THE AIRPLAY AUDIO PROTOCOL FOR NETWORK DUMMIES

CHARLES-ANTOINE
DE SALABERRY
260400928

GRAHAM
LUDWINSKI
260395716

NOVEMBER 2013

Telecommunication

PROJECT REPORT

Abstract

The goal of our project is to create an open source implementation of a receiver for Apple's Airplay protocol. The intention is to create a high fidelity receiver compatible with all devices that support airplay. There are currently a few open source Airplay implementations that have long since stopped being developed or offer incomplete implementations. We hope to expand the Airplay protocol's adoption by providing a free and easy way to work with existing Airplay devices.

Our implementation will utilize a Wi-Fi or wired network to live stream audio data between two devices. It should be capable of synchronously streaming to multiple devices at the same time.

The challenges of the project are the following: device recognition over LAN, multicast broadcasting of audio streams, synchronisation of the stream if multiple devices are being used

Note: Due to the megalomaniac characteristic of our project goal, we did not have time to create a functional implementation compatible with the AirPlay protocol. However the extensive research we made on the protocol allowed us to share our findings through a measurement project.

Introduction and Background

As experience audiophiles, we have always been trying to reduce the friction between the music and its listeners. One of the main factor of friction is availability. Not music availability as in access to songs (for which the internet is a great source), but the availability of playback devices and to enjoy it to its highest quality. Common methods to transfer music involve using cables (usually 3.5mm Jack, RCA, or Optical). In order to enjoy audio at its fullest we determined that the biggest friction to get rid of was to that cable constraint, intrinsically linked to the human laziness.

One of the most mature project on the subject has been started in September 2010 by Apple Inc. and allows you to stream media (almost) seamlessly between your devices. One of the main feature of this project however is its proprietary source code, assuring the reign of Apple in that domain and certifying the compatibility between all iDevices, but prohibiting any external application and devices to use that protocol without a lot of work invested in reverse-engineering the protocol, or paying some fees to Apple.

Our project initially intended to cover the development of a prototype allowing any machine to receive an Airplay audio stream and serve as a playback device, but the complexity of the project did not allow us to go that far. A lot of different implementation have been developed, and have long since stopped being developed due to the rapid evolution of the protocol over the successive iOS iterations.

The goal of our project is to capture and analyze network traffic caused by audio streaming using Apple's Airplay protocol. We wish to analyze Airplay's efficiency, average network utilization, and performance, especially on a congested network.

Then we will conduct our own experimental research using an Airport Express to confirm/reject the gathered informations, and make our own observations on the network utilisation of the protocol.

Methodology

Tools

- Hardware
  - mid 2010 15 inch MacBook Pro
    - OS version: MacOS X 10.9 (13A598)
    - Ethernet card: 5764m-v3.38, 0x56cdb6d5
    - Wireless Card Firmware: Broadcom BCM43xx 1.0 (5.106.98.100.22)
  - Airport express
    - Model Number: A1392
- Software
  - Wireshark: 1.10.1 Mac [8]
  - iTunes 11.1.3

Research

The first step we took was to build the skeleton of the revised airplay protocol narrowing down the field with information already available online. Because Airplay is a networking protocol, we had to identify the underlying protocols on which it is based, as well as the port used to transfer information.

Snooping around on the web, we have found multiple definition of the port mapping of the protocol. This is due to the frequent change in Apple's own requirements regarding the Airplay protocol. The most up to date version was found on Carleton.ca and describes as follow:

Port	Type	Protocol	RFC	Used by
80	TCP	HTTP	2616	AirPlay
443	TCP	HTTPS	-	AirPlay
554	UDP / TCP	TRSP	2326	AirPlay
1900	UDP	SSDP	-	Bonjour
3689	TCP	DAAP	-	AirPlay
5297	TCP	-	-	Bonjour
5298	TCP / UDP	-	-	Bonjour
5350	UDP	NAT Port Mapping Protocol	-	Bonjour
5351	UDP	NAT Port Mapping Protocol	-	Bonjour
5353	UDP	mDNS	3927	AirPlay / Bonjour
49159	UDP	mDNS	-	AirPlay / Bonjour
49163	UDP	mDNS	-	AirPlay / Bonjour

Table 1 - ports and protocols used by AirPlay [2]

Port	Type	Service or Protocol Name	RFC	Service Name	Used by / Additional Information
1900	UDP	SSDP	-	ssdp	Bonjour, Back to my Mac
5350	UDP	NAT Port Mapping Announcement	-	-	Bonjour, Back to my Mac
5351	UDP	NAT Port Mapping Protocol	-	nat-pmp	Bonjour, AirPlay, Home Sharing, Printer Discovery, Back to my Mac
5353	UDP	Multicast DNS (MDNS)	3927	mdns	Bonjour, AirPlay, Home Sharing, Printer Discovery, Back to my Mac

Table 2 - Referenced ports used by Apple (pulled from Carleton University) [6]

Protocols

In order to implement a viable and maintainable solution, Apple had to rely on existing protocols. To get a better idea of how this machinery has been engineered, we will first look at the different protocols independently, and then separate what advantage and disadvantages they bring to the Airplay protocol as a whole.

Multicast Domain Name System (mDNS)

Multicast DNS has the same role on small network with no local name server, as DNS on a bigger network including a local server. A paper presented in February 2013 by Apple outlines the primary benefits of mDNS as the following: "*(i) they require little or no administration or configuration to set them up, (ii) they work when no infrastructure is present, and (iii) they work during infrastructure failures."* Although it seems to be a really nice protocol, its goal is to give informations about the device to simplify networking setup, therefore leading to privacy issues. The NSA Security Configuration Guide for OS X actually advises to disable mDNS for security reasons. We will not dive into the proper implementation of mDNS as libraries are available on all platforms to do the job for us. [1] [10]

On Linux, the AirPlay Audio Protocol can be announced with the following command-line:

```
avahi-publish-service "{MAC_ADDRESS}@Babar’s Boombox_raop._tcp"
{PORT_NUMBER} ""sf=0x5" "fv=76400.10" "am=AirPort10,115" "vs=105.1"
"tp=TCP,UDP" "vn=65537" "pw=false" "ss=16" "sr=44100" "da=true"
"sv=false" "et=0,4" "cn=0,1" "ch=2" "txtvers=1""
```

If the TXT record is corrupted, when trying to connect with an AirPlay transmitter, that message will appear:

The remote Speaker "Babar’s Boombox" is not compatible with this version of iTunes.

In order to help us filter out the mDNS packets, we used the following filter in Wireshark:

```
dns and udp.port eq 5353
```

This is the settings we used for the Airport Express. However playing around with an AppleTV, we realised it was not acting the same way. When Airplay is enabled, it also uses port 7000 and 7100 to stream data. [3]

The pw field appears only if the AirPlay server is password protected. Otherwise it is not included in the TXT record.

Here is some valid TXT field we found along the way:

```
hostname = [PTV-3AF3.local]
txt = ["sf=0x4" "md=0,1,2" "am=AppleTV2,1" "vs=130.14" "tp=UDP" "vn=65539"
"pw=false" "ss=16" "sr=44100" "da=true" "sv=false" "et=0,1" "ek=1" "cn=0,1"
"ch=2" "txtvers=1"]

hostname = [Control4-WMB-6003C6.local]
txt = ["fv=s9459.1910.3700" "am=C4-WMB-B" "vs=141.9" "vn=65537" "tp=UDP"
"ss=16" "sr=44100" "sv=false" "pw=false" "md=0,1,2" "ft=0x44F8A00" "et=0,4"
"da=true" "cn=0,1" "ch=2" "txtvers=1"]
```

NAME	VALUE	DESCRIPTION
txtvers	1	TXT record version 1
ch	2	audio channels: stereo
cn	0,1,2,3	**audio codecs**
	0	PCM
	1	Apple Lossless (ALAC)
	2	AAC
	3	AAC ELD (Enhanced Low Delay)
et	0,3,5	**supported encryption types**
	0	no encryption
	1	RSA (AirPort Express)
	3	FairPlay
	4	MFiSAP (3rd-party devices)
	5	FairPlay SAPv2.5
md	0,1,2	**supported metadata types**
	0	text
	1	artwork
	2	progress
pw	false	does the speaker require a password?
sr	44100	audio sample rate: 44100 Hz
ss	16	audio sample size: 16-bit
tp	UDP	supported transport: TCP or UDP
vs	130.14	server version 130.14
am	AppleTV2,1	device model

Table 3 - Text field value meaning from various sources [4] [5] [16]

RTSP

RTSP is used in conjunction with **RTP** to facilitate the live streaming of data. It is used as the controlling protocol. The Apple TV supports the following commands: ANNOUNCE, SETUP, RECORD, PAUSE, FLUSH, TEARDOWN, OPTIONS, GET_PARAMETER, SET_PARAMETER, POST and GET. [11]

- **OPTIONS** simply returns the list of supported commands In its `Public` field, separated by commas.
- **ANNOUNCE** is used to send the stream properties to the server (codec and encryption keys).
- **SETUP** initialises a record session. It uses 3 separate UDP channels.
    - the **server** channel is the channel on which the audio data flows. Notice that only the server needs to announce this channel as the server does not read any audio data from the client.
    - the **control** channel (ie. `control_port=6001`) is responsible of synchronising the packets, and is used as a fallback server channel if the initial packet did not make it through on the server channel. The packet is retransmitted on the control channel.
    - the **timing** channel (ie. `timing_port=6002`) holds the master clock synchroniser, and make the client and server able to share a common clock.
- **RECORD** starts the audio streaming. It gives the server the initial RTP sequence number (seq) as well as the initial RTP timestamp (rtptime) in its `Rtp-Info` field. the servers replies with the audio latency computed.
- **FLUSH** stops the streaming, therefore pausing the audio stream. It contains the RTP sequence number at which it should pause, and the RTP timestamp.
- **TEARDOWN** kills the RTSP session. a new SETUP command should be sent to the server in order to receive data.
- **SET_PARAMETER**/**GET_PARAMETER** allows us to control the playback, such as controlling the volume, and sharing the stream metadata with the receiver:
    - **volume**: the content type of the packet is set to `text/parameters`, and contains desired volume as a float. the value can vary between -30 and 0 and represent the attenuation in dB. a value of -144 means the audio is muted.
      The text field looks like: "`volume: -15.138359`"
    - **metadata**: the content type of the packet is set to `application/x-dmap-tagged`, and the `RTP-Info` field is used to define the RTP timestamp after which the metadata is valid.
    - **artwork**: the content type of the packet is set to "`image/jpeg`". Again, RTP-Info is used to transfer the valid start time.

- progress: like volume, the content of the packet is set to `text/parameters`. It contains the RTP timestamp value at which the stream started playing, the current playback position and the timestamp at which the stream will end. The text field looks like: "`progress: 1146221540/1146549156/1195701740`"

RTP

RTP is also used with RTSP as the transport protocol. It ensures the live streaming of data and lives within the application layer. Usually RTP is used in conjunction with RTCP however, because RTCP monitors and controls quality of service, which Apple has decided against, it was not included.
RTP uses the same ports as RTSP. It conveys information as payloads in the packets sent over the network. the different possible payloads can be done over:

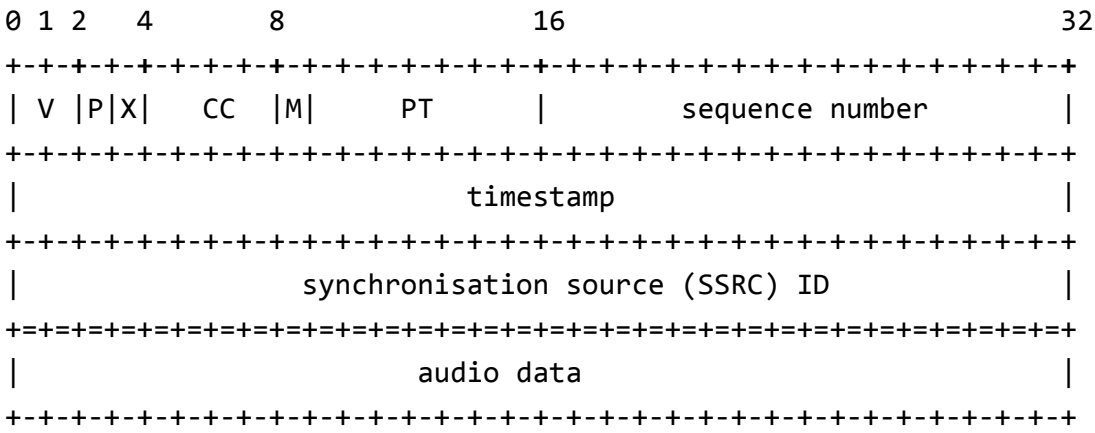
- the **server** channel: `96` to identify the content of the packet as audio data
- the **control** channel: `84` to synchronise the time, `85` to request a lost packet to be retransmitted, and `86` to identify the packet as a retransmitted one
- the **timing** channel: `82` for timing request and `83` for timing reply

As Clément Vasseur notes [4], only the RTP data (audio) packets are fully compliant; here are the ones that are not, and there specificities:

- The **sync packet** flows on the control channel, does not contain any Synchronisation sources ID (SSRC).It does actually make sense; the SSRC field is a unique identifier to differentiate the sources of the stream. In our case, the number of sources will stay at one, allowing us to get rid of that field. Furthermore, even if AirPlay supported multiple sources, the device would initiate a new session, therefore insulating our two streams. It also reduces the packet size by 4 bytes. Compared to the generic RTP audio packet that follows, the 4 bytes SSRC ID is replaced by a 8 bytes field with the current NTP time plus a 4 bytes field containing the RTP timestamp for the next audio packet.

- The **retransmission packet** does not contain any SSRC field either. It is replaced with a 2 bytes field with the sequence number for the first lost packet, followed by another 2 bytes field that contains the number of lost packets that will follow. The data that follows is a full RTP audio packet.

- The **timing packet** allows the two devices to synchronise a master clock for audio. See NTP. Here the SSRC field is replaced by a 8 bytes origin timestamp, a 8 bytes receive timestamp, and a 8 bytes transmit timestamp.



Generic RTPv2 audio packet (CSRC IDs list not displayed) [18]

- V = **version** (2 bits): the structure of the above packet is for version 2.
- P = **padding** (1 bit): If the padding bit is set, the packet contains one or more additional padding octets at the end, which are not part of the payload.
- X = **extension** (1 bit): If the extension bit is set, the fixed header is followed by exactly one header extension.
- CC = **CSRC count** (4 bits) The CSRC count contains the number of CSRC identifiers that follow the fixed header.
- M = **marker** (1 bit): The interpretation of the marker is defined by a profile. It is intended to allow significant events such as frame boundaries to be marked in the packet stream.
- PT = **payload type** (7 bits): This field identifies the format of the RTP payload and determines its interpretation by the application.
- **sequence number** (16 bits): The sequence number increments by one for each RTP data packet sent, and may be used by the receiver to detect packet loss and to restore packet sequence.
- **timestamp** (32 bits): The timestamp reflects the sampling instant of the first octet in the RTP data packet. The sampling instant must be derived from a clock that increments monotonically and linearly in time to allow synchronisation and jitter calculations
- **SSRC** (32 bits): The SSRC field identifies the synchronisation source.
- **audio data** (variable size): the data can be encrypted or not.

NTP

NTP is an application layer protocol used for clock synchronisation over the network. It allows the two devices to synchronise a master clock for audio, and is used to synchronise the playback of the audio stream on multiple audio receivers. Because each and every computers on the planet run on its own specific clock, the need to be able to find a **common clock** to which they can refer came pretty soon. The first version came from before 1985. The algorithm simply computes the round trip delay time, and tries to maintain two different clocks, one on the sender, and one on the receiver, at the same time.

Digital Audio Control Protocol (Over HTTP)

In order to control the playback of a stream directly from iTunes, AirPlay uses a special protocol called DACP. An airplay receiver does not necessary support it, but it it more convenient to integrate to your environment. DACP is announced over mDNS, just like AirPlay. On linux, it can be announced as follow (The `DACP-ID` can be found in any RTSP packet exchanged with the receiver):

```
avahi-publish-service "iTunes_Ctrl_{DACP-ID}_dacp._tcp" {PORT_NUMBER}
""txtvers=1" "Ver=131075" "DbId=63B5E5C0C201542E" "OSsi=0x1F5""
```

The playback is then controlled via http by getting values from the url: `/ctrl-int/1/{COMMAND}`
The command can be any of the following: `beginff, beginrew, mutetoggle, nextitem, previtem, pause, playpause, play, stop, playresume, shuffle_songs, volumedown, volumeup.`

Experiments

In order to identify the protocols Airplay uses, we tested streaming audio from a MacBook Pro using iTunes to an Apple Airport Express router. To analyze traffic on the network used by Airplay, we disabled all network communications used by other services and applications and then used Wireshark to inspect individual packets.

The full filtering rule for Wireshark only showing AirPlay related information was deduced from our research:

```
 tcp.port eq 80 or tcp.port eq 443 ora tcp.port eq 554 or udp.port eq 554 or
tcp.port eq 3689 or udp.port eq 5353 or udp.port eq 49159 or udp.port eq 49163
```

1 - What ports and protocols are being used by Airplay?

First, we wanted to know how Airplay discovers Airplay compatible devices on the network. We began by making sure the Airplay receiver was on the network and discoverable by iTunes. We then disconnected the Airplay receiver from the network, started Wireshark's traffic analysis, and reconnected the receiver.
Immediately, we saw that Airplay uses multicast DNS (MDNS) on port 5353 for peer discovery. Additionally, we noticed that Airplay uses Multicast Listener Discovery (MLD) over ICMPv6 to determine which peers are Airplay compatible.

We then tried to figure out how the actual audio data was transmitted. We let iTunes discover and connect to the Airport Express. We then started streaming music, started recording with Wireshark, and then stopped recording with Wireshark before pausing the music or adjusting the volume. We did this to ensure that no control packets were being sent, and only audio data was being streamed.
We observed that all of the audio data packets were sent via UDP. This matches our predictions that Airplay uses UDP because it is a streaming application and therefore doesn't need to retransmit dropped packets like TCP. Furthermore, we found that Airplay used the Address Resolution Protocol (ARP) to map IP addresses to physical addresses (in our case, an ethernet address).

2 - What is the efficiency of the Airplay protocol?

To measure the efficiency, we streamed an Apple Lossless Audio Codec (ALAC) audio file and recorded the total number of data transmitted via AirPlay. Because files are only transmitted over AirPlay as ALAC, ALAC files don't need to be converted. Therefore to measure the efficiency, we can simply **divide the size of the audio file by the total number of bytes sent** via Airplay during the playback of the entire file. We verified this data with both Wireshark and Activity Monitor.

3 - What is Airplay's average rate of transfer?

To measure Airplay's average rate of transfer, we decided to use a simpler method to capture network traffic. Using Mac OS X's built in network monitor, we collected **average upload and download speeds** during playback of many different audio files. Averaging this data gave us rough estimates for the average rates of transfer for Airplay streaming.

4 - How does upload rate and download rate compare when streaming audio?

To measure a AirPlay device's upload and download streaming rates, we used Activity Monitor to capture network speeds for both a low quality and high quality audio file.

5 - How does Airplay perform on a congested network?

To test AirPlay's performance on a congested network, we began by streaming audio to the AirPlay receiver. We noticed that Airplay chose a port for data transmission (61829) over UDP and a separate port for control (6001). To simulate a congested network, we artificially throttled the data port 61829 from about 140 kB/s to 100kB/s. Immediately, we found that Airplay dramatically reduced the amount of data sent over port 61829 and started sending UDP data packets to the control port 6001. **The audio stream's playback was uninterrupted by this throttling**.

When we artificially limited the speed of the data port to 35 kB/s, there was no interruption in playback nor was sound quality degraded. When we limited the data port to 35 kB/s, AirPlay sends approximately 3 packets over the control port for every 1 packet sent over the data port. However, once we limit the speed of the data port to lower than 33 kB/s, the audio started to skip but the sound quality didn't degrade.

We then limited the data port to 100 kB/s and the control port 6001 down to 100 kB/s. **This stopped music playback immediately**.

6 - How do bitrates for high and low quality audio files compare?

In order to test whether or not high quality songs need a higher streaming bit rate, we decided to capture network speeds over time from Mac OS X's built in Activity Monitor. This allowed us to quickly collect aggregate data about transfer rates while keeping a good level of accuracy.

Results

Although not directly related to AirPlay, our network traffic analysis showed us that many packets had incorrect header checksums. Upon further investigation, we found that many network traffic analysis tools (including Wireshark) suffer from reporting false header checksums on many modern operating systems because these OSs offload some network processing to the network interface controller (NIC) instead of the CPU (reference: http://wiki.wireshark.org/CaptureSetup/Offloading).

1 - Ports In Use

Peer discovery:	multicast DNS (MDNS) on port 5353
Airplay compatible device discovery:	Multicast Listener Discovery (MLD) over ICMPv6
Audio data transferred over:	UDP
IP to physical address mapping:	ARP

2 - Efficiency

Average packet size:	1195.487 bytes/packet
Number of packets:	29715 packets
Total Data transferred:	35523896 bytes or 34691 kB or **33.878 MB**
Audio file size:	**32.334 MB**

Leading to a transfer efficiency of $\frac{size\ of\ file}{total\ data\ transferred} = 95.44\%$ efficiency.

3 - Transfer Rate

Pulled from Table A in the Appendix:
Average Download Rate: **143.3 kB/s**
Average Upload Rate: **0.3 kB/s**

4 - Upload vs Download Rate

We averaged the upload and download rates for both files and got values of 143,328 bytes/second for downloading and 317 bytes/second for uploading, or a download to upload ratio of 452. This data makes sense seeing as the receiver should be downloading the audio stream and only uploading data for control and network overhead purposes.

5 - Behaviour on Congested Networks

Minimum speed for data port: **34 kB/s** (no sound degradation or skipping)
AirPlay uses the control port as a backup data channel in case the data port's speed is limited.

Because AirPlay audio starts to skip at a minimum bitrate, but does not reduce sound quality, this indicates that AirPlay does not down-convert audio on a congested network, unlike many media streaming applications.

6 - Bit-rate and Original File Quality

This showed that while the upload rate of the high quality file was very slightly higher than that of the lower quality file and the download rate of the high quality file was smaller than that of the lower quality file. However ultimately the differences were negligible which leads to the conclusion that iTunes must be transcoding audio files to a constant bit rate before sending it to the AirPlay receiver.

Discussion

We learned a lot about how the different protocols and IP layers interact with each other, especially at the application layer. More specifically, we were amazed at how such a complex system such as the internet can be built with such seemingly simple tools.
It would have been interesting and fun to compare it to other live streaming protocols, either internet based or otherwise. Additionally, it would have been immensely useful and educational to build a working model, even if it was extremely simple.
From the overview we had on the protocol, we saw that Airplay was no more than a big Lego structure that uses open source and already available sources, combining them to get all the features an audio streaming protocol would need in order to function properly. Also, analysing the real internals of the protocols showed us that designing a networking protocol can be really complicated, and that most of the times, if it seems to work, it is better to consider it done than try to get all cases covered (especially true for the open-source alternatives we tried.)

It was a really challenging project, not only because the subject was pretty unique and recent, but also because the materials available were simply deductions from geeks and hackers that posted it on the web. It was pretty hard to get any sample code, as each and every open-source implementations that have been posted online were not documented/commented/implemented enough for us to come and give some help, despite our good intentions. This paper will be posted online for further research (It is a public Google Doc) and anybody is already able to comment as will. We would love to see a proper implementation come out of that paper as we initially planned to do, but one that would sustain the test of time, and not fall into the abyss of the deprecated web.

References

1. Official Apple mDNS Libraries, Apple Inc. (2013, October) [Online] http://www.opensource.apple.com/source/mDNSResponder/
2. Well Known TCP and UDP port used by Apple Software Products, Apple Inc. (2013, October) [Online] http://support.apple.com/kb/TS1629
3. Ports used by the AppleTV, Apple Inc. (2012, September) [Online] http://support.apple.com/kb/HT2463
4. C. Vasseur, (2012, March). Unofficial Airplay Protocol Specifications, [Online] http://nto.github.io/AirPlay.html
5. C. Fergeau (2011, March). Specification of RAOP protocol with timing, [Online] http://git.zx2c4.com/Airtunes2/about/
6. Carleton University (2013, January). The Carlton IT Team and Airplay, http://www6.carleton.ca/iteam/airplay-protocol/
7. J. Laid (2012, April) Airport Express RSA Private Key extracted by James Laid [Online] https://mailman.videolan.org/pipermail/vlc-devel/2011-April/079148.html
8. Wireshark v1.8.8 [Online] http://www.wireshark.org/
9. Airtunes emulator from Abrasive [Online] https://github.com/abrasive/shairport
10. S. Cheshire. M. Krochmal. (2013, February). Multicast DNS. http://tools.ietf.org/html/rfc6762/
11. H. Schulzrinne. A. Rao. R. Lanphier. (1998, April). Real Time Streaming Protocol. http://tools.ietf.org/html/rfc2326
12. W. Richard Stevens. (2004). UNIX Network Programming, Volume 1. illustrated reprint. Addison-Wesley Professional.
13. Implementation of media transport with UDP sockets (in C) [Online] http://www.pjsip.org/pjmedia/docs/html/group__PJMEDIA__TRANSPORT__UDP.htm
14. M. Sekimura (2013, April), Simple Airplay Protocol Analysis [Online] https://gist.github.com/sekimura/753986
15. A Registry of Known DNS SRV [Online] http://www.dns-sd.org/ServiceTypes.html
16. AppleTV 2 Service Announcement (prior to srcvers=130.14) [Online] http://forums.whirlpool.net.au/forum-replies.cfm?t=2167093
17. Enabling AirPlay on a DLINK DIR-655 router [Online] https://discussions.apple.com/thread/3846783?start=0&tstart=0
18. J. Crowcroft, M. Handley, I. Wakeman, UCL Press (1998, December) Internetworking Multimedia [Book] http://www.cl.cam.ac.uk/~jac22/books/mm/book/node159.html

Appendix

Table A

Download rate for lossy (256kbps)	*Upload rate for lossy (256kbps)*	*Download rate for high quality(872kbps)*	*Upload rate for high quality (872kbps)*
46800	613	135000	664
150000	111	124000	54
277000	135	127000	913
129000	238	150000	3150
137000	29	156000	49
142000	108	143000	41
131000	89	149000	160
156000	1730	128000	211
161000	1050	152000	56
143000	49	127000	29
148000	50	155000	146
129000	189	156000	238
154000	218	129000	29
133000	126	130000	196
155000	37	143000	49
126000	88	137000	90
154000	37	160000	78
141000	186	146000	186
2612800	**5083**	**2547000**	**6339**

Table A - Download and upload rates (in bytes per second) of the AirPlay receiver for a high quality song and a lower quality song. the last row represent the total network speeds with our 18 samples

Table B

Average Lossy Download Speed	Average Lossy Upload Speed	Average Lossless Download Speed	Average Lossless Upload Speed
145,156 bytes/sec	282 bytes/sec	141,500 bytes/sec	352 bytes/sec

Table B - Average download/upload rates for lossy vs lossless songs