Digital Terrestrial Broadcast to IP over the Wi-Fi network

Coordinating managers: Alexandru Munteanu

Walid Sami

Michael Borroco

# Summary

Summary 2

1. Introduction 3

2. Goal 3

3. SAT>IP 3

4. Discovery state machine 5

5. RTSP state machine 6

6. RTP/UDP vs. HTTP/TCP 7

7. DVBlast open source software 8

8. Server implementation 9

9. Software code 10

10. Results and utilization 12

11. Future work 14

11. Conclusion 14

Appendix A 15

Appendix B 16

Appendix C 22

Appendix D 24

Appendix E 42

Appendix F 47

Appendix G 48

12. Reference 50

# 1. Introduction

In this paper we try to explain the concept of broadcasting over Wi-Fi networks. We cover the basic knowledge regarding the protocols used, SSDP, RTP, RTSP and the existing technologies and solutions. The following chapters will be dedicated to a better understanding of the necessary steps taken in our implementation.

In a nutshell we have designed a server, from two state machines: the first one handles the discovery by other clients and servers, and the other handles the RTP video streaming request.

# 2. Goal

The goal of the this project is to have a full working solution in order to provide terrestrial television, for any indoor Wi-Fi network, that can be access on any portable device.

In order to achieve this we have set a few requirements in order to have a standardized solution that can fulfill any demand from the user perspective.

These are the following:

* 2 or more DVB-T tuners
* Good image quality and response
* Interoperability with all devices (Android, iOS, etc.)
* Does not block your Wi-Fi for TV only
* Plug and play and easy to use
* Multiple user capabilities for watching different programs
* Interoperability with other SAT>IP servers
* EPG / Subtitles / HbbTV
* Wireless connectivity to our network
* Build-in storage / usb ports / network storage
* Recording capabilities
* Be accessible from any device that haves a browser

Before we committing to the new implementation, at the beginning we have investigated the existing ones, that a regular user can have access. Without losing objectivity, and after technical and not technical debates we have reached the conclusion that the preexisting solution does not fully convince us, and that is better to have our own EBU software. However our research, has paid of with the interesting concept of SAT>IP, that will be explained in the following chapter.

# 3. SAT>IP

SAT>IP is a protocol standardized by ETSI, and was developed by SES S.A, British Sky Broadcasting Ltd and Craftwork ApS on the 8th of January 2015, and the version we have used is 1.2.2. In this chapter will cover the importance of this standard in our project, together with a small description.

In order to have a standardized communication protocol between clients and server, we have chose SAT>IP protocol, which provided satellite broadcasting to IP through the local Wi-Fi networks. This gives the advantage of leveraging preexisting client applications, and completes the offer of live broadcasting (DVB-S and DVB-T).

SAT>IP specifies a **communication protocol**. SAT>IP is **not** a device specification. The SAT>IP protocol may be applied in different devices. Industry is left to come up with new and innovative devices using the SAT>IP protocol.

The SAT>IP protocol distinguishes between SAT>IP clients and SAT>IP servers. Actual devices may be clients or servers or both depending on their feature set.

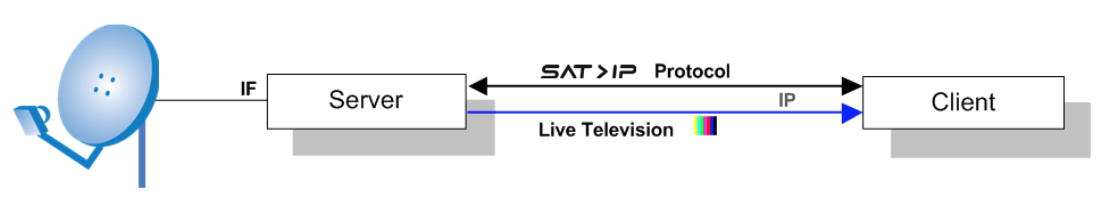


Figure 1.

*SAT>IP Clients*

SAT>IP clients provide the possibility of selecting and receiving live television programs. SAT>IP clients may be – DVB compliant Set-Top-Boxes (STBs) with an IP interface or – Software applications running on programmable hardware such as Tablets, PCs, Smartphones, Connected Televisions, NAS, Routers, etc.

*SAT>IP Servers*

SAT>IP servers will answer requests from SAT>IP clients and forward live television programs to these clients. Servers may take various forms from simple in-home IP Adapters / Multiswitches, Master STBs to ultimately IP LNBs and large MDU headends covering whole buildings or cities.

*SAT>IP Concept*

Unlike in today’s satellite distribution schemes, the SAT>IP architecture allows the reception of satellite television programs also on devices which do not have a satellite tuner directly built-in. Satellite tuners and demodulators are moved or “remoted” into SAT>IP server devices. Clients control SAT>IP servers via the SAT>IP protocol. SAT>IP is a remote tuner control protocol which provides the possibility of remotely controlling tuning devices.

This means that the reception of satellite delivered programming can be dealt with by clients purely in software, provided a SAT>IP server is available on a network. Satellite programs become available on devices, which would never be capable of, supporting satellite TV otherwise e.g. Tablets.

From the distribution point of view, satellite distribution becomes physical layer agnostic and satellite services can be forwarded over all the latest types of IP wired or wireless technologies such as Powerline (PLC), Wireless LANs, Optical Fiber Distribution, etc.

The SAT>IP protocol makes use of:

* -  UPnP for Addressing, Discovery and Description,
* -  RTSP or HTTP for Control,
* -  RTP or HTTP for Media Transport.

SAT>IP uses a subset of the UPnP/DLNA architecture and protocols [1] [2] and SAT>IP devices can be extended to also become DLNA devices. As an example a SAT>IP client could access live media streams through the SAT>IP protocol and access recorded media streams through DLNA.

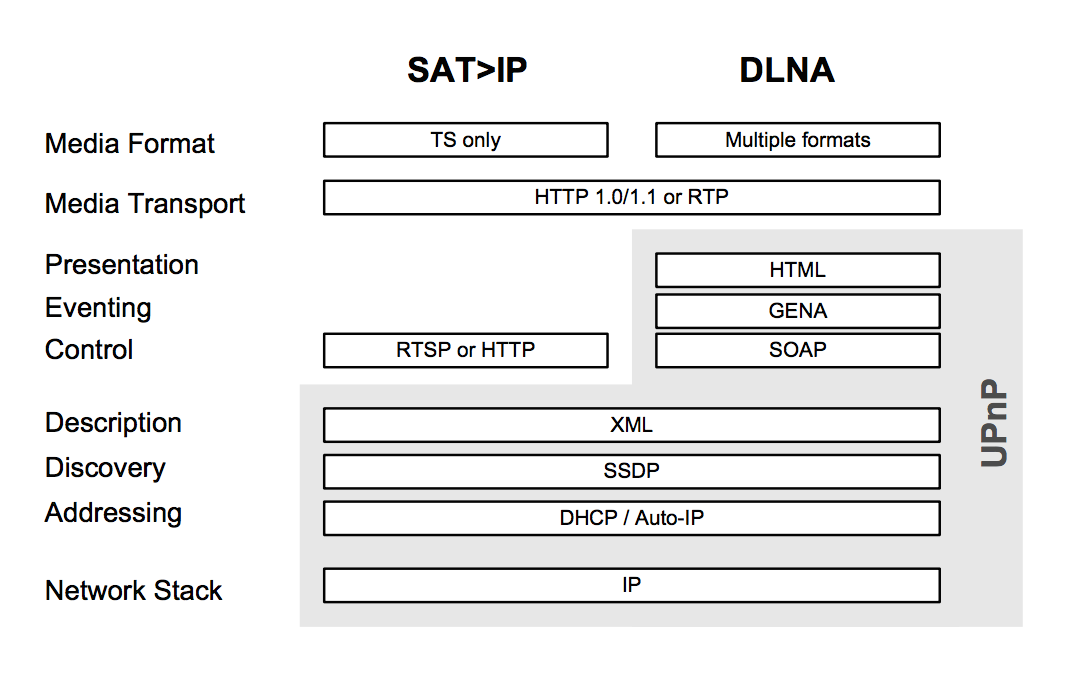


Figure 2.

SAT>IP devices successively go through the following phases: Addressing, Discovery, Description, Control and finally Media Transport.

# 4. Discovery state machine

During the discovery phase SAT>IP servers advertise their presence to other SAT>IP servers and clients. When joining a network, SAT>IP clients search the network for available SAT>IP servers.

Discovery in SAT>IP relies on the Simple Service Description Protocol (SSDP) [11] as specified in the UPnP Device Architecture 1.1 [1].

As a minimum:

* -  a SAT>IP server is a UPnP Device and a UPnP Control Point,
* -  a SAT>IP client is a UPnP Control Point.

SAT>IP servers joining a network multicast **three** different NOTIFY ssdp:alive messages to the SSDP address 239.255.255.250 port 1900. This is a requirement for UPnP root devices according to the UPnP Device Architecture 1.1 [1].

A SAT>IP server **present** on the network has to re-announce itself on a regular basis as described under CACHE-CONTROL above.

A SAT>IP server **leaving** the network needs to signal its departure by sending three different NOTIFY messages with the NTS value ssdp:byebye.

Please note that the **ssdp:byebye** messages should not include the CACHE-CONTROL, LOCATION, SERVER and DEVICEID.SES.COM headers. An example of such ssdp:byebye messages is shown in Section 3.3.3.

A SAT>IP server **changing** the network e.g. when passing from an Auto-IP network to a network with a DHCP assigned address shall signal its departure from one network by sending three NOTIFY messages with the NTS value ssdp:byebye on that network and shall announce its presence on the new network by sending three NOTIFY ssdp:alive messages.

SAT>IP **clients** (being at a minimum only UPnP Control Points) do not announce their presence. For this reason, a client leaving the network is not detectable at this level. The SAT>IP protocol however implements RTSP and IGMP which permit to detect the presence or absence of a client (RTSP session timeout, IGMP membership queries).

# 5. RTSP state machine

Control provides the functionality necessary for SAT>IP clients to request the delivery of media streams from SAT>IP servers. Device Control in SAT>IP can be handled through the use of RTSP or HTTP protocol mechanisms.

SAT>IP servers shall fully implement all protocol mechanisms specified in the current specification. Clients only need to implement those SAT>IP protocols important to their own proper operation.

*RTSP*

SAT>IP clients use RTSP over TCP to setup **RTSP sessions,** with a SAT>IP server. An RTSP session starts with a RTSP SETUP request and ends with an RTSP TEARDOWN message. A number assigned by the server uniquely identifies a session.

When setting up an RTSP session, clients define the transport mode, which will be used for delivering the actual media stream. In the SETUP message they also define or start defining through a URI query the media stream object that they want to be delivered. SAT>IP media stream objects are identified through a streamID. A media stream object is only modified through URI queries and no modification shall occur from any associated RTSP method used to invoke these queries.

Actual stream play-out is started in a session by invoking a PLAY message containing the streamID. During the course of the session, clients can change channels by invoking further PLAY messages with the URI query parameters corresponding to different requested channels.

In order to keep sessions with a server alive, clients need to issue regular RTSP messages within the timeout period (announced by the server in the original reply to the SETUP message). In SAT>IP RTSP OPTIONS messages shall be used as the default keep alive messages.

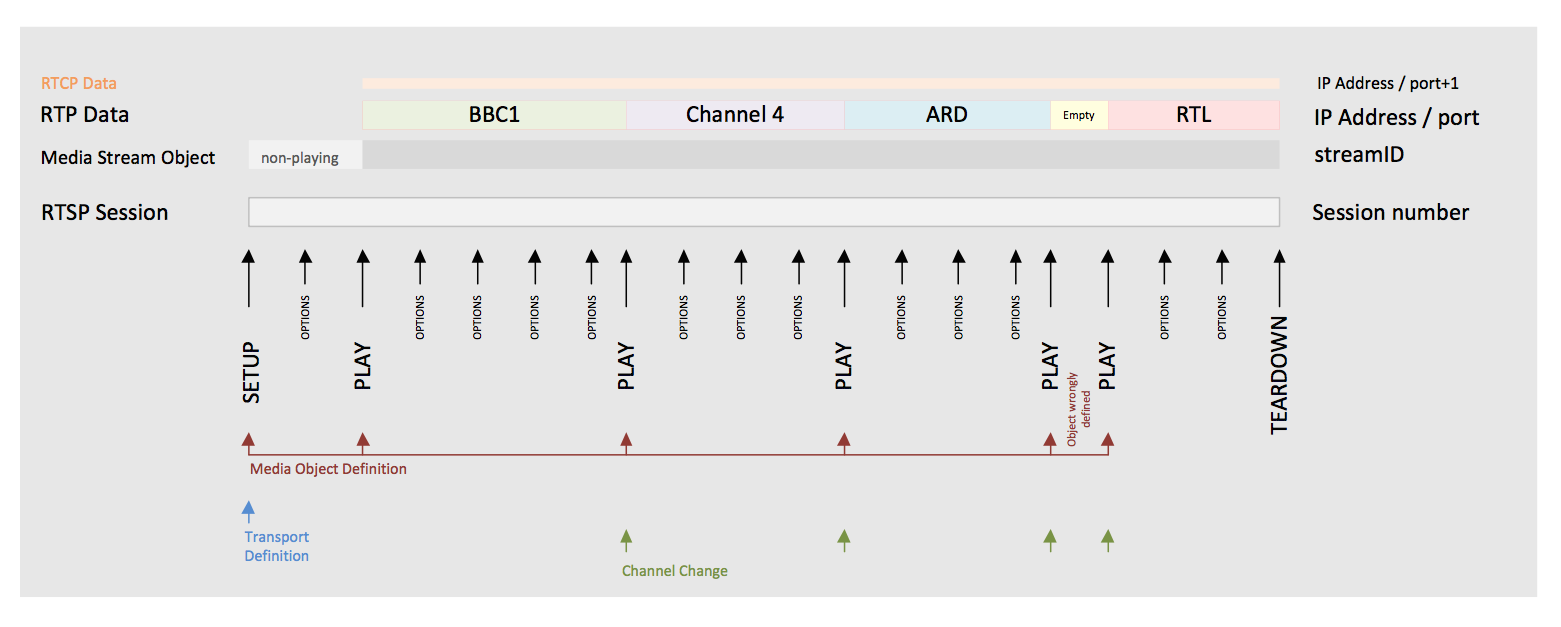


Figure3.

Clients and servers shall support RTSP version 1.0 as described by Appendix D of RFC 2326 [6].

RTSP is a text-based protocol and uses the ISO 10646 character set in UTF-8 encoding. Header field names are case-insensitive and header field values are case-sensitive in RTSP. Lines are terminated by CRLF.

SAT>IP uses the standard RTSP server port number **554**.

# 6. RTP/UDP vs. HTTP/TCP

The Real Time Transport Protocol (RTP) has been around for a long time and is often used for streaming. It's defined by [IETF RFC 3550](http://tools.ietf.org/html/rfc3550). It's a transport protocol which is built on UDP and designed specifically for real-time transfers. It's possible but unusual to use RTP with TCP. Although it sits on top of UDP (or TCP), it's still considered part of the transport layer. It's closely associated with the Real Time Control Protocol (RTCP), which operates at the session layer. The primary function of RTCP is "to provide feedback on the quality of the data distribution," allowing actions such as adjusting the data rate.

Some other protocols are typically used with RTP but aren't tightly coupled to it. The Real Time Streaming Protocol (RTSP), defined by [IETF RFC 2326](http://www.ietf.org/rfc/rfc2326.txt), is a presentation-layer protocol that is described as a "network remote control." It resembles HTTP in some ways, and it carries requests to initiate activities such as playing, pausing, and recording. The Resource Reservation Protocol, with the strained abbreviation RSVP and a spec at [RFC 2205](http://www.ietf.org/rfc/rfc2205.txt), operates at the transport level though it's used in setting up sessions. The protocol stack of RTP, RTCP, and RTSP is sometimes referred to as "RTSP."

RTP, RTCP, and RTSP all operate on different ports. Usually when RTP is on port N, RTCP is on port N+1.

An RTP session may contain multiple streams to be combined at the receiver's end; for example, audio and video may be on separate channels.

UDP URLs aren't widely supported by browsers, so a plug-in is needed to do RTP/UDP streaming to a browser. Flash is the one that's most commonly used. RTP is also used by standalone players such as RealPlayer, Windows Media Player, and QuickTime Player.

Android and iOS devices don't have RTP-compatible players as delivered. There are various third-party applications, including RealPlayer for Android.

The new trend in streaming is the use of HTTP with protocols that support adaptive bitrates. This is theoretically a bad fit, as HTTP with TCP/IP is designed for reliable delivery rather than keeping up a steady flow, but with the prevalence of high-speed connections these days it doesn't matter so much. Apple's entry is HTTP Live Streaming, aka HLS or Cupertino streaming. It was developed by Apple for iOS and isn't widely supported outside of Apple's products. Long Tail Video provides a [testing page](http://www.longtailvideo.com/html5/hls/) to determine whether a browser supports HLS. Its specification is available as an [Internet Draft](http://tools.ietf.org/html/draft-pantos-http-live-streaming-11). The draft contains proprietary material, and publishing derivative works is prohibited.

The only playlist format allowed is M3U Extended (.m3u or .m3u8), but the format of the streams is restricted only by the implementation.

Adobe HTTP Dynamic Streaming (HDS) is also known as San Jose streaming. Like Apple's HLS, it operates over HTTP. Like RTMP, it's associated with Flash. HTTP is more likely to be allowed through than other protocols, and HDS is less of a kludge than RTMP over HTTP. The [technical specs](http://www.adobe.com/products/hds-dynamic-streaming/tech-specs.html) say that Flash is required for playback, so its use is mainly in desktop environments.

DASH, for Dynamic Streaming over HTTP, is MPEG's offering in the HTTP streaming Babel. DASH's creators insist it's not a protocol but an "enabler," but that claim violates the "looks like a duck" principle. It's specified by ISO/IEC 23009-1:2012.

HTML5 needs to be mentioned here, mostly for what it isn't. HTML5 provides the <audio> and <video> tags, along with DOM properties that allow JavaScript to control the playing of the content that these elements specify. This is an application-layer protocol only, with no definition of the lower layers. HTML5 implementations can specify formats, which they process. The server is expected to download the content progressively, and it will keep downloading it completely even if paused, unless the browser completely eliminates the element. The Web Audio API allows detailed programmatic control of playback.

# 7. DVBlast open source software

DVBlast is written to be the core of a custom IRD, CID, or ASI gateway, based on a PC with a Linux-supported card. It is very lightweight and stable, designed for 24/7 operation.

DVBlast does **not** do any kind of processing on the elementary streams, such as transcoding, PID remapping or remultiplexing. If you were looking for these features, switch to [VLC](http://www.videolan.org/vlc/). It does **not** stream from plain files (have a look at [multicat](http://www.videolan.org/projects/multicat.html) instead).

DVBlast supports several input methods:

* linux-dvb-supported cards (DVB-S, DVB-S2, DVB-C, DVB-T...) with or without CI interface
* DVB-ASI cards (from [Computer Modules](http://www.computermodules.com/))
* UDP or RTP, unicast or multicast, streams carrying a transport stream

It outputs one or several RTP streams carrying transport streams with:

* hardware or software PID filtering
* PID-based or service-based demultiplexing
* optional descrambling via CAM device

optional DVB tables

# 8. Server implementation

The first implementation was deployed on a Raspberry Pi. The Raspberry Pi it is called DTT>IP server in the following figures. Using a yaggi terrestrial antenna, we have captured the broadcast signal, through a USB tuner plugged in our Raspberry Pi. Then for the network connection to the Wi-Fi, we have used a basic Ethernet cable, as we can see in the Figure 4.

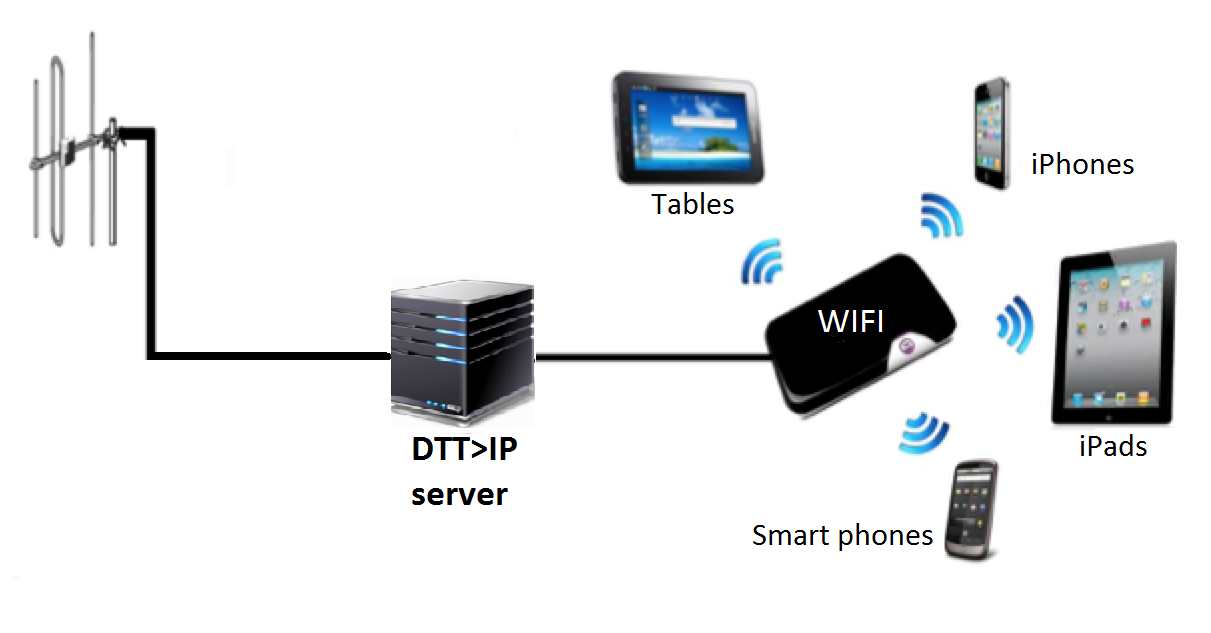


Figure 4.

Underneath the DTT>IP server, we have a Linux environment (Rapian), on which we have installed the firmware to detect the USB tuner, the DVBlast open source software, and the developed application described in chapters 4 and 5. This will allow for our server to be detected by any SAT>IP software client, and will allow a parallel coexistence with other SAT>IP servers.

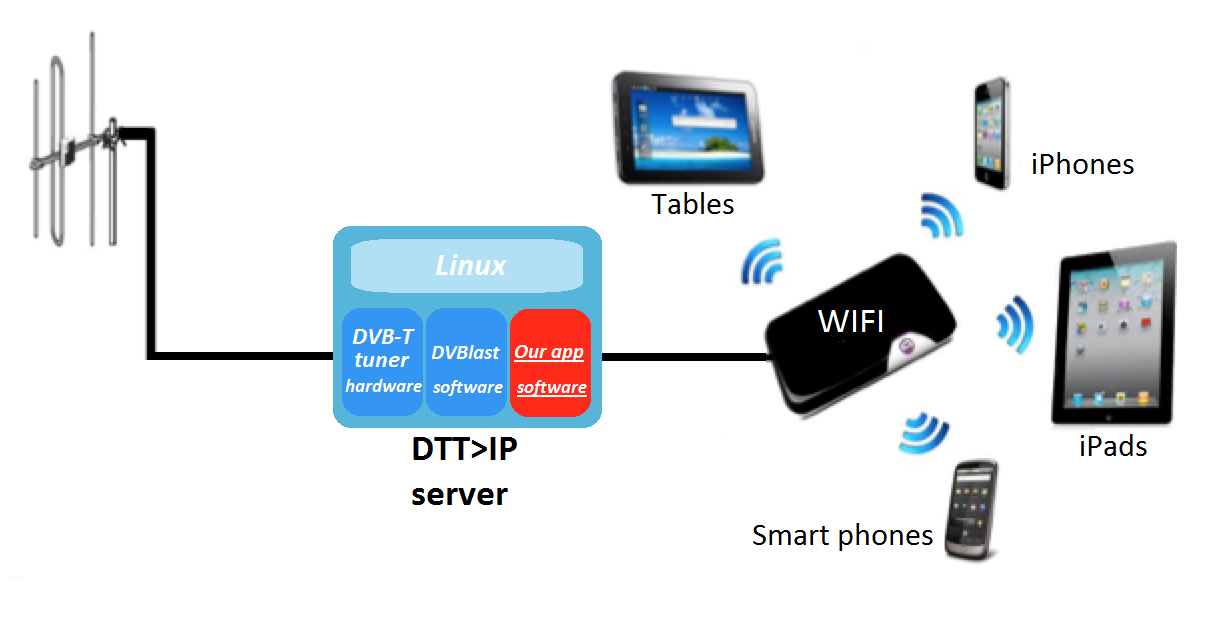


Figure 5.

The application being developed in python and because we have used a Linux distribution, we could deploy this on any hardware that could have Debian installed for example. Bearing this idea in mind, and having a Synology NAS, we went out and tested it. And with no surprise this worked from the beginning, like we anticipated.

# 9. Software code

The software code directory is as follows:

dtt2IP/

├── docs

├── test

├── t2IP

│ ├── \_\_init\_\_.py

│ ├── \_\_main\_\_.py

│ ├── t2IP.py

│ ├── discoveryServer.py

│ ├── rtspServer.py

│ ├── rtspServerWorker.py

│ ├── scanning.py

│ ├── resources.py

│ ├── netInterfaceStatus.py

│ ├── conf/

│ │ ├── discoveryServer.conf

│ │ ├── rtspServer.conf

│ ├── dvb-t/

│ │ ├── allFrequencies.txt

│ └── logs/

│ │ ├── discoveryServer.log

│ │ ├── rtspServer.log

│ │ ├── rtspServerWorker.log

│ │ ├── scanning.log

│ │ ├── resources.log

│ │ ├── t2IP.log

├── t2IP-runner.py

├── LICENSE

├── MANIFEST.in

├── README.rst

└── setup.py

The *“t2IP.py”* file is the main script from your application. In this we will run 3 threads, the first is the discovery server, the second runs the rtsp server, and the third on is to used to scan the broadcast available channels. The code is Appendix A.

The *“discoveryServer.py”* file will create an SSDP server and client. The SSDP client will serve to communicate with other servers and negotiate a unique and unused device ID. Every server must have a unique device ID in order to distinguish between them, from a client perspective. The SSDP server will announce its presence on the network to other SSDP clients. Every time a server joins a network, this will send 3xNOTIFY(ssdp: alive) multicast messages to the IP address: *“239.255.255.255”.*

If it does receive a message from other servers on the same network, saying that device ID used it is not available, before a timeout of 6 seconds, then discovery server is well setup and will continue to send at random time 3xNOTIFY messages, at least 2 times between the interval of expiration. However, if it does receive a message from the other servers before the timeout, our server will send back an acknowledgment, followed by 3xNOTIFY(ssdp: bye:bye) messages, and will restart the whole process from the beginning with an incremented device ID. This code is Appendix B.

The *“rtspServer.py”* file will create the RTSP server, which will allow clients to connect to the RTSP state machine, and make requests/teardowns for the stream that they want to watch. Every client that connects will be an instance of the class *rtspServerWorker*, which will specify the resources need. This code is Appendix C.

The *“rtspServerWorker.py”* file will handle the *rtspServerWorker* class mentioned in the *“rtspServer.py”.* In this class every instance (client) will be an entry in the client’s dictionary (*clientDict*) that will specify for all the parameters of the stream that they are requesting. Another important dictionaries that is used and updated all the time are: frequency dictionary (*freqDict*), this will map the frequencies with the adapters configured with those respective frequencies; frontends dictionary (*frontEndsDict*), which maps the adapter to the owners, the frequencies, and the number of owners. This code is Appendix D.

The *“scanning.py”* file will handle the scanning of the frequencies available. It has an interesting algorithm for keeping updated the available multiplexes, and will handle the “bad behavior” from the USB tuners that may skip sometimes some frequencies. This is also update the list of channels and parameters used by the *rtspServerWorker* class. This code is Appendix E.

The *“resources.py”* file will handle the USB tuners that are plugged into the device. These are also updated in real time, meaning that if new USB tuners become available, these are in real time detected and used at a new request for stream. This code is Appendix F.

The *“netInterfaceStatus.py”* file will handle the IP of the connected interfaces, so that we do not need to specify any IP of our *t2IP* server. However this requires an additional python package (*netifaces*) that needs to be installed prior our application. This code is Appendix G.

# 10. Results and utilization

We have implemented the application to run on Raspberry Pi and Synology NAS with good results. The real-time scanning of the available frequencies was managed for multiple users, as well as tuning to different channels at the same time. The image quality was depending most of the time on the received RF signal, occupying only 4Mbps of the bandwidth. From our observations the multicasting one channel to the network was possible, however with poor playback quality, this problem being related to the concept of multicast. In order for every connected client in the network receive the multicast packets; the channel must have a low transmission rate., which does not satisfies the our minimum transmission rate per channel.

Regarding the HD channels, we can receive them as well, with some playback degradation. The only client application, being able to display HD channel is “*tivizen*”. Another interesting remark concerning the client software, is that “*Sat>IP*” does support only one server per WiFi, where as with “*tivizen*” you could switch between them. “*Sat>IP*” offers a better user experience, with more features like: EPG, subtitles, and Teletext, whereas “tivizen” can only decode the EPG.

Depending on the platform that you want to run our application you must follow the following steps:

***Raspberry Pi:***

Install Raspian image on the SD flash card, by following these steps:

1. Download the Raspian image “*2015-05-05-raspbian-wheezy.img”* to your computer.
2. Insert the SD flash into your computer (Mac).
3. Open a terminal window.
4. Execute:

- “# diskutil list” to list the available disk, and to get the name of the SD flash on which we will pe loading the image *(“2015-05-05-raspbian-wheezy.img”)*

- “# diskutil unmountDisk /dev/disk<disk# from diskutil> “ to unmounts the SD flash, where <disk# from diskutil> must be replace with the number corresponding to your SD flash.

- “# sudo dd bs=1m if=<path to place where you downloaded the image>/2015-05-05-raspbian-wheezy.img of=/dev/rdisk<disk# from diskutil>” to load the image, where <path to place where you downloaded the image> must be replaced with the path to place where you downloaded the Raspian image.

5. Done. You now have and SD flash with an Raspian image on it.

Place the SD flash into the Raspberry Pi, and boot it up with an Internet connection. The default login to the Raspberry Pi is:

* Username: pi
* Password: raspberry

Change to the directory you want to have the application installed; execute the following commands and login with our credential:

“$ git clone -b develop <https://github.com/ebu/dtt2ip.git>”

“$ cd dtt2ip/”

“$ sudo ./raspberryInstall”

This will take care of all the packages need, and make all the necessary changes in order to have a fully working t2IP server, the next time you boot the Raspberry Pi.

Restart the Raspberry Pi and wait for 15-20 min in order for the first scan to finish, before connecting any mobile client.

“$ sudo reboot” to restart the Raspberry Pi

***Synology NAS, more complicated for the moment:***

Install on your Synology from the Package Center application, Debian Chroot by following the steps:

1. Go to Package Center and in the Community tab add the following URL:

https://synocommunity.com/packages

2. Select option install from different third party software

1. Install Debian Chroot
2. Make sure Debian Chroot application is running after installation

Install Git from the Package Center and get the following repository and place it in the location you want to have the t2IP server installed (e.g. volume1/yourDir), by following the setps:

1. Open a terminal window on your computer (Mac):
2. SSH to the Synology as root:

* “# ssh root@ip\_address\_synology”
* “> cd /volume1/yourDir/”, this will change to the directory you have placed the t2IP

server application.

* “> git clone -b develop <https://github.com/ebu/dtt2ip.git>”
* “> cd dtt2ip/”

- “> ./synologyInstall”, this will run the installation script

This will take care of all the packages need, and make all the necessary changes in order to have a fully working t2IP server, the next time you start the Debian Chroot application from Synology.

Stop and Start the Debian Chroot application from the Package Center app, and wait for 15-20 min for the first scan to finish before connecting any mobile client.

# 11. Future work

From the server perspective they may exist improvements all the time, as well as maintaining the code up to date with the new client software requirements. The server must have a configuration page where you can request new scans, configure some parameter, however it must be keeps as simple as possible. This page must provide some real-time and relevant information regarding the status of the server.

The second possible future work must be in the direction of creating our own client software that will run on the entire mobile device, this including Windows 10 phones and tables (by doing this desktops can also have this application as well). This client must provide all the features carried on the transport stream (subtitles, Teletext, HbbTv). Another future implementation can be to provide connectivity to the server through a browser, by using and leveraging MPEG-DASH. As proof of concept we have a basic demo in the “*demoTestLiveTools.tar.gz”* package. Unpack the compressed file, and run as root the “*demoTestLiveTools.py” script*. This will install the necessary tools that will be used and create and “index.html” page, that can be access directly from your host. This will display a basic video player, which will start playing off the air the broadcast channel.

# 11. Conclusion

We have manage to create, based on the Sat>IP standard, our own server for converting DVB-T RF signal to IP packets and stream it through an indoor Wi-Fi network. Being only a software implementation, and using common and well-used development environment, we have deployed it easily on different hardware platforms. Another feature that we have accomplished is the universality for the client application, which means that every user can have a freedom of choice for displaying the broadcast channels.

# Appendix A

* #!/usr/bin/python
* **from** discoveryServer **import** discoveryServer
* **from** rtspServer **import** rtspServer
* **from** scanning **import** scanning
* **import** threading
* **def** main():
* # Make sure that t2IP.log file is clean
* fLog = open('logs/t2IP.log', 'w')
* # Thread 1 discoveryServer
* t1 = threading.Thread(target=discoveryServer)
* t1.daemon = True
* t1.start()
* fLog.write('Info: discoveryServer started\n')
* fLog.close()
* # Thread 2 rtspServer
* fLog = open('logs/t2IP.log', 'a')
* t2 = threading.Thread(target=rtspServer)
* t2.daemon = True
* t2.start()
* fLog.write('Info: rtspServer started\n')
* fLog.close()
* # Thread 3 scanning (w\_scan)
* # Default period for new scan is 3600 seconds.
* # periodNewScan = 3600
* # scanningFlag = 0
* # t3 = threading.Thread(target=scanning, args=[periodNewScan, scanningFlag])
* # t3.daemon = True
* # t3.start()
* # fLog.write('Info: scanning started\n')
* # fLog.write('Info: period for new scan is ' + str(periodNewScan) + ' and scanningFlag is ' + str(scanningFlag))
* # fLog.close()
* # Keep threads alive until KeyboardInterrupt
* **try**:
* **while** t1.is\_alive() **and** t2.is\_alive():
* t1.join(timeout=1.0)
* t2.join(timeout=1.0)
* **except** (KeyboardInterrupt, SystemExit):
* **print** "Info: t2IP server stoped"
* **if** \_\_name\_\_ == '\_\_main\_\_':

    main()

# Appendix B

* #!/usr/bin/python
* # Discovery state machine
* # SSDP protocol
* **import** sys, re, os, random
* **import** threading, socket, struct, time, calendar
* **from** netInterfaceStatus **import** getServerIP
* **from** datetime **import** date, datetime
* **global** SSDP\_TERMINATE
* **global** deviceIdOk
* **global** ssdpAddr, ssdpPort, serverIP
* **global** paramDict
* **global** NT, USN, st
* **global** fLog

* paramDict = {}
* f = open("conf/discoveryServer.config", 'r')
* lines = f.readlines()
* **for** i **in** range(5, len(lines)):
* line = lines[i]
* lineArray = line.split('=')
* lineArray[0] = lineArray[0][:-1]
* lineArray[1] = lineArray[1][1:-1]
* paramDict[lineArray[0]] = lineArray[1]
* f.close()
* SSDP\_TERMINATE = 0
* deviceIdOk = False
* ssdpAddr = '239.255.255.250'
* ssdpPort = 1900
* serverIP = getServerIP()
* # Make sure that rtspServer.log file is clean
* fLog = open('logs/discoveryServer.log', 'w')
* fLog.write("Info discoveryServer: ipAddrServer = " + serverIP + '\n')
* nt1 = 'upnp:' + paramDict['upnp']
* nt2 = 'uuid:' + paramDict['uuid']
* nt3 = 'urn:' + paramDict['urn']
* NT = [nt1, nt2, nt3]
* usn1 = 'uuid:' + paramDict['uuid'] + '::upnp:' + paramDict['upnp']
* usn2 = 'uuid:' + paramDict['uuid']
* usn3 = 'uuid:' + paramDict['uuid'] + '::urn:' + paramDict['urn']
* USN = [usn1, usn2, usn3]
* # bootId = 981
* # configId = 2212703
* # deviceId = 1
* st = 'urn:' + paramDict['urn']
* # To Do make configuration file
* **def** ms\_ok(toClient):
* # MS\_OK message is used to inform the client about the presents of the DTT2IP / SAT>IP server on the network
* # or to inform the other DTT2IP / SAT>IP servers that M-SEARCH message has been correctly received
* myDate = date.today()
* currentTime = datetime.now().time()
* **if** myDate.day < 10:
* dateStr = calendar.day\_name[myDate.weekday()] + ' ' + str(myDate.weekday()) + ' ' + calendar.month\_name[myDate.month] + ' ' + str(myDate.year) + ' ' + currentTime.isoformat()[:-7] + ' ' + 'GMT'
* **else**:
* dateStr = calendar.day\_name[myDate.weekday()] + ' 0' + str(myDate.weekday()) + ' ' + calendar.month\_name[myDate.month] + ' ' + str(myDate.year) + ' ' + currentTime.isoformat()[:-7] + ' ' + 'GMT'
* **if** toClient:
* MS\_OK = 'HTTP/1.1 200 OK\r\nCACHE-CONTROL:max-age=%d\r\nDATE:%s\nEXT:\r\nLOCATION:http://%s:%d/%s\r\nSERVER:%s\nST:%s\nUSN:%s\nBOOTID.UPNP.ORG:%d\r\nCONFIGID.UPNP.ORG:%d\n\r\n' % (int(paramDict['cacheControl']), dateStr, serverIP, int(paramDict['httpPort']), paramDict['location'], paramDict['server'], st, USN[2], int(paramDict['bootId']), int(paramDict['configId']))
* **else**:
* MS\_OK = 'HTTP/1.1 200 OK\r\nCACHE-CONTROL:max-age=%d\r\nDATE:%s\nEXT:\r\nLOCATION:http://%s:%d/%s\r\nSERVER:%s\nST:%s\nUSN:%s\nBOOTID.UPNP.ORG:%d\r\nCONFIGID.UPNP.ORG:%d\r\nDEVICEID.SES.COM:%d\r\n\r\n' % (int(paramDict['cacheControl']), dateStr, serverIP, int(paramDict['httpPort']), paramDict['location'], paramDict['server'], st, USN[2], int(paramDict['bootId']), int(paramDict['configId']), int(paramDict['deviceId']))
* **return** MS\_OK
* **def** ms\_notify\_alive(nt, usn):
* # MS\_NOTIFY\_ALIVE is use to inform other DTT2IP / SAT>IP servers about our presents in the network. Used in the DEVICE ID negotiation
* MS\_NOTIFY\_ALIVE = 'NOTIFY \* HTTP/1.1\r\nHOST: %s:%d\r\nBOOTID.UPNP.ORG:%d\r\nCONFIGID.UPNP.ORG:%d\r\nCACHE-CONTROL: max-age=%d\r\nLOCATION:http://%s:%d/%s\r\nNT:%s\nNTS:%s\nSERVER:%s\nUSN:%s\nDEVICEID.SES.COM:%d\r\n\r\n' % (ssdpAddr, ssdpPort, int(paramDict['bootId']), int(paramDict['configId']), int(paramDict['cacheControl']), serverIP, int(paramDict['httpPort']), paramDict['location'], nt, 'ssdp:alive', paramDict['server'], usn, int(paramDict['deviceId']))
* **return** MS\_NOTIFY\_ALIVE
* **def** ms\_nofity\_byebye(nt, usn):
* # MS\_NOTIFY\_BYEBYE is use to infrom other servers that we are leaving the network
* MS\_NOTIFY\_BYEBYE = 'NOTIFY \* HTTP/1.1\r\nHOST: %s:%d\r\nNT:%s\nNTS:%s\nUSN:%s\nBOOTID.UPNP.ORG:%d\r\nCONFIGID.UPNP.ORG:%d\r\n\r\n' % (ssdpAddr, ssdpPort, nt, 'ssdp:byebye', usn, int(paramDict['bootId']), int(paramDict['configId']))
* **return** MS\_NOTIFY\_BYEBYE
* **def** ms\_search():
* # MS\_SEARCH is used to inform other DTT2IP / SAT>IP servers that the DEVICE ID that they want to use belongs to us. Used in the DEVICE ID negotiation
* MS\_SEARCH = 'M-SEARCH \* HTTP/1.1\r\nHOST:%s:%d\r\nMAN:"ssdp-discover"ST:%s\nUSER-AGENT:%s\nDEVICEID.SES.COM:%d' % (serverIP, ssdpPort, st, paramDict['server'], int(paramDict['deviceId']))
* **return** MS\_SEARCH
* **def** callServerReactor():
* **global** SSDP\_TERMINATE
* **global** fLog
* # Open a multicast socket
* fLog.write('Info: Discovery server started\n')
* ssdpMulticastSocket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)
* ssdpMulticastSocket.bind((ssdpAddr, ssdpPort))
* group = socket.inet\_aton(ssdpAddr)
* mreq = struct.pack('4sL', group, socket.INADDR\_ANY)
* ssdpMulticastSocket.setsockopt(socket.IPPROTO\_IP, socket.IP\_ADD\_MEMBERSHIP, mreq)
* ssdpMulticastSocket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)
* # While look until script is terminated
* **while** (**not** SSDP\_TERMINATE):
* # Wait to receive multicast messages either from client or from new servers that have joined the network
* datagram, address = ssdpMulticastSocket.recvfrom(1024)
* datagram\_array = datagram.rsplit('\r\n')
* **try**:
* first\_line = datagram\_array[0]
* # See if the multicast messages are from the clients/servers
* matchMSearch = re.search(r'M-SEARCH',first\_line)
* matchNotify = re.search(r'NOTIFY',first\_line)
* **if** matchMSearch:
* # If M-SEARCH message from client, then process it and respond to it with a unicast message
* match2 = re.search(r'ses-com',datagram)
* **if** match2:
* # fLog.write("Info: client = ip " + address[0] + ", port " + address[1] + "\n")
* fLog.write("Info: client \n")
* toClient = True
* ssdpMulticastSocket.sendto(ms\_ok(toClient), (address[0], address[1]))
* **if** matchNotify:
* # If NOTIFY message from server, then process it and see it is another DTT2IP / SAT>IP server
* matchSES = re.search(r'DEVICEID.SES.COM:([\w]+)',datagram)
* **if** matchSES:
* # If new DTT2IP / SAT>IP server that has join the network then respond to it with a unicast message
* # informing it that DEVICE ID is ours, else if this is an old DTT2IP / SAT>IP server do not do anything
* **if** int(matchSES.group(1)) == int(paramDict['deviceId']):
* ssdpMulticastSocket.sendto(ms\_search(), (address[0], address[1]))
* fLog.write("Info: MS\_SEARCH\n")
* **except**:
* fLog.write("Info: Something went wrong\n")
* ssdpMulticastSocket.close()
* **def** callClientReactor():
* **global** SSDP\_TERMINATE
* **global** deviceIdOk
* **global** fLog
* # Open a unicast socket
* fLog.write('Info: Device ID negotion started\n')
* ssdpUnicastSocket = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)
* ssdpUnicastSocket.bind((serverIP, ssdpPort))
* ssdpUnicastSocket.settimeout(5.0)
* # While look until script is terminated
* **while** (**not** SSDP\_TERMINATE):
* # DEVICE ID negotiation loop, escape only when we have valid DEVICE ID
* **while** (**not** deviceIdOk):
* # Send the first 3 SSDP NOTIFY messages
* **for** i **in** range(3):
* paramDict['bootId'] = int(paramDict['bootId']) + 1
* ssdpUnicastSocket.sendto(ms\_notify\_alive(NT[i], USN[i]), (ssdpAddr, ssdpPort))
* fLog.write("Info: MS\_NOTIFY\_ALIVE\n")
* **try**:
* # See if DEVICE ID is free, by waiting for a message or a timeout of 5 seconds
* datagram, address = ssdpUnicastSocket.recvfrom(1024)
* datagram\_array = datagram.rsplit('\r\n')
* **try**:
* # Process the message received
* first\_line = datagram\_array[0]
* matchMSearch = re.search(r'M-SEARCH',first\_line)
* **if** matchMSearch:
* # See if somebody else is using this ID
* matchSES = re.search(r'DEVICEID.SES.COM:([\w]+)',datagram)
* **if** matchSES:
* **if** int(matchSES.group(1)) == int(paramDict['deviceId']):
* # This DEVICE ID is used an we have to annouce that we are letting it go
* # by sending a MS\_OK message to the other server (i.e toClient = False)
* # and to the hole network MS\_NOTIFY\_BYEBYE mesasge.
* toClient = False
* ssdpUnicastSocket.sendto(ms\_ok(toClient), (address[0], address[1]))
* fLog.write("Info: MS\_OK\n")
* **for** i **in** range(3):
* ssdpUnicastSocket.sendto(ms\_nofity\_byebye(NT[i], USN[i]), (ssdpAddr, ssdpPort))
* fLog.write("Info: MS\_NOTIFY\_BYEBYE\n")
* # We have to increase the DEVICE ID
* paramDict['deviceId'] = int(paramDict['deviceId']) + 1
* **except**:
* fLog.write('Info: Something went wrong\n')
* **except**:
* # Change deviceIdOk to True only when we timeout (5.0 seconds)
* fLog.write("Info: DEVICE ID\n")
* deviceIdOk = True
* # We have obtain out valid DEVICE ID, we have to maintain it valid on the network by sending
* # at pseudo random periods 3 MS\_NOTIFY\_ALIVE messages. The pseudo random interval is between [0, cacheControl/2].
* # This guarantees that announcement set is repeated at least twice before it expires.
* fLog.write("Info: Device ID negotiation done. Sleep and send NOTIFY later\n")
* time.sleep(random.randint(0, int(paramDict['cacheControl'])/2))
* **for** i **in** range(3):
* paramDict['bootId'] = int(paramDict['bootId']) + 1
* ssdpUnicastSocket.sendto(ms\_notify\_alive(NT[i], USN[i]), (ssdpAddr, ssdpPort))
* fLog.write("Info: MS\_NOTIFY\_ALIVE\n")
* ssdpUnicastSocket.close()
* **def** discoveryServer():
* **global** fLog
* **global** deviceIdOk
* # DEVICE ID negotiation thread ( server <---> server communications )
* t1 = threading.Thread(target=callClientReactor)
* t1.daemon = True
* t1.start()
* # Wait for DEVICE negotiation to finish then start the Discovery protocol in order for the clients to connect to
* **while** **not** deviceIdOk:
* time.sleep(1)
* # DISCOVERY thread for the client applications ( server <---> client communications)
* t2 = threading.Thread(target=callServerReactor)
* t2.daemon = True
* t2.start()
* # Keep threads alive until KeyboardInterrupt
* **try**:
* **while** t1.is\_alive() **and** t2.is\_alive():
* t1.join(timeout=1.0)
* t2.join(timeout=1.0)
* **except** (KeyboardInterrupt, SystemExit):
* fLog.close()
* SSDP\_TERMINATE = 1

* **if** \_\_name\_\_ == "\_\_main\_\_":

    discoveryServer()

# Appendix C

* #!/usr/bin/python
* **import** sys, socket, signal, commands, re
* **from** rtspServerWorker **import** rtspServerWorker
* **from** netInterfaceStatus **import** getServerIP
* # class rtspServer:
* **def** clean():
* # Make sure that rtspServer.log file is clean
* fLog = open('logs/rtspServer.log', 'w')
* # Make sure that all dvblast sockets are deleted before you start the rtsp state machine
* cmd = 'rm -rf /tmp/dvblast\*'
* fLog.write('Info rtspServer: Cleaning dvblast sockets before starting\n')
* outtext = commands.getoutput(cmd)
* (exitstatus, outtext) = commands.getstatusoutput(cmd)
* # if not exitstatus:
* # fLog.write('Info rtspServer: Dvblast sockets clean\n')
* # Make sure that all the pidCfgFiles are clean before you start the rtsp state machine
* cmd = 'ls -l dvb-t/pid\*'
* fLog.write('Info rtspServer: Cleaning all pidCfgFiles\n')
* outtext = commands.getoutput(cmd)
* (exitstatus, outtext) = commands.getstatusoutput(cmd)
* **if** **not** exitstatus:
* linesArray = outtext.split('\n')
* **for** line **in** linesArray:
* matchPidCfgFile = re.search(r'pid([\w]+)', line)
* **if** matchPidCfgFile:
* f = open('dvb-t/pid' + matchPidCfgFile.group(1) + '.cfg', 'w')
* f.close()
* fLog.write('Info rtspServer: pidCfgFiles clean\n')
* fLog.close()
* # def main(self):
* **def** rtspServer():
* # Cleaning everything
* clean()
* # Make sure you have root privileges to run this script
* # it is necesary that we can open the "554" port
* serverPort = 554
* rtspSocket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
* ipAddrServer = getServerIP()
* rtspSocket.bind((ipAddrServer, serverPort))
* **while** (1):
* # Listen for incoming connections
* rtspSocket.listen(1)
* # Create the client profile and store necesary information
* clientInfo = {}
* clientInfo['rtspSocket'], addr = rtspSocket.accept()
* clientInfo['addr\_IP'] = addr[0]
* clientInfo['addr\_PORT'] = addr[1]
* # Run the rtspServerWorker for that specific client
* rtspServerWorker(clientInfo).run()
* rtspSocket.close()
* **if** \_\_name\_\_ == "\_\_main\_\_":
* # (rtspServer()).main()

    rtspServer()

# Appendix D

* #!/usr/bin/python
* **import** sys, traceback, threading, socket, signal, re, commands, os, time, string, random, uuid
* **from** random **import** randint
* **from** resources **import** getFrontEnds
* **from** netInterfaceStatus **import** getServerIP
* **from** scanning **import** getChList
* **from** subprocess **import** Popen, PIPE
* # global session
* # global dvblastReload
* **global** clientsDict
* **global** chList
* **global** frontEndsDict
* **global** freqDict
* **global** firstBootFlag
* **global** fLog
* # Init global variables
* clientsDict = {}    # e.g. clientsDict = { 'ip\_client\_1': {'rtpPort': '', state: 0, 'satFreq': '', stream: 0, 'src': '', 'pol': '', 'ro': '', 'msys': '', 'mtype': '', 'plts': '', 'sr': '', 'fec': '', 'status': 'sendonly', 'owner': True, 'session': '' }}
* freqDict = {}       # e.g. freqDict = {'freq': 'adapter0'}
* frontEndsDict = {}  # e.g. frontEndsDict = {'adapter0': {'owner': '0.0.0.0', 'freq': '', 'numOwners': 0}}
* chList = {}     # e.g. chList = {'satFreq': ['freq', 'pid']}
* # session = ''  # For THEARDOWN reply
* # dvblastReload = False   # e.g. Flag to trigger a reload
* firstBootFlag = True # First bootup of the device

* # Get chList, done only at the begining
* chList = getChList()
* # Make sure that rtspServerWorker.log file is clean
* f = open('logs/rtspServerWorker.log', 'w')
* f.close()
* **class** rtspServerWorker:
* # Events
* SETUP = 'SETUP'
* PLAY = 'PLAY'
* TEARDOWN = 'TEARDOWN'
* OPTIONS = 'OPTIONS'
* DESCRIBE = 'DESCRIBE'
* CLOSE\_CONNETION = 'CLOSE\_CONNETION'
* # State
* INI = 0
* READY = 1
* PLAYING = 2
* OK\_200\_OPTIONS = 0
* FILE\_NOT\_FOUND\_404 = 1
* CON\_ERR\_500 = 2
* CLOSING\_CONNECTION = 3
* OK\_404\_DESCRIBE = 4
* OK\_200\_DESCRIBE = 5
* OK\_200\_DESCRIBE\_NOSIGNAL = 6
* OK\_200\_SETUP = 7
* OK\_200\_SETUP\_PIDS = 8
* OK\_200\_PLAY = 9
* OK\_200\_TEARDOWN = 10
* SERVER\_RUNNING = 1
* clientInfo = {}
* **def** \_\_init\_\_(self, clientInfo):
* **global** clientsDict
* rtpPort = ''
* state = 0
* stream = 0
* src = ''
* freq = ''
* pol = ''
* ro = ''
* msys = ''
* mtype = ''
* plts = ''
* sr = ''
* fec = ''
* status = 'inactive'
* ownerCapabilties = True # Owner capabilties flag, in order to know if we can increase the number of owners of a tuner or not. If false that means that we are coowners, otherwise we are the only owner of a tuner or did not requested any at all.
* session = ''
* dvblastReload = False # DvblastReload flag, in order to know if we need to reload
* self.clientInfo = clientInfo
* **if** self.clientInfo['addr\_IP'] **not** **in** clientsDict:
* clientsDict[self.clientInfo['addr\_IP']] = {}
* clientsDict[self.clientInfo['addr\_IP']]['rtpPort'] = rtpPort
* clientsDict[self.clientInfo['addr\_IP']]['state'] = state
* clientsDict[self.clientInfo['addr\_IP']]['stream'] = stream
* clientsDict[self.clientInfo['addr\_IP']]['src'] = src
* clientsDict[self.clientInfo['addr\_IP']]['freq'] = freq
* clientsDict[self.clientInfo['addr\_IP']]['pol'] = pol
* clientsDict[self.clientInfo['addr\_IP']]['ro'] = ro
* clientsDict[self.clientInfo['addr\_IP']]['msys'] = msys
* clientsDict[self.clientInfo['addr\_IP']]['mtype'] = mtype
* clientsDict[self.clientInfo['addr\_IP']]['plts'] = plts
* clientsDict[self.clientInfo['addr\_IP']]['sr'] = sr
* clientsDict[self.clientInfo['addr\_IP']]['fec'] = fec
* clientsDict[self.clientInfo['addr\_IP']]['status'] = status
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = ownerCapabilties
* clientsDict[self.clientInfo['addr\_IP']]['session'] = session
* clientsDict[self.clientInfo['addr\_IP']]['dvblastReload'] = dvblastReload
* **def** run(self):
* t = threading.Thread(target=self.recvRtspRequest)
* t.daemon = True
* t.start()
* **try**:
* **while** t.is\_alive():
* t.join(timeout=1.0)
* **except** (KeyboardInterrupt, SystemExit):
* # fLog.close()
* self.SERVER\_RUNNING = 0

* **def** recvRtspRequest(self):
* """Receive RTSP request from the client."""
* connSocket = self.clientInfo['rtspSocket']
* **while** self.SERVER\_RUNNING:
* data = connSocket.recv(1024)
* **if** data:
* self.processRtspRequest(data)
* **def** processRtspRequest(self, data):
* """Process RTSP request sent from the client."""
* # global session
* # global dvblastReload
* **global** chList
* **global** clientsDict
* **global** frontEndsDict
* **global** freqDict
* **global** firstBootFlag
* **global** fLog
* # Initialize local variables
* freq = ''
* pids = ''
* delPids = 0
* delPid = 0
* fLog = open('logs/rtspServerWorker.log', 'a')
* # Get the request type
* request = data.split('\n')
* line1 = request[0].split(' ')
* requestType = line1[0]
* # Get the last part of the URI
* uriLastPart = line1[1]
* # Get the RTSP sequence number
* **for** seq\_find **in** request[1:]:
* # Word parsing for general URI request seq/client\_port
* match\_seq = re.search(r'CSeq', seq\_find)
* **if** match\_seq:
* # To do check the output
* seq = seq\_find.split(':')
* match\_client\_port = re.search(r'client\_port', seq\_find)
* **if** match\_client\_port:
* seq\_find\_array = seq\_find.split(';')
* self.clientInfo['rtpPort']= seq\_find\_array[2].split('=')[1].split('-')[0]
* clientsDict[self.clientInfo['addr\_IP']]['rtpPort'] = self.clientInfo['rtpPort']
* # Word parsing for SETUP/PLAY URI request
* **if** requestType == self.SETUP **or** requestType == self.PLAY:
* match\_pids = re.search(r'pids=([\w]+)', uriLastPart)
* **if** match\_pids:
* pids = match\_pids.group(1)
* match\_delpids = re.search(r'delpids=([\w]+)', uriLastPart)
* **if** match\_delpids:
* delPids = 1
* delPid = int(match\_delpids.group(1))
* # Process SETUP request
* **if** requestType == self.SETUP:
* # Word parsing for SETUP URI request
* match\_src = re.search(r'src=([\w]+)', uriLastPart)
* **if** match\_src:
* clientsDict[self.clientInfo['addr\_IP']]['src'] = match\_src.group(1)
* match\_freq = re.search(r'freq=([\w]+)', uriLastPart)
* **if** match\_freq:
* freq = match\_freq.group(1)
* clientsDict[self.clientInfo['addr\_IP']]['freq'] = freq
* match\_pol = re.search(r'pol=([\w]+)', uriLastPart)
* **if** match\_pol:
* clientsDict[self.clientInfo['addr\_IP']]['pol'] = match\_pol.group(1)
* match\_ro = re.search(r'ro=([\w]+...)', uriLastPart)
* **if** match\_ro:
* clientsDict[self.clientInfo['addr\_IP']]['ro'] = match\_ro.group(1)
* match\_msys = re.search(r'msys=([\w]+)', uriLastPart)
* **if** match\_msys:
* clientsDict[self.clientInfo['addr\_IP']]['msys'] = match\_msys.group(1)
* match\_mtype = re.search(r'mtype=([\w]+)', uriLastPart)
* **if** match\_mtype:
* clientsDict[self.clientInfo['addr\_IP']]['mtype'] = match\_mtype.group(1)
* match\_plts = re.search(r'plts=([\w]+)', uriLastPart)
* **if** match\_plts:
* clientsDict[self.clientInfo['addr\_IP']]['plts'] = match\_plts.group(1)
* match\_sr = re.search(r'sr=([\w]+)', uriLastPart)
* **if** match\_sr:
* clientsDict[self.clientInfo['addr\_IP']]['sr'] = match\_sr.group(1)
* match\_fec = re.search(r'fec=([\w]+)', uriLastPart)
* **if** match\_fec:
* clientsDict[self.clientInfo['addr\_IP']]['fec'] = match\_fec.group(1)
* clientsDict[self.clientInfo['addr\_IP']]['status'] = 'sendonly'
* # Process SETUP request If STATE is INI
* **if** clientsDict[self.clientInfo['addr\_IP']]['state'] == self.INI:
* fLog.write("Info rtspServerWorker: Processing SETUP, New State: READY\n")
* clientsDict[self.clientInfo['addr\_IP']]['state'] = self.READY
* # Generate a randomized RTSP session ID
* clientsDict[self.clientInfo['addr\_IP']]['session'] = uuid.uuid4().hex[:16]
* # Increment streamID for every new session
* clientsDict[self.clientInfo['addr\_IP']]['stream'] = (clientsDict[self.clientInfo['addr\_IP']]['stream'] + 1) % 65536
* # Send RTSP reply
* **if** freq **in** chList:
* f = open('dvb-t/pid' + chList[freq][0] + '.cfg', 'a')
* f.write(self.clientInfo['addr\_IP'] + ':' + clientsDict[self.clientInfo['addr\_IP']]['rtpPort'] + '\t1\t' + chList[freq][1] + '\n')
* f.close()
* clientsDict[self.clientInfo['addr\_IP']]['dvblastReload'] = True
* **if** pids == 'none' **or** pids == '':
* self.replyRtsp(self.OK\_200\_SETUP, seq[1])
* **if** pids == 'none':
* clientsDict[self.clientInfo['addr\_IP']]['status'] = 'inactive'
* **else**:
* self.replyRtsp(self.OK\_200\_SETUP\_PIDS, seq[1])
* # Process SETUP request If STATE is READY
* **elif** clientsDict[self.clientInfo['addr\_IP']]['state'] == self.READY:
* fLog.write("Info rtspServerWorker: Processing SETUP, State: READY\n")
* # Send RTSP reply
* self.replyRtsp(self.OK\_200\_SETUP, seq[1])
* # Process SETUP request If STATE is PLAYING
* **elif** clientsDict[self.clientInfo['addr\_IP']]['state'] == self.PLAYING:
* fLog.write("Info rtspServerWorker: Processing SETUP, State: PLAYING\n")
* **if** freq **in** chList:
* f = open('dvb-t/pid' + chList[freq][0] + '.cfg', 'r')
* lines = f.readlines()
* f.close()
* f = open('dvb-t/pid' + chList[freq][0] + '.cfg', 'w')
* lineToCompare = self.clientInfo['addr\_IP']
* **for** line **in** lines:
* match\_line = re.search(lineToCompare, line)
* **if** **not** match\_line:
* f.write(line)
* f.write(self.clientInfo['addr\_IP'] + ':' + clientsDict[self.clientInfo['addr\_IP']]['rtpPort'] + '\t1\t' + chList[freq][1] + '\n')
* f.close()
* clientsDict[self.clientInfo['addr\_IP']]['dvblastReload'] = True
* self.replyRtsp(self.OK\_200\_SETUP, seq[1])
* # Process PLAY request
* **elif** requestType == self.PLAY:
* # Get available frontends
* frontEndsDict = getFrontEnds(frontEndsDict)
* **if** clientsDict[self.clientInfo['addr\_IP']]['state'] == self.PLAYING **or** clientsDict[self.clientInfo['addr\_IP']]['state'] == self.READY:
* # Send response after processing and starting dvblast
* **if** clientsDict[self.clientInfo['addr\_IP']]['state'] == self.READY:
* fLog.write("Info rtspServerWorker: Processing PLAY, New State: PLAYING\n")
* clientsDict[self.clientInfo['addr\_IP']]['state'] = self.PLAYING
* **else**:
* # print "Info rtspServerWorker: Processing PLAY, State: PLAYING\n"
* fLog.write("Info rtspServerWorker: Processing PLAY, State: PLAYING\n")
* self.replyRtsp(self.OK\_200\_PLAY, seq[1])
* # START/RELOAD configuration for dvblast only if we have a streamID, the configuration file has been update and the PLAY URI is not a delete pid
* **if** clientsDict[self.clientInfo['addr\_IP']]['stream'] **and** clientsDict[self.clientInfo['addr\_IP']]['dvblastReload'] **and** delPids == 0:
* # If firstBootFlag is true this means that no frontend has been configured, so stop wasting time with searching for particular frontends
* **if** firstBootFlag:
* firstBootFlag = False
* **for** frontEnd **in** frontEndsDict:
* **if** frontEndsDict[frontEnd]['freq'] == '':
* # Start dvblast on specified freq
* cmd = 'dvblast -a ' + frontEnd[-1] + ' -c dvb-t/pid' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + '.cfg -f ' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + ' -b 8 -C -u -r /tmp/dvblast' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + frontEnd + '.sock'
* fLog.write('Info rtspServerWorker: Starting dvblast 1\n')
* self.run\_dvblast(cmd)
* clientsDict[self.clientInfo['addr\_IP']]['dvblastReload'] = False
* frontEndsDict[frontEnd]['freq'] = chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]
* frontEndsDict[frontEnd]['owner'] = self.clientInfo['addr\_IP']
* frontEndsDict[frontEnd]['numOwners'] = frontEndsDict[frontEnd]['numOwners'] + 1 # increase the number of owners
* freqDict[frontEndsDict[frontEnd]['freq']] = frontEnd
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = True
* **print** " frontEndsDict0", frontEndsDict
* **break**
* # No more first boot, the search for available and preconfigured/unconfigured frontends
* **else**:
* **print** " frontEndsDict1", frontEndsDict
* **print** " clientsDict", clientsDict
* # Search for any configured tuner with the frequency that we want to tune to
* **for** frontEnd **in** frontEndsDict:
* **if** frontEndsDict[frontEnd]['freq'] == chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]:
* cmd = 'dvblastctl -r /tmp/dvblast' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + frontEnd + '.sock reload'
* fLog.write('Info rtspServerWorker: Reloading dvblast configuration 1\n')
* self.run\_dvblast(cmd)
* clientsDict[self.clientInfo['addr\_IP']]['dvblastReload'] = False
* # Check if this was an abandoned tuner ('0.0.0.0' abandonded IP address) and take ownership if it was
* **if** frontEndsDict[frontEnd]['owner'] == '0.0.0.0':
* frontEndsDict[frontEnd]['owner'] = self.clientInfo['addr\_IP']
* frontEndsDict[frontEnd]['numOwners'] = frontEndsDict[frontEnd]['numOwners'] + 1 # increase the number of owner
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = True
* **print** "Info Alex --------------- 1"
* # Check if multiple owners ('255.255.255.255' broadcast IP address) and you have ownership capabilities,
* # then increase the number of owners by one and remove your ownership capabilities.
* **if** frontEndsDict[frontEnd]['owner'] == '255.255.255.255' **and** clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties']:
* frontEndsDict[frontEnd]['numOwners'] = frontEndsDict[frontEnd]['numOwners'] + 1 # increase the number of owner
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = False # remove the ownership capability from the client
* **print** "Info Alex --------------- 2"
* # Check if somebody else owne's it and you have ownership capabilities,
* # then make the tuner for multiple owners, increase the number of owners and remove your ownership capabilities
* **elif** frontEndsDict[frontEnd]['owner'] != self.clientInfo['addr\_IP'] **and** clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties']:
* clientsDict[frontEndsDict[frontEnd]['owner']]['owner'] = False # Remove the ownership capabilities of the previous owner
* frontEndsDict[frontEnd]['owner'] = '255.255.255.255'  # '255.255.255.255' the IP address for specifying multiple owners
* frontEndsDict[frontEnd]['numOwners'] = frontEndsDict[frontEnd]['numOwners'] + 1 # increase the number of owner
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = False # Remove the ownership capabilties of yourself
* **print** "Info Alex --------------- 3"
* **print** " frontEndsDict2", frontEndsDict
* **break**
* # If we did not find any tuner that has that frequency configured,then search for any owned tuners
* **if** clientsDict[self.clientInfo['addr\_IP']]['dvblastReload']:
* **for** frontEnd **in** frontEndsDict:
* **if** frontEndsDict[frontEnd]['owner'] == self.clientInfo['addr\_IP'] **or** frontEndsDict[frontEnd]['owner'] == '0.0.0.0':
* # Shutdown socket if used
* **try**:
* cmd = 'dvblastctl -r /tmp/dvblast' + frontEndsDict[frontEnd]['freq'] + frontEnd + '.sock shutdown'
* fLog.write("Info rtspServerWorker: Shutting down dvblast for this tuner\n")
* self.run\_dvblast(cmd)
* # print 'ALEX --- 6'
* # ALEX : ----- To be checked
* time.sleep(1)
* # Alex : -----
* **except**:
* **print** "Info rtspServerWorker: Tried to shut down tuner, however the tuner was not used"
* # Cleaning dvblast sockets is any not removed, before creating any other
* **try**:
* cmdClean = 'rm -rf /tmp/dvblast' + frontEndsDict[frontEnd]['freq'] + frontEnd + '.sock'
* fLog.write("Info rtspServerWorker: Cleaning dvblast sockets, before restarting\n")
* os.system(cmdClean)
* **except**:
* fLog.write("Info rtspServerWorker: No Cleaning for dvblast sockets, before restarting\n")
* # Start dvblast on specified freq
* cmd = 'dvblast -a ' + frontEnd[-1] + ' -c dvb-t/pid' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + '.cfg -f ' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + ' -b 8 -C -u -r /tmp/dvblast' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + frontEnd + '.sock'
* fLog.write('Info rtspServerWorker: Starting dvblast 2\n')
* self.run\_dvblast(cmd)
* clientsDict[self.clientInfo['addr\_IP']]['dvblastReload'] = False
* # Assume ownership
* **if** frontEndsDict[frontEnd]['owner'] == '0.0.0.0':
* frontEndsDict[frontEnd]['owner'] = self.clientInfo['addr\_IP']
* frontEndsDict[frontEnd]['numOwners'] = frontEndsDict[frontEnd]['numOwners'] + 1 # increase the number of owners
* frontEndsDict[frontEnd]['freq'] = chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]
* freqDict[frontEndsDict[frontEnd]['freq']] = frontEnd
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = True
* **print** " frontEndsDict3", frontEndsDict
* **break**
* # If we did not fine any owned tuners, then give it one more search before giving up. Search for nonused available tuners.
* **if** clientsDict[self.clientInfo['addr\_IP']]['dvblastReload']:
* **for** frontEnd **in** frontEndsDict:
* **if** frontEndsDict[frontEnd]['freq'] == '':
* # Start dvblast on specified freq
* cmd = 'dvblast -a ' + frontEnd[-1] + ' -c dvb-t/pid' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + '.cfg -f ' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + ' -b 8 -C -u -r /tmp/dvblast' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + frontEnd + '.sock'
* clientsDict[self.clientInfo['addr\_IP']]['dvblastReload']= False
* fLog.write('Info rtspServerWorker: Starting dvblast 3\n')
* self.run\_dvblast(cmd)
* frontEndsDict[frontEnd]['freq'] = chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]
* frontEndsDict[frontEnd]['owner'] = self.clientInfo['addr\_IP']
* frontEndsDict[frontEnd]['numOwners'] = frontEndsDict[frontEnd]['numOwners'] + 1 # increase the number of owner
* freqDict[frontEndsDict[frontEnd]['freq']] = frontEnd
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = True
* **print** " frontEndsDict4", frontEndsDict
* **break**
* # Remove corresponding pid from config file and reload for sat>ip app
* **if** clientsDict[self.clientInfo['addr\_IP']]['stream'] **and** delPids **and** delPid:
* **try**:
* f = open('dvb-t/pid' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + '.cfg', 'r')
* lines = f.readlines()
* f.close()
* f = open('dvb-t/pid' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + '.cfg', 'w')
* lineToCompare = self.clientInfo['addr\_IP']
* **for** line **in** lines:
* match\_line = re.search(lineToCompare, line)
* **if** **not** match\_line:
* f.write(line)
* lineToGet = line.split('\t')
* **print** "Info rtspServerWorker: lineToGet", lineToGet
* f.close()
* cmd = 'dvblastctl -r /tmp/dvblast' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]] + '.sock reload'
* fLog.write('Info rtspServerWorker: Reloading dvblast configuration 3\n')
* self.run\_dvblast(cmd)
* **try**:
* # Update the number of owners of frontEnd
* **if** (frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['owner'] == '255.255.255.255' **or** frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['owner'] == self.clientInfo['addr\_IP']) **and** frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] >= 1:
* frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] = frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] - 1
* clientsDict[self.clientInfo['addr\_IP']]['ownerCapabilties'] = True # Make sure that the client has ownership capabilties
* **except**:
* **print** "Info rtspServerWorker: No adapters configured with that freq 3 "
* **try**:
* # If we have only one client connected left, then the last client takes the ownership of the tuner
* **if** frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] == 1 **and** frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['owner'] == '255.255.255.255' :
* frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['owner'] = lineToGet[0][:-(len(clientsDict[self.clientInfo['addr\_IP']]['rtpPort'])+1)] # The last client that will remain in the clientsDict has to take ownership (lineToGet[0] = ip\_add:port\_num)
* **print** " frontEndsDict8", frontEndsDict
* **except**:
* **print** "Info rtspServerWorker: No adapters configured with that freq 4 "
* **print** " frontEndsDict9", frontEndsDict
* **except**:
* # print "Info rtspServerWorker: Processing PLAY DELETE PIDS\n"
* fLog.write("Info rtspServerWorker: Processing PLAY DELETE PIDS\n")
* # Process TEARDOWN request
* **elif** requestType == self.TEARDOWN:
* fLog.write("Info rtspServerWorker: Processing TEARDOWN, New State: INI\n")
* **try**:
* f = open('dvb-t/pid' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + '.cfg', 'r')
* lines = f.readlines()
* f.close()
* f = open('dvb-t/pid' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + '.cfg', 'w')
* lineToCompare = self.clientInfo['addr\_IP']
* **for** line **in** lines:
* match\_line = re.search(lineToCompare, line)
* **if** **not** match\_line:
* f.write(line)
* f.close()
* cmd = 'dvblastctl -r /tmp/dvblast' + chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0] + freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]] + '.sock reload'
* fLog.write('Info rtspServerWorker: Reloading dvblast configuration 4\n')
* self.run\_dvblast(cmd)
* **except**:
* # print "Info rtspServerWorker: processing TEARDOWN NONE\n"
* **print** " frontEndsDict7", frontEndsDict
* fLog.write("Info rtspServerWorker: processing TEARDOWN NONE\n")
* **try**:
* # Update the number of owners of frontEnd
* **if** frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] >= 1:
* frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] = frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] - 1
* **except**:
* **print** "Info rtspServerWorker: No adapters configured with that freq 1 "
* **try**:
* # If we have only one client connected left, then the last client takes the ownership of the tuner
* **if** frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['numOwners'] == 1:
* **if** clientsDict.keys()[0] == self.clientInfo['addr\_IP']:
* frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['owner'] = clientsDict.keys()[1] # The last client that will remain in the clientsDict has to take ownership
* **else**:
* frontEndsDict[freqDict[chList[clientsDict[self.clientInfo['addr\_IP']]['freq']][0]]]['owner'] = clientsDict.keys()[0] # The last client that will remain in the clientsDict has to take ownership
* **print** " frontEndsDict6", frontEndsDict
* **except**:
* **print** "Info rtspServerWorker: No adapters configured with that freq 2 "
* # Get the session value before deleting the client entry, for replay purpose
* # session = clientsDict[self.clientInfo['addr\_IP']]['session']
* # Remove client from dictinary
* # del clientsDict[self.clientInfo['addr\_IP']]
* self.replyRtsp(self.OK\_200\_TEARDOWN, seq[1])
* # Process OPTIONS request
* **elif** requestType == self.OPTIONS:
* fLog.write("Info rtspServerWorker: Processing OPTIONS\n")
* self.replyRtsp(self.OK\_200\_OPTIONS, seq[1])
* # Process DESCRIBE request
* **elif** requestType == self.DESCRIBE:
* **if** clientsDict[self.clientInfo['addr\_IP']]['session'] == '':
* fLog.write("Info rtspServerWorker: Processing DESCRIBE NONE\n")
* self.replyRtsp(self.OK\_404\_DESCRIBE, seq[1])
* **else**:
* **if** clientsDict[self.clientInfo['addr\_IP']]['dvblastReload']:
* fLog.write("Info rtspServerWorker: Processing DESCRIBE SIGNAL\n")
* self.replyRtsp(self.OK\_200\_DESCRIBE, seq[1])
* **else**:
* fLog.write("Info rtspServerWorker: Processing DESCRIBE NO SIGNAL\n")
* self.replyRtsp(self.OK\_200\_DESCRIBE\_NOSIGNAL, seq[1])
* # Process CLOSE\_CONNETION request
* **elif** requestType == self.CLOSE\_CONNETION:
* fLog.write("Info rtspServerWorker: Processing CLOSE\_CONNETION\n")
* self.SERVER\_RUNNING = 0
* self.replyRtsp(self.CLOSING\_CONNECTION, seq[1])
* fLog.close()
* **def** run\_dvblast(self, cmd):
* **global** fLog
* proc = Popen([cmd], stdout=fLog, stderr=fLog, shell=True)
* **def** updateChList():
* **global** chList
* # Get chList
* chList = getChList()
* **def** replyRtsp(self, code, seq):
* """Send RTSP reply to the client."""
* # global session
* **global** fLog
* **if** code == self.OK\_200\_OPTIONS:
* reply = 'RTSP/1.0 200 OK\r\nPublic:OPTIONS,SETUP,PLAY,TEARDOWN,DESCRIBE\r\nCSeq:1\r\n\r\n'
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* self.SERVER\_RUNNING = 0
* # Error messages
* **elif** code == self.FILE\_NOT\_FOUND\_404:
* **print** "Info rtspServerWorker: 404 NOT FOUND\n"
* fLog.write("Info rtspServerWorker: 404 NOT FOUND\n")
* **elif** code == self.CON\_ERR\_500:
* **print** "Info rtspServerWorker: 500 CONNECTION ERROR\n"
* fLog.write("Info rtspServerWorker: 500 CONNECTION ERROR\n")
* **elif** code == self.OK\_200\_DESCRIBE:
* ipServer = getServerIP()
* unicastIp = '0.0.0.0'
* serverID = 1
* serverTunerNr = len(frontEndsDict)
* tunerValues = '1,123,1,3,'
* tunerValues2 = ',' + clientsDict[self.clientInfo['addr\_IP']]['pol'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['msys'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['mtype'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['plts'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['ro'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['sr'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['fec']
* sdpString = 'v=0\r\no=- 534863118 534863118 IN IP4 %s\ns=SatIPServer:%d %d\r\nt=0 0\r\nm=video 0 RTP/AVP 33\r\nc=IN IP4 %s\na=control:stream=%d\na=fmtp:33 ver=1.0;scr=1;tuner=%s%s.00%s\na=%s\n' % (ipServer, serverID, serverTunerNr, unicastIp, clientsDict[self.clientInfo['addr\_IP']]['stream'], tunerValues, clientsDict[self.clientInfo['addr\_IP']]['freq'], tunerValues2, clientsDict[self.clientInfo['addr\_IP']]['status'])
* sdpLen = len(sdpString)
* rtspString = 'RTSP/1.0 200 OK\r\nContent-length:%d\r\nContent-type:application/sdp\r\nContent-Base:rtsp://192.168.2.61/\r\nCSeq:%s\nSession:%s\r\n\r\n' % (sdpLen ,seq, clientsDict[self.clientInfo['addr\_IP']]['session'])
* # Make the reply from the two parts: rtspString and sdpString
* reply = rtspString + sdpString
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* self.SERVER\_RUNNING = 0
* fLog.write("Info rtspServerWorker: 200 DESCRIBE\n")
* **elif** code == self.OK\_200\_DESCRIBE\_NOSIGNAL:
* ipServer = getServerIP()
* unicastIp = '0.0.0.0'
* serverID = 1
* serverTunerNr = len(frontEndsDict)
* tunerValues = '1,0,0,0,'
* tunerValues2 = ',' + clientsDict[self.clientInfo['addr\_IP']]['pol'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['msys'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['mtype'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['plts'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['ro'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['sr'] + ',' + clientsDict[self.clientInfo['addr\_IP']]['fec']
* sdpString = 'v=0\r\no=- 534863118 534863118 IN IP4 %s\ns=SatIPServer:%d %d\r\nt=0 0\r\nm=video 0 RTP/AVP 33\r\nc=IN IP4 %s\na=control:stream=%d\na=fmtp:33 ver=1.0;scr=1;tuner=%s%s.00%s\na=%s\n' % (ipServer, serverID, serverTunerNr, unicastIp, clientsDict[self.clientInfo['addr\_IP']]['stream'], tunerValues, clientsDict[self.clientInfo['addr\_IP']]['freq'], tunerValues2, clientsDict[self.clientInfo['addr\_IP']]['status'])
* sdpLen = len(sdpString)
* rtspString = 'RTSP/1.0 200 OK\r\nContent-length:%d\r\nContent-type:application/sdp\r\nContent-Base:rtsp://192.168.2.61/\r\nCSeq:%s\nSession:%s\r\n\r\n' % (sdpLen ,seq, clientsDict[self.clientInfo['addr\_IP']]['session'])
* # Make the reply from the two parts: rtspString and sdpString
* reply = rtspString + sdpString
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* fLog.write("Info rtspServerWorker: 200 DESCRIBE NOSIGNAL\n")
* self.SERVER\_RUNNING = 0
* **elif** code == self.OK\_404\_DESCRIBE:
* reply = 'RTSP/1.0 404 Not Found\r\nCSeq:%s\n\r\n' % (seq)
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* fLog.write("Info rtspServerWorker: 404 DESCRIBE\n")
* self.SERVER\_RUNNING = 0
* **elif** code == self.OK\_200\_SETUP:
* reply = 'RTSP/1.0 200 OK\r\nSession:%s;timeout=30\r\ncom.ses.streamID:%d\r\nTransport: RTP/AVP;unicast;destination=%s;client\_port=5000-5001\r\nCSeq:%s\n\r\n' % (clientsDict[self.clientInfo['addr\_IP']]['session'], clientsDict[self.clientInfo['addr\_IP']]['stream'], self.clientInfo['addr\_IP'], seq)
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* fLog.write("Info rtspServerWorker: 200 SETUP\n")
* self.SERVER\_RUNNING = 0
* **elif** code == self.OK\_200\_SETUP\_PIDS:
* reply = 'RTSP/1.0 200 OK\r\nSession:%s;timeout=30\r\ncom.ses.streamID:%d\r\nTransport: RTP/AVP;unicast;destination=%s;client\_port=5000-5001\r\nCSeq:%s\r\n\r\n' % (clientsDict[self.clientInfo['addr\_IP']]['session'], clientsDict[self.clientInfo['addr\_IP']]['stream'], self.clientInfo['addr\_IP'], seq)
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* fLog.write("Info rtspServerWorker: 200 SETUP PID\n")
* self.SERVER\_RUNNING = 1
* **elif** code == self.OK\_200\_PLAY:
* reply = 'RTSP/1.0 200 OK\r\nRTP-Info:url=//192.168.2.61/stream=%d;seq=50230\r\nCSeq:%s\nSession:%s\r\n\r\n' % (clientsDict[self.clientInfo['addr\_IP']]['stream'], seq, clientsDict[self.clientInfo['addr\_IP']]['session'])
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* fLog.write("Info rtspServerWorker: 200 PLAY\n")
* self.SERVER\_RUNNING = 0
* **elif** code == self.OK\_200\_TEARDOWN:
* reply = 'RTSP/1.0 200 OK\r\nContent-length:0\r\nCSeq:%s\nSession:%s\r\n\r\n' % (seq, clientsDict[self.clientInfo['addr\_IP']]['session'])
* connSocket = self.clientInfo['rtspSocket']
* connSocket.send(reply)
* # Remove client from dictinary
* **del** clientsDict[self.clientInfo['addr\_IP']]
* fLog.write("Info rtspServerWorker: 200 TEARDOWN\n")

            self.SERVER\_RUNNING = 0

# Appendix E

* #!/usr/bin/python
* **import** commands, re, os.path, time
* **global** numIter
* **global** valTimer
* **global** valTimerCheck
* **global** fLog
* # Initialize global variables
* numIter = 0
* valTimer = 3
* valTimerCheck = 4
* # Make sure that "scanning.log" file is clean
* fLog = open('logs/scanning.log', 'w')
* fLog.close()
* **def** getChList():
* **global** fLog
* fLog = open('logs/scanning.log', 'a')
* # e.g. chList = {'satFreq': ['freq', 'pid']}
* # Initialize chList
* chList = {}
* f = open('conf/rtspServer.config', 'r')
* lines = f.readlines()
* **for** i **in** range(5, len(lines)):
* line = lines[i]
* lineArray = line.split(' ')
* **if** lineArray[0] != '#':
* chList[lineArray[0]] = lineArray[1:-1]
* f.close()
* fLog.write('Info: Available channels obtained\n')
* fLog.close()
* **return** chList
* **def** scanning(periodNewScan, scanningFlag):
* **global** numIter
* **global** valTimer
* **global** valTimerCheck
* **global** fLog
* # Increment number of iteration for scanning
* numIter = numIter + 1
* fLog = open('logs/scanning.log', 'a')
* # Make sure that rtspServer.config file is clean
* f = open('conf/rtspServer.config', 'w')
* f.write('# Please be carefull when editing this file. \n')
* f.write('# The syntax is :\n')
* f.write('# fakeFrequency' 'dvbtFrequency' 'bandwidth' 'modulationType' 'pid' ' (for the moment no bandwith or modulation)\n')
* f.write('# Use "#' '" to comment line\n')
* f.write('\n')
* f.close()
* # Statically asigned frequencies from 19.2 degrees E satelite
* # First value signifies that freq is free "0" or occupied "1"
* # Second value signifies the timer that the freq is still available.
* # Starts from valTimer and gets decremented everytime and gets incremented mod valTimer
* # if freq is found on polls different from valTimerCheck, except the first time.
* # If timer gets to "0", then remove that frequency from the rtspServer.config file and
* # make the corresponding satFreq available and reinitialize valTimer.
* satFreq = { '10729':[0,valTimer], '10743':[0,valTimer], '10773':[0,valTimer], '10788':[0,valTimer], '10818':[0,valTimer], '10832':[0,valTimer],
* '10847':[0,valTimer], '10862':[0,valTimer], '10876':[0,valTimer], '10979':[0,valTimer], '11023':[0,valTimer], '11038':[0,valTimer],
* '11097':[0,valTimer], '11156':[0,valTimer], '11171':[0,valTimer], '11303':[0,valTimer], '11318':[0,valTimer], '11362':[0,valTimer],
* '11436':[0,valTimer], '11464':[0,valTimer], '11479':[0,valTimer], '11509':[0,valTimer], '11538':[0,valTimer], '11568':[0,valTimer],
* '11597':[0,valTimer], '11627':[0,valTimer], '11671':[0,valTimer], '11686':[0,valTimer], '11720':[0,valTimer], '11739':[0,valTimer],
* '11758':[0,valTimer], '11778':[0,valTimer], '11798':[0,valTimer], '11817':[0,valTimer], '11836':[0,valTimer], '11856':[0,valTimer],
* '11876':[0,valTimer], '11895':[0,valTimer], '11914':[0,valTimer], '11934':[0,valTimer], '11954':[0,valTimer], '11973':[0,valTimer],
* '11992':[0,valTimer], '12012':[0,valTimer], '12032':[0,valTimer], '12051':[0,valTimer], '12070':[0,valTimer], '12090':[0,valTimer],
* '12110':[0,valTimer], '12129':[0,valTimer], '12148':[0,valTimer], '12168':[0,valTimer], '12188':[0,valTimer], '12207':[0,valTimer],
* '12226':[0,valTimer], '12246':[0,valTimer], '12266':[0,valTimer], '12285':[0,valTimer], '12304':[0,valTimer], '12324':[0,valTimer],
* '12344':[0,valTimer], '12363':[0,valTimer], '12382':[0,valTimer], '12402':[0,valTimer], '12422':[0,valTimer], '12441':[0,valTimer],
* '12460':[0,valTimer], '12480':[0,valTimer], '12515':[0,valTimer], '12545':[0,valTimer], '12552':[0,valTimer], '12574':[0,valTimer],
* '12581':[0,valTimer], '12604':[0,valTimer], '12610':[0,valTimer], '12633':[0,valTimer], '12640':[0,valTimer], '12663':[0,valTimer],
* '12670':[0,valTimer], '12692':[0,valTimer], '12699':[0,valTimer], '12722':[0,valTimer], '12728':[0,valTimer] }
* **while** scanningFlag < 4:
* # cmd = 'w\_scan > dvb-t/allFrequencies.txt'
* fLog.write('Info: Scaning all available frequencies from your antenna\n')
* # outtext = commands.getoutput(cmd)
* # (exitstatus, outtext) = commands.getstatusoutput(cmd)
* # if not exitstatus:
* f = open('dvb-t/allFrequencies.txt', 'r')
* lines = f.readlines()
* f.close()
* **for** line **in** lines:
* # Search for the frequencies available from the scan
* matchFreq = re.search(r':([\w]+)', line)
* **if** matchFreq:
* freq = matchFreq.group(1) + '000'
* line = line[::-1]
* # Search for the PID's corresponding to the frequencies detected
* matchPid = re.search(r'([\w]+):([\w]+):([\w]+):([\w]+):', line)
* **if** matchPid:
* pid = matchPid.group(4)[::-1]
* # Create all the necesary '.cfg' files
* **if** **not** os.path.isfile('dvb-t/pid' + freq + '.cfg'):
* cmd = 'touch dvb-t/pid' + freq + '.cfg'
* **print** 'Info: About to do this = ', cmd
* outtext = commands.getoutput(cmd)
* (exitstatus, outtext) = commands.getstatusoutput(cmd)
* **if** **not** exitstatus:
* **print** 'Info: Creating missing pid' + freq + '.cfg file'
* fLog.write('Info: Creating missing pid' + freq + '.cfg file\n')
* **if** numIter == 1:
* **print** "Info: first iteration"
* fLog.write('Info: Update rtspServer.config with new freq= ' + freq + ' and pid= ' + pid + ' found\n')
* **for** currentFreq **in** sorted(satFreq):
* **if** satFreq[currentFreq][0] == 0:
* satFreq[currentFreq][0] = 1
* f = open('conf/rtspServer.config', 'a')
* f.write(currentFreq + ' ' + freq + ' ' + pid + ' \n')
* f.close()
* **break**
* # Alex TO DO -- what if we ran out of available sat freq
* **else**:
* **print** "Info: second/other iteration"
* newFreqFlag = True
* fLog.write('Info: Update timer for freq= ' + freq + ' and pid= ' + pid + ' \n')
* **for** freqPid **in** chDict.values():
* **if** freq == freqPid[0] **and** pid == freqPid[1]:
* newFreqFlag = False
* # Increase timer (i.e second value from satFreq) mod valTimer
* satFreq[chDict.keys()[chDict.values().index([freq, pid])]][1] = (satFreq[chDict.keys()[chDict.values().index([freq, pid])]][1] + 1) % (valTimer + 2)
* **break**
* **if** newFreqFlag:
* **print** "Info: Update rtspServer.config with new freq = " + freq + ' found'
* fLog.write('Info: Update rtspServer.config with new freq = ' + freq + ' found.')
* **for** currentFreq **in** sorted(satFreq):
* **if** satFreq[currentFreq][0] == 0:
* satFreq[currentFreq][0] = 1
* # satFreq[currentFreq][1] = satFreq[currentFreq][1] - 1
* f = open('conf/rtspServer.config', 'a')
* f.write(currentFreq + ' ' + freq + ' ' + pid + ' \n')
* f.close()
* **break**
* # Alex TO DO -- what if we ran out of available sat freq
* # Every iteration decrease valTimer
* **if** numIter > 1:
* **for** freqPid **in** chDict.values():
* satFreq[chDict.keys()[chDict.values().index([freqPid[0], freqPid[1]])]][1] = satFreq[chDict.keys()[chDict.values().index([freqPid[0], freqPid[1]])]][1] - 1
* # Clean unavailable frequencies every period (multiples of valTimerCheck)
* **if** numIter % valTimerCheck == 0:
* **print** "Info: clean unavailable frequencies"
* fLog.write('Info: clean unavailable frequencies\n')
* **print** "satFreq", satFreq
* **for** currentFreq **in** sorted(satFreq):
* **if** satFreq[currentFreq][1] == 0:
* f = open('conf/rtspServer.config', 'r')
* lines = f.readlines()
* f.close()
* f = open('conf/rtspServer.config', 'w')
* **for** line **in** lines:
* matchFreq = re.search(currentFreq, line)
* **if** **not** matchFreq:
* f.write(line)
* f.close()
* satFreq[currentFreq][1] = valTimer
* numIter = numIter + 1
* scanningFlag = scanningFlag + 1
* fLog.write('Info: W\_SCAN has finished. All configuration files have been update\n')
* chDict = getChList()
* # Check every hour for new frequencies. It can be changed to longer periods from site
* fLog = open('logs/scanning.log', 'a')
* fLog.write('Info: W\_SCAN is enteringn sleeping mode. Will wake up in ' + str(periodNewScan) + ' seconds\n')
* time.sleep(periodNewScan)
* fLog.close()
* **if** \_\_name\_\_ == '\_\_main\_\_':
* # Default period for new scan is 3600 seconds.
* # It can be changed from the site interface
* periodNewScan = 10
* scanningFlag = 0

    scanning(periodNewScan, scanningFlag)

# Appendix F

* #!/usr/bin/python
* **import** commands, re
* # Make sure that resources.log file is clean
* fLog = open('logs/resources.log', 'w')
* fLog.close()
* **def** getFrontEnds(frontEndDict):
* # e.g. frontEndDict = {'adapter0': {'owner': '0.0.0.0', 'freq': '', 'numOwners': 0, 'valid': True}}
* # Initialize frontEnd dictionary
* # frontEndDict = {}
* fLog = open('logs/resources.log', 'a')
* # Update the frontEndDict values (i.e number of owners, owners)
* **if** frontEndDict:
* **for** frontEnd **in** frontEndDict:
* **if** frontEndDict[frontEnd]['numOwners'] == 0:
* frontEndDict[frontEnd]['owner'] = '0.0.0.0'
* # List the available adapters
* cmd = 'ls -l /dev/dvb/'
* fLog.write('Info: about to do find all available adapters\n')
* outtext = commands.getoutput(cmd)
* (exitstatus, outtext) = commands.getstatusoutput(cmd)
* **if** **not** exitstatus:
* linesArray = outtext.split('\n')
* # Verify is adapter are still valid, and not removed.
* **for** frontEnd **in** frontEndDict:
* frontEndDict[frontEnd]['valid'] = False
* # Update the adapters
* **for** line **in** linesArray:
* matchAdapter = re.search(r'adapter([\w]+)', line)
* **if** matchAdapter:
* adapter = 'adapter' + matchAdapter.group(1)
* **if** **not** frontEndDict.has\_key(adapter):
* # print "Info resources: new adapter"
* frontEndDict[adapter] = {}
* frontEndDict[adapter]['owner'] = '0.0.0.0'
* frontEndDict[adapter]['freq'] = ''
* frontEndDict[adapter]['numOwners'] = 0
* frontEndDict[adapter]['valid'] = True
* fLog.write('Info: Available ' + adapter + ' detected\n')
* **else**:
* frontEndDict[adapter]['valid'] = True
* # Remove not valid adapters from the dictionary
* **for** frontEnd **in** frontEndDict:
* **if** frontEndDict[frontEnd]['valid'] == False:
* **del** frontEndDict[frontEnd]
* **else**:
* frontEndDict = {}
* fLog.write('Info: NO AVAILABLE adapters detected\n')
* fLog.close()
* **return** frontEndDict
* **if** \_\_name\_\_ == '\_\_main\_\_':
* frontEndDict = {}
* frontEndDict = getFrontEnds(frontEndDict)

**print** frontEndDict

# Appendix G

* #!/usr/bin/python
* **import** netifaces
* # import Error

* **def** getServerIP():
* ipAddrServer = '127.0.0.1'
* ipAddrArray = getNetworkInt()
* **for** ipAddr **in** ipAddrArray:
* **if** ipAddr["name"] != "lo" **and** ipAddr["status"] == "up":
* ipAddrServer = ipAddr["ip"]
* # print "Info netInterfaceStatus: ipAddrServer = " + ipAddrServer + '\n'
* **return** ipAddrServer
* **def** getNetworkInt():
* '''''
* getNetworkInt() function that checks the all network interfaces on the machine.
* This will return array vector with all the available interfaces.
* If not available interface (all interfaces are down) will return,
* The last interface found with loop ip address (127.0.0.1) with status "down".
* e.g. :
* if ok :
* [{'status': 'up', 'ip': '10.50.219.250', 'mac': '3c:15:c2:e2:97:a0', 'name': 'en0'}]
* else :
* [{'status': 'down', 'ip': '127.0.0.1', 'mac': '42:01:38:ad:79:a2', 'name': 'awdl0'}]
* '''
* networkServer = []
* netInts = netifaces.interfaces()
* **for** netInt **in** netInts:
* netIntDesc = netifaces.ifaddresses(netInt)
* # print "netIntName", netInt
* networkServerDic = {}
* #Description for each number
* # better use the netifaces.AF\_LINK instead of 18, it may change over time
* # 18 --- is the AF\_LINK (which means the link layer interface, e.g. Ethernet)
* #  2 --- is the AF\_INET (normal internet address)
* # 30 --- is the AF\_INET6 (IPv6)
* **try**:
* networkServerDic["name"] = netInt
* networkServerDic["mac"] = netIntDesc[netifaces.AF\_LINK][0]["addr"]
* networkServerDic["ip"] = netIntDesc[netifaces.AF\_INET][0]["addr"]
* networkServerDic["status"] = "up"
* networkServer.append(networkServerDic)
* **except** Exception as err:
* **if** int(str(err)) == netifaces.AF\_INET:
* networkServerDic["name"] = netInt
* networkServerDic["mac"] = netIntDesc[netifaces.AF\_LINK][0]["addr"]
* networkServerDic["ip"] = "127.0.0.1"
* networkServerDic["status"] = "down"
* **if** networkServer == []:
* networkServer.append(networkServerDic)
* # print networkServer
* **return** networkServer
* **if** \_\_name\_\_ == '\_\_main\_\_':

    getNetworkInt()

# 12. Reference

Add reference