TITLE: Study and proposal of a distributed and scalable real-time media production platform

DEGREE: MSc Telecommunication Engineering and Master's degree in Applied Telecommunications and Engineering Management (MASTEAM)

AUTHOR: Gerard Castillo Lasheras

ADVISOR: David Cassany Viladomat

SUPERVISOR: David Rincón Rivera

DATE: October 7, 2015
Resum

La indústria de producció de continguts audiovisuals (e.g.: cadenes de radiodifusió, productores de baix pressupost) ha estat, i encara està, emprant tecnologies rígides i difícils d’escalar per al transport i gestió dels seus fluxos a través de les seves cadenes de producció. Tot i que des de principis de l’any 2000 s’està portant a terme una migració a tecnologies basades en xarxes IP, l’adopció està essent timida i lenta.

A més, la majoria d’aquestes tecnologies encara vigents impliquen grans costos de desplegament i manteniment (e.g.: maquinari específic, cablejat específic i costós). Per aquest motiu, es proposa l’estudi de tecnologies IP i, específicament, tecnologies relacionades amb el concepte de la computació distribuïda, i al núvol, per tal de proposar solucions per a abaratir els costos i millorar les possibilitats de producció de continguts audiovisuals.

Concretament, aquesta tesi s’enfoca en analitzar, proposar, desenvolupar i demostrar tecnologies específiques de virtualització, monitoratge i aplicació, que ofereixen solucions als reptes esmentats.

Pel que fa a la virtualització s’utilitzen tecnologies basades en Linux Containers, concretament contenidors Docker. Gràcies a la capa de gestió que ofereix aquesta tecnologia s’assoleix l’empaquetament, distribució i execució d’aplicacions de forma distribuïda a la xarxa. A més a més, s’assegura una plataforma escalable ja que aquest tipus de tecnologia permet el manteniment, gestió i replicació d’aplicacions de forma ràpida i robusta.

L’aplicació d’eines de monitorització és una peça clau per a oferir a les aplicacions i a la pròpia plataforma el control de l’estat d’aquestes i així permetre aplicar polítiques d’actuació a temps real de forma eficient. En concret, s’utilitzen les tecnologies Collectd i Graphite. Aquestes eines també permeten ser gestionades dins de contenidors per tal de poder ser distribuïdes per la xarxa en paral·lel a les aplicacions que conformen la plataforma.

Finalment, es demostra que el nucli de la plataforma, el LiveMediaStreamer framework, assoleix els requisits per a ser utilitzada com a servei al núvol per a la producció de continguts audiovisuals a temps real gràcies a les tecnologies esmentades anteriorment, la implementació d’una capa d’estadístiques (de xarxa i de rendiment intern) i el desenvolupament d’un software intermig que ofereix una API REST.
Overview

The audio-visual media content production industry (e.g.: broadcasters, small production companies) has been, and already is, employing rigid and difficult to scale technologies to transport and manage their streams through their processing chain. Although since early 2000s a gradually adoption of IP technologies has been happening, the process is still slow.

Furthermore, most of the existing technologies involve large deployment and maintenance costs (e.g.: specific hardware, specific and costly wiring). For this reason, the study of IP technology is proposed, specifically technology related to the distributed cloud computing concept, in order to propose solutions to reduce costs and increase the audiovisual content production’s possibilities.

Particularly, this thesis focuses on analysing, proposing, developing and demonstrating specific virtualization, monitoring and application technologies in order to provide solutions to these mentioned issues.

Regarding virtualization, technologies based on Linux Containers are used, specifically Docker containers. Thanks to the managing layer offered by Docker containers the shipment, distribution and execution of applications over the network is achieved. Moreover, platform scalability is assured because the maintenance, management and replication of applications containerized within this technology are fast and reliable.

The use of monitoring tools is a key point to offer application status management to applications and to the platform itself and to allow the application of actuation policies in real-time in an efficient manner. Specifically, Collectd and Graphite are the selected tools. Moreover, these tools are able to be managed inside containers in order to be simultaneously deployed over the network together with the applications’ platform.

Finally, as it is demonstrated, the core of the platform, the LiveMediaStreamer framework, achieves the requirements in order to be used as a real-time cloud service for audiovisual media content production. This is thanks to the technologies above-mentioned, the statistics layer implemented for monitoring (network and performance) and the development of a REST API middleware.
Dedicat a tothom que m’ha permés i ajudat a arribar fins aquí.
# CONTENTS

Introduction ................................................. 1

CHAPTER 1. State of the art ................................. 5
  1.1. Media content production ........................... 5
      1.1.1. Standard formats .............................. 5
      1.1.2. Convergence to IP ............................ 7
  1.2. Migration to cloud ..................................... 9
      1.2.1. Cloud computing ............................. 9
      1.2.2. Virtualization ............................... 11
      1.2.3. Monitoring .................................. 14
  1.3. LiveMediaStreamer framework ....................... 16

CHAPTER 2. Problem statement and proposal ............... 19
  2.1. Architecture analysis ............................... 19
      2.1.1. Virtualization ................................ 20
      2.1.2. Monitoring layer ............................. 22
      2.1.3. Application layer ............................. 24
  2.2. Architecture proposal ............................... 26
  2.3. Task planning ....................................... 28

CHAPTER 3. Application ................................... 29
  3.1. REST API ............................................ 29
  3.2. Network metrics .................................... 30
      3.2.1. Input network metrics ....................... 30
      3.2.2. Output network metrics ..................... 32
  3.3. Pipeline metrics ................................... 33
      3.3.1. Delay .......................................... 34
      3.3.2. Losses ......................................... 35

CHAPTER 4. Virtualization ................................. 37
4.1. Creating and managing generic containers .................................................. 37
  4.1.1. Basic LMS container ................................................................. 39
  4.1.2. HTTP REST API container ....................................................... 40

4.2. Linking containers ...................................................................................... 41
  4.2.1. Same OS ...................................................................................... 41
  4.2.2. Separate OS ................................................................................... 42

4.3. Running multiple processes within a container ............................................. 42

4.4. Best practices ............................................................................................. 43

CHAPTER 5. Monitoring layer ............................................................................ 45

5.1. Monitoring containers .................................................................................. 45
  5.1.1. From the container point of view ....................................................... 46
  5.1.2. From the OS point of view .............................................................. 47

5.2. Showcasing monitoring ................................................................................ 48

CHAPTER 6. Platform deployment tests, and results ............................................ 51

6.1. Platform deployment .................................................................................... 51
  6.1.1. Isolated deployments ...................................................................... 52
  6.1.2. Generic scenario deployment ............................................................ 54

CHAPTER 7. Conclusions .................................................................................... 59

7.1. LiveMediaStreamer .................................................................................... 59

7.2. Collectd and Graphite ................................................................................ 60

7.3. Docker ....................................................................................................... 61

7.4. Platform ..................................................................................................... 62

7.5. Next steps .................................................................................................. 62

Acronyms ........................................................................................................... 65

Bibliography ........................................................................................................ 67

APPENDIX A. LiveMediaStreamer architecture .................................................. 73
APPENDIX B. LMS HTTP RESTful API ........................................... 85

APPENDIX C. Specific application metric methods ....................... 87
C.1. Input network metric method ............................................. 87
C.2. Output network metric method ......................................... 88
C.3. Pipeline delay metric method ........................................... 89

APPENDIX D. Docker cheat sheet ............................................. 91

APPENDIX E. Docker, Nginx and Collectd configuration files .......... 105
E.1. Basic LMS container ....................................................... 105
E.2. HTTP REST API container .............................................. 106
E.3. Running multiple processes within a container ...................... 107
E.4. Nginx server file configuration example ............................ 108
E.5. Sharing volumes within containers .................................... 110
E.6. Collectd client container ............................................... 110
E.7. Collectd client configuration ........................................... 111
E.8. Collectd server and Graphite container .............................. 111
E.9. Collectd server configuration ........................................... 113
E.10. Collectd tail plugin example configuration ......................... 113

APPENDIX F. Source codes .................................................... 117

APPENDIX G. Exchanged e-mails with Live555 developer mailing list ......................................................... 119
LIST OF FIGURES

1 Broadcasting example based on open-source softwares [2] .................. 2

1.1 Cloud computing layers .................................................. 10

2.1 Generic platform architecture ........................................... 19
2.2 QEMU with KVM or hypervisor type2 to type1 ......................... 21
2.3 Docker with LXC .......................................................... 22
2.4 Monitoring layer architecture proposal .................................. 24
2.5 LiveMediaStreamer framework statistics gathering and logging proposal ................................................................. 26
2.6 Proposal of the platform architecture .................................. 27
2.7 Example of specific platform deployment ................................ 27
2.8 Task planning - Gantt proposal .......................................... 28

3.1 RESTfull API middleware architecture .................................. 29
3.2 Input network metrics structure ......................................... 31
3.3 Output network metrics structure ....................................... 33
3.4 LMS framwork's internal pipeline structure example ................ 34

5.1 Detailed monitoring architecture ........................................ 45

6.1 Configuration of the scenario for the isolated deployments .......... 52
6.2 Isolated scenarios - CPU usage ......................................... 53
6.3 Isolated scenarios - average pipeline processing time ............... 53
6.4 Isolated scenarios - pipeline accumulated lost blocks ............... 54
6.5 Configuration of the scenario for the generic deployment .......... 55
6.6 Generic scenario - video mixing configuration result ................ 55
6.7 Generic scenario - average CPU usage ................................ 56
6.8 Generic scenario - paths average processing time .................... 56
6.9 Generic scenario - paths accumulated lost blocks .................... 57

7.1 Generic scenario - derivative function of the video path accumulated lost blocks ....................................................... 60
7.2 Transcoder scenario - Collectd bandwidth usage (bits per second) 60
7.3 Audio and video mixer scenario - Collectd bandwidth usage (bits per second) ................................................................. 61
LIST OF TABLES

6.1 Deployment environment characteristics ........................................ 51
INTRODUCTION

The audio-visual media content production industry (e.g.: broadcasters, small production companies) has been, and already is, employing rigid and difficult to scale technologies to transport and manage their streams through their processing chain. But, since early 2000s, a gradually adoption of IP technologies has been happening. Key examples of this adoption are that TV content media production is already digital based (i.e.: broadcasting channels) and its media transport layer is circuit oriented. Moreover, IP networks offer enhancements over operational and cost issues and it is the next step to the audiovisual media production.

Since few years ago, the broadcasting industry is pushing on to adopt IP as the transport technology because of several benefits such as:

- Enhanced agility and flexibility of the broadcast workflows
- Convergence of services (i.e.: audio, video, metadata and generic data)
- Format agnosticism to support the adoption of coming new UHDTV formats such as 4K and 8K
- Economy of scale by integrating broadcasting industry into the far more massive IT industry

But there is still a lot of work to be done to achieve complete IP convergence, and lots of new possibilities thanks to different architectures and configurations that can be applied on OTT (Over-The-Top [1]) content management systems.

Since almost all post-production environments have become file-based, they can now be completely migrated to IP infrastructures. Main examples are some control and management services, and media content distribution over Internet (i.e.: live or on demand media streaming to end-users) or over proprietary networks (i.e.: IPTV [1]).

Nevertheless, the live production environment has yet to be migrated to IP. The main challenges are some stringent constraints such as levels of synchronization, extremely low packet loss levels, high-bandwidth demand or jitter variation, because these are not all intrinsically assured by current IP-based technologies, which offer a best-effort service with no guarantee.

Figure 1 helps understanding where this thesis is focusing and what is its goal: to offer specific solutions and tools for real-time media content production over cloud infrastructures. Therefore, this thesis is mainly focused on the "Live production" step of Figure 1, where related participants (i.e.: people, cams, mixers, ...) are aimed to be networked or moved to an IP and virtualized environment.

The following topics are going to be studied together with the aim to propose and develop a platform prototype. Moreover, this platform prototype will be deployed in an experimental environment in order to demonstrate and validate the proposal.

- Virtualization layer

The virtualization paradigm offers specific solutions to improve scalability, robustness and reliability of any platform and services to be deployed over a cloud en-
Monitoring layer

The monitoring layer is a crucial system tool to be deployed which offers the capability of getting non-stop feedback from deployed services in order to actuate in real-time over any defined alarm. This layer gives also the chance to find and solve infrastructure/platform bottle-necks. So, in this thesis environment, the performance of the services deployed will be monitored (e.g.: LiveMediaStreamer framework processing latency and losses, and external network performance parameters such as bandwidth usage, packet losses and delay variation).

Application layer

The application layer is the core service itself (the audio and video production core service) that must be adapted in order to offer full compatibility with previous mentioned layers and to become a cloud service. Therefore, a HTTP REST API and application statistics gathering will be developed.

All the items above-mentioned are related to this thesis execution period that is aligned with specific goals of the i2CAT Foundation Audiovisual Unit [4], which has specific research challenges such as to study and propose enhancements for networked media and interactive and immersive media related topics. Specifically, this thesis is one of the next steps related to i2CAT’s LiveMediaStreamer framework project [5], an open-source software developed in the Audiovisual Unit's technical team. One of this main functionalities
is the capability of working as a software-based audio and video mixer, and this is the scenario that this thesis is going to focus on as the main service to be analysed.

Finally, this thesis is organized as follows: Chapter 1 describes the state of the art of related technologies of the audiovisual content production, where the steps already done for reaching IP convergence are briefly explained. Furthermore, an introduction of the cloud concept and the topics related of this thesis are also shown. In Chapter 2, the problem statement is presented with a proposal solution, based on the topics already explained. Chapter 3 (application), 4 (virtualization) and 5 (monitoring) are each one focused on how each part of the solution are developed. Chapter 6 describes the deployment, tests and demonstrations of the software. Chapter 7 concludes the thesis and describes the future lines of development.
CHAPTER 1. STATE OF THE ART

This chapter aims to provide a vision of how the audio-visual media production sector has been, and still is, converging to IP.

Moreover, this chapter will be focused on the topics related to media production over an IP environment and, specifically, the ones related on this thesis study and proposal. These are virtualization, monitoring and application core service.

Within the broadcast workflow two main scenarios can be distinguished: production and diffusion (i.e.: broadcasting). It is important to remark that this thesis is going to be focused only on the production scenario (i.e.: live production, see Figure 1).

1.1. Media content production

This section summarises the standard formats currently used in audio-visual media content production and how they are adapted to reach IP convergence.

1.1.1. Standard formats

In production environment, broadcasters manage their audio and video content in an uncompressed or slightly compressed state. The main reason is to ensure they deliver the best quality possible (i.e.: lossless quality) of their digitized data. Besides, the goal to deliver best quality possible is also intended for diffusion environments but taking into account bandwidth usage constraints.

1.1.1.1. Media formats in production environment

In 1989, SMPTE (The Society of Motion Picture and Television Engineers [6]) standardized a family of digital video and audio interfaces based on coaxial cable, called SDI (Serial Digital Interface [7]), which is used for transport of uncompressed digital video and audio in a television studio environment.

Around SDI there are other related standards which are focused in specific solutions (i.e.: to support higher video resolution, media qualities, media formats, frame rates, audio channels, synchronization, ...). An example is the HD-SDI [7] standard (High-Definition Serial Digital Interface, defined in SMPTE 292M) which provides enough bandwidth to transport HD video formats (i.e.: up to 1,485 Gbps). Other SDI related standards are the 3G-SDI [8] and 12G UHD-SDI [9], which give support to FHD (i.e.: Full HD, 1080p resolution up to 2,970 Gbps bitrates) and 4K (i.e.: UHD cinema resolution up to 12 Gbps bitrates) video formats, respectively.

The main parameters related to uncompressed audio and video signals are:
- For video:
  
  - Color depth: this is also referred to as bits-per-pixel (BPP), and defines how many colors can be represented by each pixel in the video frames (i.e.: still images). A number \( n \) of bits per pixel provides \( 2^n \) colours per pixel.
• Video resolution: this is measured by the number of pixels wide by the number of pixels high of a video stream.

• Frame rate: this is the number of still images (i.e.: video frames) per second (i.e.: fps) sent as part of the video stream.

Then, the bitrate generated by an uncompressed video stream can be calculated as:

$$\text{bitrate}(\text{bits/s}) = \text{color depth}(\text{bits/pixel}) \cdot \text{frame size}(\text{pixels/frame}) \cdot \text{frame rate}(\text{frames/s})$$  \hspace{1cm} (1.1)

- For audio:

  • Sample rate: This is the average number of audio samples obtained in one second and its measurement unit is hertz (Hz).
  
  • Bit depth: this is the number of bits an audio sample is recorded at.
  
  • Number of channels: this is the number of separate streams of the audio information.

Then, the bitrate generated by an uncompressed audio stream can be calculated as:

$$\text{bitrate}(\text{bits/s}) = \text{sample rate}(\text{samples/s}) \cdot \text{bit depth}(\text{bits/sample}) \cdot \text{channels}$$  \hspace{1cm} (1.2)

A typical HD-SDI stream with 10 bits per sample and YUV\(^1\) 4:2:2 color encoding scheme is applied with Y sampled at 74.25 Msamples per seconds and \(C_B\) (i.e.: U) and \(C_R\) (i.e.: V) sampled at 37.125 Msamples per seconds. So, the amount of bandwidth accepted is:

$$10\text{bits/sample} \cdot (74.25\text{Msamples/s} + 2 \cdot 37.125\text{Msamples/s}) = 1.485\text{Gbps}$$  \hspace{1cm} (1.3)

This number includes both the active part of the image and the inactive part (synchronization).

If the visible part is calculated, a video sampled at 50 frames per second at interleaved frame rate, with frame size of 1920 x 1080 pixels and bit depth of YUV 4:2:2 fits inside HD-SDI:

$$25 \cdot 1920 \cdot 1080 \cdot 10 \cdot [1 + 0.5 + 0.5] = 1.036\text{Gbps}$$  \hspace{1cm} (1.4)

1.1.1.2. Media formats in diffusion environment

The wide adoption of low-delay encoding (e.g.: JPEG2K, AVC, AVCi, VC-2) for high quality video streams could represent a new opportunity to reduce the bandwidth consumption in several scenarios (e.g.: diffusion). Likewise, high-compression mechanism as MPEG4 H264 or HEVC could be useful to transport media content through very limited network resources scenarios (as Internet or cloud-based systems).

\(^1\)In YUV, ‘Y’ represents the brightness, or luma value, and ‘UV’ represents the color, or chroma values (U is the blue-difference and V is the red-difference). In contrast, the values of the RGB encoding scheme represent the intensities of red, green and blue channels in each pixel. YUV is used because the human eye perceives chroma worse than luma, and therefore a chroma subsampling factor of 2 or 4 can be applied (i.e.: only half or a fourth of color samples are taken, compared with the luma or black-and-white signal.)
To remark that visually lossless compression formats (e.g.: the lossless encoding profile of JPEG2K) are also considered for production environments.

Related to the above statements, the recent advances in chip designs by industry leaders (such as Intel, Broadcom, Xilinx and Altera) have eased a strong movement towards consolidation of complex functions (e.g.: encoding, transcoding, conversion) into a single device, instead several disparate platforms. Additionally, these hardware advances imply the chance of using software-centric frameworks, which provide for greater flexibility and customization from a business standpoint. Both advances are enabling a broader adoption of upcoming media technologies at a reduced cost, without compromise on quality, flexibility nor capability.

1.1.2. Convergence to IP

This section focuses on IP convergence, from the transport and media types points of view, and describes how media content is transported in an IP environment over tools that are going to be developed during this thesis, to reach the goal of convergence of the media content production to IP.

Since SDI standard was released, new encapsulation audio (i.e.: AES67-2013 [10]) and video (i.e.: SMPTE 2022-6 [11]) standards have appeared for the transport of high-quality media signals over IP Networks. Also, further specific solutions, such as managing packet loss recovery using FEC (SMPTE 2022-5 [12]), provide higher robustness.

In the middle of 2014, the Video Services Forum (VSF [13]) has formed a new working group (SVIP [14], which focuses on defining and researching requirements for video over IP without SDI encapsulation) looking at new encapsulation mechanisms for audio, video and ancillary data into IP without using SDI framing (raw data) to develop or recommend a standard for video over IP without SDI encapsulation.

Moreover, in 2013 SMPTE, VSF and EBU (European Broadcasting Union) created the JT-NM task force (JT-NM) to drive the broadcasting industry towards a full IP adoption by providing guidelines to enable a successful migration. Currently, the JT-NM is working to develop a reference architecture to help all involved layers to agree on all cross issues while defining specific requirements over concrete use cases to uncover missing definitions to address the general scenario [15].

Both groups aim to study and define the requirements for video over IP/Ethernet within plant (e.g.: video, audio, ancillary data, bundles, timing, sequencing and latency) in order to research over current and proposed solutions so that to report on gaps between requirements and existing solutions (especially regarding existing SMPTE 2022 Standards) and finally to propose scope for follow on activity, if required.

1.1.2.1. Layer 2 - Data Link

Besides, it is important to remark the role of the Ethernet [16] protocol, which was standardised in 1983 and since then it has been increasing its speed rate from the initial 10 Mbps to 100 Gbps (foreseen 400 Gbps by IEEE P802.3bs [17] Task Force), with currently easily affordable 10 Gbps and 40 Gbps interfaces. These rates are enough to accommodate current broadcast formats (e.g.: HD-SDI at 1.5 Gbps, 3G-SDI at 3 Gbps and
UHD at 12 Gbps) and future formats, thanks to the nature of the packet technologies that make them completely agnostic to the upper formats and indeed transparent for future formats in contrast with current media transport technologies which are completely bounded with the transported formats (i.e.: standard cable video formats used over broadcast environments). On the other hand, Ethernet does not have any timing awareness or QoS assurance, and therefore it is difficult to accommodate current broadcast requirements over this technology. Since Ethernet is widely used in the IT industry as COTS\(^2\) (Commercial Off-The-Shelf) switches, the next logical step is to use this technology in the broadcast industry deployments and assess specific necessary features as the latency deviation or packet loss.

To address some of the inherent Ethernet limitations, Audio Video Bridging (AVB, IEEE 802.1BA-2011 standard [18]) appeared in 2011. AVB is a set of standard extensions to the Ethernet IEEE 802.1 [19] focusing on timing and QoS guarantees within local area networks. Its approach is a plug-and-play platform to ease transition from current transport technologies to the newer ones using the same workflows, but the current version is still limited to local premises and limited topologies. Since November 2012, because more varied industry sectors joined the task group, a more general name, the Time Sensitive Networking (TSN task group [20]), was created to carry on with the new developments.

A new paradigm, SDN (Software-Defined Networking) is emerging [21]. SDN separates the control and forwarding plane besides, creating northbound interfaces to interact with external applications, enables new flexible and customised network operations and deployments. There are a lot of foreseen benefits from this approach but to be fully capable of supporting all type of streams some extensions should appear, such as the specific extension which has been released by the ONF (Open Networking Forum) to address timing restrictions [21].

1.1.2.2. Layer 3 - Network

IP is the de facto standard and within its protocol suite there are some solutions which help to transport media content efficiently. For instance, IP supports the multicast [22] paradigm operation using widely supported routing protocols (e.g.: IGMP), but the computational and scalable complexity of these protocols tends to difficult and limit deployments.

In terms of QoS, IP has a field known as ToS/DSCP [23] which marks the header packets along their way to help mappings with lower-layer protocols (e.g.: Ethernet or MPLS) to implement QoS at the buffer level.

1.1.2.3. Layer 4 - Transport

UDP has been preferred over TCP for real-time transport because of its connectionless nature and avoidance of unnecessary retransmission and congestion avoidance techniques for live streams. RTP (Real Time Protocol), whose most deployed version is RFC 3550, was initially introduced to audio and video services, to add supplementary data fields in order to enhance the UDP protocol (i.e.: timestamp field, sequence number, payload iden-

\(^2\)COTS refers to products that are commercially available and can be bought ready to use. The use of COTS products/services may imply a reduction of overall development, deployment and maintenance time and cost.
tification), together with the protocol for control purposes (i.e.: RTCP, which stands for Real Time Control Protocol). RTCP takes care of monitoring the QoS of the audiovisual transmission. Recently, new extensions have appeared introducing new header options to support the adoption of services related to media production workflows. Specifically, RTP and RTCP extension have been proposed to accommodate media specific info over IP.

1.1.2.4. Signaling and metadata

In the signaling layer, protocols such as RTSP (Real Time Streaming Protocol) for end-to-end session control or SDP for service description provide capabilities for the stream management. Furthermore, media wrappers aim to gather different types of programme media and associated information, as well as generically identify this information. Different media wrapper formats are in use at this time, but, for the media industry, it is important that the wrappers have characteristics like openness, extensibility and performance. The MXF (Material Exchange Format) is a container format [24], which supports a number of different streams of coded based by enabling interoperability between different platforms. This is done by encoding in any type of video and audio compression formats, together with a metadata wrapper which describes the material contained within the MXF file.

1.2. Migration to cloud

This section is related to the previous one but focusing on how OTT\(^3\) [1] content technologies (i.e.: video delivery techniques) are giving new chances to enhance audio-visual media production to IP convergence, concretely within the cloud computing concept.

1.2.1. Cloud computing

Cloud computing describes the delivery of shared computing resources (software and/or data) on demand through the Internet. Cloud computing is defined by the NIST recommendations [25]. So, for many reasons like flexibility, scalability, security, data protection, agility and cost many organisations are migrating to cloud computing environments.

Nowadays, cloud computing defines three fundamental models named SaaS, PaaS and IaaS, as seen in Figure 1.1, that are organized through application/service, platform and infrastructure layers.

Moreover, there are different deployment models depending on the product to be delivered (e.g.: specific service or application), which are related to the resources from the entity that is offering or using such product. The main deployment models are:

- Public: when applications/services run over resources that are open for public use, which may be free. The fact of being public/opened implies much more complexity in terms of security issues.

\(^3\)OTT refers to the service you use over the network services of your service provider. An example of OTT service is Youtube, which lets you playback video content on top of the infrastructure of several ISPs (Internet Service Providers).
Figure 1.1: Cloud computing layers

- Private: when infrastructure is operated solely for a single organization, whether managed internally or by a third-party, and hosted either internally or externally. This cloud type might be similar in terms of architecture design from the public one.

- Hybrid: when a composition of two or more clouds (private or public ones) are treated as distinct entities but are bound together, offering the benefits of multiple deployment models. Hybrid cloud allows to extend the capabilities of a cloud service by aggregation, integration or customization with another cloud service.

High-performance computing (e.g.: GPU based clouds [26]) and software-defined networking (SDN) can improve solutions to current cloud issues such as security, processing performance and full processing chain control through specific SLAs (Service Level Agreements), among others.

So, in many terms, the cloud concept is a key solution to help media producers create better content more quickly. There are lots of examples to focus on, but let us introduce the ones that will provide flexible and scalable ways to access the benefits that cloud computing brings to media production:

- Low-cost initial expenditures
  Media production tends to require an enormous initial investment in technology infrastructure and the technical staff to manage it. In that sense, cloud computing technology offers to the creative industries to ease the need to invest heavily in technology that would rapidly become obsolete. Cloud computing allows the media production industry to provision only the technology they need, when they need it, avoiding excessive CAPEX.

- Cost forecasting
  Infrastructure as a Service (IaaS) prices are predictable and granularly treated. It allows prediction on a per project basis with detailed cost analysis precision. As done by many IaaS providers (e.g.: Amazon and Wowza), each resource used in a media production workflow is metered, and companies pay only for what they use.
• Dynamic infrastructure deployment

Cloud computing helps production entities take advantage of the on-demand basis deployments. Media production companies can quickly provision servers to meet the demands of specific projects and shut them down when they are no longer needed. Moreover, cloud computing provides many infrastructure services such as content storage, transcoding, ingestion, . . .

Moreover, cloud computing can improve media production at many different media services requirements planes, such as:

• Media asset management
• Granular costs measurement
• Cloud transcoding
• High-speed file transfer
• Automated content verification
• Elastic deployment
• Real-time and full monitoring
• Video quality control

And, expected overall outcomes might be:

• Increased performance
• Lowered costs
• Improved cross collaboration

1.2.2. Virtualization

Cloud computing is usually strongly related and implemented with different kinds of virtualization. Many virtualization methods are commonly implemented at datacenters where platforms and services are going to be deployed over different infrastructure architectures. Nevertheless, deploying virtualization at data centers does not automatically mean running over a cloud and it is possible to deploy clouds without virtualization. Furthermore, the cloud computing concept started to be widely used from 2000’s, virtualization technologies such as virtual desktops can be traced back to the 1960’s, but others can only be traced back a few years, such as virtualized applications.

Specifically, virtualization under computing environments means creating a virtual version of any possible piece of actual hardware or software so that we can use system resources effectively. Despite the many ways to define current virtualization methods, we can summarize the types and levels as follows:
• Types: based on specific computer/server resources virtualization. We can distinguish:
  
  – Data virtualization: when an application is able to retrieve and manipulate data without requiring technical details of such data.
  
  – Memory virtualization: when, in a cluster, volatile random access memory (i.e.: RAM) resources are decoupled from physical machines in order to be aggregated with other RAM resources and to become a virtualized memory pool.
  
  – Network virtualization: when combining hardware and software network resources and network functionalities into a single and software-based management entity.
  
  – Storage virtualization: when pooling data from multiple and different storage devices into a virtual device that is managed from a central console.

• Levels: based on abstract and generic virtualization concepts. The following cases can be distinguished:
  
  – Application virtualization: when encapsulating an application software from the underlying operating system on which it is executed. It involves separating the physical client device from the management of the application itself.
  
  – Environment virtualization: when virtualizing at operating system level. It is a virtualization method where the kernel of an operating system allows for multiple isolated user-space instances, instead of just one.
  
  – Hardware virtualization: when hiding the physical characteristics of a computing platform from a user point of view and showing another abstract computing platform. It means computer or operating system virtualization by creating virtual machines. Nowadays, the software that manages virtualization is called hypervisor or virtual machine monitor.

Therefore, in terms of cloud computing benefits, virtualization can increase agility, flexibility, and scalability while creating significant cost savings. Workloads might be deployed faster, performance and availability increases and operations can become fully automated, resulting in a cloud with ease to be managed.

This section focuses on the previous defined virtualization layers, which are of interest for this thesis development.

So, starting from the upper layer, the current technologies for application virtualization are:

• Desktop virtualization: when separating part or all of the desktop environment and associated applications from the physical client device that is used to access remotely or locally it. This improves portability, manageability and compatibility of a personal computer’s desktop environment. A common implementation of this approach is to host multiple desktop operating system instances on a server hardware platform running a hypervisor. This is generally referred to as Virtual Desktop Infrastructure (i.e.: VDI). Some commercial examples are Microsoft RemoteApp and the Citrix Seamless Windows.
• Application streaming: when delivering pieces of the application’s code, data, and settings when they’re first needed, instead of the entire application being delivered before startup. Running the packaged application may require the installation of a lightweight client application. Packages are usually delivered over a protocol such as HTTP, CIFS or RTSP. Some examples are Microsoft App-V and Citrix XenApp Streaming.

The next lower layer is the intermediate layer of environment virtualization. The pioneer implementation of this layer was FreeBSD’s jails mechanism, allowing system administrators to partition a FreeBSD-based computer system into several independent mini-systems called jails. There are many other examples of environment virtualization, but all of them are OS-based virtualization with differences such as its kernel operating system (i.e.: FreeBSD, Solaris, Unix-like, and Windows) and the level of isolation in terms of resources utilization (i.e.: types of virtualization, explained above), security and ease of delegation.

Currently, the environment virtualization method of Linux containers (LXCs) are widely enhancing application/services development, testing, packaging, deployment and managing methodologies. Specifically, containers represent one of the leading trends in computing today. With this technology it is possible to run multiple isolated Linux systems (i.e.: containers) on a single Linux control host. LXC combines kernel’s cgroups\(^4\) and support for isolated namespaces\(^5\) to provide an isolated environment for applications without the need for starting any virtual machine.

Finally, the lower environment virtualization layer is the hardware-centric one. Different methods can be distinguished as follows:

• Full virtualization: when simulating enough hardware to allow using an isolated guest operating system in a virtual machine. There are many examples of implementation like Parallels, VirtualBox, OracleVM, VMware and QEMU among other platforms.

• Hardware-assisted virtualization: is a full virtualization enhancement that uses specific hardware capabilities by improving hardware simulation efficiency. There many implementations’ examples like Linux KVM and Xen among others platforms.

• Partial virtualization: was the previous virtualization technology of the full virtualization. The main differences resides on the address space virtualization, in which each virtual machine consists of an independent address space. This fact implies that a full operating system is not able to run in a virtual machine but many of its applications.

• Paravirtualization: when a virtual machine does not implement full hardware virtualization, but offers a special API for a guest with a modified version of the operating system. This type of virtualization is also implemented in most of the widely used virtualization platforms like VMware, Parallels and Xen.

\(^4\)The control groups (cgroups) is a Linux kernel feature that limits, accounts for, and isolates the resource usage (CPU, memory, disk I/O, network, etc.) of a collection of processes.

\(^5\)Namespace isolation refers to a Linux kernel feature where specific groups of processes are separated such that they cannot interact with resources in other groups (i.e.: namespaces).
1.2.3. Monitoring

Strongly related to cloud reliability is the monitoring concept. In order to reach maximum cloud reliability it is important to observe and check the progress and/or quality of key parameters over certain periods of time and to keep them under systematic review in order to create proper reactions, if required.

Therefore, this implies monitoring the cloud infrastructure (e.g.: servers, virtual or physical) and related services (e.g.: applications). Here appear the QoS (Quality of Service) and QoE (Quality of Experience) terms, respectively.

QoS is the network-centric monitoring of underlying infrastructure components such as servers, routers and its network traffic. QoS metrics are generally device-related (e.g.: CPU and memory load, CPU temperature, disk space or HDD health) or transport-oriented (e.g.: packet loss, delay, bandwidth usage or jitter).

Although QoS can be fully affordable due to the robustness and redundancy of current infrastructures (e.g.: back-up services, network rerouting and error correction), this does not mean that any end user might be feeling comfortable by using deployed services (e.g.: searching on a e-commerce webpage) over a QoS-assured infrastructure. Then, QoE monitoring term evaluates the quality delivered to a user and it is done by analysing parameters when connecting to such services like a user. Therefore, QoE performance indicators are user-centric (e.g.: webpages response time or measuring video and audio quality (e.g.: Mean Opinion Score test, MOS).

Common network monitoring protocols for distributed infrastructures management are:

- **SNMP**: Simple Network Management Protocol is a widely known and used Internet standard protocol for managing IP-capable devices. SNMP is based on monitoring stations (i.e.: traps) which implement registry (i.e.: Management information base, MIB) polling to specific equipments (IP-capable devices supporting SNMP), which offer data of interest such as disk usage, link status, CPU usage,... Moreover, it can be configured in an asynchronous mode.

- **WMI**: Windows Management Instrumentations is a Microsoft’s implementation of the Web-Based Enterprise Management (WBEM) and Common Information Model (CIM) standards from the Distributed Management Task Force (DMTF). It offers a detailed set of properties and methods (offering data metrics similar as done by SNMP) for access by an authenticated user but it is all done through Windows proprietary definitions.

- **NetFlow**: a Cisco protocol for network switches and routers. It is meant to be used as a network flow analyser (by identifying and analysing each configured flow, which is an unidirectional statistical packet sequence)

Usually, these protocols are used to measure QoS, but there are complex algorithms that processes those QoS measurements parameters of interest in order to measure the QoE too. Nevertheless, there are specific applications to define and perform specific QoE measurements.

Focusing our attention in the topics of this thesis, the QoE measurements are relevant for audiovisual content services because bad network performance may highly affect the
user’s experience. This is mainly because these contents are compressed and coded, and have low redundancy. Moreover, when designing systems, for referenced analysis, several elements in the video production and delivery chain may introduce distortion by degrading the content (i.e.: from the transcoding system, transport network, access network, home network to end device).

An important concept is the referenceless\(^\text{6}\) analysis, which is based on the idea that end users do not know about the original content. In this case, instead of measuring the QoE by comparing the original data to the delivered one, this is done by trying to detect artefacts (i.e.: blockiness, blur or jerkiness for video frames).

Obviously, the automation of critical cloud performance monitoring tasks is crucial for ensuring availability, providing efficient services and reducing common errors, costs and complexity. So, the use of OTT applications are crucial in order to process such quantities of data flows and display outcome parameters of interest.

There are many tools that offer monitoring capabilities to be integrated and of-the-shelf. Such monitoring capabilities can be organized as:

- **System monitoring**
  
  Single server/computer/instance resources monitoring (e.g.: CPU and memory loads or processes utilization). Typical examples are the system monitoring tools that each operating system includes by default.

- **Network monitoring**
  
  Related to the previous item, but specific to network resources monitoring (e.g.: monitoring input and output accumulated bytes of a single computer network interface or monitoring specific network hardware like accumulated incoming UDP packets of a router in a LAN). There are many examples of tools and services for network monitoring which go from desktop applications (e.g.: netstat [30]) to specific router and switch daemons (e.g.: MRTG [31]).

- **Infrastructure monitoring**
  
  When system and network monitoring are coupled together by adding specific tools and interfaces to monitor distributed resources within the infrastructure. There are many examples of tools (e.g.: cacti [32] or monitis [33]) and services (e.g.: new relic [34] or pingdom [35]) at this monitoring level.

The associated database model for collecting such amount of data is the widely known Round Robin Database [36], which stores data in a circular buffer based database where the system storage remains fixed by handling time-series data, and data is stored at different levels of time granularity by consolidating the more granular data into coarse time scales (e.g.: 5 minutes - 1 hour - 1 day - 1 week).

Finally, it is important to remark that network managers have the capability to minimize the storage and network resources by allocating only the resources that are required thanks to monitoring evaluation services and tools (i.e.: infrastructure optimization).

\(^\text{6}\)Referenceless analysis is a technique studied and implemented by the AGH Multimedia Team [27] and it has been applied within European projects [28] where i2CAT Foundation’s Media Internet area and AGH Multimedia Team has been collaborating [29].
1.3. **LiveMediaStreamer framework**

This section is devoted to the framework which the core service (i.e.: audio and video mixer software) to be analysed is implemented with, the LiveMediaStreamer (LMS) framework.

The aim of the LiveMediaStreamer framework is to offer multiple audio and video streams manipulation in real-time in many ways. It is designed following a pipeline pattern so that it consists in a number of filters (i.e.: encoders, decoders, receivers, transmitters, dashers, mixers and resamplers) that can be concatenated or connected with each other in order to process a data flow. The framework is developed under a Linux environment, currently being the only supported platform, using C++ standard libraries and it makes use of several mature media-related libraries, which are:

- **Live555** [37] – Network streaming media library which implements the RTP standard protocol.
- **ffmpeg** [38] – A complete, cross-platform solution to record, convert and stream audio and video.
- **OpenCV** [39] – Open source computer vision and machine learning software library.
- **x264** [40] – Free software library and application for encoding video streams into the H.264/MPEG-4 AVC compression format.
- **x265 HEVC Encoder** [41] – Open source HEVC encoder.
- **LAME** [42] – High quality MPEG Audio Layer III (MP3) encoder, under LGPL license.
- **Opus** [43] – Totally open, royalty-free, highly versatile audio codec.
- **WebM VPX** [44] – VP8/VP9 Codec SDK. A open, royalty-free, media file format designed for the web.

The framework is designed to be managed remotely through a simple network interface based on JSON-formatted TCP socket messages.

The LMS framework project stated to be developed since two years ago. A first implementation was based on the network core of the open-source software UltraGrid [45]. One year after, a new core has been developed based on the network streaming library Live555. This change has improved system performance and eased further developments. Live555 library enables implementing any RTP and RTSP module, standard or not, among offering out of the box specific audio and video RTP payload formats support like H264, HEVC or VP9 for video codecs and G711, OPUS or AAC for audio codecs.

Currently, the framework does not have a RESTful API, but has a web API based middleware, that loads and manages an audio and video mixer scenario, written in Ruby programming language. It implements Sinatra framework for the web service interface side.

The main reason to provide a REST API is due to its decoupled architecture and low bandwidth usage, which makes the REST architecture style suitable for developing applications over cloud environments.
Appendix A includes more information about specifics of the LMS architecture in order to understand the following chapters (mostly, in problem statement and proposal’s section 2.1.3. and Chapter 3, the solution's implementation).

Finally, note that there are many existing similar solutions but most of them are proprietary or closed solutions (i.e.: Wowza Streaming Engine or Adobe Flash Media Server). Moreover, these solutions does not provide video and audio mixing features which means that the use of other external tools is required if they are aimed to be used as real-time media production tools.
CHAPTER 2. PROBLEM STATEMENT AND PROPOSAL

The goal of this chapter is to provide a structured vision of the specific problems of each one of the points to be developed within this thesis.

The main challenges are to prepare and/or adapt current technologies to be ready for a cloud deployment. This fact implies studying the existing technologies and tools in order to create specific services that will be interconnected between them. Finally, the required developments will be carried out in order to assure the goals behind and ease demonstrate the results.

The main goals of this thesis are:

- To implement a high-level HTTP REST API in order to manage and monitor the LiveMediaStreamer framework.
- To demonstrate that LMS is able to be deployed in a virtualized environment.
- To enhance LMS with specific metrics measurements in order to demonstrate its capabilities as a real-time media streaming framework.
- To offer a set of tools which gather the aforementioned metrics and present them in order to demonstrate the previous statements.
- To include all previous statements within an architecture proposal.
- To demonstrate previous statements in specific testing deployments.

2.1. Architecture analysis

The main requirement is to define the platform’s architecture to be prototyped in order to have a global and generic insight. So, taking into account the pieces required for this thesis (i.e.: service layer, monitoring layer, virtualization/deployment layer), such architecture should contain the different high level layers as shown in Figure 2.1.

![Generic platform architecture](image)

Figure 2.1: Generic platform architecture

The "LiveMediaStreamer REST API" layer will contain the service that will be offered to different and external applications (communicating over HTTP) in order to manage the
"LiveMediaStreamer instance" layer by creating different audio and video production scenarios. Both layers are the core layers of the platform.

Moreover, in order to offer a centralized monitoring system, the "Statistics collector and display" layer becomes as the generic box for this requirement.

It is also required to provide an orchestrator that manages the deployment and distribution of the possible configurations for the previous introduced layers. This will be done thanks to the "Virtualization manager" layer.

Finally, an important point is to define how the communication between each layer is going to be carried out. This is described in the following sections.

### 2.1.1. Virtualization

This subsection aims to introduce the possibilities that different virtualization technologies can offer in our project requirements, and to decide between one of them.

First of all, the following points show which are the expected outcomes for using virtualization and what requirements should the selected technology fulfill:

- To manage and maintain a system of small pieces of services
- To have flexibility in order to quickly instantiate (e.g.: start, restart, stop) the required instances (e.g.: to assure real-time scalability) and to deploy any possible required scenario/configuration
- To offer ease of continuous development and deployment of the different parts of the architecture
- To have a virtualized application version control system for having different version tags for the architecture modules (e.g.: a development and a production container of the same REST API service)
- To assure full compatibility for the core layers’ operating system (right now only Linux environments are supported)
- To assure full compatibility for the hardware to work with (mainly x86 processors are in the scope)

Moreover, it is also required to use a technology that is as much lightweight as possible. Since we want to virtualize each piece of the platform architecture.

Under Linux environments there are many virtualization options to analyse (most of them proprietary), but let us focus on the ones that are open-source and have wider and active communities behind. These are:

- **KVM (Kernel-based Virtual Machines) [46]**
  It is a FreeBSD and Linux kernel module that offers a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It
consists of a loadable kernel module that provides the core virtualization infrastructure and a processor specific module. Usually, KVM runs with the QEMU (Quick Emulator) [47] which is a complete and standalone emulation suite that performs hardware virtualization.

KVM with QEMU are able to offer virtualization for x86, PowerPC, and S/390 guests. For instance, when the target architecture is the same as the host architecture, QEMU can make use of KVM particular features in order to not emulate CPU nor memory by using the offered by the host kernel.

![Figure 2.2: QEMU with KVM or hypervisor type2 to type1](image)

Figure 2.2 showcases how KVM can convert a type2 hypervisor (i.e.: QEMU) into a type1 hypervisor (known as a bare metal hypervisor) which increases overall application performances.

- **LXC (Linux Containers) [48]**

  It is an operating-system-level virtualization environment able to run multiple isolated Linux systems (known as containers) on a single Linux central host.

  Linux kernel itself provides the cgroups functionality that allows limitation and prioritization of resources (CPU, memory, block I/O, network, etc.) without the need for starting any virtual machines, and namespace isolation functionality that allows complete isolation of an applications’ view of the operating environment, including process trees, networking, user IDs and mounted file systems.

  Nowadays virtualization tendencies are focusing on the Docker project [49], which is a platform that provides an additional layer of abstraction and automation of operating-system-level virtualization on Linux, Mac OS and Windows.

  Using LXC with Docker mean resource isolation to allow independent containers to run within a single Linux instance, avoiding the overhead of starting and maintaining
virtual machines. Moreover, the Docker project automates the deployment of applications inside software containers and offers different tools to manage them (i.e.: CLI, API and file configurations), as shown in Figure 2.3.

Thus, it seems that the technology that best suits this thesis requirements is the Docker project. The main reasons are the capabilities of ease maintenance, test and quickly deploy each container.

Finally, it is important to point out that another key feature of the Docker solution is that it allows containers to intercommunicate containers that are in the same physical server by isolating its network layer. This means much more security and performance over other virtualization solutions (at least of the simplicity point of view when intercommunicating instances).

2.1.2. Monitoring layer

We now provide an insight on the minimum requirements for the monitoring module, together with an analysis of available tools. Following the same criteria used previously, we focus on open-source tools.

The main requirements for the monitoring layer implementation are:

- To ease distributed deployments.
- To offer full control for specific configuration requirements and to not depend on external/enterprise services.
- To be fully deployable.
• To be as lightweight as possible in terms of:
  – Processing capacity
  – Memory utilization
  – Network throughput

• To be supported under a Linux environment.

• To be flexible and scalable enough to be used in future extensions of the project.

• To support RRD (Round Robin Database) as a DBMS (Database Management System) in order to control data storage size.

The data to be gathered includes:

• CPU and memory usage per process.

• Network usage (per process involved and per media flow)

• Internal core service performance parameters (i.e.: LiveMediaStreamer core performance):
  – Processing time
  – Data block losses ratio

We propose to split the monitoring layer in order to ensure stated requirements. The proposal, as usually done in many monitoring tools such as system monitoring, follows this model:

• Gathering layer
  It will only be responsible of the gathering, distribution and storing of the metrics.

• Display layer
  It will only be responsible of displaying data in answer of user-specific queries.

An alert layer could be considered as a future development. This thesis is focused on measuring and demonstrating that such platform is feasible but it could be enhanced with a set of alarms and actuators system in order to become as automated as possible.

The following list describes some of the tools available that might fit this thesis’s requirements as previously exposed:

• Munin [51]
  A cross-platform web-based network monitoring and graphing tool designed as a front-end application. Written in Perl.

• RRDTool [52]
  A de-facto industry standard, it is a high performance data logging and graphing system for time series data. Written in C and runs under GNU/Linux and Windows platforms.
• **Collectd [53]**  
Small daemon which collects system information periodically and provides mechanisms to store and monitor the values in a variety of ways. Written in C and support any Unix-like OS.

• **Graphite [54]**  
Tool for monitoring and graphing the performance of computer systems, which collects, stores, and displays time series data in real time.

Many other solutions have been discarded due to its dependence on specific enterprises’ roadmap or because they do not fit under the aforementioned requirements.

Finally, Collectd and Graphite are the selected tools. The main reasons are due to being fully configurable and its core design and philosophy. Collectd has a data distribution system based on a push model and it can be single deployed (i.e.: gathering and display layer), but it is going to be used with Graphite as the storage and display tool. This fact assures high performance for the containers where the Collectd will be gathering and transmitting metrics through UDP. Moreover, Graphite software will be deployed as a centralized storing and displaying tool which will be fed from many Collectds. Another reason to select Collectd is its full compatibility with Docker (Collectd can be deployed inside a container and there is also an official plugin to monitor Docker containers in an OS). Finally, Graphite has also been selected because it offers a HTTP API which enhances its scalability.

Therefore, the monitoring layer is proposed to be designed as shown in Figure 2.4.

![Figure 2.4: Monitoring layer architecture proposal](image)

2.1.3. **Application layer**

This subsection aims to compile the minimum requirements that the core service (i.e.: the LiveMediaStreamer framework) should implement to fit within the platform. This means assuring that it can be demonstrated as an efficient real-time cloud production platform, as a prototype.
First of all, let us identify the main goals that LiveMediaStreamer framework should provide:

- To assure minimal performance cost in computational terms when measuring internal metrics in order to not interfere on the overall performance of the framework, whatever the scenario setup.

- To calculate specific metrics of interest. Common metrics should be:
  
  - For network usage statistics
    - Bandwidth usage per incoming and outgoing streams
    - Packet loss ratio per incoming and outgoing streams
    - Delay variation per incoming and outgoing streams
    - Delay from far-end clients to LMS per incoming and outgoing streams
  
  - For media statistics
    - Pipeline losses
    - Pipeline delay

- To compile the metrics in an efficient manner in order to let them be gathered by Collectd or any other application.

Currently, as of revision 0.2, the LiveMediaStreamer framework does not implement any metrics gathering neither logging.

Regarding network usage statistics, LiveMediaStreamer does not support any internal metrics gathering at network modules (i.e.: receiver and transmitter) where the Live555 library is implemented. However, the Live555 library allows a quick implementation of network statistics measurements (i.e.: bandwidth, losses, jitter, delay).

Thanks to Live555 examples (i.e.: the testProgs folder inside the source code library path) it becomes easy to understand how to gather network statistics at the input side. But, regarding output network statistics, it is not so obvious which is the optimal solution. This fact is mainly due to having two options which imply to implement the metrics gathering by doing some methods’ re-implementation of the RTP (i.e.: first option) or the RTCP (i.e.: second option) main classes, but this specific problem statement will be treated in Chapter 3.

Regarding media statistics, current LiveMediaStreamers’ version of the framework does not implement any metrics gathering. So, in order to minimize computational cost of such measurements we propose to implement:

- A specific method to measure processing delays (i.e.: the time a frame takes to be processed within a path).

- A specific method to measure processing losses.

Then, in order to implement the metrics presentation to be logged for external applications, a middleware API is required to be developed. Current API is not at all an API but a web GUI API for an specific scenario (i.e.: AVMixer scenario, as mentioned in Chapter 1 Section 1.3.), which means that it is required to implement a more generic middleware API to let
external applications to log the LMS internal metrics as API definition reclaims. Therefore, the proposal for solving this issue is to develop a RESTfull API in order to present the metrics from the LMS instance. This implementation is an opportunity to develop a full API for managing the LiveMediaStreamer framework over HTTP. Moreover, such middleware will ease to develop new applications over the LMS framework by evolving it as a SaaS, which means a proper enhancement to fit in a cloud environment.

Figure 2.5: LiveMediaStreamer framework statistics gathering and logging proposal

Figure 2.5 illustrates how previous statements are proposed to be implemented. As it can be seen in Appendix A, each filter can have one or more readers and writers (both are inheritors of the IOInterface class). This issue will be described in Chapter 3.

Finally, the Pipelinemanager class instance is the class responsible for compiling the metrics, filter by filter, when they are requested by the Controller class instance.

### 2.2. Architecture proposal

Once the problem statement has been described for each defined layer (i.e.: virtualization, monitoring and application) of the platform, a global architecture can be proposed. This is shown in Figure 2.6 which tries to be as generic as possible in order to illustrate the possibilities of the platform. There are different physical servers shown in order to demonstrate the aim to also support flexibility and scalability. In the following chapters (specifically the conclusions Chapter 6) we will demonstrate the feasibility of the approach.

Moreover, each physical server has different types of containers where different and possible combinations and configurations of the technologies are shown that might be deployed and used for specific use cases (scenarios).
For example, a possible and specific architecture configuration could be a transcoder scenario. Figure 2.7 shows a simple example of a transcoding service behind a web application. This use case includes an user, which configures and manages the LiveMediaStreamer’s scenario configured for a transcoding service (e.g.: starting with a HTML input form for incoming inputs, transcoding parameters and RTSP server configuration). Specifically, at the architecture level, there is a container with an LMS daemon running, another one with the REST API that receives queries from the HTTP server from another container,
which is serving the web application to the end user (e.g.: web browser). Then, each container has a collectd client that parses the logs of interest and sends the metrics to the Graphite service. Therefore, through the web interface the user can see specific graphics (e.g.: CPU usage, data losses, bandwidth usage, . . . ) from the graphite server (in another container) in order to monitor the overall performance and actuate if required.

Finally, it is important to point out that what will be deployed in order to demonstrate the platform feasibility is an specific scenario, but it is as generic as possible. This will be detailed in Chapter 6.

### 2.3. Task planning

This section describes the tasks to be done and its precedence.

![Figure 2.8: Task planning - Gantt proposal](image)

Figure 2.8 is the Gantt proposal for the task planning organization. It takes into account a period of six months for the execution of this thesis and it shows specific periods per task. For example, some tasks might seem overloaded, but this is taking into account that the whole time will not be spent on working in the thesis but other tasks related to tasks carried out at the i2CAT Foundation.

Note that the fact of working with a team inside an organization means extra tasks such as internal demonstrations periods in order to carry out specific validations.

The following chapters describes the work done in each one of the topics: Application (Chapter 3), virtualization (Chapter 4), monitoring (Chapter 5) and testing and deployment (Chapter 6).
CHAPTER 3. APPLICATION

The main goal of this chapter is to develop the tasks related to the application, including to prepare LMS to be deployed as a cloud service and to give support for internal and external monitoring.

3.1. REST API

As mentioned previous chapters, it is required to develop an API ready to be used over cloud environments in order to ease creating specific and new applications over the LMS framework, and thus to demonstrate the viability of this thesis prototype.

Nowadays, the common and widely used format for cloud services intercommunication is JSON, as it is also used for the TCP socket API of the LMS framework. Therefore, this API middleware is going to follow such requirement.

A suitable technology to work with JSON formatted messages is Node.js [55] which is widely known for its good performance. Node.js is an open source, cross-platform runtime environment for server-side and networking applications. It provides an event-driven architecture and a non-blocking I/O API that optimizes application’s throughput and scalability. This technology is commonly used for real-time web applications.

Working with Node.js means avoiding serialization of the JSON messages by increasing services intercommunication performance (i.e.: less computational cost and less processing time).

A common Node.js framework for developing web applications and REST APIs is Express.js [56]. It is the de-facto standard server framework for Node.js. So, our middleware development is going to use Express.js routing system (URIs with HTTP request methods like GET, POST, PUT and DELETE).

Figure 3.1 illustrates the software structure proposal for developing the HTTP RESTful API middleware, which is responsible for translating the TCP socket API of the LMS framework.
• HTTP REST API layer
  This layer handles HTTP queries from external applications. It implements specific routes to handle specific HTTP queries. The first implementation will not implement multiple LMS management but single, as shown in Figure 3.1.

• Interface layer
  This layer handles the body messages from the previous layer’s HTTP queries and manipulates them in order to create an as much generic as possible API by adapting the messages to be sent through following TCP socket layer.

• TCP socket layer
  This layer is the responsible for sending and receiving JSON-formatted TCP socket messages to and from the targeted LMS instance.

As presented in Section 1.3, and detailed in Appendix A, there are two different management layers: the generic one and the filter specific one. So, by following this organization, the proposed API’s structure is as described in Appendix B.

Note that this API is not implementing persistence\(^1\) because the state (managed through ‘State’ method) is given by the LMS instance itself. The unique sign of persistence is regarding the LMS host and port which the middleware is connected to (managed through ‘Connect’ and ‘Disconnect’ methods). Higher levels of persistence should be implemented by external applications, which implies specific scenarios and requirements (i.e.: specific persistence).

For more details about how this is structured and the overall middleware is implemented check Appendix F with the code.

### 3.2. Network metrics

Network metrics could be treated as external metrics. This is because these metrics are specially dependent from sources which transmit to LMS, the receivers from LMS and the state of the network itself. Obviously, the performance of the LMS affects to the metrics gathered too, but this effect is intended to be minimized, at least in a gathering and presentation of the metrics’ point of view.

#### 3.2.1. Input network metrics

As mentioned in Section 2.1.3., input network metrics are going to be implemented by carrying out methods re-implementations of methods provided by the Live555 library, which is the library that will manage network streams.

By following the LiveMediaStreamer architecture structure, input network implementations are going to be developed inside the ‘liveMediaInput’ structure. Specifically, a new

---

\(^1\)Persistence, in computer science, refers to the characteristic of state that outlives the process that created it. This is usually solved by storing the state’s application as data structures in databases.
class is developed, called ‘SCSSubsessionStats’. This class is managed by the ‘Stream-
CleanState’ class, which is a class related to each stream ‘Session’ class managed by
the ‘SourceManager’ class. This last class is a ‘HeadFilter’ class. Figure 3.2 shows the
inter-class structure.

By initializing new RTP or RTSPClient sessions (i.e.: network inputs), a group of subes-
sions is associated per each stream (i.e.: an RTP session has one subsession associated
and the RTSPClient session has as many subsessions as accepted from the SDP that
defines different RTP sessions).

When a new subsession is set, a new RTPReceiverStats class is automatically initialized.
This Live555 object implements RTCP stats measurement which are only required to be
treated outside. This is done at SCSSubsessionState, which creates a new schedule to
periodically measure and save current state (a default granularity of 1 second is set). The
implemented method is called periodically as shown in Appendix C Section C.1. This
implementation prepares the metrics that are going to be presented when a new state
query is received. The code in Appendix C Section C.1. shows how metrics from the
Live555 library are obtained. For a more detailed insight of the overall implementation see
Appendix F. The metrics that are presented per each new state query are shown next (a
default granularity of 1 second is set):

- Bitrate: maximum, minimum and average in kbps.
- Packet loss percentage: maximum, minimum and average.
- Inter-packet gap: maximum, minimum and average in milliseconds.
- Jitter: maximum, minimum and current inter-packet gap variation in microseconds.
All these metrics are measured through previous algorithm shown which is executed each second (it is the default value set for all metrics gathered indeed). Specifically, bitrate is measured by dividing the total number bytes, received during each scheduled period, by the elapsed time given by the Live555 library implementation:

$$\text{average bitrate(kbps)} = \frac{\text{kbits received}}{\text{elapsed seconds}}$$  \hspace{1cm} (3.1)

The maximum and the minimum bitrates are the last maximum and minimum average bitrates obtained, and this is done for all other metrics.

Jitter is measured as the estimate of the statistical variance of the RTP data interarrival time inserted in the interarrival jitter field of reception reports (in microseconds), and this is already internally done by the Live555 library. So, what is presented is the current jitter value at the beginning of each new schedule.

Another metric that might be of interest is the delay from the stream source to the LMS instance but it is discarded due to not being offered from Live555 library. Moreover, it has been discarded to be implemented at SCSSubsessionStats class level due to its computational cost and complexity to develop such requirement. Note that this metric is not relevant because network performance problems can be detected through other metrics already gathered (i.e.: jitter and packet loss ratio).

### 3.2.2. Output network metrics

As done in the previous section, output network metrics are going to be implemented by carrying out re-implementations of methods given by the Live555 library.

This implementation has been much more difficult due to not having control of the creation or deletion of the RTPSink class of the Live555 library. Previous developments before the final version were based over the RTPSink re-implementation already done per each OnDemandServerMediaSubsession (Live555 library class), which is also re-implemented by QueueServerMediaSubsession (LMS framework class). But the implementation was still losing the specific RTPSink instance of specific subsession.

In order to continue with the development, an e-mail was sent to the Live555 developers mailing list and a solution was provided by Ross Finlayson as shown in Appendix G.

The best option, as suggested by Ross, was to re-implement the RTCPInstance class per each inheriting class of the OnDemandServerMediaSubsession class, specifically the inheriting classes of the QueueServerMediaSubsession class.

Figure 3.3 shows the relationship between the SinkManager class with both possible types of connections (RTP or RTSP). Each Connection object has a map of objects of the re-implemented RTCPInstance class, called ConnRTCPInstance (i.e.: RTCP instances per each connection). Then, each time a new specific RTP connection or any QueueServerMediaSubsession (RTSP connection, from RTSP server) is created, a ConnRTCPInstance is associated in order to start gathering the statistics offered from Live555 library. This is shown in Appendix C Section C.2., with the code of the method that is periodically called (default periodicity value is set to 1 second). In this case, the delay metric is gathered from the Live555 library.
The metrics are:

- Bitrate: maximum, minimum and average in kbps.
- Packet loss percentage (ratio): maximum, minimum and current.
- Round trip delay: maximum, minimum and current in milliseconds.
- Jitter: maximum, minimum and current inter-packet gap variation in microseconds.

All these metrics are measured by following the same expressions used for the input network metrics.

### 3.3. Pipeline metrics

The following metrics are presented as the internal metrics. Please, note that the measurement of these metrics interfere the overall performance of the LMS framework. Therefore, to achieve the minimum possible computational cost is a must. But, first of all, let us pick up the example figure of a pipeline from Appendix A in order to showcase the internal pipeline structure of the LMS framework. As described in Appendix A, this will help understanding the following two implementations.

Note that in Figure 3.4 the arrows are the queues which interconnect each filter’s writer with another filter’s reader. Writers and readers are subclasses of the IOInterface class.
3.3.1. Delay

This metric is related to the time that data (i.e.: a video frame, an audio sample, . . . ) takes to be processed from an origin point to an end point by an unique given path, and this is measured from a given and required time: the data’s timestamps. Data inside the LMS source code is known as a “frame”, which can be an audio frame (i.e.: sample), a H264 video NAL unit, a raw frame, . . .

The delay time is not measured per each filter in order to not affect the overall performance, but each measured time involves an origin filter which resets the timestamps. These origins are the Head (i.e.: receiver) and OneToMany (i.e.: audio and video mixer) filters. These filters reset the frames’ timestamps in order to reach and control an internal synchronization, and this is due to the fact that these filters have many outputs (i.e.: writers) or many inputs (i.e.: readers) and they require synchronizing their outputs in order to assure one point of time control inside the LMS framework, whatever the scenario configured.

So, in order to measure the delay it is important to note that a pipeline is not composed by an unique path (see Appendix A for clarifications) but multiple paths. This fact implies that it is not possible to measure an unique overall delay time per frame which goes over the pipeline\(^2\), or at least it is not suitable for performance issues, but it can be done for external applications which know the scenario configuration and gathers such metrics. Therefore, this is solved by splitting the measurements by paths. And this is an optimal measurement: the delay is given by the differential time measured by the last reader of the

\(^2\)There is a special case when there is only a path that defines the pipeline itself (e.g.: a path for only transcoding video with one quality: a receiver, a decoder, a re-sampler, an encoder and a transmitter, all connected within a path that defines the pipeline of this specific scenario)
path (i.e.: the reader of the destination filter). Appendix C Section C.3. shows the method which implements the measurement inside the Reader class.

In summary, the code measure the average delay of a frame by a given window time (default is configured to 1 second) with a resolution of microseconds. And this means measuring from the origin time, which, as said, is set by the origin filters (i.e.: the beginning of a path, starting from a writer), to each reader of the path (i.e.: each filter). But, the delay time presented is from the beginning (i.e.: initial writer of the path) until the last reader. Figure 3.4 illustrates different path examples.

### 3.3.2. Losses

This metric follows a similar criteria as the previous one. This is solved by measuring at the same point, but it is done when flushing frames at the reader side (as presented in Section 2.1.3.). In order to reach minimal computational cost, the measurement is just a counter of the overall data losses when calling the flush methods. What is done is a method encapsulation by defining a parent method that is just incrementing its reader counter and then it calls the specific flush implementation per each data/frame type.

In order to properly present it to external applications it is required to be presented at a path level as done for the delay metrics. So, in this case it is only required to sum up the overall losses of the path’s readers.

It is important to note that this metric is not referenced to a total data processed or at any time point, but measuring its continuity over time permits detect losses with different thresholds. This means that if this value is incrementing gradually then the system is not working properly. Such detections might imply fast increments on its continuity (differential increase).

Note that Pipeline metrics are measured when flushing frames (discarding) or when a reader is able to remove a frame from its belonging queue.

Finally, once previous developments are carried out a first step to offer a cloud real-time media production service is achieved. This means, offering a RESTful API and a status layer for monitoring external and internal performance of the platform.
CHAPTER 4. VIRTUALIZATION

This chapter focuses on the preparation of generic containers for a media production platform prototype with the technologies and tools introduced in Chapter 1. We study and test the Docker technology in order to assure that LMS is ready to properly fit in a containerized environment and to evaluate different possibilities to reach maximum flexibility when configuring and deploying different scenarios.

4.1. Creating and managing generic containers

The minimal containerized entities are the following ones:

- The core container with a LiveMediaStreamer instance already deployed inside and ready to use.
- The HTTP REST API container with the middleware interface inside and ready to use

First of all, let us describe how and where Docker is installed. The host operating system where tests are going to be carried out is an Ubuntu 14.04 LTS [57], which is the Linux distribution version where the LMS is being developed. Note that previous list does not handle monitoring requirements, but this topic will be treated in Chapter 5. this is because our initial interest is related to testing how LiveMediaStreamer can be deployed inside a containerised environment.

Note that main Docker requirements for Ubuntu 14.04 are to be under a 64-bit installation and kernel must be at version 3.10 or higher (lower versions are buggy and unstable).

Some procedures must be taken into account in order to properly install and assure a best fit possible of the Docker technology inside the OS (and also to be ready for a cloud environment). These procedures include:

- To create a docker group: in order to avoid user permission issues.
- To adjust memory and swap accounting: in order to not suffer memory overhead and performance degradation.
- To enable UFW forwarding: if UFW is enabled it is required to properly configure its forwarding policy (if UFW’s enabled it will drop all forwarding and incoming traffic).
- To configure a DNS server: since Docker defaults to using an external DNS nameserver, it cannot use the local one.

It is strongly recommended to check documentation’s web page of the official Docker project site [49] in order to know how these configurations can be done.

Once Docker is properly installed as a daemon in the OS let us focus on interesting possibilities that this technology offers in order to create and manage containers:
### Creating containers

- **Using a Docker file**

  This is the main configuration file for a Docker container set up. It can be seen as an initial script to build a specific docker container. Also, it is like setting up a local git repository to later distribute it, but at a container level instead of software level. There you can define many configuration parameters in order to install and properly configure required dependencies and tools to run inside.

- **Using Docker pull, commit and push for images**

  This is not recommended for creating an original container, but it is appropriate when starting with Docker technology. However, it is recommended to be used once an initial container has been build (from a Docker file) in order to maintain and to distribute different versions and deployments of this (as said, as a higher level git repository). It is important to note that a Docker image consists of a series of layers. Docker makes use of union file systems to combine these layers into a single image (the container itself). Union file systems allow files and directories of separate file systems, known as branches, to be transparently overlaid, forming a single coherent file system. This last fact is a key point due to its capacity of also offering deployment layers inside a container (i.e.: more than one tool/technology inside a container).

### Managing containers

- **Using Docker Hub service**

  It is a public registry of Docker images’ repositories (there are also private ones with specific paying plans) from Docker official site. There is a list of basic (e.g.: CentOS, Ubuntu, Debian, ...) and complex (e.g.: CentOs with Nginx, Ubuntu with Nginx and Wordpress, Debian with Node.js and MongoDB, ...) containers containing clean and/or OS environments.

- **Using Docker Registry and Repository service**

  This item is a key point when looking for local management of images’ repositories. This tool let us deploy an external (e.g.: private) registry if some enhancements over Docker Hub are desired (e.g.: specific user credentials, high level security layer,...).

- **Using Docker Compose**

  This tool let us create a localhost orchestrator of docker containers. This means managing/running more than one container and linking them (if required) at same time from the same point.

For a more specific detailed list of options and possibilities from Docker containers, please check Appendix D.
4.1.1. Basic LMS container

Once previous brief of Docker possibilities to work with has been introduced let us start containerizing a single LiveMediaStreamer.

As shown in the wiki page at GitHub’s LiveMediaStreamer (see Appendix A), the framework has some requirements and dependences that should be previously solved (i.e.: installed). Appendix E Section E.1. lists the first and basic Docker file which installs and configures the image in order to run a LMS instance.

Now, let us focus on what is done in order to explain it better:

- **FROM**: this command indicates which is the image base to be used, in this case, as previously mentioned, Ubuntu 14.04 is the selected environment.
- **MAINTAINER**: this command tags the maintainer/creator of such container image.
- **RUN**: specific command which runs specific bash scripts (e.g.: apt-get, mkdir, adduser and any other available system command from the base image)
- **USER**: this command is used in order to specify the system user that is going to be loaded in such container. This is mainly for security reasons (e.g.: avoiding root user).
- **EXPOSE**: this command handles the ports to be exposed from the container itself. This does not imply that later ports couldn’t be exposed through the command line interface, but it is used in order to list suitable ports to be required. In this case, the exposed ports are a range of UDP ports where streams will be input or outputted (i.e.: RTP), a TCP range for the RTSP and the TCP port for managing the LMS framework through TCP socket messages.
- **CMD**: this configures the command that will be executed when running the image itself. It is important to point out that any user can later enter inside the container avoiding the execution of the default CMD defined (e.g.: executing bash for development purposes inside the container and creating a new container’s version).

There are many other commands that could be used inside a Docker file (see Appendix D) but they are not required in this case.

Once the Docker file is defined it is time to build the image as follows:

```bash
$ docker build \
-t <origin repository registry>/<image name>:<version tag> \
<Docker file folder path>
```

Note that in order to start working with different images and managing them in different environments (i.e.: different servers/computers) a Docker Hub account has been created. So, let us define each parameter in order to name and tag each image to built during this thesis. In this case:

- Origin repository registry is “gerardcl” (see https://hub.docker.com/r/gerardcl)
The last command builds the image by following the defined script inside the Docker file already defined. Finally, the command to execute the image is:

```
$ docker run --rm -p <host port>:<container port> \
--name single-lms gerardcl/lms:single
```

This last command runs the previous defined and built image by exposing internal TCP port (i.e.: \(<container port>\), which has been configured in CMD method with 7777) to another defined TPC port at host side (i.e.: \(<host port>\)). Flag \(--rm\) is used in order to be able to run again the same command by defining (i.e.: identifying) the running image with "single-lms" name (i.e.: \(--name\) flag). Note that for testing purposes flags \(-i -t\) (or \(-it\), it is just the same) might be also added in the command in order to run as an interactive process (this means allocating a TTY for the container process, so exposing the STDIN, STDOUT and STDERR standard streams). Therefore, the effect is similar to executing the process over the same OS too.

Finally, in order to expose other ports it is just required to add the \(-p\) flag as many times as ports required. If an UDP port is also required to be exposed it is required to add the udp tag as shown:

```
$ docker run --rm -p <host port>:<container port> \
-p <stream1 host port>:<stream1 container port>:udp \
--name single-lms gerardcl/lms:single
```

### 4.1.2. HTTP REST API container

The following container to be built is the HTTP REST API middleware developed in Chapter 3. Since it is a Node.js application, it requires to be built with Node.js and the NPM which is the official Node.js package manager. NPM is used to to install middleware's dependencies. This is shown in Appendix E Section E.2..

In this case, in order to build and manage such image, these are the image parameters which define this image:

- Origin repository registry is "gerardcl" (see https://hub.docker.com/r/gerardcl)
- Image name is "lms-rest-api"
- Image version tag is "single"

The building command is:

```
$ docker build \n-t gerardcl/lms-rest-api:single \n<Docker file folder path>
```
Note that the middleware implementation support changing the listening port of the application (default is 8080) by just adding the "-e" flag and defining the "PORT" environment variable of the container. An example is shown:

```bash
$ docker run -it -e "PORT=9000" -p 8080:9000 \
--name lms-rest-api --rm gerardcl/lms-rest-api:single
```

This last command sets the internal "PORT" environment variable to 9000 and binds it to the 8080 port of the host.

### 4.2. Linking containers

In order to play with the previously built containers, it is required to know how they might be interconnected (i.e.: linked).

#### 4.2.1. Same OS

If containers are running on the same OS it is important to keep in mind that each environment is isolated and its network environment is isolated too. Then, in order to let the HTTP REST API container connect to and manage the LMS instance container it is required to share its network environment with the LMS instance container. Here is how each container might be executed:

```bash
$ docker run -it -p 8080:8080 --name lms-rest-api \
--rm gerardcl/lms-rest-api:single

$ docker run -it --rm --net container:lms-rest-api \
--name lms gerardcl/lms:single
```

Flag `--net container:<container id>` helps solving this issue. This configures the "lms" container to use the network environment of the "lms-rest-api" container. So, both containers are on the same localhost, but isolated. The only exposed port is the 8080, which is required to get access to REST API.

To point out that in this case it is not required to expose the TCP socket API port because it is internally linked through the REST API container. This is a key point of the Docker technology, which lets isolate a container from external world.

In order to demonstrate it let us show what a `$ netstat -putaneo` command execution returns:

- Looking for port 8080:

  **Proto LocalAddress ForeignAddress State User PID/Program name**
  tcp6 :::8080 ::::* LISTEN 0 21735/docker-proxy

- Looking for port 7777:

  No result obtained so isolation is reached on port 7777
4.2.2. Separate OS

In order to deploy each container in separate OS it is required to expose required ports for achieve intercommunication. The Docker "run" command might be:

- For LMS container (only exposing TCP socket API control ports):

  $ docker run -it --rm -p 7777:7777 --name lms gerardcl/lms:single

- For REST API container:

  $ docker run -it -p 8080:8080 --name lms-rest-api --rm \
  gerardcl/lms-rest-api:single

Then, in order to reach connection from REST API to LMS the host to set in the connect JSON parameter is the host IP of the OS which is running the LMS instance containerized. And, obviously, in order to play with the REST API and control the remote LMS instance, the host URI must be the OS IP of the running REST API container.

4.3. Running multiple processes within a container

Finally, it is important to note that a Docker image is only able to run a single process through the CMD method. But there is a solution to reach executing more than one process (i.e.: multiple services). This fact will help developing Chapter 5.

The solution for running multiple services inside the same container remains on a common and widely used tool, called supervisord. Supervisord is a client/server system that allows monitoring and controlling any number of processes on UNIX-like operating systems, which is meant to be used to control processes related to a project or a customer (i.e.: a Docker container), and is meant to start like any other program at boot time.

An example for supervisor to be required is due to the fact that LMS framework is able to encapsulate audio and/or video streams to MPEG-DASH\(^2\) segments, which a key point feature of the LMS framework in order to offer live or on demand streaming to browser applications. Therefore, in order to let a browser play obtained segments an HTTP server is required, which serves the required files to be sent to the browser. Appendix E Section E.3. describes the Docker file for building such container, which will have a LMS instance and a HTTP server (i.e.: Nginx).

This is quite similar to the first Docker file introduced in this chapter but:

- Nginx and Supervisord are also installed
- Specific /var/run and /var/log folders are created per each Supervisord process inside the supervisord.conf file

\(^2\)Dynamic Adaptive Streaming over HTTP (DASH), also known as MPEG-DASH, is an adaptive bitrate streaming technique that enables high quality streaming of media content over the Internet delivered from conventional HTTP web servers.
- This Docker file uses the command ADD in order to add specific configuration files for the Nginx and Supervisord container's servers (they are later showcased)

- What is executed now is the Supervisord daemon

- The port for the Nginx server is exposed

- The user that is going to be used for this container is its root user due to require being used by supervisord service itself

The Nginx configuration file is as shown in Appendix E Section E.4. In this configuration file, apart from typical Nginx configurations (which are out of scope of this thesis), some access control methods are added in order to treat common HTTP server issues like CORS (Cross-Origin Resource Sharing). Note that the root file system specified is the one which the LMS instance should use for saving the MPEG-DASH output files too (i.e.: the dasher filter should be configured with this folder).

Finally, the last file to show is the supervisord.conf configuration file:

```
[supervisord]
nodaemon=true

[program:livemediastreamer]
command=/usr/local/bin/livemediastreamer 7777

[program:nginx]
command=/usr/sbin/nginx
```

This supervisord configuration file defines both services to be executed, LiveMediaStream and Nginx. Then, as done with previous Docker files, let us show how it should be built:

```
$ docker build -t gerardcl/lms:dash <Docker file folder path>
```

An example command to run this container’s image is shown next:

```
```

So, this container will serve the MPEG-DASH files on http://host:8090/ and it will also be listening on port 7777 in order to get TCP socket configuration messages and it will be listening on port 5004 in order to receive an audio or a video on that port.

### 4.4. Best practices

As illustrated in the official Docker documentation web site [50], one of the best practices that should be taken into account is to avoid creating complex images and try to split as much as possible them.
For example, the previous section is just an example of a scenario where running multiple services inside a container is a must (i.e.: when there are no other options or the solution becomes much complex or the customer demands it), which should be avoided.

So, let us try to showcase a best practice for solving previous use case. By using the "volumes" feature that Docker offers, this last container with multiple services inside could be splitted into two: one specific container for a Nginx server (with same configuration file as shown before) and the other the single LMS container introduced at the beginning of this chapter.

First of all, let us describe which is the Docker file generated for the Nginx container in Appendix E Section E.5.

Then, the build command might be as follows:

```bash
$ docker build \
-t gerardcl/lms-nginx:single \
<Container file folder path>
```

Finally, let us show in order which should be the commands to execute:

```bash
$ docker run -v <host volume folder>:/home/lms/dashSegments \
-p 7777:7777 -p 5004:5004/udp -it --rm \
--name lmsdash gerardcl/lms:single

$ docker run -it -p 8090:8090 --rm --name nginx \
--volumes-from lmsdash:ro gerardcl/lms-nginx:single
```

With `-v` flag is specified the "host volume folder", which is the host folder shared with the internal folder which LMS container will write the MPEG-DASH files. If not specified, default permissions are read and write. Then, the Nginx container will use the `--volumes-from <container id>:<permissions mode>` flag by specifying the containers from which will share its volume and its file permissions. In this case file permissions are specified to be in read-only mode in order to not avoid file modifications from the server/container side.

So, scalability, performance and control of the set of containers are assured by applying best practices like the last one.

Finally, note that thanks to these illustrations of Docker file configurations there is a step forward to demonstrate this thesis's goals.
CHAPTER 5. MONITORING LAYER

As already discussed, in order to properly manage a cloud computing environment it is strongly required to use monitoring tools in order to gather information of interest by improving the environment itself or by finding out issues and solving them as fast as possible.

This chapter describes how the selected monitoring tools can fit in the architecture that has been proposed and how they can be used. This includes preparing the environment to support Collectd (i.e.: monitoring and gathering) and Graphite (i.e.: storing and presenting) tools.

Such tools are helpful to demonstrate that LiveMediaStreamer framework could be deployed as the core of a real-time media production platform, as shown in Chapter 6).

Note that the fact of creating small and reusable containers is the main goal of this chapter and, of course, the goal of this platform architecture to prototype. And, thanks to the selected monitoring tools, which are lightweight and ease configuration flexibility, this issue might be properly solved.

Figure 5.1 illustrates the proposal of the monitoring architecture, which describes the relationship between different containers and the whole Collectd and Graphite deployment.

![Detailed monitoring architecture](image)

**Figure 5.1: Detailed monitoring architecture**

### 5.1. Monitoring containers

This section is based on how Collectd can be configured and deployed in order to monitor and properly gather the metrics of interest.
5.1.1. From the container point of view

From a container point of view and by following the requirement to build containers as reusable as possible, what is proposed to implement is a container to gather the logged stats from an LMS container. This, as shown in Figure 5.1, implies sharing a Docker volume (as introduced in previous Chapter 4) from LMS container to the Collectd client which is using the tail\(^1\) plugin as an input. Moreover, in order to send specific logged metrics to the Collectd server container it is also using the network\(^2\) plugin as done in previous Collectd client configuration.

First of all, to deploy this Collectd configuration in a container is required. The specific Docker file is shown in Appendix E Section E.6..

Then, we have to specify a bash script to run as CMD. This runs collectd but after envtpl python’s package sets the environment variables:

```
#!/bin/bash
envtpl /etc/collectd/collectd.conf.tpl
collectd -C /etc/collectd/collectd.conf -f
```

Thanks to the envtpl package it is possible to run the container with specific environment variables in order to configure following parameters:

- **LMS NAME**: this will be used to identify the LMS instance in a container which Collectd is monitoring.
- **GRAPHITE HOST**: this is to set the address of the remote/local container where the Collectd server is listening and pushing the metrics inside the Graphite’s tools.
- **GRAPHITE PORT**: this is the port where the Collect server and Graphite’s tools container is listening to.

These parameters are set in the Collectd configuration file as shown in Appendix E Section E.6. (among the specific plugins).

This Collectd configuration example file loads specific system loggers plugins as inputs to be sent through the network plugin to the Collectd server.

Chapter 6 shows an example of use of the Collectd tail plugin by using regular expressions. As shown in Figure 5.1, the tail plugin is listening in an specific folder which is shared through the Docker’s volume functionality with the LMS container, which logs its metrics in the same volume.

Finally, the Docker run command where the specific environment variables are set is:

```
$ docker run -it -e "LMS_NAME=lms" \
-e "GRAPHITE_HOST=<IP address>" -e "GRAPHITE_PORT=25826" \
--rm --name cdc \
-p 25826:25826/udp gerardcl/lms-collectd-client
```

\(^1\)The Tail plugin can be used to “tail” log files. Each line is given to one or more “matches” which test if the line is relevant for any statistics using a POSIX extended regular expression.

\(^2\)The Network plugin can send values to other instances (i.e.: client) and receive values from from other instances (i.e.: server). Which action is taken depends on the configuration.
This will start immediately sending the defined container stats to the Graphite container specified by the environment parameters. The following section showcases the Graphite side to be deployed.

This example of deployed container for isolated Collectd clients is a key point in the general monitoring architecture due to the fact of being easily configurable and reusable.

5.1.2. From the OS point of view

The fact of using Collectd means a wide community behind, which probably have already developed some of the functionalities a project could require. And this is the case: the requirement of monitoring each of the containers that a host OS might have can be solved by configuring already existing plugins or similar tools for Collectd from the Docker community. The selected tool is using the stats API introduced since Docker 1.5 version. The reported container’s stats are:

- Network
- Memory usage
- CPU usage

The plugin is called “collectd-docker” and its documentation can be found in GitHub [59]. But, in order to follow the idea of clustering Collectd, it has been modified in order to use the network plugin instead of using the write graphite plugin. Since both run on the same OS, it requires changing the UDP ports through the metrics are sent by avoiding port binding issues. Moreover, previous Collectd client is also configured as a proxy server by re-configuring the network plugin as shown next:

```xml
<Plugin network>
  Server "{{ GRAPHITE_HOST }}" "{{ GRAPHITE_PORT | default("25826") }}"
  Listen "*" "25827"
  Forward true
  ReportStats true
</Plugin>
```

Therefore, it is not a plugin itself but a Docker container, which lists the docker daemon socket of the system (as proposed in Figure 5.1) and monitors each one of the containers running.

This container implements the same envtpl python package in order to define specific environment parameters such as the collectd server host and port (check its documentation for a detailed configuration explanation). So, in order to showcase how this and the previous Collect client are interconnected, the following Docker run commands are shown:

---

3 The Write Graphite plugin stores values in Carbon, the storage layer of Graphite. It keeps the TCP connection to Carbon open in order to minimize the connection handshake overhead. It buffers the data in a buffer to send many lines at once, rather than generating lots of small network packets. The size of this buffer (1428 bytes) is dimensioned so that the buffer as well as the TCP and IP header fit into one Ethernet frame and can (hopefully) be delivered without fragmentation.
48 Study and proposal of a distributed and scalable real-time media production platform

- Collectd tail client (from the container point of view):

```bash
docker run -it --rm --name cc -v /home/gerardcl/logs:/home/lms/logs \
-p 25826:25826/udp -e "LMS_NAME=lmsAVMixingStats" \
-e "GRAPHITE_HOST=192.168.1.140" \
gerardcl/lms-collectd-client
```

- Collectd docker socket API reader (from the OS point of view):

```bash
docker run -v /var/run/docker.sock:/var/run/docker.sock \
-e GRAPHITE_HOST=127.0.0.1 -e COLLECTD_HOST=lmsOS \
-e COLLECTD_DOCKER_APP=lmsAVMixingStats -e GRAPHITE_PORT=25827 \
-it --rm --name collector --net="container:cc" \
gerardcl/lms-collectd-collector
```

In order to avoid port binding issues, the previous code sets up the Docker collectd collector container to use the same network environment of the Collectd tail client.

### 5.2. Showcasing monitoring

This section focuses on the storage and presentation side of the metrics already logged and gathered. This is proposed to be done within a container built with a Collectd server (network data inputs served to Graphite) and a Graphite system (storing and presenting stats).

Installing and configuring Graphite is not trivial and not as easy as installing and configuring Collectd. This can be seen in the Docker file of the Appendix E Section E.8. in order to build such container, as shown in Figure 5.1.

In this case, specific “collectd” and “graphite” users are created (among other environment configurations as shown). Then, a bunch of different and specific configuration files for graphite are added (i.e.: ADD command) to their specific configuration folders. And, finally, specific command executions for database synchronization (i.e.: sqlite3 [60], which is required for specific features for the Graphite web application) and other final configurations required for Graphite are also done. Regarding Collectd, it is installed in a similar way as previously done but it is now configured as shown in Appendix E Section E.9..

So, here, Collectd is configured as a server by listening from anywhere at the default port for Collectd clustering. Moreover, the write graphite plugin is loaded in order to work as an output to the Graphite’s Carbon tool, which receives the data to be stored in the Whisper RRD of the Graphite installation.

It is required to configure this container to be able to run multiple processes (i.e.: Graphite and Collectd). Therefore, by following the instructions of the previous section regarding how to run multiple processes inside a container in Chapter 4, the Supervisord system is used. This time it is configured by splitting the processes configurations into two parts, Collectd and Graphite. The later configures the Graphite web application and the Graphite’s Carbon cache.
• Collectd:

```
[program:collectd]
user=collectd
directory=/
command=collectd -C /etc/collectd/collectd.conf -f
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s_error.log
```

• Graphite web:

```
[program:graphite-web]
user=graphite
directory=/opt/graphite/webapp/
command=/opt/graphite/env/bin/gunicorn -w 1 -b 0.0.0.0:8080 \ 
    --pythonpath /opt/graphite/webapp/graphite graphite_wsgi
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s_error.log
```

• Carbon cache:

```
[program:carbon-cache]
user=graphite
directory=/
env=PYTHONPATH=/opt/graphite/lib/
command=/opt/graphite/bin/carbon-cache.py --debug start
stdout_logfile=/var/log/supervisor/%(program_name)s.log
stderr_logfile=/var/log/supervisor/%(program_name)s_error.log
```

An important configuration, that must be previously decided and configured by knowing the requirements for why monitoring is required, implies defining the storage schemas, which detail retention rates for storing metrics by using the Round-Robin Database storage type, as mentioned in Chapter 2. So, in order to work over a real-time media production platform it is important to achieve as much time accuracy as possible regarding the specific metrics of interest (e.g.: bandwidth usage, losses, pipeline delays, ...). There are more files regarding the Graphite’s tools configuration, but most of them are set to default values which are the recommended ones.

Before presenting the results, it is important to describe example commands for both Collectd client, and Collectd server and Graphite’s containers:

```
$ docker run -it -e "LMS_NAME=lms" -e "GRAPHITE_HOST=<IP of >" \ 
    --rm --name collectd -p 25826:25826/udp gerardcl/lms-collectd-client
```

```
$ docker run -it --rm --name graphite -p 25826:25826/udp -p 8080:8080 \ 
    gerardcl/lms-collectd-graphite
```
As shown, the collectd (i.e.: Collectd client) container sends to the default port (i.e.: UDP protocol) of the graphite host's container (with Collectd server and Graphite tools). Moreover, the graphite container opens the HTTP port 8080 in order to enable browsers to get access to its web application anywhere.

As conclusion, thanks to previous monitoring technology treated a generic and scalable monitoring layer is achieved.
In order to demonstrate that the LiveMediaStreamer is a suitable tool to be used as the core framework of a cloud real-time media production platform it is required to test how it performs over the cloud.

6.1. Platform deployment

In order to demonstrate how LMS fits the project requirements two scenarios, with different complexity, are deployed.

- Isolated deployments
  The main goal of this deployment is to demonstrate how LMS performs inside a Docker container by comparing its performance in the same OS but without running inside a container (i.e.: system installation).
  In this scenario LMS is configured to act as a transcoder service. This means applying one pipeline per stream type (i.e.: one video and one audio paths).

- Generic deployment scenario
  This scenario aims to showcase a suitable and as much generic as possible cloud real-time media production scenario. LMS is configured to receive eight streams (i.e.: four audio and four video streams), mix them and transmit them through RTP/RTSP.

The environment where the deployments are done is composed of two laptops. The LMS container and the Collectd client container are deployed in laptop described in Table 6.1. The second laptop is a Dual-Core PC with the Graphite container running. Note that the first laptop described has the highest computational cost because all the audio and video processing is done in. The second laptop only stores the statistics, displays the Graphite graphs in a browser, plays the LMS's output streams and transmits a video source (2 Mbps H.264 encoded video stream at 25 fps and at 1280x720 pixels resolution) as input for the LMS.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware type</td>
<td>Sony VAIO laptop</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel core i7-3632QM at 2.20 GHz</td>
</tr>
<tr>
<td>RAM</td>
<td>6 GB (4 + 2) DDR3</td>
</tr>
<tr>
<td>Operating system</td>
<td>XUbuntu 14.04 - 64 bits (x86 64)</td>
</tr>
<tr>
<td>Kernel version</td>
<td>3.13.0-55-generic</td>
</tr>
<tr>
<td>Docker version</td>
<td>1.6.8</td>
</tr>
</tbody>
</table>

As seen, the deployment has not been carried out in any type of specific server or high performance cluster environment. The main goal is to demonstrate flexibility on the deployment (i.e.: in a laptop) and portability of the platform (not only the cloud itself). All of
these characteristics are achieved thanks to the performance of the platform itself and the LMS (the core).

Note that all tests have been carried out in a 1 Gbps local area network with a router with both laptops connected (see Figure 6.1 and Figure 6.5). The measurements have been carried out during 10 minutes and a second of granularity.

### 6.1.1. Isolated deployments

This section compares the performance of the LMS installed on system (i.e.: in the host OS) and the same LMS inside a container. A C/C++ script has been developed, which configures the LMS framework as shown in Figure 6.1. Moreover, in order to test the performance, the pipeline metrics are logged once per second (i.e.: pipeline losses and delay) and gathered by a Collectd client container properly configured. Then the Collectd client sends the data to the Graphite container. Check Appendix E Section E.10. to see an example of how the logging and collecting is implemented for these demonstrations.

![Figure 6.1: Configuration of the scenario for the isolated deployments](image)

The Collectd client container, which reads from the folder where the LMS is logging its metrics, uses the `tail` plugin (previously explained in Chapter 5) with specific regular expressions in order to parse the metrics from the LMS logs.

Both isolated scenarios are the same but one is running the LMS on the system and the other is running the same configured LMS but containerized. The second OS is the one which runs the Docker container with the Collectd and Graphite tools. Moreover, this OS acts as the receiver of the transcoded streams through the RTSP protocol and also acts as the transmitter of the source stream.

Figure 6.2 shows the results. These are mainly focused on the pipeline performance metrics (i.e.: internal LMS performance). The figure describes the CPU usage gathered at the Collectd client side and presented for the Graphite web GUI for both isolated scenarios. Regarding system installation the CPU average usage of the averages given by Graphite
CHAPTER 6. PLATFORM DEPLOYMENT TESTS, AND RESULTS

is around 4,015% and around 4,111% for the containerized one. So, there is not so difference about running system installation or the same application containerized.

Figure 6.2: Isolated scenarios - CPU usage

Figure 6.3 illustrates the average pipeline delay introduced by the LMS system, which in both video and audio cases is almost the same. Regarding video, system installation reaches an average of 216.8 milliseconds and the containerized case reaches an average of 215.9 milliseconds. Regarding audio, system installation reaches an average of 25.6 milliseconds and the containerized reaches an average of 25.2 milliseconds.

Figure 6.3: Isolated scenarios - average pipeline processing time
Regarding pipeline losses, as shown in Figure 6.4, both pipelines within both system installation and containerized scenarios do not introduce any data loss. Therefore, LMS is a good option to work with, not only on system installation but also in a containerized environment in order to be a portable service over a cloud infrastructure.

6.1.2. Generic scenario deployment

This last scenario is a generic and basic example demonstration of audio and video production in a cloud environment. Figure 6.5 illustrates how the scenario is configured.

This demonstration is quite similar to the previous but this time LMS is only configured and running inside a container. This LMS configuration is also a C/C++ script which configures the framework, as shown in Figure 6.5, inside the LMS container, specifically.

In this case what is deployed is an audio and video mixer which receives four audio streams and four video streams encoded with OPUS and H264 codecs, respectively, which are streamed through its standard RTP encapsulation (i.e.: specific payload headers).

Regarding the audio mixing, all input streams are mixed using the logarithmic mixing algorithm in order to not saturate the signal of the resulting audio stream. Regarding the video mixing, the HD inputs (at 1280x720 pixels resolution) are mixed as shown in Figure 6.6. All video inputs are resized (through the pre-resampler filter to the video mixer, see Figure 6.5) to half of its size in order to fit into a resulting HD video stream as shown in Figure 6.5.
Let us focus now on the results regarding the pipeline performance parameters. The CPU usage of the containerized audio and video mixer obtained by averaging the averages, shown in Figure 6.7, is 23.02%.
The audio and video average pipeline delay introduced in this scenario is shown in Figure 6.8. There are two path groups, the "receiver to mixer" paths and the "mixer to transmitter" path (see Figure 6.5).

The average delay introduced for the audio "receiver to mixer" paths averages is 8.6 milliseconds and the audio "mixer to transmitter" path average delay is 23.1 milliseconds. Then, by adding the maximum average path (16.2 ms), a total average value of 39.3 milliseconds of processing time involving the audio pipeline is achieved. Regarding the pipeline's video path, an average of 9.89 milliseconds is obtained by averaging the "receiver to mixer" video paths average processing times. By adding the average of the "mixer to transmitter" path processing time of 67 milliseconds to the maximum average obtained in the "receiver to mixer" path (12.8 ms) a total average of 79.8 milliseconds of delay introduced for the video pipeline is obtained. Therefore, the generic scenario demonstrates that LMS achieves real-time\footnote{Real-time parameters \cite{61} mean a maximum delay of 150 milliseconds between origin and destination.} processing time values.
Figure 6.9: Generic scenario - paths accumulated lost blocks

Figure 6.9(a) illustrates that the audio paths of the accumulated lost blocks remains to zero, meaning that the audio pipeline is not discarding any data at any filter, which is an important fact because losing any byte of audio would mean noticing some effects (i.e.: audio clips).

Then regarding the video "receiver to mixer" paths there aren’t accumulated data losses. But, "mixer to transmitter" path reaches around 52,308 lost data blocks. The fact of having lost data blocks is due to the transitory period of the mixer filter. However, this accumulated losses remains constant, meaning that there are no more data blocks lost.

So, although this scenario is being deployed in an Intel i7 processor laptop, it is able to real-time mix four couples of audio and video streams without issues.

Finally, we want to emphasize that the signal discontinuities that appear in some figures are due to the fact of transmitting origin streams in a pseudo-live mode, which means audio and video loops. Therefore, this noise appears when the origin streams restart.
CHAPTER 7. CONCLUSIONS

This chapter aims to be a corollary of the whole thesis, summarizing the results and conclusions obtained in each one of the topics previously treated, which are related to prototype a cloud real-time media production platform.

The main goals of the thesis have been achieved:

- To develop an HTTP REST API middleware.
- To implement statistics measurement for the LMS framework.
- To create a virtualized environment which is suitable for cloud infrastructures.
- To create a logging, gathering, storing and displaying system for the platform metrics.

Therefore, the obtained results demonstrate that this thesis offers a set of tools that have been tested and prepared for real-time media content production over cloud infrastructures.

7.1. LiveMediaStreamer

As seen in Chapter 6, LiveMediaStreamer is performing properly and seems to be able to fit as the core part for real-time video and audio streams manipulation.

Figures 6.2 and 6.7 illustrates the scalability of the core in terms of stream processing capabilities. This means that if for an audio and video stream treatment (Figure 6.1) the average CPU usage is around the 4%, and for four pairs of audio and video streams treatment (Figure 6.5) the CPU usage is around the 23% of an eight core CPU, then LMS can be considered as a reliable and a scalable core framework. This is also confirmed by the fact of not suffering data losses inside the pipeline (Figures 6.4 and 6.9). However, it is important to note that in the video mixing pipeline there are accumulated losses, which appear due to the fact of synchronizing different streams with different frame rates. Figure 7.1 illustrates a constant index of losses which means that this data losses are not incrementing in time, so the system is stable.

It is important to note that in Figure 7.1 (also in previous Chapter 6 figures) there is some noisy data which are caused by the use of working static files to simulate live audio-video streaming. This means that when a file ends some seconds of discontinuity are noticed.

Related to the LMS framework development plan, it is demonstrated that the selected third party libraries are suitable to continue being used (i.e.: Live555, LibAV/ffmpeg, x264 and OpenCV, among others) because they achieve an acceptable performance (i.e: the results show that the filters where they are implemented do not add critical processing time). And, regarding behaviour debugging tools for the LMS, it is possible to know which filters are critical or which should be improved thanks to the implemented metrics gathering system (i.e.: developers are now able to know how many time a filter takes to process a frame).

It is also important to remark that thanks to the developments done related to metrics gathering and the tools deployed around the LMS, it is easier to detect issues in the future.
Figure 7.1: Generic scenario - derivative function of the video path accumulated lost blocks

For example, Figure 6.8(a) showed that the processing time for the audio "receiver to mixer" path was not working properly due to not cleaning up the queues when the source is sending in a pseudo-live mode (i.e.: a loop, a periodic reinitialization of the audio stream).

Finally, note that the generic scenario, which performs audio and video mixing from multiple inputs, demonstrates that the video pipeline introduces around 80 milliseconds of processing time. However, most of this delay is due to the encoder (i.e.: x264 library) and the mixer (i.e.: OpenCV library) filters (the "mixer to transmitter" path).

7.2. Collectd and Graphite

The selection of these tools for the monitoring layer is a good choice when the goal is to have a lightweight and highly configurable system.

A critical issue about clustering such tools (i.e.: Collectd clustering) is how its bandwidth usage may affect the whole system performance. Figures 7.2 and 7.3, which are a filtered wireshark capture of the UDP stream bandwidth usage from the Collectd client to the Collectd and Graphite server, demonstrates that despite the huge amount of parameters to be sent, the communication protocol is lightweight and is not an issue that the system might suffer from. In the worst case (i.e.: generic scenario) the bandwidth usage average is below 70 kbps.

Figure 7.2: Transcoder scenario - Collectd bandwidth usage (bits per second)
Regarding the Graphite tool it is important to remark its huge amount of features to present and treat the metrics, which help to carry out proper graphical data mining. It is also remarkable its compatibility with external tools in order to implement alarm systems or near real-time actuators over the system which is being monitored. This fact is a key point in order to implement a cloud real-time media production platform service.

Regarding the amount of data to be stored, thanks to the RRD, once the periodicity and granularity of time points are defined the required storage capacity is known and remains constant in time. This is a key point that ensure the system performance will not be affected because of the HDD usage.

There is a last point to highlight about Graphite, which is its capability to offer graphics and specific data through its HTTP API. This means that specific applications are able to display data performance, in many ways, in near real-time.

### 7.3. Docker

Regarding this virtualization system implemented. Docker has demonstrated to be a proper tool to encapsulate each one of the required pieces of the platform in order to develop, test, distribute and play with them.

Moreover, Docker offers to this platform prototype to become a multiplatform tool, which means that any piece of this prototype can be executed in any OS which Docker gives support (i.e.: currently supported OS are Linux/Unix, MacOSX and Windows).

Thanks to this tool, the global platform becomes fast to scale due to its fast start-up, restart and shut-down times. And this is also due to its ease to quickly replicate any containerized application or service, if required.

Despite its strengths, it is not still a mature technology (the company behind, called dot-Cloud, and Docker project itself were born in 2013) but it is evolving really fast. This is demonstrated by the high growth that its community is taking place year by year.

An area to be improved, which should be solved by Docker itself or through external solutions, are security issues. This is mainly due to the fact that when running a containerized LMS this container does not completely isolate LMS from the security considerations of running it directly on the host; in fact it adds more security vulnerabilities. The most important of these is that the Docker daemon's API does not require any kind of authentication. It is important to make sure that there is a good firewall configured to isolate the host machine from outside the host, or it might be prone to external attacks. Docker's bridged networking as well as mounted file system support allows for possible security holes into...
and out of the container itself, but this might happen also with traditional virtualization. Of course, the topic of container security is an extensive subject, but it is an issue to be strongly considered when creating a container in order to guarantee the security of Docker (and LXC).

Despite the aforementioned issues, Docker offers other interesting solutions like direct hardware access (i.e.: non specific GPU drivers nor emulator are required) and security at infrastructure management level (i.e.: fast replication and relocation of the containers when specific containers are running with troubles like external network attacks to the service).

### 7.4. Platform

The platform developed and tested in this project can be considered the tip of the iceberg of a cloud real-time and live media production platform. Therefore, the outcomes of this thesis are specific tools/solutions which are suitable to be used together or separated, which might lead to future implementations of new specific and higher level tools (e.g.: dynamic platform configuration through browser application).

We can conclude the main goals of this project has been achieved and are ready to be used.

In order to comply the premise to be as open as possible, a web site of the LMS framework has been published at [http://livemediastreamer.i2cat.net](http://livemediastreamer.i2cat.net) in order to start building a community of users and developers.

### 7.5. Next steps

New user requirements should be gathered, guiding new developments, but assuring the requirement to offer a set of tools as much configurable as possible.

In order to improve some filter steps a possibility is to work over GPGPUs (General Purpose Graphic Processing Units). In this case, OpenCV is already offering specific APIs to work over CUDA. Such implementations could lead to develop UHD video pipelines.

Regarding the network side, and thanks to the APIs implemented, it seems feasible to analyse possibilities to deploy these thesis tools within SDN environments, which should improve management and resource optimization.

Other possible future lines could be related to implement new codec and new media formats in order to keep up with the codec market.

In order to improve streaming robustness, a FEC implementation is also a point to explore but keeping in mind to optimize the bandwidth and delay (i.e.: processing time) overheads. There are specific and proprietary solutions but we propose to implement the standard described in RFC 5109 (RTP Payload Format for Generic Forward Error Correction). It is also important to support specific synchronization mechanisms (in order to assure accurate streams synchronizations), such as PTP (Precision Time Protocol - IEEE 1588-2008 standard), which is a protocol used to synchronize clocks throughout a computer network.
Finally, in order to offer a complete set of tools for private media production use cases and to maintain specific license agreements (i.e.: IPR - Intellectual Property Rights) it is also proposed to implement DRM (Digital Rights Management) technology.
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAC</td>
<td>Advanced Audio Coding</td>
</tr>
<tr>
<td>AES</td>
<td>Audio Engineering Society</td>
</tr>
<tr>
<td>API</td>
<td>Application Programme Interface</td>
</tr>
<tr>
<td>AVB</td>
<td>Audio Video Bridging</td>
</tr>
<tr>
<td>AVC</td>
<td>Advanced Video Coding</td>
</tr>
<tr>
<td>AVCi</td>
<td>Advanced Video Coding - Intra</td>
</tr>
<tr>
<td>BPP</td>
<td>Bits Per Pixel</td>
</tr>
<tr>
<td>CAPEX</td>
<td>Capital Expenditures</td>
</tr>
<tr>
<td>CIFS</td>
<td>Common Internet File System</td>
</tr>
<tr>
<td>CIM</td>
<td>Common Information Model</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>CORS</td>
<td>Cross-Origin Resource Sharing</td>
</tr>
<tr>
<td>COTS</td>
<td>Commercial Off-The-Shelf</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DASH</td>
<td>Dynamic Adaptive Streaming over HTTP</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database Management System</td>
</tr>
<tr>
<td>DMTF</td>
<td>Distributed Management Task Force</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name Service</td>
</tr>
<tr>
<td>DVMRP</td>
<td>Distance Vector Multicast Routing Protocol</td>
</tr>
<tr>
<td>EBU</td>
<td>European Broadcasting Union</td>
</tr>
<tr>
<td>FHD</td>
<td>Full High Definition</td>
</tr>
<tr>
<td>GPL</td>
<td>GNU General Public License</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HD</td>
<td>High Definition</td>
</tr>
<tr>
<td>HDD</td>
<td>Hard Disk Drive</td>
</tr>
<tr>
<td>HEVC</td>
<td>High Efficiency Video Coding</td>
</tr>
<tr>
<td>HTML</td>
<td>HyperText Markup Language</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IGMP</td>
<td>Internet Group Management Protocol</td>
</tr>
<tr>
<td>IO</td>
<td>Input and Output</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>IPTV</td>
<td>Internet Protocol Television</td>
</tr>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>JPEG</td>
<td>Joint Photographic Experts Group</td>
</tr>
<tr>
<td>JSON</td>
<td>Javascript Object Notation</td>
</tr>
<tr>
<td>JT-NM</td>
<td>Joint Task Force on Networked Media</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
</tr>
<tr>
<td>LAN</td>
<td>Local Area Network</td>
</tr>
<tr>
<td>LGPL</td>
<td>GNU Lesser General Public License</td>
</tr>
<tr>
<td>LMS</td>
<td>LiveMediaStreamer framework</td>
</tr>
<tr>
<td>LTS</td>
<td>Long Term Service</td>
</tr>
<tr>
<td>LXC</td>
<td>Linux Containers</td>
</tr>
<tr>
<td>Acronym</td>
<td>Abbreviation</td>
</tr>
<tr>
<td>---------</td>
<td>--------------</td>
</tr>
<tr>
<td>MIB</td>
<td>Management Information Base</td>
</tr>
<tr>
<td>MOS</td>
<td>Mean Opinion Score</td>
</tr>
<tr>
<td>MPEG</td>
<td>Moving Picture Experts Group</td>
</tr>
<tr>
<td>MPLS</td>
<td>Multiprotocol Label Switching</td>
</tr>
<tr>
<td>MXF</td>
<td>Material Exchange Format</td>
</tr>
<tr>
<td>NAL</td>
<td>Network Abstraction Layer</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Functions Virtualization</td>
</tr>
<tr>
<td>NIST</td>
<td>National Institute of Standards and Technology</td>
</tr>
<tr>
<td>ONF</td>
<td>Open Networking Foundation</td>
</tr>
<tr>
<td>OS</td>
<td>Operating System</td>
</tr>
<tr>
<td>OTT</td>
<td>Over The Top</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>PIM</td>
<td>Protocol Independent Multicast</td>
</tr>
<tr>
<td>PTPv2</td>
<td>Precision Time Protocol version 2</td>
</tr>
<tr>
<td>QEMU</td>
<td>Quick Emulator</td>
</tr>
<tr>
<td>QoE</td>
<td>Quality of Experience</td>
</tr>
<tr>
<td>QoS</td>
<td>Quality of Service</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>SDN</td>
<td>Software-Defined Networking</td>
</tr>
<tr>
<td>SDI</td>
<td>Serial Digital Interface</td>
</tr>
<tr>
<td>SDP</td>
<td>Session Description Protocol</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>SLA</td>
<td>Service Level Agreement</td>
</tr>
<tr>
<td>SMPTE</td>
<td>Society of Motion Picture and Television Engineers</td>
</tr>
<tr>
<td>SNMP</td>
<td>Simple Network Management Protocol</td>
</tr>
<tr>
<td>SVIP</td>
<td>Define and research requirements for Video over IP without SDI encapsulation</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>RFC</td>
<td>Request for Comments</td>
</tr>
<tr>
<td>RRD</td>
<td>Round Robin Database</td>
</tr>
<tr>
<td>RTP</td>
<td>Real-time Transport Protocol</td>
</tr>
<tr>
<td>RTCP</td>
<td>Real-time Control Protocol</td>
</tr>
<tr>
<td>RTSP</td>
<td>Real Time Streaming Protocol</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>ToS/DSCP</td>
<td>Type Of Service / Differentiated Services Code Point</td>
</tr>
<tr>
<td>TSN</td>
<td>Time Sensitive Networks</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>UFW</td>
<td>Uncomplicated Firewall</td>
</tr>
<tr>
<td>UHD</td>
<td>Ultra High Definition</td>
</tr>
<tr>
<td>UHDTV</td>
<td>Ultra High Definition Television</td>
</tr>
<tr>
<td>VC-2</td>
<td>Dirac Pro - Video Codec Level 2</td>
</tr>
<tr>
<td>VDI</td>
<td>Virtual Desktop Infrastructure</td>
</tr>
<tr>
<td>VSF</td>
<td>Video Service Forum</td>
</tr>
<tr>
<td>WBEM</td>
<td>Web-Based Enterprise Management</td>
</tr>
<tr>
<td>WDI</td>
<td>Windows Management Instrumentations</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY


[33] Monitis, part of TeamViewer Family “All-in-one application monitoring platform”. http://www.monitis.com/ 15
[34] New Relic, Inc. “Software analytics done with SaaS”. http://newrelic.com/about


[38] FFmpeg library. Homesite: http://ffmpeg.org/


[41] x265 library. Homesite: https://bitbucket.org/multicoreware/x265


[48] Linux containers project. Homesite: https://linuxcontainers.org/


[53] Collectd project. Homesite: https://collectd.org/


[56] Expressjs project. Homesite: http://expressjs.com/

[57] Ubuntu project. Homesite: http://www.ubuntu.com/

[58] MPEG-DASH project. Homesite: http://dashif.org/about/

[60] SQLite project. Homesite: https://www.sqlite.org/ 48

APPENDICES
APPENDIX A. LIVEMEDIASTREAMER ARCHITECTURE

This appendix points to main links to learn LMS framework basics, which go to the LMS framework official web site developed during August 2015 (http://livemediastreamer.i2cat.net/)

- LMS architecture: http://livemediastreamer.i2cat.net/architecture/
- LMS core API: http://livemediastreamer.i2cat.net/tcpsocketapi/
- LMS REST API: http://livemediastreamer.i2cat.net/apirest/

Finally, next pages contain the LMS architecture site content attached.
Architecture

Table of contents

Introduction

Platform and libraries

Framework layers

Data flow layer

Execution layer

Control layer

Main classes

FrameQueue

Frame

BaseFilter

Reader & Writer

Path

PipelineManager

WorkersPool

Controller

Structures

Filters

Receiver

Transmitter

Video decoder

Video resampler

Video encoder

Video mixer

Audio decoder

Audio encoder
Introduction

LiveMediaStreamer framework has been designed based on two main concepts: modularity and simplicity. Therefore, using it, a wide range of streaming applications can be designed due to its architecture. The main idea is constructing pipelines (paths and/or multipaths) of interconnected modules that process frames. Moreover, its architecture ease addition to support new audio, video, container formats and transmission protocols.

The framework has been designed for Linux platforms using C/C++ language. It's the environment supported by all the underlying libraries:

- Live555 – Internet Streaming Media, Wireless, and Multicast technology, services, & standards
- ffmpeg – A complete, cross-platform solution to record, convert and stream audio and video
- OpenCV – Open source computer vision and machine learning software library
- x264 – Free software library and application for encoding video streams into the H.264/MPEG-4 AVC compression format
- x265 HEVC Encoder – Open source HEVC encoder.
- LAME – High quality MPEG Audio Layer III (MP3) encoder licensed under the LGPL
- Opus – Totally open, royalty-free, highly versatile audio codec
- WebM VPX – VP8/VP9 Codec SDK. A open, royalty-free, media file format designed for the web

LMS enables to use and manage platform resources with full control, which is a critical aspect for time sensitive applications.

Framework layers

In order to keep the software modular and configurable, with a simple TCP socket access API, it has been organized into three layers:

- Data flow layer
- Execution layer
- Control layer

Data flow layer

The software data flow has been designed to follow the pipes and filters design pattern. In computer science a pipeline is known to be a sequence or a chain of several processors (routines, threads, etc.) placed in a manner that the output of each processor is the entrance of the next. Called pipeline thanks to the analogy to a physical pipeline. The information that flows inside a pipeline usually is a stream of records, bytes or bits that is read from a buffer or queue (pipe) by the filter (processor) that after processing it is written to another buffer to keep data flowing. Pipelines are usually implemented in multithreaded environments so it take advantage of parallel programming, so the processors may run in parallel and not in strict sequence. This is a very suitable pattern for multimedia live streaming as in such scenarions the data to process are continuous audio and video flows.

Execution Layer

The execution layer of the software is where the parallelization takes part. Each filter of the pipeline implements a Runnable pure abstract class which represents the interface between the filters and the WorkerPool class (an automated pool of workers, which has automatic management of threads execution). Each filter is automatically assigned inside the worker pool within a list of workers to be executed. Workers are totally independent of the pipeline composition and the filter nature, they only know about Runnables which is a simple interface that runs the processing routine of a single frame of a filter. This way independence between the parallelization and the pipeline configuration is kept. Therefore, this is an implementation of a regular thread-pool pattern where each thread consumes tasks from a task queue.

Keeping the software as a lock free implementation is important as the procedure to signal conditions between threads and awake and sleep threads might be time consuming. It may help to keep optimal usage of the CPU, but it might be waste of time, which is critical for time sensitive applications.

Moreover this approach gives freedom on how the parallelization is done and let to the management layers define the criteria to follow. Again this is important in order to determine bottlenecks and to test different criteria (i.e. audio prioritization).
Control Layer

The control and management of the pipeline prepared to be done using a self-defined TCP socket protocol so it can be managed remotely and easily presented as a cloud service as it can be operated remotely and it is pretty simple to develop web services or applications on top of it without having to actually get into the C++ code. See TCP socket API section for all the details of this protocol. The Controller class is the responsible of dispatching received events to the PipelineManager class, which is the class that has full overview of the system, it has the filters pool and their interconnections (the paths). PipelineManager class, in fact, is where execution and data flow layers meet and the Controller class owns an instance of PipelineManager.

Main classes

In the following figure there is a conceptual diagram of the logical structure of the control and dataflow layers.

In this diagram we can see a video transcoding scenario. In this case, a video is received from the network and decoded. After that, it is encoded in two different ways with different parameters (size, bitrate, framerate). Finally, each encoded video is transmitted independently.

As it can be seen in the diagram there is a receiver and a transmitter as head and tail filters of the pipeline, respectively. Moreover there is a video decoder, which is connected to the receiver. Two video resamplers, each one connected to a video encoder, share the output of the video decoder. Finally, each video encoder is connected to the transmitter.

Three different paths form this pipeline:

The first is defined by the receiver as origin filter and the video decoder as destination filter. In this case, the origin writer must be specified because the receiver could have more than one writer. Regarding destination reader, video decoder only has one reader so it is necessary to specify it (default id for any writer/reader is 1).

The second is defined by the video decoder as origin filter and the transmitter as destination filter. Moreover, there are two mid filters: video resampler and video encoder. It is not necessary to specify the origin writer because video decoder has only one writer. However, transmitter has multiple readers so it is important to specify a destination reader.

The third is similar to the second path. It has the same origin filter and the same origin writer. Although the destination filter is the same, the destination reader must be different (note that the writers can be shared by different paths, unlike readers).

Finally it also can be seen the PipelineManager, which contains all the filters and paths, and the Controller which manages control messages and dispatches the resulting events to the corresponding filters or to the PipelineManager.

The main structure of this approach is designed around the following 10 major classes or structures described in the following subsections:

FrameQueue

This is the pure abstract class that represents buffering structure of the pipeline. It is a circular queue prepared that owns a certain number of pre-allocated frames. It is relevant that frames are pre-allocated and reused all the time during the execution of the program, allocating memory is an expensive operation for the OS, as it implies a complete lock of the whole process memory, so keeping memory allocation and deallocation to the minimum expression is a must in order to optimize time sensitive applications.

Another important aspect in terms of optimization is that the queue must be thread safe. To avoid complex or time consuming mutex operations it have been designed in a simple lock free structure, assuming and forcing in the code single consumer and single producer scenarios.

This class is implemented by: AVFramedQueue (discret audio or video frames), AudioCircularBuffer (continuous raw audio byte array), X264VideoCircularBuffer (specific
for x264 in order to tread NAL units as discrete independent frames).

**Frame**

This is a very simple abstract class that handles byte array of a video or audio frame and some other frame related information, such as timestamp, is planar, sequence number, etc. This class is implemented by: AudioFrame (it adds audio related data, such as sample rate, bits per sample, etc.), VideoFrame (it adds video related data, such as codec, colour space, etc.), X264VideoFrame (it adds NAL units management to VideoFrame).

**BaseFilter**

This is one of the most important classes, is where all the data processing is done. The most relevant elements that contains this class might be list of Readers and the list of Writers (see iOInterface description). Each filter may have from none to many inputs and from none to many outputs. Each input is determined by a single Reader and each output by single Writer. Readers and Writers job is to put to and to take out frames to and from the filter respectively. Each filter implementation must define the routine doProcessFrame, which is the responsible of processing frames given by the readers and to fill the resulting frames given by the writer with the processed data.

Another important element of filters is that they have a priority events queue. This queue stores events requested by the user in order to modify the filter configuration (frame rate, bitrate, etc.). Priority criteria is time based, events may be targeted to take place after certain amount of time (this way it is possible to program events).

Finally, the filters are responsible for connecting themselves, so it implements the connect, which connect a certain Writer from the filter to a Reader from another filter, here is where and when the FrameQueue is created and its frames allocated. Is in this connection process when the Frames and FrameQueues are initiated according to the specific data they will handle (i.e. AudioCircularBuffer for raw audio bytes or an AVFramedQueue with VideoFrames for raw video frames). The same way using disconnect routine destroy the FrameQueue that unifies two filters and destroys allocated frames. It cannot exist a FramedQueue if two filters are not connected.

This class is implemented by each processing filter of the system (audio or video encoders and decoders, receivers, transmitters, mixer, resamplers, etc.). However between each specific filter implementation and the BaseFilter class there are:

- **OneToOneFilter**: a filter limited to a single input stream and single output stream
- **OneToManyFilter**: a filter with a single input stream and several output streams
- **HeadFilter**: this is a filter that represents the origin of one or several streams, none inputs, i.e. the receiver filter is a HeadFilter
- **TailFilter**: this filter is the ending filter of a pipeline, none outputs, i.e. the transmitter filter is a TailFilter
- **ManyToOneFilter**: a filter that has several input streams and a single output stream, i.e. the video mixer filter is a ManyToOneFilter

Finally, it’s important to point out that there are two different types of the Filter class: Regular (main type of filter) and Server (it is an special case for network filters adapted to live555 library needs, for transmitting and receiving RTP streams).

**Reader & Writer**

These are two independent classes, but conceptually tightly related, because they share the FramedQueue object when connected. These are Reader and Writer objects. They are pretty simple and they only encapsulate a FrameQueue object. Each of them contain a reference of a FrameQueue, being a reference to the same object if they are connected with each other. A Writer has the routines to add frames to the queue and the Reader has the routines to read (and mark them as obsolete) frames of the queue.

**Path**

This class is an important object in order to determine the pipeline configuration, filters interconnections and different data paths. As the filters may have from zero to many inputs and outputs the pipeline is likely to present different branches, conjunctions and bifurcations. A Path is just a list for all the filters connected to a same branch, in fact, a Path is defined by an origin filter (HeadFilter or OneToManyFilter), a destination filter (TailFilter or ManyToOneFilter) and a sorted list of all the filters (only OneToOneFilter) sequentially connected between origin and destination filters (see Figure 1). Note that the Path doesn’t own or contain the filters; it just records their ID. The container of all the filters of the whole pipeline is the PipelineManager class. Path only stores the interconnections of a particular branch of the pipeline. The Path becomes handy when having to identify and distinguish different instances of a filter of a kind.

**PipelineManager**

This class as the name suggests is the class that defines the pipeline as a whole, it’s the only object that has all related information to existing filters, paths and their interconnections. Moreover it represents the contact between the data flow, control and execution layers.

In PipelineManager there are three relevant attributes: a list of Paths and a list of Filters. This class implements all the routines needed to create, destroy, assign and connect all these elements.

**WorkersPool**

This is a pure abstract class that is mostly in charge of executing the process method of one or several filters in a dedicated thread. Note that, in general, it’s isolated and unaware of filter type. The workers pool basically contains a list of Runnables and related C++ thread objects. A Runnable is an interface implemented by BaseFilter, which has some basic methods such as processFrame and processEvent in order to process a single frame of the filter. First processEvent is executed (the filter gets reconfigured...
When a Worker starts it runs their specific thread in an infinite loop that keeps running Runnable methods that performs specific filter actions. In order to control data consuming cadence the worker might be defined to process at a certain frame-rate, this way each iteration of its Runnable is determined by the frame-rate. If there is no frame-rate defined the WorkerPool has a best-effort behaviour, trying to consume data as fast as possible.

Controller

The controller is the class that handles the interface to outside world, it is the class that defines the control protocol. The protocol is simple, each received packet by the controller is a JSON or an array of JSONs, each JSON represents an event. There are two types of events, internal event (–ANCHOR–) or filter events (–ANCHOR–). Filter events are dispatched to the specific filter’s events queue and internal events are handled by the PipelineManager. So the controller handles the TCP listening socket and parses incoming events format and consistency.

Structures

Filters

In this section a simple description of the filters that are going to be included in the software is presented. Relevant information such as inputs/outputs, relevant parameters or tips are described using a common structured table to enable information access at a glance.

Receiver

It manages the reception of streams into the system. Supported transport protocols are RTP and RTSP. For each input stream it creates a new writer, identified by the reception port.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Receiver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It manages RTP/RTSP input sessions</td>
</tr>
<tr>
<td>Input</td>
<td>Network</td>
</tr>
<tr>
<td>Output</td>
<td>Number of outputs: N</td>
</tr>
<tr>
<td></td>
<td>Data type: codec audio/video frames</td>
</tr>
<tr>
<td>Parameters</td>
<td>None</td>
</tr>
<tr>
<td>Tips</td>
<td>For each RTP session, it creates a writer identified with the session (see Connecting filters and Path chapters)</td>
</tr>
</tbody>
</table>

Transmitter

It manages the muxing and the transmission of output streams. Supported muxing formats are DASH and MPEG-TS. Supported protocols are RTP/RTSP and HTTP (only for DASH format).

Streams can be transmitted without muxing them into a container using RTP/RTSP.

It is also possible to transmit streams directly over RTP, without using RTSP to manage the session, defining the destination IP and port.

This module can have multiple readers, each one representing a stream. Each stream can be simultaneously muxed in different formats and/or transmitted using different RTP/RTSP sessions.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Transmitter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It manages RTP/RTSP output sessions</td>
</tr>
<tr>
<td>Input</td>
<td>Number of inputs: N</td>
</tr>
<tr>
<td></td>
<td>Data type: coded audio/video frames</td>
</tr>
<tr>
<td>Output</td>
<td>Network</td>
</tr>
<tr>
<td>Parameters</td>
<td>None</td>
</tr>
<tr>
<td>Tips</td>
<td>Each reader can be used in many output sessions at the same time</td>
</tr>
</tbody>
</table>

Video decoder

It decodes video coded frames, outputting raw video frames. It has only one reader and one writer.
Supported codecs are the ones supported by libavcodec (which includes H264, HEVC, VP8 and VP9 decoding support)

Decoded frames size and pixel format are determined by the coded frames.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Video decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It decodes the input video frame, outputting a raw video frame</td>
</tr>
</tbody>
</table>
| Input | Number of inputs: 1  
Data type: coded video frames |
| Output | Number of outputs: 1  
Data type: raw video frames |
| Parameters | None |
| Tips | Supported input codecs: H264, H265 and VP8  
Output frames pixel format and size is defined by the input frame (this module does not resize nor change pixel format, use video resampler for this). |

Video resampler

It resizes and/or changes the pixel format of raw frames, outputting resampled raw frames.

Supported pixel formats are the same supported by libavcodec (e.g. YUV420, RGB24).

Configurable parameters are video output size and pixel format.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Video resampler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It resizes and/or changes the pixel format of the input frame</td>
</tr>
</tbody>
</table>
| Input | Number of inputs: 1  
Data type: raw video frames |
| Output | Number of outputs: 1  
Data type: raw video frames |
| Parameters | Output size, output pixel format, discard period |
| Tips | Supported pixel formats:  

Video encoder

It encodes raw video frames, outputting coded video frames.

Supported codecs are the ones supported by libavcodec (which includes H264, HEVC, VP8 and VP9 encoding support using libx264, libx265 and libvpx).

Configurable parameters are codec, frame rate, GoP length and bitrate.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Video encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It encodes the raw input video frame, outputting coded frame NALUs</td>
</tr>
</tbody>
</table>
| Input | Number of inputs: 1  
Data type: raw video frames |
| Output | Number of outputs: 1  
Data type: H264 coded NALUs |
| Parameters | Framerate, bitrate, GoP |
Tips

Supported output codecs: H264 and H265

Supported input pixel format: YUV420P, YUV422P

Video mixer

It has multiple readers, each one associated to a channel. Its main task is composing a layout using frames of its different channels, outputting mixed frames.

It only supports RGB24 raw frames as input; output frames are also in this pixel format.

Configurable parameters are channel number and layout size. Channel configurable parameters are size, upper left corner position, opacity, layer and enabled/disabled.

It is important to consider that video mixer does not resize its input frames. This means that channel input frames and its size configuration must coincide.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Video mixer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It composes a layout using frames of its different channels</td>
</tr>
<tr>
<td>Input</td>
<td>Number of inputs: N</td>
</tr>
<tr>
<td>Data type: raw video frames</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>Number of outputs: 1</td>
</tr>
<tr>
<td>Data type: raw video frames</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>Channel number, layout size</td>
</tr>
<tr>
<td>Channel: size, upper left corner position, opacity, layer, enabled/disabled</td>
<td></td>
</tr>
<tr>
<td>Tips</td>
<td>Video mixer does not resize channel frames, so input frames size must be equal to channel size configuration (using video resample module)</td>
</tr>
<tr>
<td>Supported input pixel format: RGB24</td>
<td></td>
</tr>
<tr>
<td>Output pixel format: RGB24</td>
<td></td>
</tr>
</tbody>
</table>

Audio decoder

It decodes audio coded frames, outputting raw audio frames. It has only one reader and one writer.

Supported codecs are the ones supported by libavcodec (e.g. OPUS, AAC, MP3). Supported sample formats are the ones supported by libavresample (e.g S16, FLT).

Configurable parameters are output sample rate, sample format and channel number.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Audio decoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It decodes the input coded audio, outputting a raw audio frame</td>
</tr>
<tr>
<td>Input</td>
<td>Number of inputs: 1</td>
</tr>
<tr>
<td>Data type: coded audio frames</td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>Number of outputs: 1</td>
</tr>
<tr>
<td>Data type: raw audio frames</td>
<td></td>
</tr>
<tr>
<td>Parameters</td>
<td>Output sample rate, channels and sample format</td>
</tr>
<tr>
<td>Tips</td>
<td>Supported input codecs: PCM, PCMU, OPUS</td>
</tr>
<tr>
<td>Supported output configuration: 48k, stereo, signed 16bit Planar</td>
<td></td>
</tr>
</tbody>
</table>

Audio encoder

It encodes raw audio samples, grouping them into frames and coding these ones. It has only one reader and one writer.

Supported codecs are the ones supported by libavcodec (e.g. OPUS, AAC, MP3).

Configurable parameters are output codec, sample rate, sample format and channel number. Some of these parameters may be defined by the codec, as long as the number of samples per frame (which is not externally configurable).

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Audio encoder</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It encodes the input frame, outputting the coded frame</td>
</tr>
</tbody>
</table>
### Audio mixer

It mixes raw audio samples (grouped in frames), outputting the resulting mixed samples (grouped frames). It has only one reader and one writer.

It has multiple readers, each one associated to a channel.

It only supports signed 16bit planar (s16p) PCM samples as input; output samples are also in this format.

Configurable parameters are output codec, sample rate, sample format and channel number. Some of these parameters may be defined by the codec, as long as the number of samples per frame (which is not externally configurable).

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Audio mixer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It mixes input channels frames, outputting a mixed audio frame</td>
</tr>
</tbody>
</table>
| Input             | Number of inputs: N  
Data type: raw audio frames |
| Output            | Number of outputs: 1  
Data type: raw audio frames |
| Parameters        | Channel number, master volume, channel volume  
Output sample rate, channels and sample format |
| Tips              | Default output configuration: 48k, stereo, S16P  
Default mixing channels: 8 |

### Demuxer

It is a wrapper of the ffmpeg framework which enables LMS to receive all supported input network protocols by ffmpeg. It is designed as the receiver filter, it is a Head filter.

It receives streams from the configured input URI (i.e.: RTMP, ...) and outputs its media streams to the following filters (usually an audio or a video decoder). So, it has multiple writers each one associated to each output stream.

Configurable parameters are the input URI.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Demuxer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It manages input network sessions</td>
</tr>
<tr>
<td>Input</td>
<td>Network</td>
</tr>
</tbody>
</table>
| Output            | Number of outputs: N  
Data type: codec audio/video frames |
| Parameters        | Input URI |
| Tips              | For each stream associated to input URI, it creates a writer |

### Dasher

It is a Tail filter as it is the transmitter filter also. It is configured with different input streams (audio and video) which are encapsulated to MPEG-DASH.

It has multiple readers, each one associated to a dasher segmenter. Resulting outputs (MPD, init and segments) are stored in previous defined system folder path.

It supports H.264 and H.265 as video input codecs and AAC as audio input codecs.
Configurable parameters are input codecs associated to each input stream which are also associated to a dasher segmenter. The duration of the segments, the maximum number of segments and minimum buffer time are also input configuration parameters. Output parameters are the destination folder and the files base name.

<table>
<thead>
<tr>
<th>Filter name</th>
<th>Dasher</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process frame description</td>
<td>It manages MPEG-DASH stream encapsulation and file creation</td>
</tr>
<tr>
<td>Input</td>
<td>Number of inputs: ( N )</td>
</tr>
<tr>
<td></td>
<td>Data type: coded audio/video frames</td>
</tr>
<tr>
<td>Output</td>
<td>MPEG-DASH files (MPD, init and segments) to be served through an HTTP server</td>
</tr>
<tr>
<td>Parameters</td>
<td>Segment duration, maximum number of segments, minimum buffer time, destination folder and files base name</td>
</tr>
<tr>
<td>Tips</td>
<td>Each reader is associated to a segmenter</td>
</tr>
</tbody>
</table>

**Pipelines**

A pipeline is defined by a group of paths and its associated filters which process media frames. Its head filter is always a receiver or a demuxer, which is capable of capturing network streams and feed different filters with the received data. Its tail filter is always a transmitter or a dasher, which is fed by different filters and is responsible for muxing this data and sending it to the network. Between them, each filter process its input frames and feeds the filter/s connected to it.

Different pipelines correspond to different use cases. For example, a transcoding scenario is described in "ANCHOR". Another example could be a video production scenario, using video mixer and audio mixer filters along with encoders, decoders and resamplers.

**Connecting filters**

Filters need to be connected between them in order to create a useful scenario.

Filters are connected using frame queues. Each filter consumes frames from its input queue/s, processes them and feeds its output queue/s with these processed frames.

Regarding this, there are four types of connection:

- One to one: filter A output is connected to filter B input

- Shared output: filter A output is connected to various filters, which share output frames. This means that the same frame is read by many filters at once concurrently.

- Many to one: many filters are connected to different inputs of the same filter, which must support multiple inputs (e.g. audio mixer, video mixer)

- One to many: many filters are connected to different outputs of the same filter, which must support multiple outputs. The difference between this and "Shared output" connection is that in this case the different outputs represent frames with different information (e.g. a video splitter, where each output represents a part of a video frame).

**Paths**

A path represents a series of one to one connections between filters. It is defined by:

- Origin filter: path head filter ID

- Destination filter: path tail filter ID

- Origin filter output: head filter writer ID (if origin filter supports multiple outputs)

- Destination filter input: tail filter reader ID (if destination filter supports multiple inputs)

- Middle filters: filter IDs between head and tail (head-to-tail ordered). These filters must only have one input and one output.

**Management**

There are two management levels in order to control the whole system at real-time

**TCP API**
The system core works using a TCP socket communication system based on JSON messages exchange. The system is listening to a specific TCP port, waiting for new connections. For each connection, the system waits for a JSON message that can be composed by many event JSONs.

An event JSON message contains:

- **action**: the method/action to be executed
- **filter_id**: the id of the filter associated to the action (0 if internal event)
- **delay**: the execution delay
- **params**: parameters involved in the action

Each event JSON is parsed and an event is created using its information. Events are pushed to the corresponding filter event queue and executed by delay order. However, some events correspond to general management events (internal events), so they cannot be executed by any filter specifically. These events are identified by filter_id and are executed by the PipelineManager.

A JSON describing possible errors (if any) is sent as response.

**HTTP REST API**

The remote management is achieved using a web framework middleware, which manages HTTP REST requests transforming them to TCP socket management messages (core events) to the LMS instance to work with.

A REST request can involve more than one core event. In this case, the system core management message is formed by a list of event JSONs, each corresponding to an event. This way multiple events can be transmitted at once and facilitate higher level events implementations, for instance, there could be a predefined event in the middleware named reduceQuality which could be composed by multiple core events configuring multiple filters such as the Video resampler and the Video encoder.

It’s important to note that the middleware as it implements a RESTful API it is developed using more web appropriate technologies, which are Node.js and the Express.js framework. This is important as the implementation of this middleware becomes easy to develop, maintain and extend with other possible higher level events. So this is really easy to adapt and customize to a specific use case or high level application needs, without even having to modify the core software.

Moreover this middleware is responsible for starting and stopping the core binary.
APPENDIX B. LMS HTTP RESTFUL API

Proposed HTTP RESTful API’s structure for managing an instance of the LiveMediaStream:

- Generic management queries
  - Connect
    Checks if an existing instance of LMS is running and sets the LMS port and LMS host to work with.
    
    POST http://<host>:<port>/api/connect
    JSON
    
    
    "port":<lms-port>,
    "host":"<lms-host>"

  - Disconnect
    Resets the running LMS instance and sets lms-port and lms-host to null in order to connect again to the same or any another LMS instance running.
    
    GET http://<host>:<port>/api/disconnect

  - State
    Gets the state object of the current LMS instance connected to (JSON object with the configured filters and paths).
    
    GET http://<host>:<port>/api/state

  - Create a filter
    Creates a filter (current types are: receiver, transmitter, demuxer, dasher, audioDecoder, audioEncoder, videoDecoder, videoResampler, videoEncoder, audioMixer, videoMixer) with an unique identifier.
    
    POST http://<host>:<port>/api/createFilter
    JSON
    
    "id": filterID,
    "type": "type"

  - Create a path of filters
    Create a path of filters. It interconnects each filter as specified.
    
    POST http://<host>:<port>/api/createPath
    JSON
    
    "id" : pathId,
    'orgFilterId' : orgFilterId,
    'dstFilterId' : dstFilterId,
    'orgWriterId' : orgWriterId,
    'dstReaderId' : dstReaderId,
    'midFiltersIds' : [filterID1, filterID2,...]

  - Remove a path of filters
    Deletes a path of filters. It disconnects each path’s filters.

DELETE http://<host>:<port>/api/path/:path_id

- **Specific filter management queries**
  - **Configure an existing filter**
    
    Configures an existing filter (each filter has its own actions defined, check APPENDIX A)

    PUT http://<host>:<port>/api/filter/:filterID
    JSON
    
    ```json
    {
      "action":"the action",
      "params":{
        "param1":param1,
        "param2":"param2",
        "param3":"param3",
        "param4":true
      }
    }
    ```

    So, with previous API proposal, the whole LMS’s TCP socket API becomes simplified. Moreover, specific responses format to client has been proposed as shown:

    - **Success messages**
      
      It may be a string, bool, array or another object, depending on the request method

      JSON
      
      ```json
      {
        "message": { the incoming message }
      }
      ```

    - **Error messages**
      
      JSON
      
      ```json
      {
        "error": "the error message"
      }
      ```
This appendix is showing concrete algorithm of interest in order to help understanding how application methods work.

C.1. Input network metric method

```c
void SCSSubsessionStats::
periodicStatMeasurement(struct timeval const& timeNow)
{
    unsigned secsDiff = timeNow.tv_sec - measurementEndTime.tv_sec;
    int usecsDiff = timeNow.tv_usec - measurementEndTime.tv_usec;
    double timeDiff = secsDiff + usecsDiff/1000000.0;
    measurementEndTime = timeNow;

    RTPReceptionStatsDB::Iterator statsIter(fSource->receptionStatsDB());
    RTPReceptionStats* stats = statsIter.next(True);
    if (stats != NULL) {
        double kBytesTotalNow = stats->totNumKBytesReceived();
        double kBytesDeltaNow = kBytesTotalNow - kBytesTotal;
        kBytesTotal = kBytesTotalNow;
        double kbpsNow = timeDiff == 0.0 ? 0.0 : 8*kBytesDeltaNow/timeDiff;
        if (kbpsNow < 0.0) kbpsNow = 0.0; // in case of roundoff error
        if (kbpsNow < kbitsPerSecondMin) kbitsPerSecondMin = kbpsNow;
        if (kbpsNow > kbitsPerSecondMax) kbitsPerSecondMax = kbpsNow;

        unsigned totReceivedNow = stats->totNumPacketsReceived();
        unsigned totExpectedNow = stats->totNumPacketsExpected();
        unsigned deltaReceivedNow = totReceivedNow - totNumPacketsReceived;
        unsigned deltaExpectedNow = totExpectedNow - totNumPacketsExpected;
        totNumPacketsReceived = totReceivedNow;
        totNumPacketsExpected = totExpectedNow;

        double lossFractionNow = deltaExpectedNow == 0 ? 0.0 : 1.0 - 
            deltaReceivedNow/(double)deltaExpectedNow;
        if (lossFractionNow < packetLossFractionMin) {
            packetLossFractionMin = lossFractionNow;
        }
        if (lossFractionNow > packetLossFractionMax) {
            packetLossFractionMax = lossFractionNow;
        }
    }
```
C.2. Output network metric method

void ConnRTCPInstance::
periodicStatMeasurement(struct timeval const& timeNow)
{
    unsigned currentNumBytes;
    double currentElapsedTime;

    RTPTransmissionStatsDB::Iterator
    statsIter(fSink->transmissionStatsDB());

    fSink->getTotalBitrate(currentNumBytes, currentElapsedTime);
    avgBitrate = currentElapsedTime == 0 ? 0.0 :
                ((8*currentNumBytes/currentElapsedTime)/1000.0);
    if(minBitrate > avgBitrate) minBitrate = avgBitrate;
    if(maxBitrate < avgBitrate) maxBitrate = avgBitrate;

    RTPTransmissionStats* stats;
    while((stats = statsIter.next()) != NULL){
        SSRC = stats->SSRC();
        packetLossRatio = stats->packetLossRatio();
        if(minPacketLossRatio > packetLossRatio)
            minPacketLossRatio = packetLossRatio;
        if(maxPacketLossRatio < packetLossRatio)
            maxPacketLossRatio = packetLossRatio;

        roundTripDelay = stats->roundTripDelay();
        if(minRoundTripDelay > roundTripDelay)
            minRoundTripDelay = roundTripDelay;
        if(maxRoundTripDelay < roundTripDelay)
            maxRoundTripDelay = roundTripDelay;

        jitter = stats->jitter();
        if(minJitter > jitter) minJitter = jitter;
        if(maxJitter < jitter) maxJitter = jitter;
    }
}
void Reader::measureDelay()
{
    if(lastTs.count() < 0){
        lastTs = frame->getPresentationTime();
    }

    if (lastTs == frame->getPresentationTime()) {
        return;
    }

    timeCounter += frame->getPresentationTime() - lastTs;
    lastTs = frame->getPresentationTime();

    if(timeCounter >= windowDelay){
        avgDelay = delay / frameCounter;
        timeCounter = std::chrono::microseconds(0);
        delay = std::chrono::microseconds(0);
        frameCounter = 0;
    }

    delay += std::chrono::duration_cast<std::chrono::microseconds>(
        std::chrono::system_clock::now() - frame->getOriginTime());
    frameCounter++;
}
APPENDIX D. DOCKER CHEAT SHEET

This is a continually expanded GitHub repository where Docker users contribute with specific usages of the Docker technology:

Docker cheat sheet

Current version of previous Docker cheat sheet is attached within next pages:
Docker Cheat Sheet

NOTE: This used to be a gist that continually expanded. It’s now a github project because it’s considerably easier for other people to edit, fix and expand on Docker using Github. Just click README.md (https://github.com/wsargent/docker-cheat-sheet/blob/master/README.md), and then on the "writing pen" icon on the right to edit.

- Why
- Prerequisites
- Installation
- Containers
- Images
- Registry and Repository
- Dockerfile
- Layers
- Links
- Volumes
- Exposing Ports
- Best Practices
- Tips

Why

"With Docker, developers can build any app in any language using any toolchain. “Dockerized” apps are completely portable and can run anywhere – colleagues’ OS X and Windows laptops, QA servers running Ubuntu in the cloud, and production data center VMs running Red Hat.

Developers can get going quickly by starting with one of the 13,000+ apps available on Docker Hub. Docker manages and tracks changes and dependencies, making it easier for sysadmins to understand how the apps that developers build work. And with Docker Hub, developers can automate their build pipeline and share artifacts with collaborators through public or private repositories.

Docker helps developers build and ship higher-quality applications, faster." -- What is Docker (https://www.docker.com/whatisdocker/#copy1)

Prerequisites
I use Oh My Zsh (https://github.com/robbyrussell/oh-my-zsh) with the Docker plugin (https://github.com/robbyrussell/oh-my-zsh/wiki/Plugins#docker) for autocompletion of docker commands. YMMV.

**Linux**

The 3.10.x kernel is the minimum requirement (http://docs.docker.com/installation/binaries/#check-kernel-dependencies) for Docker.

**MacOS**

10.8 “Mountain Lion” or newer is required.

**Installation**

**Linux**

Quick and easy install script provided by Docker:

```
curl -sSL https://get.docker.com/ | sh
```

If you're not willing to run a random shell script, please see the installation (https://docs.docker.com/installation/) instructions for your distribution.

If you are a complete Docker newbie, you should follow the series of tutorials (http://docs.docker.com/linux/started/) now.

**Mac OS X**

Download and install Docker Toolbox (https://www.docker.com/toolbox) . If that doesn't work, see the installation instructions (http://docs.docker.com/installation/mac/) .

Docker used to use boot2docker, but you should be using docker machine now. The Docker website has instructions on how to upgrade (https://docs.docker.com/installation/mac/#migrate-from-boot2docker) . If you have an existing docker instance, you can also install the Docker Machine (https://docs.docker.com/machine/install-machine/) binaries directly.

Once you've installed Docker Toolbox, install a VM with Docker Machine using the VirtualBox provider:

```
docker-machine create --driver=virtualbox default
docker-machine ls
eval "$(docker-machine env default)"
```

Then start up a container:

```
docker run hello-world
```
That's it, you have a running Docker container.

If you are a complete Docker newbie, you should probably follow the series of tutorials (http://docs.docker.com/mac/started/) now.

**Containers**

Your basic isolated Docker process (http://etherealmind.com/basics-docker-containers-hypervisors-coreos/). Containers are to Virtual Machines as threads are to processes. Or you can think of them as chroots on steroids.

**Lifecycle**

- **docker create** (https://docs.docker.com/reference/commandline/create) creates a container but does not start it.
- **docker run** (https://docs.docker.com/reference/commandline/run) creates and starts a container in one operation.
- **docker stop** (https://docs.docker.com/reference/commandline/stop) stops it.
- **docker start** (https://docs.docker.com/reference/commandline/start) will start it again.
- **docker restart** (https://docs.docker.com/reference/commandline/restart) restarts a container.
- **docker rm** (https://docs.docker.com/reference/commandline/rm) deletes a container.
- **docker kill** (https://docs.docker.com/reference/commandline/kill) sends a SIGKILL to a container.
- **docker attach** (https://docs.docker.com/reference/commandline/attach) will connect to a running container.
- **docker wait** (https://docs.docker.com/reference/commandline/wait) blocks until container stops.

If you want to run and then interact with a container, **docker start**, then spawn a shell as described in Executing Commands (https://github.com/wsargent/docker-cheat-sheet/#executing-commands).

If you want a transient container, **docker run --rm** will remove the container after it stops.

If you want to remove also the volumes associated with the container, the deletion of the container must include the `-v` switch like in **docker --rm -v**.

If you want to poke around in an image, **docker run -t -i <myimage> <myshell>** to open a tty.

If you want to poke around in a running container, **docker exec -t -i <mycontainer> <myshell>** to open a tty.

If you want to map a directory on the host to a docker container, **docker run -v $HOSTDIR:$DOCKERDIR**. Also see Volumes (https://github.com/wsargent/docker-cheat-sheet/#volumes).

If you want to integrate a container with a host process manager (https://docs.docker.com/articles/host_integration/), start the daemon with `-r=false` then use **docker start -a**.

If you want to expose container ports through the host, see the exposing ports section.
Restart policies on crashed docker instances are covered here (http://container42.com/2014/09/30/docker-restart-policies/).

**Info**

- **docker ps** [shows running containers](https://docs.docker.com/reference/commandline/ps).
- **docker logs** [gets logs from container](https://docs.docker.com/reference/commandline/logs).
- **docker inspect** [looks at all the info on a container (including IP address)](https://docs.docker.com/reference/commandline/inspect).
- **docker events** [gets events from container](https://docs.docker.com/reference/commandline/events).
- **docker port** [shows public facing port of container](https://docs.docker.com/reference/commandline/port).
- **docker top** [shows running processes in container](https://docs.docker.com/reference/commandline/top).
- **docker stats** [shows containers’ resource usage statistics](https://docs.docker.com/reference/commandline/stats).
- **docker diff** [shows changed files in the container’s FS](https://docs.docker.com/reference/commandline/diff).

```
docker ps -a  shows running and stopped containers.
```

**Import / Export**

- **docker cp** [copies files or folders between a container and the local filesystem](http://docs.docker.com/reference/commandline/cp).
- **docker export** [turns container filesystem into tarball archive stream to STDOUT](https://docs.docker.com/reference/commandline/export).

**Executing Commands**

- **docker exec** [to execute a command in container](https://docs.docker.com/reference/commandline/exec).

To enter a running container, attach a new shell process to a running container called foo, use: `docker exec -it foo /bin/bash`.

**Images**

Images are just templates for docker containers [https://docs.docker.com/introduction/understanding-docker/#how-does-a-docker-image-work].

**Lifecycle**

- **docker images** [shows all images](https://docs.docker.com/reference/commandline/images).
- **docker import** creates an image from a tarball.
- **docker build** creates image from Dockerfile.
- **docker commit** creates image from a container.
- **docker rmi** removes an image.
- **docker load** loads an image from a tar archive as STDIN, including images and tags (as of 0.7).
- **docker save** saves an image to a tar archive stream to STDOUT with all parent layers, tags & versions (as of 0.7).

**Info**

- **docker history** shows history of image.
- **docker tag** tags an image to a name (local or registry).

Docker image ids are sensitive information and should not be exposed to the outside world. Treat them like passwords.

**Registry & Repository**

A repository is a **hosted** collection of tagged images that together create the file system for a container.

A registry is a **host** -- a server that stores repositories and provides an HTTP API for managing the uploading and downloading of repositories.

Docker.com hosts its own index to a central registry which contains a large number of repositories. Having said that, the central docker registry does not do a good job of verifying images and should be avoided if you’re worried about security.

- **docker login** to login to a registry.
- **docker search** searches registry for image.
- **docker pull** pulls an image from registry to local machine.
- **docker push** pushes an image to the registry from local machine.

**Run local registry**

Registry implementation has an official image for basic
setup that can be launched with docker run -p 5000:5000 registry
Note that this installation does not have any authorization controls. You may use option -P -p 127.0.0.1:5000:5000 to limit connections to localhost only. In order to push to this repository tag image with repositoryHostName:5000/imageName then push this tag.

Dockerfile

The configuration file. Sets up a Docker container when you run docker build on it. Vastly preferable to docker commit. If you use jEdit, I've put up a syntax highlighting module for Dockerfile you can use. You may also like to try the tools section.

Instructions

- .dockerignore
- FROM
- MAINTAINER
- RUN
- CMD
- EXPOSE
- ENV
- ADD
- COPY
- ENTRYPOINT
- VOLUME
- USER
- WORKDIR
- ONBUILD

Tutorial

- Flux7's Dockerfile Tutorial

Examples

- Examples
- Best practices for writing Dockerfiles
- Michael Crosby has some more Dockerfiles best practices / take 2
Best Practices

This is where general Docker best practices and war stories go:

- The Rabbit Hole of Using Docker in Automated Tests
  (http://gregoryszorc.com/blog/2014/10/16/the-rabbit-hole-of-using-docker-in-automated-tests/)
- Bridget Kromhout (https://twitter.com/bridgetkromhout) has a useful blog post on running Docker in production (http://sysadvent.blogspot.co.uk/2014/12/day-1-docker-in-production-reality-not.html) at Dramafever.
- There's also a best practices blog post (http://developers.lyst.com/devops/2014/12/08/docker/) from Lyst.
- Discourse in a Docker Container (http://samsaffron.com/archive/2013/11/07/discourse-in-a-docker-container)

Layers

The versioned filesystem in Docker is based on layers. They're like git commits or changesets for filesystems (https://docs.docker.com/terms/layer/).

Note that if you're using aufs (http://en.wikipedia.org/wiki/Aufs) as your filesystem, Docker does not always remove data volumes containers layers when you delete a container! See PR 8484 (https://github.com/docker/docker/pull/8484) for more details.

Links

Links are how Docker containers talk to each other through TCP/IP ports (https://docs.docker.com/userguide/dockerlinks/). Linking into Redis (https://docs.docker.com/examples/running_redis_service/) and Atlassian (http://blogs.atlassian.com/2013/11/docker-all-the-things-at-atlassian-automation-and-wiring/) show worked examples. You can also (in 0.11) resolve links by hostname (https://docs.docker.com/userguide/dockerlinks/#updating-the-etchosts-file).

NOTE: If you want containers to ONLY communicate with each other through links, start the docker daemon with `-icc=false` to disable inter process communication.

If you have a container with the name CONTAINER (specified by `docker run --name CONTAINER`) and in the Dockerfile, it has an exposed port:

```
EXPOSE 1337
```

Then if we create another container called LINKED like so:
Then the exposed ports and aliases of CONTAINER will show up in LINKED with the following environment variables:

- $ALIAS_PORT_1337_TCP_PORT
- $ALIAS_PORT_1337_TCP_ADDR

And you can connect to it that way.

To delete links, use `docker rm --link`.

If you want to link across docker hosts then you should look at Swarm (http://docs.docker.com/swarm/). This link on stackoverflow (http://stackoverflow.com/questions/21283517/how-to-link-docker-services-across-hosts) provides some good information on different patterns for linking containers across docker hosts.

### Volumes

Docker volumes are free-floating filesystems (http://docs.docker.com/userguide/dockervolumes/). They don’t have to be connected to a particular container. You should use volumes mounted from data-only containers (https://medium.com/@ramangupta/why-docker-data-containers-are-good-589b3c6c749e) for portability.

Volumes are useful in situations where you can’t use links (which are TCP/IP only). For instance, if you need to have two docker instances communicate by leaving stuff on the filesystem.

You can mount them in several docker containers at once, using `docker run --volumes-from`.

Because volumes are isolated filesystems, they are often used to store state from computations between transient containers. That is, you can have a stateless and transient container run from a recipe, blow it away, and then have a second instance of the transient container pick up from where the last one left off.

See advanced volumes (http://crosbymichael.com/advanced-docker-volumes.html) for more details. Container42 is also helpful (http://container42.com/2014/11/03/docker-indepth-volumes/).

For an easy way to clean abandoned volumes, see `docker-cleanup-volumes` (https://github.com/chadoe/docker-cleanup-volumes).

As of 1.3, you can map MacOS host directories as docker volumes (http://docs.docker.com/userguide/dockervolumes/#mount-a-host-directory-as-a-data-volume) through boot2docker:

```
docker run -v /Users/wsargent/myapp/src:/src
```

You can also use remote NFS volumes if you're feeling brave (http://www.tech-d.net/2014/03/29/docker-quicktip-4-remote-volumes/).
You may also consider running data-only containers as described here (http://container42.com/2013/12/16/persistent-volumes-with-docker-container-as-volume-pattern/) to provide some data portability.

**Exposing ports**

Exposing incoming ports through the host container is fiddly but doable (https://docs.docker.com/reference/run/#expose-incoming-ports).

The fastest way is to map the container port to the host port (only using localhost interface) using `-p`:

```
$ docker run -p 127.0.0.1:$HOSTPORT:$CONTAINERPORT --name CONTAINER -t someimage
```

If you don't want to use the `-p` option on the command line, you can persist port forwarding by using `EXPOSE` (https://docs.docker.com/reference/builder/#expose):

```
EXPOSE <CONTAINERPORT>
```

If you're running Docker in Virtualbox, you then need to forward the port there as well, using `forwarded_port` (https://docs.vagrantup.com/v2/networking/forwarded_ports.html). It can be useful to define something in Vagrantfile to expose a range of ports so that you can dynamically map them:

```
Vagrant.configure(VAGRANTFILE_API_VERSION) do |config|
  ...
  (49000..49900).each do |port|
    config.vm.network :forwarded_port, :host => port, :guest => port
  end
  ...
end
```

If you forget what you mapped the port to on the host container, use `docker port` to show it:

```
$ docker port CONTAINER $CONTAINERPORT
```

**Tips**

Sources:

Commit with command (needs Dockerfile)

```bash
alias dl='docker ps -l -q'
docker run ubuntu echo hello world
docker commit 'dl' helloworld
```

Get IP address

```bash
docker inspect 'dl' | grep IPAddress | cut -d '"' -f 4
```

or

```bash
wget http://stedolan.github.io/jq/download/source/jq-1.3.tar.gz
tar xzvf jq-1.3.tar.gz
cd jq-1.3
./configure && make && sudo make install
docker inspect 'dl' | jq -r ".[0].NetworkSettings.IPAddress"
```

or using a go template (https://docs.docker.com/reference/commandline/inspect)

```bash
docker inspect -f "{{ .NetworkSettings.IPAddress }}" <container_name>
```

Get port mapping

```bash
```

Find containers by regular expression

```bash
for i in $(docker ps -a | grep "REGEXP_PATTERN" | cut -f1 -d " "); do echo $i; done
```

Get Environment Settings

```bash
docker run --rm ubuntu env
```

Kill running containers

```bash
docker kill $(docker ps -q)
```

Delete old containers

```bash
docker ps -a | grep 'weeks ago' | awk '{print $1}' | xargs docker rm
```
Delete stopped containers

```bash
docker rm -v `docker ps -a -q -f status=exited`
```

Delete dangling images

```bash
docker rmi $(docker images -q -f dangling=true)
```

Delete all images

```bash
docker rmi $(docker images -q)
```

Show image dependencies

```bash
docker images -viz | dot -Tpng -o docker.png
```

Slimming down Docker containers Intercity Blog

- Cleaning APT
  ```bash
  RUN apt-get clean
  RUN rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/*
  ```

- Flatten an image
  ```bash
  ID=$(docker run -d image-name /bin/bash)
  docker export $ID | docker import - flat-image-name
  ```

- For backup
  ```bash
  ID=$(docker run -d image-name /bin/bash)
  (docker export $ID | gzip -c > image.tgz)
  gzip -dc image.tgz | docker import - flat-image-name
  ```

Monitor system resource utilization for running containers

To check the CPU, memory and network i/o usage, you can use:

```bash
docker stats <container>
```

for a single container or

```bash
docker stats $(docker ps -q)
```
to monitor all containers on the docker host.
APPENDIX E. DOCKER, NGINX AND COLLECTD
CONFIGURATION FILES

This appendix is listing specific Docker, Nginx and Collectd configuration files.

E.1. Basic LMS container

Next Docker file installs latest development commit into the container to be built and prepares the image to run LMS as a service.

# LiveMediaStreamer Container
FROM ubuntu:14.04
MAINTAINER Gerard CL <gerardcl@gmail.com>
RUN apt-get update && apt-get -y upgrade
RUN apt-get -y install git cmake autoconf automake build-essential \ 
    libass-dev libtheora-dev libtool libvorbis-dev pkg-config zlib1g-dev \ 
    libcppunit-dev yasm libx264-dev libmp3lame-dev libopus-dev \ 
    libvpx-dev liblog4cplus-dev libtiniyxml2-dev opencv-data \ 
    libopencv-dev mercurial cmake-curses-gui vim libcurl3 wget curl
RUN adduser --disabled-password --gecos '' lms && adduser lms sudo \ 
    && echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers
USER lms
RUN hg clone https://bitbucket.org/multicoreware/x265 /home/lms/x265 \ 
    && cd /home/lms/x265 && make -j && sudo make install && sudo ldconfig
RUN git clone https://github.com/mstorsjo/fdk-aac.git/ /home/lms/fdk-aac \ 
    && cd /home/lms/fdk-aac && libtoolize && ./autogen.sh \ 
    && ./configure && make -j && sudo make install && sudo ldconfig
RUN cd /home/lms && wget http://ffmpeg.org/releases/ffmpeg-2.7.tar.bz2 \ 
    && tar xjvf ffmpeg-2.7.tar.bz2 && cd ffmpeg-2.7 \ 
    && ./configure --enable-gpl --enable-libass --enable-libtheora \ 
    --enable-libvorbis --enable-libx264 --enable-nonfree --enable-shared \ 
    --enable-libopus --enable-libmp3lame --enable-libvpx \ 
    --enable-libfdk_aac --enable-libx265 && make -j \ 
    && sudo make install && sudo ldconfig
RUN cd /home/lms && wget \ 
    && tar xaf live555-latest.tar.gz && cd live \
RUN git clone https://github.com/ua-i2cat/livemediastreamer.git \
/home/lms/livemediastreamer && cd /home/lms/livemediastreamer \
&& git checkout development && ./autogen.sh \
&& make -j && sudo make install && sudo ldconfig

EXPOSE 5000-5017/udp
EXPOSE 8554-8564
EXPOSE 7777

CMD ["/usr/local/bin/livemediastreamer","7777"]

E.2. HTTP REST API container

Next Docker file installs Node.js and clones latest HTTP REST API for LMS from the GitHub repository. It also prepares the image in order to run the HTTP REST API middleware for the LMS framework.

# LiveMediaStreamer Container
FROM ubuntu:14.04

MAINTAINER Gerard CL <gerardcl@gmail.com>

RUN apt-get update && apt-get -y upgrade

RUN apt-get -y install git npm

RUN adduser --disabled-password --gecos '' lms \
&& adduser lms sudo \
&& echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers

USER lms

RUN cd /home/lms \
&& git clone https://github.com/ua-i2cat/LMStoREST.git \
/home/lms/LMStoREST && cd /home/lms/LMStoREST && npm install

EXPOSE 8080
CMD ["nodejs", "/home/lms/LMStoREST/lms-middleware.js"]
E.3. Running multiple processes within a container

This Docker file installs Nginx and LMS inside the same container in order to be ready to serve through HTTP Nginx server the MPEG-DASH files generated for the LMS framework (it externally requires to be configured as a transcoder to MPEG-DASH). Then, superviورد, which is also installed, runs as the default image process. Supervisord is managing each defined process to be executed.

```bash
# LiveMediaStreamer Container
# and Nginx server for MPEG-DASH streaming
FROM ubuntu:14.04
MAINTAINER Gerard CL <gerardcl@gmail.com>

RUN apt-get update && apt-get -y upgrade
RUN apt-get -y install git cmake autoconf automake build-essential \ 
    libass-dev libtheora-dev libtool libvorbis-dev pkg-config zlib1g-dev \ 
    libcppunit-dev yasm libbx264-dev libmp3lame-dev libopus-dev \ 
    libvpx-dev liblog4cplus-dev libtinyxml2-dev opencv-data \ 
    libopencv-dev mercurial cmake-curses-gui vim libcurl3 wget curl

RUN apt-get -y install nginx supervisor
RUN mkdir -p /var/lock/nginx /var/run/nginx \ 
    /var/lock/livemediastreamer /var/run/livemediastreamer \ 
    /var/log/supervisor
ADD ./nginx.conf /etc/nginx/nginx.conf
ADD ./supervisord.conf /etc/supervisor/conf.d/supervisord.conf

RUN adduser --disabled-password --gecos '' lms \ 
    && adduser lms sudo \ 
    && echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers

RUN mkdir -p /home/lms/dashSegments

USER lms

RUN hg clone https://bitbucket.org/multicoreware/x265 /home/lms/x265 \ 
    && cd /home/lms/x265 && cmake -G "Unix Makefiles" ./source \ 
    && make -j && sudo make install && sudo ldconfig

RUN git clone https://github.com/mstorsjo/fdk-aac.git/ /home/lms/fdk-aac \ 
    && cd /home/lms/fdk-aac && libtoolize && ./autogen.sh \ 
    && ./configure && make -j && sudo make install && sudo ldconfig

RUN cd /home/lms && wget http://ffmpeg.org/releases/ffmpeg-2.7.tar.bz2 \ 
    && tar xjvf ffmpeg-2.7.tar.bz2 && cd ffmpeg-2.7
```
&& ./configure --enable-gpl --enable-libass --enable-libtheora 
--enable-libvorbis --enable-libx264 --enable-nonfree --enable-shared 
--enable-libopus --enable-libmp3lame --enable-libvpx 
--enable-libfdk_aac --enable-libx265 && make -j 
&& sudo make install && sudo ldconfig

RUN cd /home/lms && wget \
&& tar xaf live555-latest.tar.gz && cd live \
&& ./genMakefiles linux-with-shared-libraries && make -j \
&& sudo make install && sudo ldconfig

RUN git clone https://github.com/ua-i2cat/livemediastreamer.git \\
/home/lms/livemediastreamer && cd /home/lms/livemediastreamer \
&& git checkout development && ./autogen.sh \\
&& make -j && sudo make install && sudo ldconfig

USER root

EXPOSE 5000-5017/udp
EXPOSE 8554-8564
EXPOSE 7777
EXPOSE 8080

CMD ["/usr/bin/supervisord"]

E.4. Nginx server file configuration example

This is a basic example to configure the Nginx server in order to server the files generated for the LMS. It is not recommended to define servers in the same nginx.conf file (servers should be defined through the sites available/enabled directories) but this is done that way in order to demonstrate and example a complete but basic Nginx server configuration.

# this sets the user nginx will run as,
# and the number of worker processes
user nobody nogroup;
worker_processes 1;

# setup where nginx will log errors to
# and where the nginx process id resides
error_log /var/log/nginx/error.log;
pid /var/run/nginx.pid;

events {
    worker_connections 1024;
    # set to on if you have more than 1 worker_processes
    accept_mutex off;
http {
  include /etc/nginx/mime.types;

  default_type application/octet-stream;
  access_log /tmp/nginx.access.log combined;

  # use the kernel sendfile
  sendfile  on;
  # prepend http headers before sendfile()
  tcp_nopush  on;

  keepalive_timeout  5;
  tcp_nodelay  on;

  gzip  on;
  gzip_vary;
  gzip_min_length 500;

  gzip_disable "MSIE [1-6]\.(?!.*SV1)";
  gzip_types text/plain text/xml text/css
      text/comma-separated-values
      text/javascript application/x-javascript
      application/atom+xml image/x-icon;

  # configure the virtual host
  server {
    # replace with your domain name
    server_name localhost;
    root /home/lms/dashSegments;
    # port to listen for requests on
    listen 8090;
    # maximum accepted body size of client request
    client_max_body_size 4G;
    # the server will close connections after this time
    keepalive_timeout 5;

    add_header Access-Control-Allow-Origin "*";
    add_header Access-Control-Allow-Methods "GET, OPTIONS";
    add_header Access-Control-Allow-Headers "origin, authorization, accept";
    add_header Cache-Control no-cache;

    location / {
      add_header Access-Control-Allow-Origin "*";
      add_header Access-Control-Allow-Methods "GET, OPTIONS";
      add_header Access-Control-Allow-Headers "origin, authorization, accept";
      add_header Cache-Control no-cache;
    }
  }
}
E.5. Sharing volumes within containers

This container installs Nginx and runs it as default image command. This is an isolated Nginx server container which is serving the LMS files through the shared volumes between the basic LMS container and this one.

```
FROM ubuntu:14.04
MAINTAINER Gerard CL <gerardcl@gmail.com>

RUN apt-get update && apt-get install --fix-missing && apt-get -y upgrade

RUN apt-get -y install nginx

ADD ./nginx.conf /etc/nginx/nginx.conf

RUN adduser --disabled-password --gecos '' lms && adduser lms sudo && echo '%sudo ALL=(ALL) NOPASSWD:ALL' >> /etc/sudoers

RUN mkdir -p /home/lms/dashSegments

EXPOSE 8090

CMD ["/usr/sbin/nginx"]
```

E.6. Collectd client container

This is a basic Collectd client container build.

```
FROM ubuntu:14.04
MAINTAINER Gerard CL <gerardcl@gmail.com>

ENV DEBIAN_FRONTEND noninteractive

RUN apt-get update
RUN apt-get -y install collectd curl python-dev python-pip
ADD collectd.conf.tpl /etc/collectd/collectd.conf.tpl

RUN pip install envtpl
ADD start_container /usr/bin/start_container
RUN chmod +x /usr/bin/start_container
CMD start_container

E.7. Collectd client configuration

This is a basic Collectd client configuration template file.

Hostname "{{ LMS_NAME }}"
FQDNLookup true

Interval 1
Timeout 4
ReadThreads 5

LoadPlugin syslog
LoadPlugin cpu
LoadPlugin load
LoadPlugin memory
LoadPlugin network

<Plugin "syslog">
    LogLevel "info"
    NotifyLevel "OKAY"
</Plugin>

<Plugin network>
    Server "{{ GRAPHITE_HOST }}" "{{ GRAPHITE_PORT | default("25826") }}"
    ReportStats true
</Plugin>

E.8. Collectd server and Graphite container

This Docker file installs and configures all the dependences required to run a basic Graphite system which is served by a Collectd server which listens from external Collectd container-ized clients.

# LiveMediaStreamer Container
FROM ubuntu:14.04
MAINTAINER Gerard CL <gerardcl@gmail.com>

RUN apt-get update
RUN apt-get install -y python-cairo collectd-core libgcrypt11 \ python-virtualenv build-essential python-dev supervisor sudo

RUN adduser --system --group --no-create-home collectd \ && adduser --system --home /opt/graphite graphite

RUN sudo -u graphite virtualenv --system-site-packages `graphite/env

RUN echo "django \n python-memcached \n django-tagging \n twisted \n gunicorn \n whisper \n carbon \n graphite-web" > /tmp/graphite_reqs.txt

RUN sudo -u graphite HOME=/opt/graphite /bin/sh -c "/bin/bash --no-input && pip install -r /tmp/graphite_reqs.txt"

ADD collectd/collectd.conf /etc/collectd/\nADD supervisor/ /etc/supervisor/conf.d/\nADD graphite/settings.py /opt/graphite/webapp/graphite/\nADD graphite/local_settings.py /opt/graphite/webapp/graphite/\nADD graphite/mkadmin.py /opt/graphite/webapp/graphite/\nADD graphite/storage-schemas.conf /opt/graphite/conf/

RUN cp /opt/graphite/conf/carbon.conf.example \ /opt/graphite/conf/carbon.conf\nRUN cp /opt/graphite/conf/graphite.wsgi.example \ /opt/graphite/webapp/graphite/graphite_wsgi.py\nRUN cp /opt/graphite/conf/graphite.wsgi.example \ /opt/graphite/conf/graphite.wsgi\nRUN cp /opt/graphite/conf/storage-aggregation.conf.example \ /opt/graphite/conf/storage-aggregation.conf

RUN sed -i "s#\((SECRET_KEY = \(.*\))#"'python -c 'import os; import base64; \ print(base64.b64encode(os.urandom(40)))'"#" \ /opt/graphite/webapp/graphite/app_settings.py\nRUN sudo -u graphite HOME=/opt/graphite PYTHONPATH=/opt/graphite/lib/ \ /bin/sh -c "cd ~/webapp/graphite && ~/env/bin/python manage.py syncdb --noinput"

EXPOSE 8080 25826/udp

CMD exec supervisord -n
E.9. Collectd server configuration

This is a Collectd server configuration example.

Hostname "localhost"
FQDNLookup true
Interval 1

LoadPlugin syslog
LoadPlugin network
LoadPlugin write_graphite

<Plugin syslog>
LogLevel "info"
NotifyLevel "OKAY"
</Plugin>

<Plugin network>
Listen "*" "25826"
ReportStats true
</Plugin>

<Plugin write_graphite>
<Node "graphing">
Host "localhost"
Port "2003"
Protocol "tcp"
LogSendErrors true
Prefix "collectd."
StoreRates true
AlwaysAppendDS false
EscapeCharacter "_"
</Node>
</Plugin>

E.10. Collectd tail plugin example configuration

Next tail plugin configuration is performing specific regular expression matching to bind logged metrics from the LMS demo script. Each logged line to stdout is formatted as shown next:

|PATHID|value|avgDelay-5004|value|lostBlocks-5004|value|PATHID|value|avgDelay-5006|value|lostBlocks-5006|value|RXmediaType|video
|videoRXavgBitRateInKbps|value|videoRXavgPacketLossPercentage|value|videoRXavgInterPacketGapInMiliseconds|value|videoRXcurJitterInMicroseconds|value|RXmediaType|audio
|audioRXavgBitRateInKbps|value|audioRXavgPacketLossPercentage|value|RXmediaType|audio
Then, each new line is parsed through the tail plugin of the Collectd configuration. An example is shown next:

```xml
<Plugin tail>
<File "/home/lms/logs/lms.log"/>
Instance "lmsTest"
<Match>
Regex ".*avgDelay-5004\|([0-9]*)\.*" DSType "CounterAdd"
Type counter
Instance "VideoAVGdelay-us"
</Match>
<Match>
Regex ".*lostBlocks-5004\|([0-9]*)\.*" DSType "CounterAdd"
Type counter
Instance "VideoLostBlocks"
</Match>
<Match>
Regex ".*avgDelay-5006\|([0-9]*)\.*" DSType "CounterAdd"
Type counter
Instance "AudioAVGdelay-us"
</Match>
<Match>
Regex ".*lostBlocks-5006\|([0-9]*)\.*" DSType "CounterAdd"
Type counter
Instance "AudioLostBlocks"
</Match>
<Match>
Regex ".*RXmediaType\|video.*videoRXavgBitRateInKbps\|([0-9]*)\.*" DSType "CounterAdd"
Type counter
Instance "videoRXavgBitRateInKbps"
</Match>
<Match>
Regex ".*RXmediaType\|video.*videoRXavgPacketLossPercentage\|([0-9]*)\.*" DSType "CounterAdd"
Type counter
Instance "videoRXavgPacketLossPercentage"
```
<Match>
Regex ".*RXmediaType\|video.*videoRXavgInterPacketGapInMilliseconds\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "videoRXavgInterPacketGapInMilliseconds"
</Match>

<Match>
Regex ".*RXmediaType\|video.*videoRXcurJitterInMicroseconds\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "videoRXcurJitterInMicroseconds"
</Match>

<Match>
Regex ".*RXmediaType\|audio.*audioRXavgBitRateInKbps\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "audioRXavgBitRateInKbps"
</Match>

<Match>
Regex ".*RXmediaType\|audio.*audioRXavgPacketLossPercentage\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "audioRXavgPacketLossPercentage"
</Match>

<Match>
Regex ".*RXmediaType\|audio.*audioRXavgInterPacketGapInMilliseconds\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "audioRXavgInterPacketGapInMilliseconds"
</Match>

<Match>
Regex ".*RXmediaType\|audio.*audioRXcurJitterInMicroseconds\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "audioRXcurJitterInMicroseconds"
</Match>

<Match>
Regex ".*TXavgBitrateInKbps-1\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "VideoTXavgBitrateInKbps"
</Match>

<Match>
Regex ".*TXpacketLossRatio-1\|([0-9]*)\.*"
DSType "CounterAdd"
Type counter
Instance "VideoTXpacketLossRatio"
Regex ".*TXjitterInMicroseconds-1\|([0-9]*)\.*" 
DSType "CounterAdd"
Type counter
Instance "VideoTXjitterInMicroseconds"
</Match>

<Match>
Regex ".*TXroundTripDelayMilliseconds-1\|([0-9]*)\.*" 
DSType "CounterAdd"
Type counter
Instance "VideoTXroundTripDelayMilliseconds"
</Match>

<Match>
Regex ".*TXavgBitrateInKbps-2\|([0-9]*)\.*" 
DSType "CounterAdd"
Type counter
Instance "AudioTXavgBitrateInKbps"
</Match>

<Match>
Regex ".*TXpacketLossRatio-2\|([0-9]*)\.*" 
DSType "CounterAdd"
Type counter
Instance "AudioTXpacketLossRatio"
</Match>

<Match>
Regex ".*TXjitterInMicroseconds-2\|([0-9]*)\.*" 
DSType "CounterAdd"
Type counter
Instance "AudioTXjitterInMicroseconds"
</Match>

<Match>
Regex ".*TXroundTripDelayMilliseconds-2\|([0-9]*)\.*" 
DSType "CounterAdd"
Type counter
Instance "AudioTXroundTripDelayMilliseconds"
</Match>
</File>
</Plugin>
APPENDIX F. SOURCE CODES

Full source codes can be found in the GitHub’s web page of the Media Internet Area’s developers team of the i2CAT Foundation:

- **HTTP REST API**
  This is the source code repository for the RESTfull API middleware.
  https://github.com/ua-i2cat/LMtoREST

- **LiveMediaStreame framework**
  This is the source code repository for the LiveMediaStreamer framework.
  https://github.com/ua-i2cat/liveMediaStreamer

  Concretely:

  - **Statistics for network inputs**
  - **Statistics for network outputs**
  - **Statistics for pipeline metrics**
    https://github.com/ua-i2cat/liveMediaStreamer/blob/development/src/I0Interface.cpp
APPENDIX G. EXCHANGED E-MAILS WITH LIVE555 DEVELOPER MAILING LIST

This appendix is showing the e-mail conversation I had with the CEO and CTO of the Live555 library, which helped a lot for developing a proper solution to gather the Live555 statistics (i.e.: network statistics).

E-mail sent:

Sender: Gerard Castillo Lasheras <gerard.castillo@i2cat.net>
Receiver: LIVE555 Streaming Media - development & use <live-devel@ns.live555.com>

Hi Ross,

I’m implementing statistics on our software (liveMediaStreamer framework) and I’d like to have access to the RTPTransmissionStatsDB. But, I do not see how to get the RTPSink object (which has the RTPTransmissionStatsDB and its stats).

Which should be the proper way to get the RTPSink object related to my OnDemandServerMediaSubsession childs? I’ve seen that OnDemandServerMediaSubsession has a friend classe StreamState which has the RTPSink associated but, anyway, I’m not able to have access to it.

Thanks in advance,

Kind regards,

The reply:

Sender: Ross Finlayson <finlayson@live555.com>
Receiver: LIVE555 Streaming Media - development & use <live-devel@ns.live555.com>

First of all, note that while a "OnDemandServerMediaSubsession" object refers to a track of streamable media, a "RTPSink" object refers to a receiving client (or possibly multiple clients if "reuseFirstSource" is True). So there’s (in general) a one-to-many relationship between "OnDemandServerMediaSubsession" and "RTPSink". Thus, it doesn’t make sense to talk about *the* RTPSink object for your "OnDemandServerMediaSubsession".

However...
There are at least two possible ways to get access to the "RTPSink" objects:

1/ Note the pure virtual function "createNewRTPSink()" that you
have implemented in your "OnDemandServerMediaSubsession" subclass. You can use your implementation of this function to get access to the "RTPTransmissionStatsDB" for the new "RTPSink", after you’ve created it.

The drawback of this approach, though, is that you don’t know when the "RTPSink" object later gets deleted, so - if you’re not careful - you may end up holding a reference or pointer to a "RTPTransmissionStatsDB" that has been deleted.

2/ Define a subclass "myRTCPInstance" of the "RTCPInstance" class. Then, in your "OnDemandServerMediaSubsession" subclass, reimplement the "createRTCP()" virtual function to create a "myRTCPInstance" object, rather than a "RTCPInstance" object. Note that "createRTCP()" contains a "sink" parameter, pointing to a "RTPSink", from which you can get the "RTPTransmissionStatsDB".

The advantage of this approach over approach 1/ is that - by defining a subclass of "RTCPInstance", you can learn when the "RTPInstance" object gets deleted, and thus when the "RTPSink" object gets deleted. (The "RTCPInstance" object always gets deleted immediately before the "RTPSink" object.) Thus, you can use your "myRTCPInstance" destructor to figure out when the "RTPTransmissionStatsDB" should no longer be used.