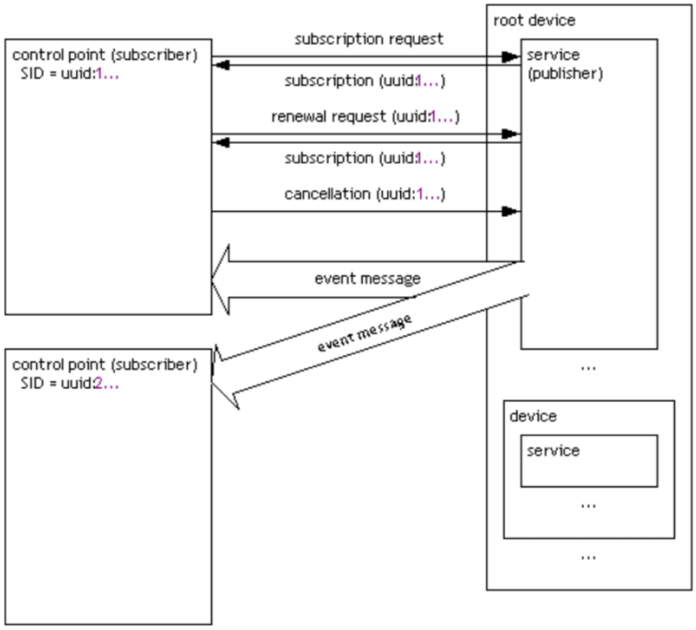

Figure 18 UPnP GENA example

3.2.6 Presentation

The final step in UPnP networking is presentation. If a device has a URL for presentation, then the control point can retrieve a page from this URL, load the page into a web browser, and depending on the capabilities of the page, allow a user to control the device and/or view the device status. The degree to which each of these can be accomplished depends on the specific capabilities of the presentation page and device.

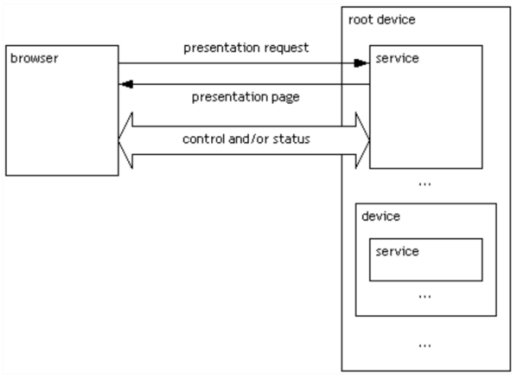

Figure 19 UPnP presentation example

3.3 UPnP AV architecture and services

3.3.1 UPnP AV architecture

This section describes the overall UPnP AV architecture, which forms the foundation of the UPnP AV Device and Service templates. The AV Architecture defines the general interaction between UPnP control points and UPnP AV devices. It is independent of any particular device type, content format, and transfer protocol. It supports a variety of devices such as TVs, VCRs, CD/DVD players, MP3 players, still-image cameras, camcorders, electronic picture frames (EPFs), and the PCs. The AV Architecture allows devices to support different types of formats for the entertainment content (such as MPEG2, MPEG4, JPEG, MP3, Windows Media Architecture (WMA), bitmaps (BMP), NTSC, PAL, ATSC, etc.) and multiple types of transfer protocols (such as IEC-61883/IEEE-1394, HTTP GET, RTP, HTTP PUT/POST, TCP/IP, etc.). The following clauses describe the AV Architecture and how the various UPnP AV devices and services work together to enable various end-user scenarios. (UPnP, 2002)

The UPnP AV architecture was explicitly defined to meet the following goals:

- To support arbitrary transfer protocols and content formats.
- To enable the AV content to flow directly between devices without any intervention from the control point.
- To enable control points to remain independent of any particular transfer protocol and content format. This allows control points to transparently support new protocols and formats.
- Scalability, support of devices with very low resources, especially memory and processing power as well as full-featured devices.
- Synchronized playback to multiple rendering devices. access control, content protection, and digital rights management.

Most AV scenarios involve the flow of (entertainment) content (i.e. a movie, song, picture, etc.) from one device to another. An AV control point interacts with two or more UPnP devices acting as source and sink, respectively. Although the control point coordinates and synchronizes the behaviour of both devices, the devices themselves interact with each other using a non-UPnP (‘out-of-band’) communication protocol. The control point uses UPnP to initialize and configure

both devices so that the desired content is transferred from one device to the other. However, since the content is transferred using an 'out-of-band' transfer protocol, the control point is not directly involved in the actual transfer of the content. The control point configures the devices as needed, triggers the flow of content, then gets out of the way. Thus, after the transfer has begun, the control point can be disconnected without disrupting the flow of content. In other words, the core task (i.e. transferring the content) continues to function even without the control point present.

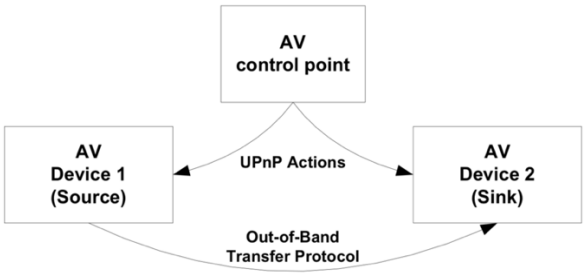

Figure 20 UPnP AV device interaction model

As described in the above scenario, three distinct entities are involved: the control point, the source of the media content (called the '*MediaServer*'), and the sink for the content (called the '*MediaRenderer*'). All three entities are described as if they were independent devices on the network. Although this configuration may be common (i.e. a remote control, a VCR, and a TV), the AV Architecture supports arbitrary combinations of these entities within a single physical device. For example, a TV can be treated as a rendering device (e.g. a display). However, since most TVs contain a built-in tuner, the TV can also act as a server device because it could tune to a particular channel and send that content to a *MediaRenderer* (e.g. its local display or some remote device such as a tuner-less display). Similarly, many *MediaServer*s and/or *MediaRenderer*s may also include control point functionality. For example, an MP3 renderer will likely have some UI controls (e.g. a small display and some buttons) that allow the user to control the playback of music.

The following figure shows a general device architecture where a control point gets media content from a *MediaServer* and reproduces it into a *MediaRenderer* device.

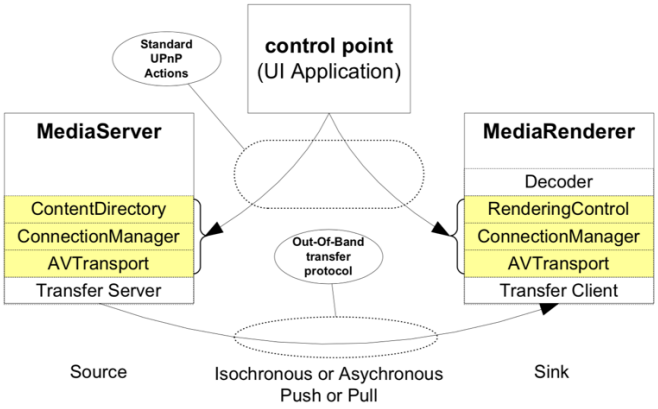

Figure 21 UPnP AV general device architecture

As described above, the AV architecture consists of three distinct components that perform well-defined roles. In some cases, these components will exist as separate, individual UPnP devices. However, this need not be the case. Device manufacturers are free to combine any of these logical entities into a single physical device. In such cases, the individual components of these combo devices may interact with each other using either the standard UPnP control protocols (e.g. SOAP over HTTP) or using some private communication mechanism. In either case, the function of each logical entity remains unchanged.

The *MediaServer* and *MediaRenderer* do not control each other via UPnP actions. However, in order to transfer the content, the *MediaServer* and *MediaRenderer* use an 'out-of-band' (e.g. a non-UPnP) transfer protocol to directly transmit the content. The control point is not involved in the actual transfer of the content. It simply configures the *MediaServer* and *MediaRenderer* as needed and initiates the transfer of the content. Once the transfer begins, the control point 'gets out of the way' and is no longer needed to complete the transfer.

3.3.2 MediaServer services

The *MediaServer* is used to locate content that is available via the home network. *MediaServer*s include a wide variety of devices including VCRs, DVD players, satellite/cable receivers, TV tuners, smartphones, radio tuners, CD players, audio tape players, MP3 players, PCs, etc. A *MediaServer*'s primary purpose is to allow control points to enumerate (i.e. browse or search for) content items that are

available for the user to render. The *MediaServer* contains a *ContentDirectory* service and a *ConnectionManager* service.

1. ContentDirectory service: This service provides a set of actions that allow the control point to enumerate the content that the Server can provide to the home network. The primary action of this service is *Browse.* This action allows control points to obtain detailed information about each Content Item that the Server can provide. This information (i.e. meta-data) includes properties such as its name, artist, date created, size, etc. Additionally, the returned meta-data identifies the transfer protocols and data formats that are supported by the Server for that particular content item. The control point uses this information to determine if a given *MediaRenderer* is capable of rendering that content in its available format.
2. ConnectionManager service: This service is used to manage the connections associated with a particular device. The primary action of this service (within the context of a *MediaServer*) is *PrepareForConnection*. When implemented, this optional action is invoked by the control point to give the Server an opportunity to prepare itself for an upcoming transfer. Depending on the specified transfer protocol and data format, this action may return the *InstanceID* of an *AVTransport* service that the control point can use to control the flow of this content (e.g. stop, pause, seek, etc). Some *MediaServer*s are capable of transferring multiple content items at the same time, e.g. a hard-disk-based audio jukebox may be able to simultaneously stream multiple audio files to the network. In order to support this type of *MediaServer*, the *ConnectionManager* assigns a unique Connection ID to each 'connection' (i.e. each stream) that is made.

3.3.3 MediaRenderer services

The *MediaRenderer* is used to render (e.g. display and/or listen to) content obtained from the home network. This includes a wide variety of devices including TVs, stereos, speakers, hand-held audio players, music-controlled water-fountain, etc. Its main feature is that it allows the control point to control how content is rendered (e.g. brightness, contrast, volume, mute, etc). Additionally, depending on the transfer protocol that is being used to obtain the content from

the network, the *MediaRenderer* may also allow the user to control the flow of the content (e.g. stop, pause, seek, etc). The *MediaRenderer* includes a RenderingControl service, a *ConnectionManager* service, and an optional AVTransport service (depending on which transfer protocols are supported).

In order to support rendering devices that are capable of handling multiple content items at the same time (e.g. an audio mixer such as a karaoke device), the RenderingControl and AVTransport services contain multiple independent (logical) instances of these services. Each (logical) instance of these services is bound to a particular incoming connection. This allows the control point to control each incoming content item independently from each other.

Multiple logical instances of these services are distinguished by a unique *InstanceID* which references the logical instance. Each action invoked by the control point contains the Instance ID that identifies the correct instance.

1. RenderingControl service: This service provides a set of actions that allow the control point to control how the Renderer renders a piece of incoming content. This includes rendering characteristics such as brightness, contrast, volume, mute, etc. The *RenderingControl* service supports multiple, dynamic instances, which allows a Renderer to 'mix together' one or more content items (e.g. a Picture-in-Picture window on a TV or an audio mixer device).
2. ConnectionManager service: This service is used to manage the connections associated with a device. Within the context of a *MediaRenderer*, the primary action of this service is the *Get Protocol Info* action. This action allows a control point to enumerate the transfer protocols and data formats that are supported by the *MediaRenderer*. This information is used to predetermine if a *MediaRenderer* is capable of rendering a specific content item. A *MediaRenderer* may also implement the optional *PrepareForConnection* action. This action is invoked by the control point to give the Render an opportunity to prepare itself for an upcoming transfer.
3. AVTransport service: This service is used by the control point to control the flow of the associated content. This includes the ability to play, stop, pause, seek, etc. Depending on transfer protocols and/or data formats that

are supported, the Renderer may or may not implement this service. In order to support *MediaRenderers* that can simultaneously handle multiple content items, the AVTransport service may support multiple logical instances of this service.

4. OpenHome library

So far, the UPnP protocol, two different architectures and services have been defined using the documentation provided. How can someone after reading the documentation start developing? This section will try to explain how to start the UPnP protocol implementation by using the OhNet (Open Home) library.

As said before, there is a lack of documentation and the only starting point is a website where you can download the compiled libraries. However, what do they mean? How do you start developing? That was my situation but after researching and some trial and error situations I have been able to put all the pieces together.

4.1 Overview

OpenHome Networking is an open, modern and cross platform UPnP stack that includes both a control point and a device stack that can be used together or independently. It runs on Linux, Windows, Mac, iOS and Android.

OhNet is the first UPnP stack support Linn UPnP extensions that provide :

- Multiple *MediaRenderers* and *MediaServers*
- Multiple Control Point support
- On device playlists -- no requirement for the control point to be always on, or always connected to the network
- Full preamp integration
- The Linn UPnP extensions only affect the communication between the *MediaRenderer* and the control point and they do not affect communication with *MediaServer*. Thus, the *OpenHome MediaRenderer* will work with any UPnP compliant UPnP/DLNA *MediaServer*.

4.2 OpenHome list of GitHub projects

As most open source projects, OhNet can be found in a Github repository. For those still unsure about it, Github is a web-based hosting service for version control using Git. It is mostly used for computer code. It offers all of the distributed version control and source code management (SCM) functionality of Git as well as adding its own features. It provides access control and several collaboration

features such as bug tracking, feature requests, task management, and wikis for every project, also known as repository.

The link for the OhNet repository can be found on the website and it redirects to the main *OpenHome* Github page[17] where, apart from the OhNet repository, there are others with similar names. They are projects based on OhNet that perform more or less the same functionalities but more focused on specific goals (i.e. building a media player). The one this project relates to is in fact the first one and it is called 'OhNet'.

Github offers the possibility for downloading or cloning a repository. Depending on the usage, users can get the content using either way. In my case I want to clone the project so I will be able to fetch new changes in the future easily. So, before I finish this section, I will open the terminal and I will type the clone command for having the project in my computer and start making use of it.

```
Ramons-MBP:~ ramonharomarques$ git clone https://github.com/openhome/ohNet.git ~/Desktop/UOC/NetworkMediaStreamingProject/ohnet
Cloning into '/Users/ramonharomarques/Desktop/UOC/NetworkMediaStreamingProject/ohnet'...
remote: Enumerating objects: 90, done.
remote: Counting objects: 100% (90/90), done.
remote: Compressing objects: 100% (59/59), done.
remote: Total 42219 (delta 58), reused 50 (delta 30), pack-reused 42129
Receiving objects: 100% (42219/42219), 108.43 MiB | 529.00 KiB/s, done.
Resolving deltas: 100% (32087/32087), done.
Ramons-MBP:~ ramonharomarques$
```

Figure 22 OhNet Github clone process

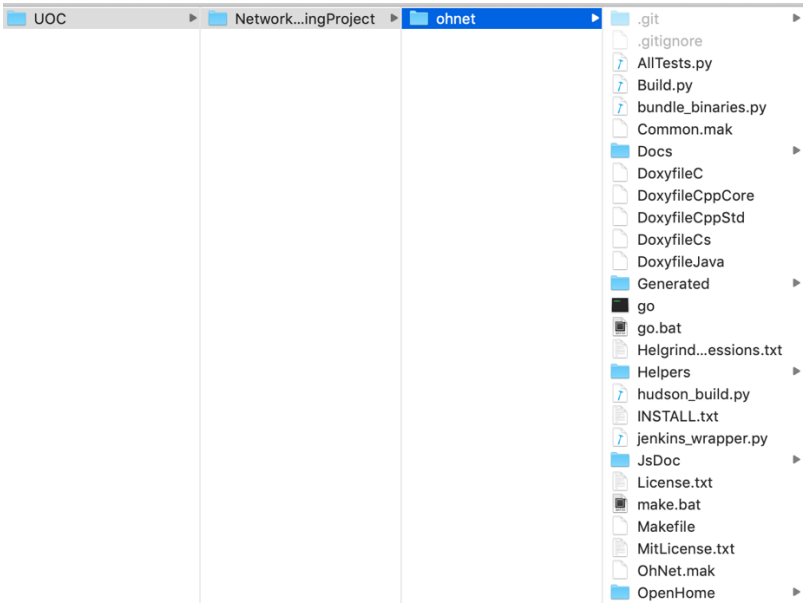

Figure 23 OhNet project cloned to the computer

4.3 Services

As mentioned before, the services are the way different devices interact between each other. I have only described the basic services for implementing basic *MediaServer*s and *MediaRenderers*. However, more pre-existing services can be added and even new services can be created according to new necessities.

The folder '*/OpenHome/Net/Service'* contains a file called '*services.xml*' and a folder called 'UPnP'. The first one declares what services the library will compile and it contains the location where the service XML is located. On the right side, the folder contains the services XML's and each of these will define what actions, variables and notifications the service handles.

```
<servicelist>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ConnectionManager</type>
        <version>1</version>
    </service>
    <service>
        <publisher>OpenHome</publisher>
        <dcpdir></dcpdir>
        <domain>av.openhome.org</domain>
        <type>Product</type>
        <version>1</version>
    </service>
    <service>
        <publisher>OpenHome</publisher>
        <dcpdir></dcpdir>
        <domain>av.openhome.org</domain>
        <type>Sender</type>
        <version>1</version>
    </service>
    <service>
        <publisher>OpenHome</publisher>
        <dcpdir>Test</dcpdir>
        <domain>openhome.org</domain>
        <type>TestBasic</type>
        <version>1</version>
    </service>
    <service>
        <publisher>OpenHome</publisher>
        <dcpdir></dcpdir>
        <domain>openhome.org</domain>
        <type>SubscriptionLongPoll</type>
        <version>1</version>
    </service>
</servicelist>
```

Figure 24 OhNet servce.xml file

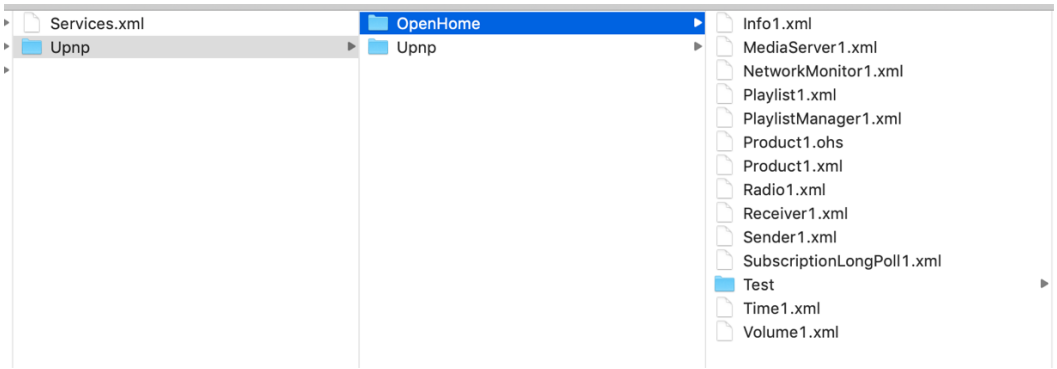

Figure 25 OhNet XML services folder

As seen in the pictures above, the library contains a lot more XML services that the ones define. It also contains a bunch of services under the folder *OpenHome*. These services will not be implemented in this project but they are intended to enhance the media renderer either by implementing a queue playlist service or by offering the possibility of sharing playlists on the media server among others.

Before finishing this section, I will modify the '*services.xml*' file for defining which services I want to be compiled for the next section.

```
<servicelist>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>AVTransport</type>
        <version>1</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>AVTransport</type>
        <version>2</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ConnectionManager</type>
        <version>1</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ConnectionManager</type>
        <version>2</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ContentDirectory</type>
        <version>1</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ContentDirectory</type>
        <version>2</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ContentDirectory</type>
        <version>3</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>RenderingControl</type>
        <version>1</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>RenderingControl</type>
        <version>2</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ScheduledRecording</type>
        <version>1</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>MediaServer_3</dcpdir>
        <domain>upnp.org</domain>
        <type>ScheduledRecording</type>
        <version>2</version>
    </service>
    <service>
        <publisher>Upnp</publisher>
        <dcpdir>LightingControls_1</dcpdir>
        <domain>upnp.org</domain>
        <type>SwitchPower</type>
        <version>1</version>
    </service>
```

Figure 25 OhNet services.xml services declaration

Having a closer look about what this declaration means, the following information can be extracted:

1. Publisher: Refers to the company, organisation or the common name for a collection of services. Also, it says the root folder where this collection is located under the folder mentioned before.
2. dcpdir: Common name and version for a service collection that also specifies the folder which it is the child of the previous. There can be older versions of the same service inside the folder.
3. Domain: Domain name for which the services are associated.
4. Type: Service name.
5. Version: Specifies the version number.

4.4 Compiling the library

Now it is time to build the library. Before I start, I will have a closer look at the '*Makefile'* file located in the root directory of the library. The file is written using phyton and by going through the code, I can confirm that:

- From a Mac computer I can only compile for Apple devices for the following architectures:
    - iOS ARM7: Old iPhone devices with 32 bits architecture.
    - iOS ARM64: New iPhone devices with 64 bits architecture.
    - iOS x64: Emulator for 64 bits computers.
    - iOS x86: Emulator for 32 bits computers.
    - Mac x64: For Mac 64 bits computers.
    - Mac x86: For Mac 32 bits.
- The compiled library will be in C, C++ and C#.
- For compiling into C#, the computer has to have Microsoft visual studio installed which provides the libraries required for this task.
- All the languages above will be compiled and cannot be omitted.
- Line 169 of the code sets up the minimum iOS version to be 2.2 and that will force the library to be compiled as libstdc++. It took time to figure out but after raising a question into *'*StackOverflow*[18]*' I found out the answer. The only thing to be done is replace the 2.2 to 7 and that will force the library to be compiled as libc++.

Now everything is setup for start the compilation process.

4.4.1 Steps for compiling for different structures

One of the first things that developers are told to do in the OhNet Github page is to run a command to regenerate the *makefiles*, *proxies* or *providers*. These last two refer to services and I will show later why they are named as this. Since I want to regenerate or generate the files needed for the compilation process, to compile my services I run the following commands.

```
Ramons-MBP:ohnet ramonharomarques$ make generate-makefiles uset4=yes
CROSS_COMPILE:
Machine reported by compiler is: x86_64-apple-darwin18.0.0
Machine reported by uname is: Darwin
Building for system Mac and architecture x86
mkdir -p Build/Tools/
rsync OpenHome/Net/T4/UpnpServiceXml/UpnpServiceDescription.xsd Build/Tools/
mkdir -p Build/Tools/
rsync OpenHome/Net/T4/UpnpServiceXml/UpnpServiceTemplate.xsd Build/Tools/
mkdir -p Build/Tools/
```

Figure 26 OhNet regeneration terminal process

As the initial page in Github says, this process creates a new folder with all the tools that the compilation process will need.

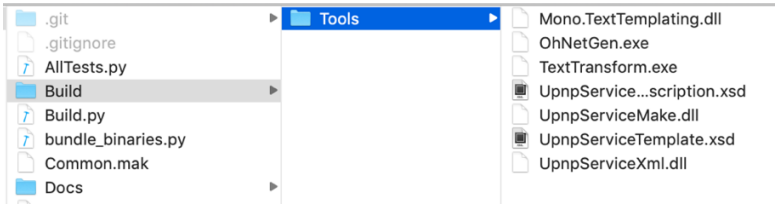

Figure 27 OhNet build folder generated

The following process is about compiling (or making) the library for each architecture. I will compile for each type mentioned before and compile a debug and release version for type. The shell make for *Makefile* takes as parameters the architecture and the target. If the target is not specified, it will compile for a release target by default. From now on, I will use the word ‘bundle’ for referring to each compiled library for an architecture and target.

```
Ramons-MBP:ohnet ramonharomarques$ make iOs-armv7=1 ; make iOs-armv7=1 debug=1
CROSS_COMPILE:
Machine reported by compiler is: x86_64-apple-darwin18.0.0
Machine reported by uname is: Darwin
Building for system iOs and architecture armv7
mkdir -p Build/Obj/iOS-armv7/Release/
mkdir -p Build/Include
mkdir -p Build/Include/OpenHome
mkdir -p Build/Include/OpenHome/Private
mkdir -p Build/Include/OpenHome/Net
mkdir -p Build/Include/OpenHome/Net/Private
mkdir -p Build/Include/OpenHome/Net/Private/Tests
```

Figure 28 OhNet compilation for iOs-armv7 debug and release using the terminal

If the process has been successful, three new folders have been created inside the Build folder:

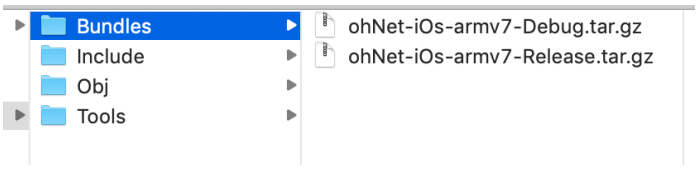

Figure 29 Build folder after first compilation

1. Bundles: Contains all the compressed files for all the compiled libraries.
2. Include: Contains all the headers needed for using functions, variables, and others inside the library.
3. Obj: Folder containing all the compiled objects. This may be needed but not currently for this project.

After this first compilation, the process needs to be repeated for every single architecture. Note that I can use semicolon for executing different commands one after the other in a single command line.

```
Ramons-MBP:ohnet ramonharomarques$ make iOs-arm64=1 ; make iOs-arm64=1 debug=1 ; make iOs-x86=1 ; make iOs-x86=1 debug=1 ; make iOs-x64=1 ;
make iOs-x64=1 debug=1 ; make mac-64=1 ; make mac-64=1 debug=1 ; make mac-86=1 ; make mac-86=1 debug=1
CROSS_COMPILE:
Machine reported by compiler is: x86_64-apple-darwin18.0.0
Machine reported by uname is: Darwin
Building for system iOs and architecture arm64
mkdir -p Build/Obj/iOS-arm64/Release/
mkdir -p Build/Include
mkdir -p Build/Include/OpenHome
mkdir -p Build/Include/OpenHome/Private
mkdir -p Build/Include/OpenHome/Net
mkdir -p Build/Include/OpenHome/Net/Private
mkdir -p Build/Include/OpenHome/Net/Private/Tests
```

Figure 30 Multi compile command process

The process of compilation is done by Xcode libraries. At end of the process I have shown all the generated Bundles.

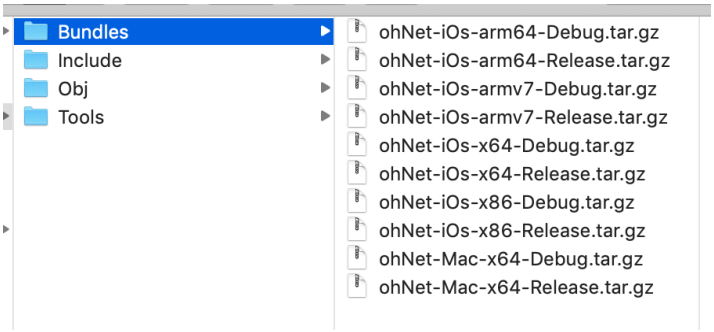

Figure 31 Final compilation bundles

The compilation process failed for mac-86 using MacOS 10.14 Mohave. I assume the new OS deprecated the tools needed for the compilation as it worked with

MacOS10.13 Sierra High. Nevertheless, this project won't need it and nowadays it is hugely unusual to see a machine running this architecture.

At this point I will create a new folder called 'OhNetGenerated' and I will start generating the final library that will be used in the project. Inside this folder I have created the following sub-folders:

1. Bundles: Contains all the compressed bundles generated in the previous step.
2. BundlesUncompressed: Contains all the uncompressed bundles generated in the previous step.
3. OhNet-iOS: Contains all the fat libraries organised by debug and release, and the headers folder which are common for any bundle.

Exploring an uncompressed bundle, I would like to show the structure and the key factors. The following picture shows the static libraries that will be used.

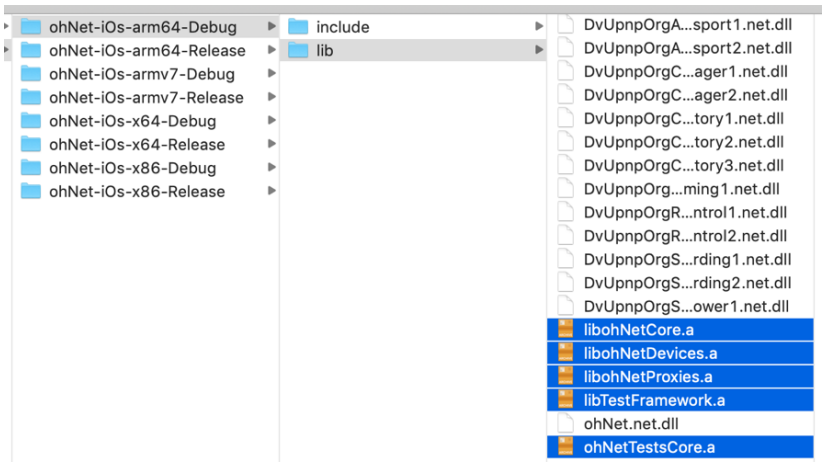

Figure 32 OhNet lib folder for arm64 bundle

On the other side this is how the 'include' folder looks:

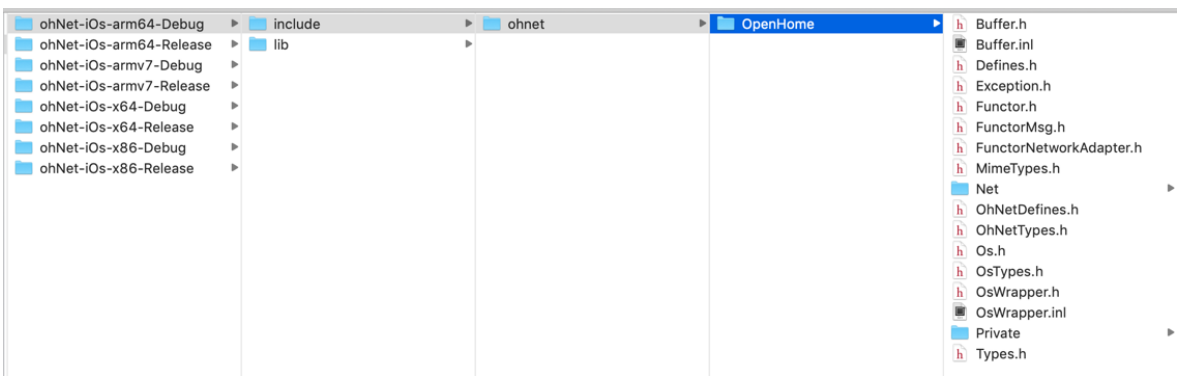

Figure 33 Include folder for arm64 bundle

4.4.2 Fat library creation

Once I have all the bundles, it is time to create a universal static library so it can be used with different architecture devices. In the Apple development world, this type of libraries are called fat libraries and it is achieved with the help of 'lipo'.

This process will compile a fat library for debug and release for the following static libraries:

- libOhNetCore.a
- libOhNetDevices.a
- libOhNetProxies.a
- libTestFramework.a
- OhNetTestsCore.a

These static libraries are the ones showed on the previous chapter and they will need to be done for the iOS architectures mentioned before.

Before starting with the process, I am going to confirm that the static libraries are compiled with libc++ and not with libstdc++.

```
Ramons-MBP:BundlesUncompressed ramonharomarques$ nm ohNet-iOs-arm64-Debug/lib/libohNetCore.a | c++filt | grep string | head -1
0000000000000120 T OpenHome::Net::CpDeviceCpp::GetAttribute(char const*, std::__1::basic_string<char, std::__1::char_traits<char>, std::__1::alloc
ator<char> >&) const
Ramons-MBP:BundlesUncompressed ramonharomarques$
```

Figure 34 OhNet terminal command to check the .a file is libc++

The presence of std:__1 tells that the library is compiled with libc++.

Once checked, the fat library compilation is achieved like this:

```
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Debug/lib/libohNetCore.a ohNet-iOs-armv7-Debug/lib/libohNetCore.a oh
Net-iOs-x64-Debug/lib/libohNetCore.a ohNet-iOs-x86-Debug/lib/libohNetCore.a -output ../OHNetBuild/OHNet-iOS/OHNetDebugLibs/libohNetCore.a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Release/lib/libohNetCore.a ohNet-iOs-armv7-Release/lib/libohNetCore.
a ohNet-iOs-x64-Release/lib/libohNetCore.a ohNet-iOs-x86-Release/lib/libohNetCore.a -output ../OHNetBuild/OHNet-iOS/OHNetReleaseLibs/libohNetCore.
a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Debug/lib/libohNetDevices.a ohNet-iOs-armv7-Debug/lib/libohNetDevice
s.a ohNet-iOs-x64-Debug/lib/libohNetDevices.a ohNet-iOs-x86-Debug/lib/libohNetDevices.a -output ../OHNetBuild/OHNet-iOS/OHNetDebugLibs/libohNetDev
ices.a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Release/lib/libohNetDevices.a ohNet-iOs-armv7-Release/lib/libohNetDe
vices.a ohNet-iOs-x64-Release/lib/libohNetDevices.a ohNet-iOs-x86-Release/lib/libohNetDevices.a -output ../OHNetBuild/OHNet-iOS/OHNetReleaseLibs/l
ibohNetDevices.a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Debug/lib/libohNetProxies.a ohNet-iOs-armv7-Debug/lib/libohNetProxie
s.a ohNet-iOs-x64-Debug/lib/libohNetProxies.a ohNet-iOs-x86-Debug/lib/libohNetProxies.a -output ../OHNetBuild/OHNet-iOS/OHNetDebugLibs/libohNetPro
xies.a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Release/lib/libohNetProxies.a ohNet-iOs-armv7-Release/lib/libohNetPr
oxies.a ohNet-iOs-x64-Release/lib/libohNetProxies.a ohNet-iOs-x86-Release/lib/libohNetProxies.a -output ../OHNetBuild/OHNet-iOS/OHNetReleaseLibs/l
ibohNetProxies.a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Debug/lib/libTestFramework.a ohNet-iOs-armv7-Debug/lib/libTestFramew
ork.a ohNet-iOs-x64-Debug/lib/libTestFramework.a ohNet-iOs-x86-Debug/lib/libTestFramework.a -output ../OHNetBuild/OHNet-iOS/OHNetDebugLibs/libTest
Framework.a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Release/lib/libTestFramework.a ohNet-iOs-armv7-Release/lib/libTestFr
amework.a ohNet-iOs-x64-Release/lib/libTestFramework.a ohNet-iOs-x86-Release/lib/libTestFramework.a -output ../OHNetBuild/OHNet-iOS/OHNetReleaseLi
bs/libTestFramework.a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Debug/lib/ohNetTestsCore.a ohNet-iOs-armv7-Debug/lib/ohNetTestsCore.
a ohNet-iOs-x64-Debug/lib/ohNetTestsCore.a ohNet-iOs-x86-Debug/lib/ohNetTestsCore.a -output ../OHNetBuild/OHNet-iOS/OHNetDebugLibs/ohNetTestsCore.
a
Ramons-MBP:BundlesUncompressed ramonharomarques$ lipo -create ohNet-iOs-arm64-Release/lib/ohNetTestsCore.a ohNet-iOs-armv7-Release/lib/ohNetTestsC
ore.a ohNet-iOs-x64-Release/lib/ohNetTestsCore.a ohNet-iOs-x86-Release/lib/ohNetTestsCore.a -output ../OHNetBuild/OHNet-iOS/OHNetReleaseLibs/ohNet
TestsCore.a
```

Figure 35 OhNet terminal command to generate all the fat libraries

4.4.3 Composing the OhNet folder

In the previous section I started creating the composite folder which will be the one to be used at the end for the Xcode project. This folder is called ‘OhNetBuild’. It contains a subfolder for the iOS libraries called ‘OhNet-iOS’ and inside that one it contains a folder for each target and the include folder. For each target the folder will contain the fat libraries generated during the last step.

As mentioned, the include header can be generated for a bundle but I will use the one generated in the Build folder from the OhNet project.

Figure 36 Include folder from OhNet

Figure 37 OhNetBuild folder completed

Now I am ready to start the app development in Xcode. The next section will show how to import what I have built into an Xcode project and how to work with C libraries.

5. iOS APP Project

Once the library has been compiled, it is time to start the Xcode project. In this section I will show how to import the OhNet library and how to setup the project to work with Swift.

5.1 XCode

For all who do not know what Xcode is, I would describe it as an integrated development environment (IDE) for macOS containing a suite of software development tools developed by Apple to develop software for macOS, iOS, watchOS, and tvOS. First released in 2003, the latest stable release is version 10.0 and is available via the Apple Mac app Store free of charge for macOS users. Registered developers can download preview releases and prior versions of the suite through the Apple Developer website[19]. Despite strongly not recommending it, if you are a non-macOS user you can develop apps using the Xamarin[20] library that it is incorporated within Microsoft visual studio.

Xcode offers the possibility to write and test code using a simulator. However, for testing on a real device and pushing apps to the *AppStore*, the developer needs to sign up for a developer account which costs annually around seventy pounds.

So, for this project I will use my personal apple developer account but, because I have compiled the OhNet library for iOS x86 and iOS x64, the project will be able to run using the simulator. I must say I haven't tried myself and the behaviour may be unexpected.

5.2 Mac and iOS programming languages

Once we have Xcode installed, I need to define which programming languages are used for developing apps for apple devices. Android decided to write a whole library in java for creating apps and, in the early days, you could simply write an app using eclipse.

However, Apple, as well as creating its own IDE, they also created a new object oriented programming **Objective-C** which is general-purpose language that is developed on top of C Programming language by adding features of Small Talk

programming language, making it an object-oriented language. It is primarily used in developing iOS and Mac OS X operating systems as well as its applications.

Initially, Objective-C was developed by NeXT, which was Steve Job's previous company before returning to Apple, for its NeXTSTEP OS from whom it was taken over by Apple for its iOS and Mac OS X.

Objective-C was created in 1980 and is able to power great programs and applications nowadays but in 2014 Apple presented **Swift** as its replacement. The new programming language is more powerful and intuitive, and after releasing the version 3.0 it was stable enough to no longer needing to use its predecessor. The goal for this project isn't to explain how swift works but in the code there will be comments explaining its characteristics. For now, I can describe some of their differences.

1. Swift runs almost as fast as C++. And, with the newest versions of Xcode from 2015, it's even *faster* (Carey, 2017).
2. Swift is easier to read and easier to learn than Objective-C. Objective-C is over thirty years old, and that means it has a more clunky syntax. Swift streamlines code and more closely resembles readable English, similar to languages like C#, C++, JavaScript, Java and Python. Developers already versed in these languages can expect to pick Swift up pretty quickly. Also, Swift requires less code whereas Objective-C is verbose when it comes to string manipulation. Swift employs string interpolation, without placeholders or tokens.
3. Unified files make code easier to maintain. Again, an old standard of the C language holds Objective-C back: a two-file requirement. This means that programmers have to update and maintain two separate files of code, whereas in Swift, these become one. That means less work for programmers, but not at the cost of speed on the front end.
4. Better compilers equals better coding experience for programmers. Swift is built with the Low Level Virtual Machine (LLVM), a compiler that is used by languages like Scala, Ruby, Python, C# and Go. The LLVM is faster and smarter than previous C compilers, so more workload is transferred from the programmer to Xcode and the compiler.

5. No pointers means Swift is ‘safer’. Objective-C, like other C languages, uses pointers which is a method for exposing values that gives programmers more direct access to data. The problem with pointers is that they can cause vulnerabilities in security. They also create a barrier to finding and fixing bugs. With Swift, however, if the code’s pointer is missing a value (a *nil* value), rather than continuing to run the app, it causes the app to crash and allows you to locate and fix bugs on the spot. Developers have cleaner code and spend less time looking for bugs down the road.
6. Better memory management. Memory leaks can occur in object-oriented programming and apps, and they decrease available memory for an app to run causing the application to fail. Typically, Cocoa Touch APIs support Automatic Reference Counting (ARC), a streamlined way to handle memory management. But in the context of the Core Graphics API, ARC is not available and it is up to the developer. This is a common pitfall when an app is using big data buffers, video, or graphics. When too much memory is used during a memory leak, an app can get shut down by the operating system. To fix this, Swift supports ARC across all APIs, and this stability means less time programmers have to spend focusing on memory management.
7. Swift brings Objective-C extension one step further. Classes can extend their functionalities and they are not anymore private to the class that declares the extensions.
8. Swift has more powerful enumeration types and a better ‘Get/Set/WillSet/DidSet’ properties management.

To find out more about Swift and Objective-C I will add links to the developer website where readers of this project can find more about. I do not expect the information provided so far to be understood completely but I aim to offer clear documented code when writing the code which will be in Swift.

5.3 First steps for developing the APP project

After having a quick overview about Xcode and how to start programming, it is time to start creating the project. In this section I will cover the basic parts and

how I write the OhNet Swift implementation for a moderate experienced Swift programmer to start writing the UI using my interpretation of the OhNet library.

To offer a fully comprehensive outline about my UI implementation, the code will be available, commented on and I will try to give different approaches for achieving the same goal.

5.3.1 Creating projects and files

The first thing required to do to create a project is to open Xcode and chose the option ‘Create new project’. Then a new window requests what type of project you want to create. Under ‘Application’, different templates are available but for creating a mobile application from scratch ‘Single View App’ is the option needed.

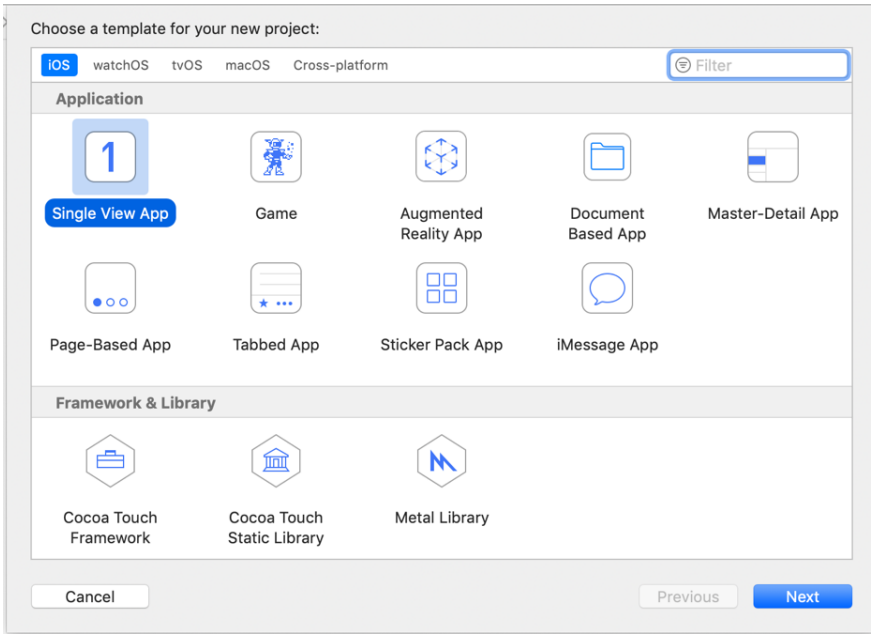

Figure 38 Xcode iOS project type choices

As seen below, the next step is to give a name to the project, chose the Apple developer account and the bundle identifier for the project. Also, for this project I will not use neither core data (Database) or unit test so I leave them unselected.

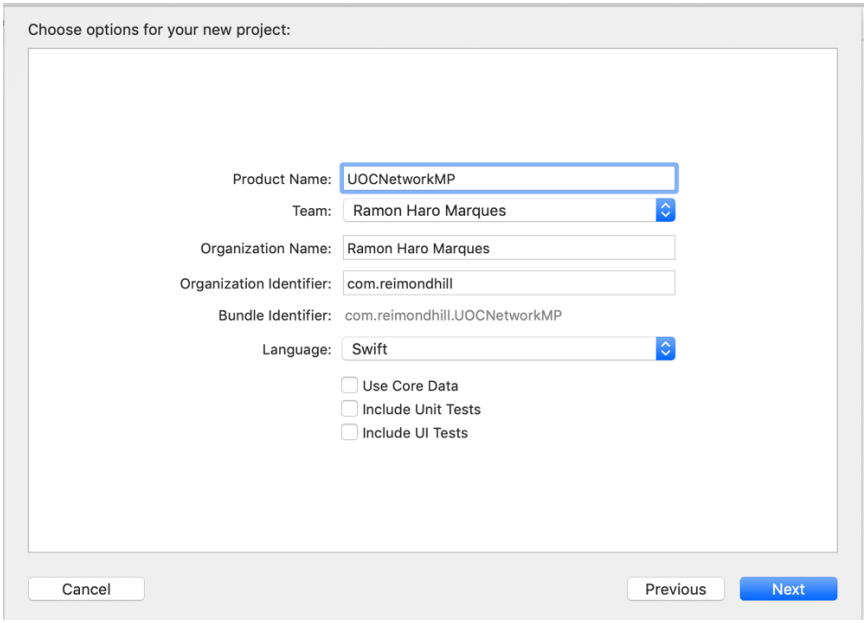

Figure 39 Xcode project basic configuration information

After setting up the details, I am ready to go and I am presented with the main interface. By clicking on the top of project navigator, I can access the project settings.

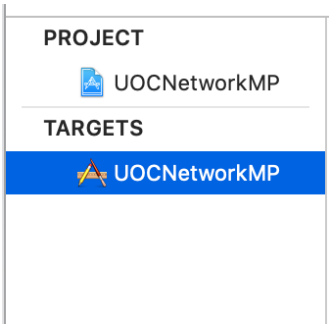

Figure 40 Xcode project and targets view

As seen above, we have different settings, the project and the target.

A project contains one or more targets, which specifies how to build products. It defines default build settings for all the targets in the project. However, each target can specify its own build settings, thus overriding the default project level settings.

A target, on the other hand, specifies a product to build and contains the instructions for building the product from a set of files in a project. A target defines a single product and organises the inputs into the build system. Projects can

contain one or more targets, each of which produces one product. Mostly all the configuration will be done under the target.

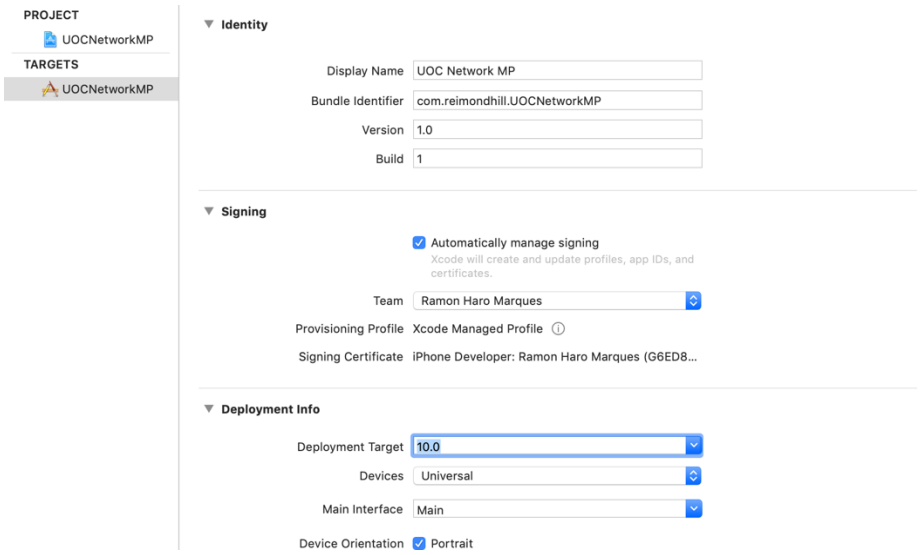

Figure 41 Xcode general target configuration

By navigating to general tab in the target I can set up the minimum iOS the app will run, version, the name, and others.

The next thing that needs doing is to set the app icon. To achieve that, under 'Assets.xcassets' (Xcode assets file) there is an option called 'AppIcon'. There are some websites that generate the whole collection of icons for the apps. The one I normally use is called 'MakeAppIcon[21]'. The strong points of this website are that it is free, efficient, easy to use and it also generates the icons for Android.

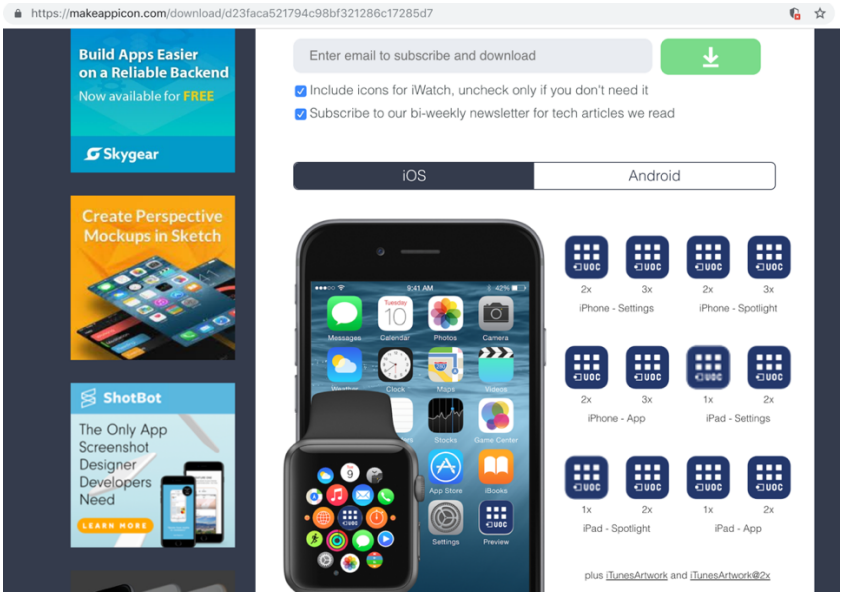

Figure 42 Website to generate the app icons

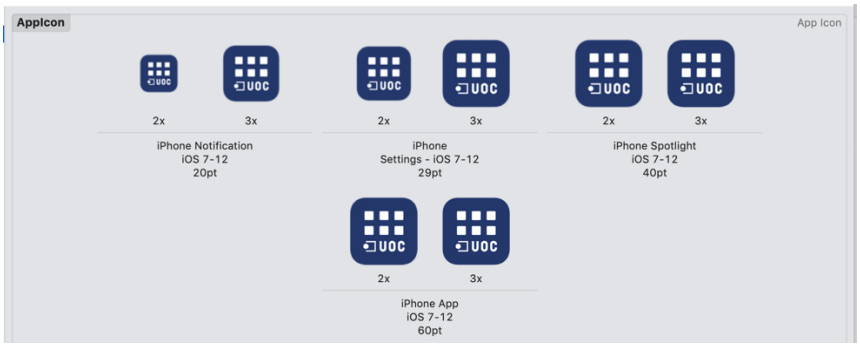

Figure 43 Xcode app icon setup

For adding more images and icons, they all need to be declared in the *xcassets* file. Not doing that will lead to a very disorganised project.

5.3.2 Setting up an Xcode project for working with C libraries

The next step is to import the OhNet library and configurate the Xcode projects to work with C libraries.

Swift works slightly different than Objective-C when non Swift files (C, C++ and Objective-C) need to be exposed to a Swift file. To achieve that a file called 'bridging header' needs to be created. There are different ways to create the special header but the simplest way is to create a new Objective-C file and if it is the first time, Xcode will ask the user to create it automatically.

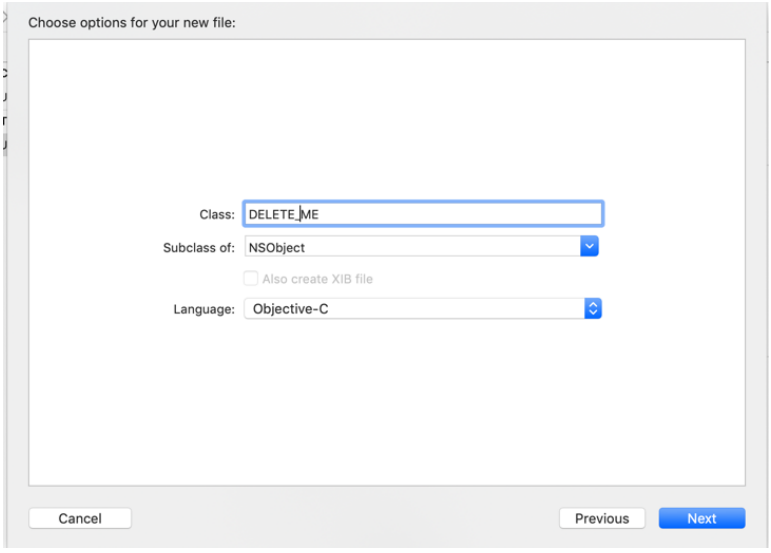

Figure 44 Xcode dummy objective-C creation

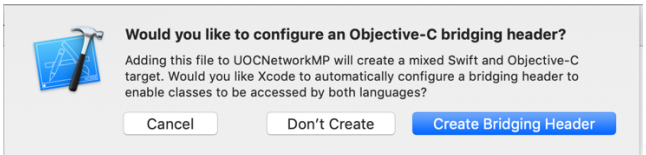

Figure 45 Xcode bridging header creation request

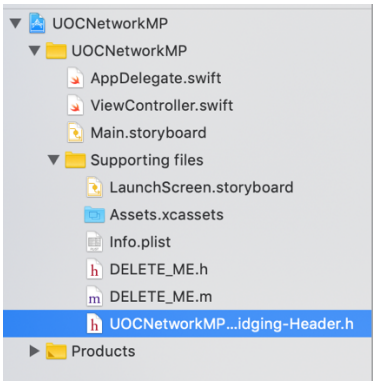

Figure 46 Xcode prove that the bridging header is created

As mentioned before, in the bridging header, all the C, C++ and Objective-C headers need to be imported if I want Swift to be able to reach them.

A similar process is available for exposing Swift files to only Objective-C files but it will not be covered in this project because I will not use either Objective-C files or classes at all.

Before I start importing the headers from OhNet, I need to copy the OhNet generated folder I created into the project's root folder and to import the .a files into the Build Phases tab – Link binary with libraries inside the target view.

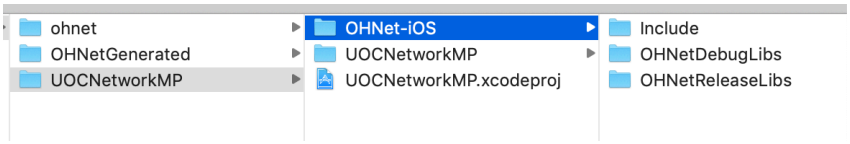

Figure 47 OhNet generated folder copied into the Xcode project's root folder

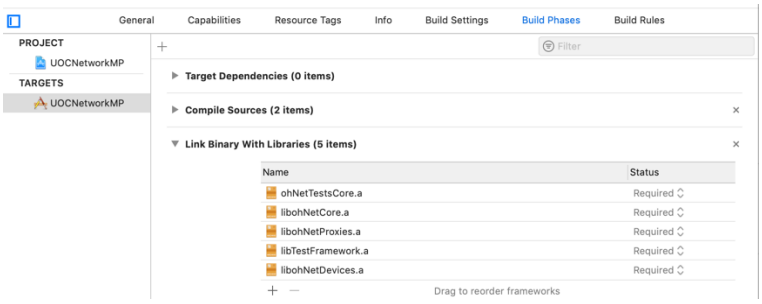

Figure 48 .a imports to Xcode

As seen before, there are two different types of .a files, debug and release. It does not matter which I include here but, to tell Xcode which type it needs, an extra setup needs to be done under ‘Target’ – ‘Build Settings’ – ‘Search Paths’ – Library search paths. This step will configure which path Xcode needs to choose to find the right type of library.

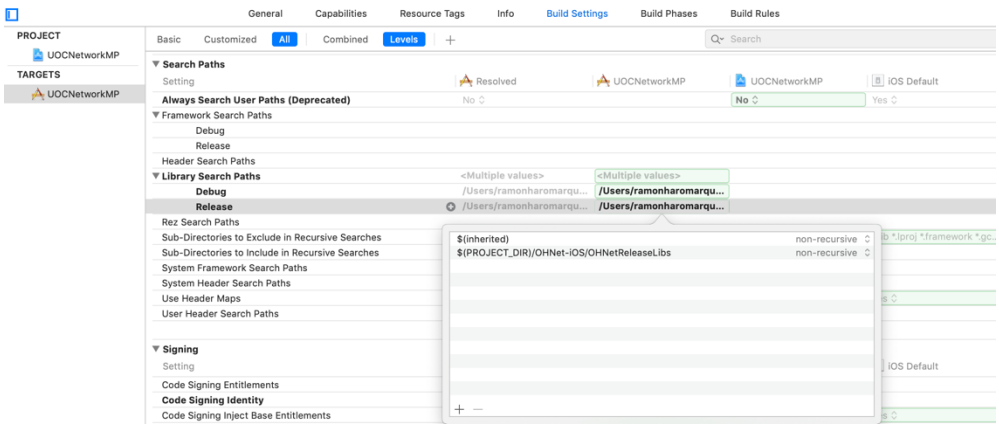

Figure 49 Xcode search path configuration for release

After this and in the same section, I need to tell Xcode where to find the headers I will declare in my bridging file. That can be done in Header Search Paths which is just above Library Search Path.

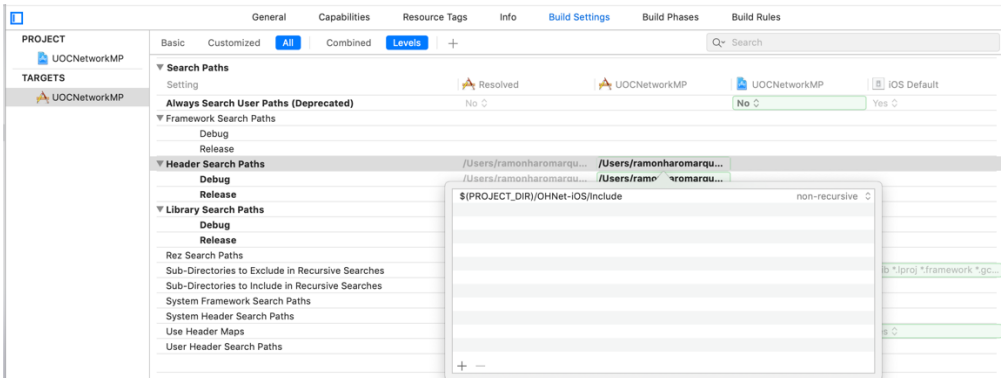

Figure 50 Xcode header search path configuration

The last configuration before declaring the headers into the bridging header is to add a *C++* flag into the linker configuration, and to choose the standard library for *c++* which will be *libc++*. At the very beginning of the project I said that Apple did deprecate *stdlibc++* and I have shown the changes that were needed.

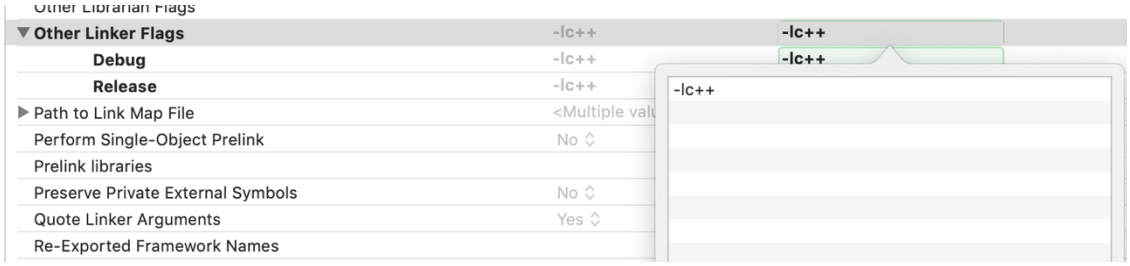

Figure 51 Xcode c++ linker flag configuration

Figure 52 Xcode standard CLANG C++ library configuration

Finally, I am ready to show how I import the headers and which headers I have to import. Also, I will explain as promised what the differences are between proxies and providers. They will be referenced throughout the project using the '*prx' or 'cp'* prefix when a class or variable refers to a proxy or with the 'prv' or 'dv' prefix when a class or variable refers to a provider.

```
/*** CP -PROXIES- ***/
/** Header That Includes all the CP Related Headers **/
#import "OpenHome/Net/C/CpStack.h"

/** CP Services **/
/*UPnP*/
#import "OpenHome/Net/C/CpUpnpOrgConnectionManager1.h" //Server - Renderer
#import "OpenHome/Net/C/CpUpnpOrgContentDirectory1.h" //Server
#import "OpenHome/Net/C/CpUpnpOrgRenderingControl1.h" //Renderer
#import "OpenHome/Net/C/CpUpnpOrgAVTransport1.h" //Renderer


/*** DV -PROVIDERS- ***/
/** Header That Includes all the DV Related Headers **/
#import "OpenHome/Net/C/DvStack.h"

/** DV Services **/
/* UPnP */
#import "OpenHome/Net/C/DvUpnpOrgConnectionManager1.h" //Server - Renderer
#import "OpenHome/Net/C/DvUpnpOrgContentDirectory1.h" //Server
#import "OpenHome/Net/C/DvUpnpOrgRenderingControl1.h" //Renderer
#import "OpenHome/Net/C/DvUpnpOrgAVTransport1.h" //Renderer
```

Figure 53 Xcode bridging header declaring the OhNet headers to import

As shown above, the OhNet library contains a header that includes all the necessary headers for importing the related files that manage either a proxy or a provider. The rest of headers are the services that the app will implement.

There will be one provider in this project because, as the name suggests, it provides all the service functionalities to the other devices on the network.

On the other hand, there will one or more proxy. All cp files will be used to discover which services each device implements and to communicate with them.

5.3.3 Dependency manager in iOS

Most modern languages come with an official solution for code distribution. In today's world of modern mobile development, it is essential to re-use the code already written by developers. Code reuse can be achieved by creating and distributing the packages from central repository. A package manager is a tool that simplifies the process of working with code from multiple sources (Jagtap, 2017). Typical Package Manager should have the ability to do the following things:

- Centralised hosting of packages and source code with public server, with access to developers or contributors
- Download the source code at the run time, so that it does not need to be included in the repository.
- Link the source code to a personal working repository by including source files.
- Allow packages to be versioned

Examples of popular package managers are Ruby Gems for Ruby , Composer for PHP or NPM for NodeJS.

The basic usage of most package managers generally has the following in common:

- Define the file with all required dependencies (Gemfile, composer.json, package.json).
- Download the dependencies with specific command (gem/bundle install, composer install, npm install). This will create lock file with all the installed versions of the dependencies. (e.g. Gemfile.lock, Composer.lock …). This will also create a directory with downloaded source code at the specified location.
- Update the dependencies for new versions (gem/bundle update, composer update, npm update, pod update). This will update lock files with new versions.

- Add required package by browsing packages on the hosting server.

The Swift dependency management for iOS does exactly the same thing but it has an additional task of building the Xcode scheme. Apple has released its own package manager called 'Swift Package Manager' to share and distribute Swift packages but they are still not available for iOS. It is good to know that Apple is working on a replacement of the current package managers in the iOS development world which are CocoaPods and Carthage. In this project I will use Carthage, this is mainly due to CocoaPods issues around organising the project, alongside confusing features. It is recommendable to install Carthage using Homebrew by running '*brew install carthage'*.

To implement Carthage, I need to create a file called 'Cartfile' in the project's root directory.

```
Ramons-MBP:UOCNetworkMP ramonharomarques$ touch Cartfile
Ramons-MBP:UOCNetworkMP ramonharomarques$ nano Cartfile
```

Figure 54 Carthage cartfile creation

After that I will declare all the links where the libraries are located (mostly GitHub).

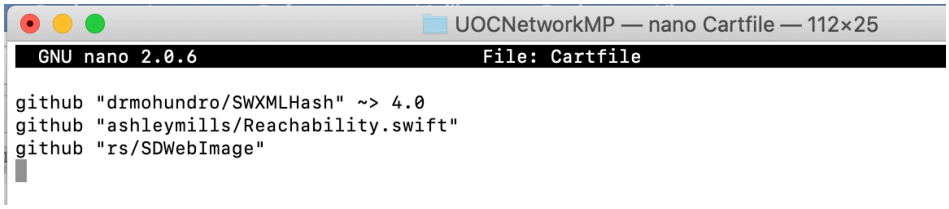

Figure 55 Carthage dependency declaration

The libraries used are:

- Reachability: Great library for managing network status and changes. The app will only work through Wi-Fi so it is important to have this powerful library.
- SWXMLHash: All the communication between devices is done though XML. The classes that Apple provides for this manner are inefficient. With this library I can achieve much better control.
- SDWebImage: Downloading and caching images are another challenging part. This library stunningly manages all the asynchronous connections,

caches the images already downloaded for recycle and handles all the disposing.

The next part is to run the command to start all the process. Carthage also offers the option to only get the dependencies for a particular platform which, in this case, is iOS by including the flag platforms.

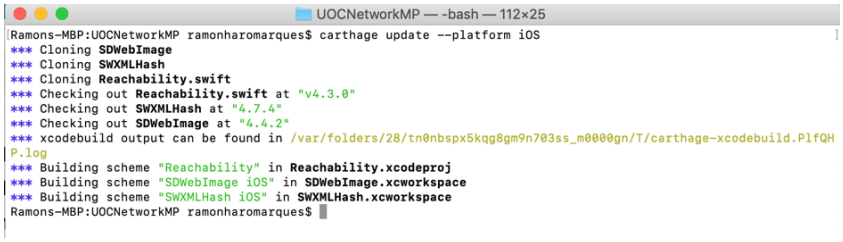

Figure 56 Carthage dependencies installation process

If everything is successful the lock file with the extension resolved and the folder should have been created. If I have not mentioned it before, the libraries for Xcode are called frameworks.

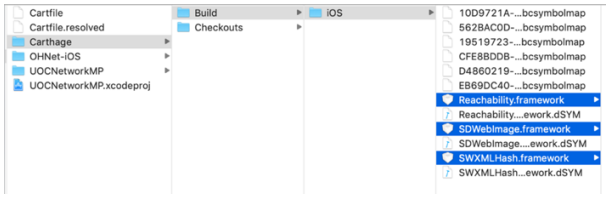

Figure 57 Carthage downloaded dependencies

The last thing I need to use the frameworks for within the project is to add them to the Build Phases (same places that the .a files) and import a run script for copying them to the app folder when the app is compiled. All the Carthage documentation can be found on the website.

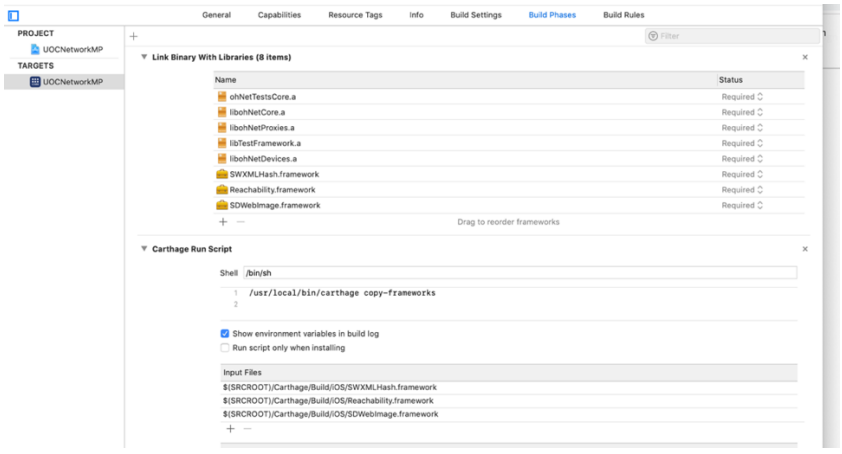

Figure 58 Carthage frameworks imported to an Xcode target

5.4 Implementing the library in Swift

After a long process of building and configuration I am ready to start writing the library implementation and the UI. In this last section I will cover how to implement the library call-backs for the proxy side and how to enable services on the provider side. Please note I will not be covering the model structure, the XML parse or the player management, for more information please refer to the code.

5.4.1 Proxy implementation and delegation

As mentioned several times, proxy files are in charge of discovering services from other devices in the same network and all the functions and variables can be easily identified because they start with either ‘cp’ or ‘prx’.

Having a closer look at one of the proxy files that implements a service, for example, the *ConnectionManager,* four different types of functions can be identified.

1. Constructor and De-constructor: Used for initialising and de-initialising a service, at the moment of destroying it the library needs to notify and clear the variables and the dependencies.

```
/**
 * Constructor.
 * @return  Handle which should be used with all other functions in this header
 */
DllExport THandle STDCALL CpProxyUpnpOrgConnectionManager1Create(CpDeviceC aDevice);
/**
 * Destructor.
 * @param[in]  aHandle   Handle returned by CpProxyUpnpOrgConnectionManager1Create
 */
DllExport void STDCALL CpProxyUpnpOrgConnectionManager1Destroy(THandle aHandle);
```

Figure 59 OhNet connection manager Constructor and De-constructor

2. Synchronous actions: Functions used for triggering service actions such as play, *PrepareForConnection*, and others. These type of actions block the main thread until the action is completed so I very much do not recommend using them.

```
DllExport int32_t STDCALL CpProxyUpnpOrgConnectionManager1SyncPrepareForConnection(THandle aHandle, const char*
    aRemoteProtocolInfo, const char* aPeerConnectionManager, int32_t aPeerConnectionID, const char* aDirection,
    int32_t* aConnectionID, int32_t* aAVTransportID, int32_t* aRcsID);
```

Figure 60 Synchronous action example

3. Asynchronous actions: Same as the previous actions but this action is performed asynchronously. I will be using this one within the project. As shown below these type of actions are composed for two functions and they can be distinguished because they have the words ‘begin’ or ‘end’ in the function declaration. To start the actions begin need to be called and when the action is completed, the call-back will be received using the variable *OhNetCallbackAsync* and inside this the end function needs to be called passing as an argument the call-back received. If it is not the same call-back, the action will fail. The last parameter of the begin action is a pointer that can be used for sending any object and it will be returned inside within the call-back. For services I will be using a custom class call *PrxServiceCallback* that will be useful for passing data between these functions.

```
DllExport void STDCALL CpProxyUpnpOrgConnectionManager1BeginPrepareForConnection(THandle aHandle, const char*
    aRemoteProtocolInfo, const char* aPeerConnectionManager, int32_t aPeerConnectionID, const char* aDirection,
    OhNetCallbackAsync aCallback, void* aPtr);
```

Figure 61 OhNet Connection manager asynchronous begin action

```
DllExport int32_t STDCALL CpProxyUpnpOrgConnectionManager1EndPrepareForConnection(THandle aHandle, OhNetHandleAsync
    aAsync, int32_t* aConnectionID, int32_t* aAVTransportID, int32_t* aRcsID);
```

Figure 62 OhNet Connection manager asynchronous end action

From Swift 3 (currently 4.2) working with C and C call-backs are much easier and they can be managed inside a completion block (kind of call-back).

```
public func beginPrepareForConnection(connectionInfo:ConnectionInfo, completion:((PrxServiceResponseCallback) -> ())?){

    print("\(logClassName) beginPrepareForConnection \(deviceName)")

    let prxServiceCallback = PrxServiceCallback.init(prxService: self)
    prxServiceCallback.completion = completion

    prxServiceCallback.dataPassed[.connectionInfo] = connectionInfo

    CpProxyUpnpOrgConnectionManager1BeginPrepareForConnection(prxHandleId, connectionInfo.protocolInfo, connectionInfo.peerConnectionManager,
        connectionInfo.peerConnectionID, connectionInfo.direction.rawValue, { (pointer, ohNetAsync) in

        let prxServiceCallback : PrxServiceCallback = bridgeToTypeRetained(ptr: pointer!)
        let connectionInfoPassed = prxServiceCallback.dataPassed[.connectionInfo] as! PrxConnectionManager.ConnectionInfo
        (prxServiceCallback.prxService as! PrxConnectionManager).endPrepareForConnection(connectionInfo: connectionInfoPassed, async: ohNetAsync!, completion:
            prxServiceCallback.completion)

    }, bridgeToPointerRetained(obj: prxServiceCallback))

}
```

Figure 63 Swift begin action

```
private func endPrepareForConnection(connectionInfo:ConnectionInfo, async: OhNetHandleAsync, completion:((PrxServiceResponseCallback) -> ())?){

    print("\(logClassName) endPrepareForConnection \(deviceName)")

    let responseCallback = PrxServiceResponseCallback()

    let resultCode = CpProxyUpnpOrgConnectionManager1EndPrepareForConnection(prxHandleId,
                                                                             async,
                                                                             &connectionInfo.connectionID,
                                                                             &connectionInfo.avTransortID,
                                                                             &connectionInfo.aRcsID)

    if resultCode == 0{

        responseCallback.response[.connectionInfo] = connectionInfo

    }

    responseCallback.success = resultCode == 0
    responseCallback.resultCode = resultCode

    completion?(responseCallback)

}
```

Figure 64 Swift end action

To know the equivalent Swift to C type, refer to the Swift developer website.

4. Notifications: These functions are used to send events when a state variable (as mentioned in the UPnP documentation) changes its value. The state variables can also be identified because the functions that return the value have the name property in the declaration. The library call-back strategy would be similar to that used for the asynchronous actions but in this case *OhNetCallback* will be the variable used that will also return the pointer I specify.

```
DllExport void STDCALL CpProxyUpnpOrgConnectionManager1SetPropertySourceProtocolInfoChanged(THandle aHandle, OhNetCallback
    aCallback, void* aPtr);
```

Figure 65 Function for setup de notification

```
DllExport int32_t STDCALL CpProxyUpnpOrgConnectionManager1PropertySourceProtocolInfo(THandle aHandle, char**
    aSourceProtocolInfo);
```

Figure 66 Function to be called after the notification is triggered for obtaining the updated value

```
CpProxyUpnpOrgConnectionManager1SetPropertySourceProtocolInfoChanged(prxHandleId, { (pointer) in
    let prxConnectionManager:PrxConnectionManager = bridgeToTypeUnretained(ptr: pointer!)
    var aSourceProtocolInfo:UnsafeMutablePointer<CChar>? = nil

    CpProxyUpnpOrgConnectionManager1PropertySourceProtocolInfo(prxConnectionManager.prxHandleId,
        &aSourceProtocolInfo)

    prxConnectionManager.sourceProtocolInfoArray = String(cString:
        aSourceProtocolInfo!).components(separatedBy: ",")
    prxConnectionManager.delegate?.sourceProtocolInfoArrayDidChange(to:
        prxConnectionManager.sourceProtocolInfoArray)

}, bridgeToPointerRetained(obj: self))
```

Figure 67 Swift implementation for OhNet notification

To subscribe to a notification, I need to use the *CpProxySubscribe* function.

Finally, I need to create protocols (similar to Java's Interface) to notify whoever subscribes to my Swift service class that the notification event happened.

```
//MARK: - PrxConnectionManager Protocol Implementation
public protocol PrxConnectionManagerDelegate{

    func sourceProtocolInfoArrayDidChange(to newSourceProtocolInfoArray:[String])
    func sinkProtocolInfoArrayDidChange(to newSinkProtocolInfoArray:[String])
    func connectionIDsArrayDidChange(to newConnectionIDsArray:[Int32])

}
```

Figure 68 Swift Connection Manager proxy protocol

```
public var delegate:PrxConnectionManagerDelegate?
```

Figure 69 Swift Connection Manager proxy protocol variable

5.4.2 Provider types and lists of actions for services

For this project I will implement the renderer provider part so the app will be able to play and queue music and videos. The difference between the proxy services and the provider service implemented is that my provider will also implement the *OpenHome* services apart from the UPnP services. Showing how these new services work will give a new approach to decide whether or not they can be implemented on the proxy side. The main advantages for the *OpenHome* services are:

- Better transport control.
- Better play queue list, because *AVTransport* from UPnP only implements the current track and the next.
- Credential services for playing music from Spotify, tidal and others.
- Radio services for streaming live content.

In the proxy implementation I described the main parts which are the constructor, state variables (AKA properties) and actions. The steps required for defining a provider service are:

1. Constructor and de-constructor: The functions are similar to the one described for the proxies but they will be identified because they start with DV instead of CP.
2. Enable properties: The process for choosing which properties or state variables the service will expose to the other devices.

```
DvProviderUpnpOrgConnectionManager1EnablePropertySourceProtocolInfo(prvHandleId)
DvProviderUpnpOrgConnectionManager1EnablePropertySinkProtocolInfo(prvHandleId)
```

Figure 70 SourceProtocol and SinkProtocol available

3. Properties initialisation: Every property needs to have a value and it cannot ever be a null value. Sending a non-initialised property will cause the library to crash. The Get/Set of a property is done through the corresponding provider function. When a variable is set, the notification for the provider will be triggered automatically. There are different approaches however I have decided to use Swift computed variables because it saves a lot verbose every time I need to set or get a property.

```
/** Protocol Info **/
public private (set) var sourceProtocolInfoArray:String{
    get{
        var aSource:UnsafeMutablePointer<CChar>? = nil
        DvProviderUpnpOrgConnectionManager1GetPropertySourceProtocolInfo(prvHandleId, &aSource)
        return String(cString: aSource!)
    }
    set(value){
        DvProviderUpnpOrgConnectionManager1SetPropertySourceProtocolInfo(prvHandleId, value, &aChanged)
    }
}
public private (set) var sinkProtocolInfoArray:String{
    get{
        var aSink:UnsafeMutablePointer<CChar>? = nil
        DvProviderUpnpOrgConnectionManager1GetPropertySinkProtocolInfo(prvHandleId, &aSink)
        return String(cString: aSink!)
    }
    set(value){
         DvProviderUpnpOrgConnectionManager1SetPropertySinkProtocolInfo(prvHandleId, value, &aChanged)
    }
}
```

Figure 71 Swift computed variable for source protocol info property

4. Enable actions: Similar to enable properties, all the actions that the service will expose to other devices needs to be declared. This actions will return to the proxy or set from the proxy the properties, or it will simply perform and action such as play, *PrepareForConnnection*, or others.

The provider side seems easier than the provider, because, from my point of view I define the capabilities of my device. On the other hand, the proxies need to manage different behaviours from different devices.

All the documentation about UPnP and *OpenHome* services can be found on the corresponding website.

6. App flow and test

After coding the iOS app, it is time to test it and show its flow. This app:

- Is available into the Apple iOS app store under the name 'UOC Net MP'.
- Can be installed using the IPA files (one for each version) using the program 'Cydia Impactor'. For this process an iOS device needs to be plugged into the computer. After downloading and opening the program, the IPA needs to be dragged into the program. After entering the iTunes credentials the process will start and, if it has been success, the app will be installed into the device. This process allows the app to be active for 30 days maximum. After that the app will crash when it is being initialised.

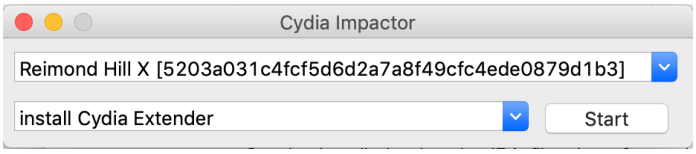

Figure 72 Cydia Impactor interface

- Can be compiled and installed onto any iOS device using a valid apple developer account.
- Can be compiled using the emulator but results may be unexpected. This chapter will assume that a previous Xcode project review has been performed.

6.1 Verifying the provider implementation and inspecting proxies

Before starting to use the app, I will perform a device network inspection to discover which devices on the network implement the UPnP protocol. In addition, I will test that the provider implementation shows the services declared and implemented.

There are various ways to test whether or not the media renderer provider has been implemented successfully. One of them is to test it with other devices but, despite being good for testing performance, it is uncertain how many services or actions a device implements. In this section I will use a java application called 'cling-workbench[22]' to inspect the UPnP devices on the network. There is not a lot of documentation for this application but it is quite straight forward.

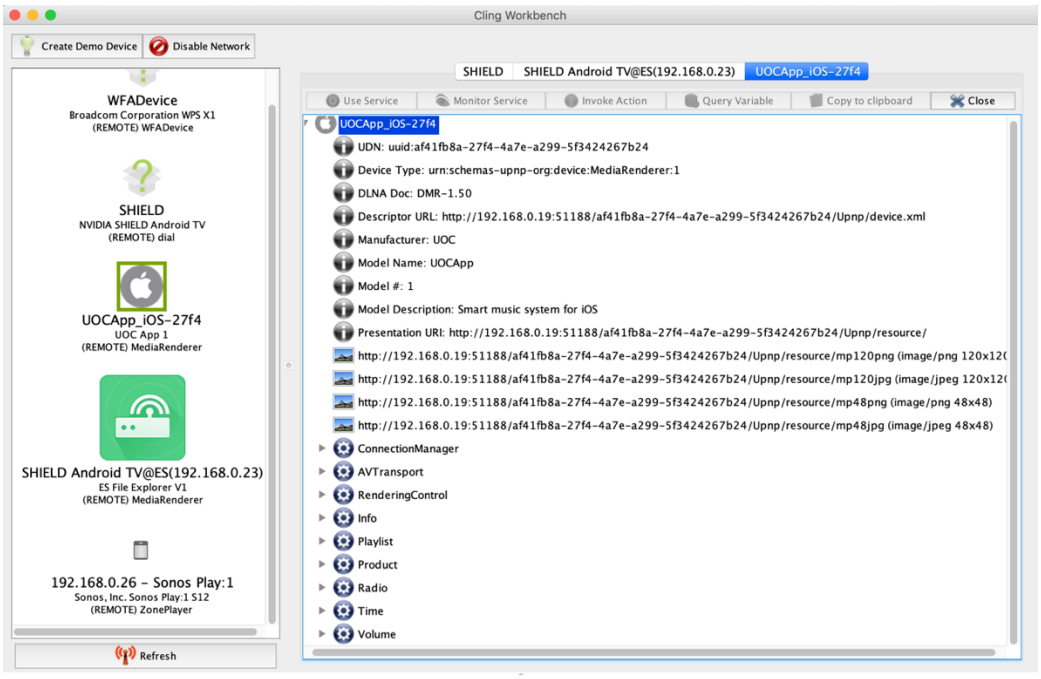

Figure 73 Network device inspection

As shown above, on the left side, cling-workbench shows all the devices on the same network and what type of devices they are (Server/Renderer). Moreover, on the right side, it shows the device description and the services it implements. By unfolding one of the services from the list, all the actions and properties are shown.

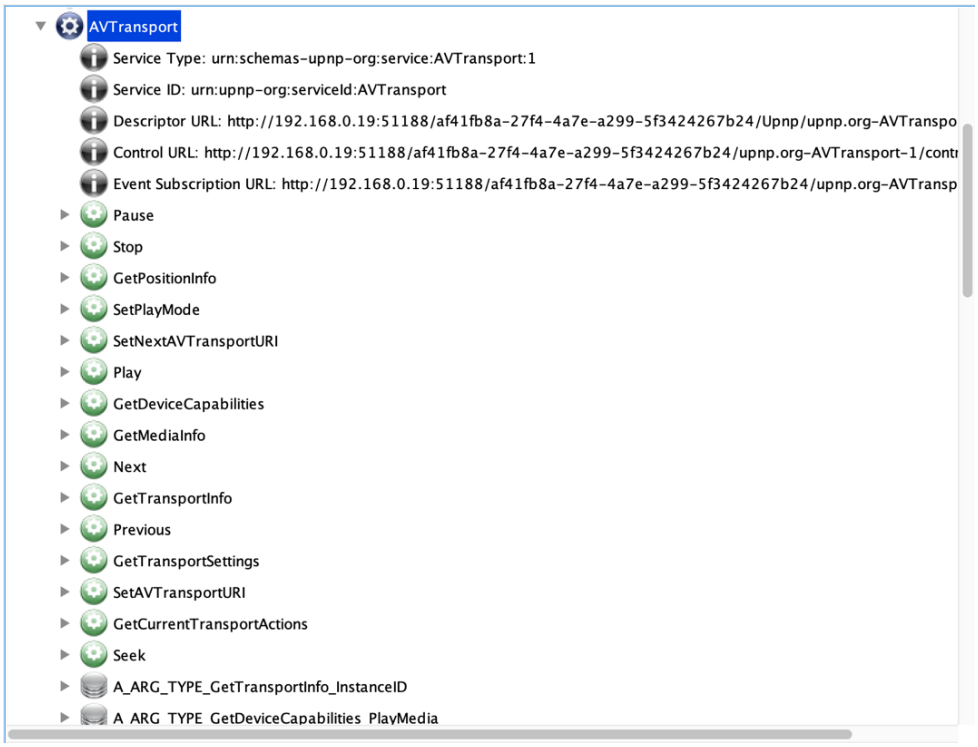

Figure 74 List of actions and properties for AVTransport service

One of the strongest functionalities of this application is the possibility to perform actions by selecting an action from the list and by clicking the 'Invoke action' button situated on the top menu.

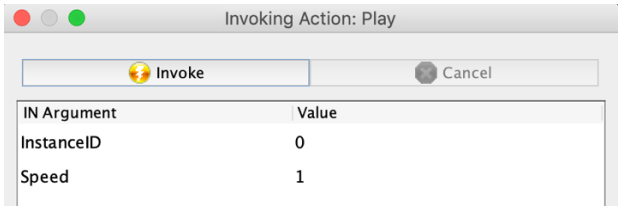

Figure 75 Invoking play action for AVTransport

Cling-workbench is a great tool and, as well as showing devices and services, it also proves that all the services, actions and properties are correctly declared. Now it is time to see if they work correctly by using the app.

6.2 App flow and functionalities

In this section I will show the app flow and how to interact with other devices' services and actions. It is worth mentioning that the interaction would be from the proxy point of view. Even the app provider will be treated as a proxy.

6.2.1 Initialisation and default devices selection

The app has been coded for demo purpose and for a one time only use. In other words, it will always start asking which server and renderer the app should start using by default but it will not remember it for the next time. That can be easily fixed by storing the current server and renderer's UDN into the UserDefaults, core data, etc. During the app lifecycle these two devices can be changed.

On the other hand, the OhNet initialisation occurs during the *OpenHome*Object singleton class' init method so, when it is first called in SelectServerViewController, it gets initialised and it starts implementing the call-backs for when a device is added or removed. When a device is added, all the services and call-backs for this device are also initialised.

The initials UIViewControllers are for selecting the default server and renderer.

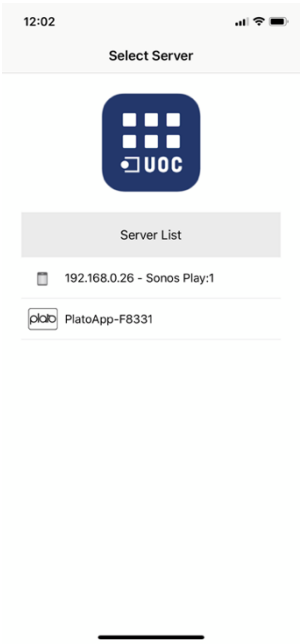

Figure 76 Server selection

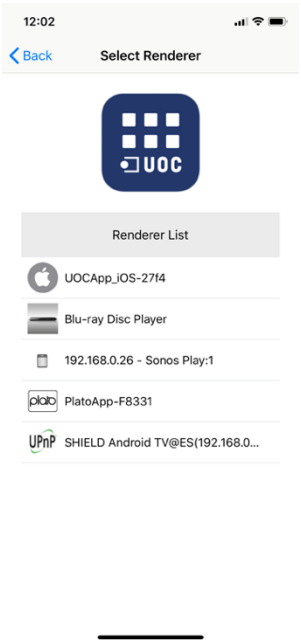

Figure 77 Renderer selection

As seen above, there are more renderers than servers. For this demo, the Android device 'PlatoApp-F8331' will be used. Having fewer servers makes this protocol harder to use.

If the selection cannot happen the app will not go to the main UIViewController.

From now on I will use UIViewController to refer to what is known in Android as 'Activity'. A UIViewController can be thought as an agent that controls (and has the implementations of methods) inside the MVC (Model-View-Controller) model.

6.2.2 Main UIViewControllers

After selecting the server and renderer, the main UIViewController is shown and it embeds three sub UIViewControllers that can be accessed using the segmented control. Each UIViewController implements a different service (sometimes two) and, depending on if it is a server or renderer service a dropdown view will be available to select another device. The 3 sub UIViewControllers that can be found are:

1. Library: Implements the server's *ContentDirectory*'s browse action and it will browse from the root directory.

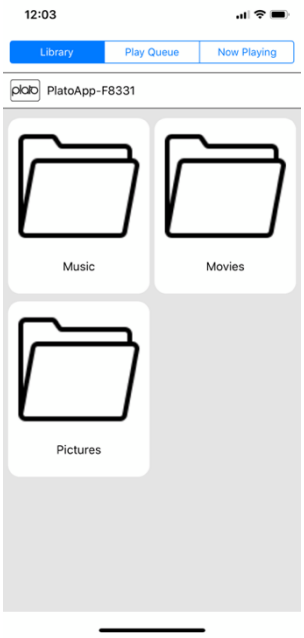

Figure 78 Library view controller

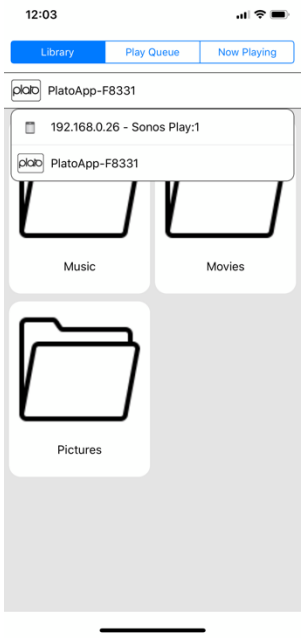

Figure 79 Library view controller with servers dropdown

This UIViewController implements the following protocols:

- CPDeviceListDelegate: Notifies when the current server changes.
- PrxContentDirectoryDelegate: Notifies when the system ID changes. Every time it changes, a new browse needs to be performed as this notification alerts that a change (addition, deletion, etc) in the *ContentDirectory* occurred.

By clicking into 'Music' and then 'Tracks' a list of audio tracks will appear and, by pressing any track, the play menu popup will appear. This menu implements the media addition methods for the renderers' *AVTransport* service and it has a dropdown view for selecting other renderers available. The track selected is 'Every day I Love you less and less' by 'Kaiser Chief' browsed from the server selected at the beginning. With the help of the play menu I can send the URL of this song to:

- UOCApp_iOS-27f4
- Blue-ray Disc Player
- 192.168.0.26 – Sonos Play:1

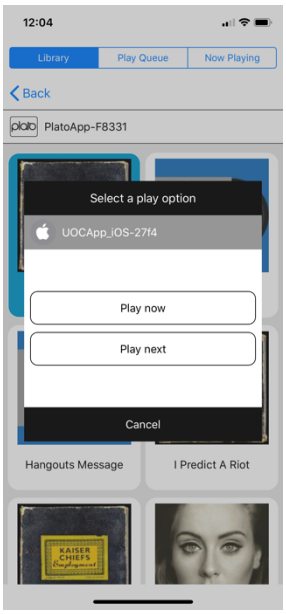

Figure 80 Popup play menu

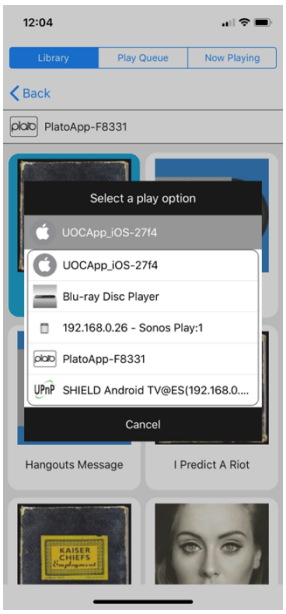

Figure 81 Popup play menu with renderers dropdown view

2. Play queue: Implements the renderer's *AVTransport*'s *getMediaInfo*. The view is a basic table view that displays the current track and the next track information. At the moment no selection has been considered. It has a renderers dropdown view to swap between devices and see what media is set in each device.

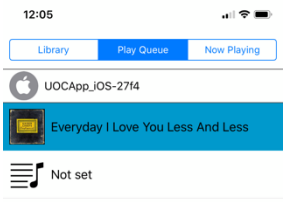

Figure 82 Play queue UOCApp_iOS-27f4

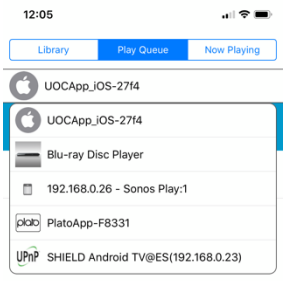

Figure 83 Play queue UOCApp_iOS-27f4 with renderers dropdown view

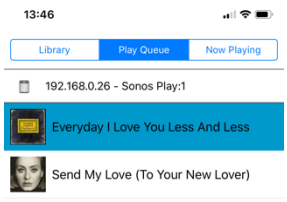

Figure 84 Play queue 192.168.0.26 - Sonos Play:1

This UIViewController implements the following protocols:

- CPDeviceListDelegate: Notifies when the current renderer changes.
- PrxAVTransportDelegate: Notifies when either the current or next track changes.

3. Now playing: Implements the renderer's *AVTransport* service for displaying the current media item information and the *RenderingControl* service for performing actions such as control play, pause, next and previous. Furthermore, the *RenderingControl* service is used for seeking an absolute time position within the media item.
   Finally, to update the progress time when a media item is playing, a timer is declared for performing the '*beginGetPositionInfo' AVTransport's* action every second.
   As mentioned before, if the device had implemented the *OpenHome* services, I would have used the *Time* service and I would have used the *PrxTimeDelegate* for this matter.

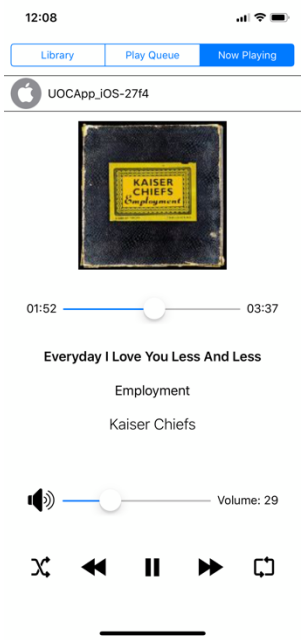

Figure 85 Now playing UOCApp_iOS-27f4

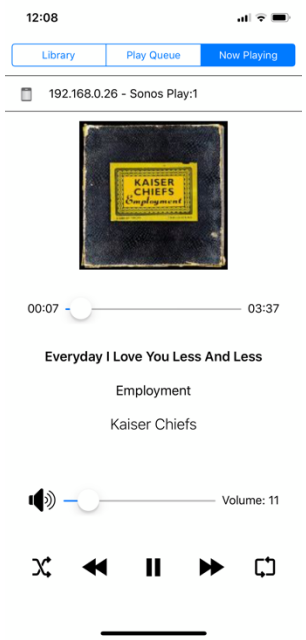

Figure 86 Now playing 192.168.0.26 - Sonos Play:1

This UIViewController implements the following protocols:

- CPDeviceListDelegate: Notifies when either the current server or the current renderer has changed.
- PrxAVTransportDelegate: Notifies when the current track, the transport state (play or pause) or play mode (shuffle or repeat) has changed.
- PrxRenderingControl: Notifies when the volume or the mute has changed.

To know more about the protocols, refer to the code as some of the protocols are used for different purposes with the help of the Swift optional protocol methods. In other words, not all the methods inside a protocol need to be used.

7. Conclusions

One of the challenges I faced at the beggining of this project was to be able to summarise and structure all the content that I have learnt during the last year and a half. As mentioned at beginning, documentation to start working was one of the main challenges I faced when I started to develop this protocol. By reading the selected UPnP documentation I have made and, once the Xcode project is setup, going through the library declarations is more or less complicated depending on the developer experience.

Despite the complexity and depth of this project, I strongly believe that, through this project, I have been able to describe and implement an iOS application that implements the UPnP protocol providing the proxies implementation and the media renderer provider. In other words, the app can read, send and play content.

As for the results achieved with the app, I think they are quite satisfactory. After some testing I can confirm that the app can interact with:

- iOS devices that have this app open. It may also work with others.
- Android devices with a UPnP app such as Plato open. However, this app is more *OpenHome* services friendly.
- Sonos devices.
- Smart TV's connected on the same network. The positive part is that, as long as the television is on, the UPnP is working continuously. Some televisions stop the current task to start playing the audio of the video sent and some others return an error if the user is not on the music player section.
- Some Blu Ray players, to my surprise, are also UPnP ready.

On the other hand, some of the negatives I have experienced during my testing are:

- The app crashes sometimes during initialisation when some network devices such as NAS or Microsoft networks are attached. This crash has only happened when I am running the app without Xcode compilation.
- The library has an internal error when changing the playback mode in the *PrxRenderingControl*.

The mobile application is incomplete but I strongly believe that, after reading the code provided, the implementation for the media server provider can be achieved using different techniques. These ones can be either using core data (mentioned in chapter 5), Codable protocol (from Swift 4) or SQLite frameworks (however in my opinion I would not recommend this one).

Furthermore, the code has been structured in a way that can be easily moved to a custom framework. This was an initial idea at the start of the project but the complexity required was too extensive to include it in this project. Also and as mentioned before, when Swift dependency manager reaches iOS, this process will be different.

To summarise, the next steps to work in the future for this project are:

- Media server provider implementation with both UPnP and *OpenHome* services.
- *OpenHome* proxy's services implementation.
- UI Improvement for tablets.
- Integration for streaming and sharing media services such as Spotify, *TuneIN* (Radio), YouTube, among others.

8. Bibliography

Carey, W. (2017). *Swift vs. Objective-C*. Retrieved from Upwork: https://www.upwork.com/hiring/mobile/swift-vs-objective-c-a-look-at-ios-programming-languages/

Contributors, W. (2018). *Android*. Retrieved from Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/Android

Contributors, W. (2018). *Netflix*. Retrieved from Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/Netflix

Contributors, W. (2018). *Sonos*. Retrieved from Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/Sonos

Contributors, W. (2018). *Spotify*. Retrieved from Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/Spotify

Contributors, W. (2018). *UPnP (Universal Plug and Play)*. Retrieved from https://en.wikipedia.org/wiki/Universal_Plug_and_Play

Contributors, W. (2018). *YouTube*. Retrieved from Wikipedia, The Free Encyclopedia: https://en.wikipedia.org/wiki/YouTube

Jagtap, S. (2017). *Swift Dependency Management for iOS*. Retrieved from Medium: https://medium.com/xcblog/swift-dependency-management-for-ios-3bcfc4771ec0

McDonald, N. (2018). *Digital in 2018: World's internet users pass the 4 billion mark*. Retrieved from We are social: https://wearesocial.com/us/blog/2018/01/global-digital-report-2018

Orf, D. (2016). *A brief history of iOS*. Retrieved from Gizmodo: https://gizmodo.com/a-brief-history-of-ios-1780790760

UPnP. (2002). *UPnP AV Architecture 1.0.* Retrieved from UPnP: http://www.upnp.org/specs/av/UPnP-av-AVArchitecture-v1-20020625.pdf

UPnP. (2008). *UPnP Device Architecture 1.0.* Retrieved from UPnP: http://www.upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.0-20081015.pdf

[1] Internet, social media and smartphones users in 2018, accessed 22 September, 2018, https://wearesocial-net.s3.amazonaws.com/wp-content/uploads/2018/01/DIGITAL-IN-2018-001-GLOBAL-OVERVIEW-V1.00.png

[2] Sonos Play device, accessed 24 September, 2018, https://store.storeimages.cdn-apple.com/4667/as-images.apple.com/is/image/AppleInc/aos/published/images/H/KL/HKL12/HKL12?wid=1144&hei=1144&fmt=jpeg&qlt=95&op_usm=0.5%2C0.5&.v=1475452916178

[3] Samsung Smart TV device, accessed 24 September, 2018, https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcS2_IM1-sWV5Qpx1sF5QJNm-ZjU_2_Gcu3xerC_T8sL41dlG3wL

[4] Amazon echo device, accessed 24 September, 2018, https://media.4rgos.it/i/Argos/7549423_R_Z001A?Web&$DefaultPDP570$

[5] Google Chromecast device, accessed 24 September, 2018, https://pisces.bbystatic.com/image2/BestBuy_US/images/products/4397/4397400_sa.jpg

[6] Convert Technologies websites, accessed 28 September, 2018, http://www.convert-av.com/

[7] UOC Network MP bitbucket repository, accessed 03 January, 2019, https://bitbucket.org/reimondhill/uocnetworkmp

[8] Cydia impactor official website, accessed 03 January, 2019, http://www.cydiaimpactor.com/

[9] iOS iPhone 1 image, accessed 1 October, 2018, https://fm.cnbc.com/applications/cnbc.com/resources/img/editorial/2017/06/28/104556364-original-iPhone.600x337.jpg?v=1498757777

[10] iOS device collection, accessed 1 October, 2018, http://www.ndimensionz.com/kb/wp-content/uploads/2017/06/apple_products.png

[11] iPhone completing a transaction with Apple Pay , accessed 1 October, 2018, https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcTCYXuOF3zObT-xmzqpr_fhwvq4M6xoScrrjZt8tT3-1KXllkx-

[12] iPhone completing a transaction with Apple Pay , accessed 3 October, 2018, http://www.applemust.com/wp-content/uploads/2016/11/Apple-Watches.jpg

[13] Apple TV with Siri remote control, accessed 5 October, 2018, https://www.myistore.co.za/media/catalog/product/cache/1/image/1600x/9df78eab33525d08d6e5fb8d27136e95/a/p/appletv4k-2.png

[14] iOS HomePod speaker, accessed 5 October, 2018, https://cdn.vox-cdn.com/thumbor/m28bFiqixPhr3yX8tMZnf93Cmvs=/0x0:2040x1360/1200x800/filters:focal(857x517:1183x843)/cdn.vox-cdn.com/uploads/chorus_image/image/58885767/jbareham_180202_2266_0003.0.jpg

[15] Android Pixel 3 next to iPhone X, accessed 7 October, 2018, https://cdn.vox-cdn.com/thumbor/m28bFiqixPhr3yX8tMZnf93Cmvs=/0x0:2040x1360/1200x800/filters:focal(857x517:1183x843)/cdn.vox-cdn.com/uploads/chorus_image/image/58885767/jbareham_180202_2266_0003.0.jpg

[16] Official UPnP website, accessed 4 November, 2018, http//:upnp.org

[17] OpenHome Github repository website, accessed, 15 November, 2018, https://github.com/openhome/OhNet.

[18] URL for the question raised about the issue, accessed 17 November, 2018, https://stackoverflow.com/questions/46927007/libstdc-is-deprecated-move-to-libc-wdeprecated-but-changing-produces-com

[19] Apple developer website, accessed, 28 November, 2018, https://developer.apple.com/

[20] Xamarin website, accessed, 28 November, 2018, https://visualstudio.microsoft.com/xamarin/

[21] Make app icon website, accessed, 28 November, 2018, https://makeappicon.com/

[22] Cling-workbench download website, accessed 30 December, 2018, http://4thline.org/m2/org/fourthline/cling/cling-workbench/2.1.0/