

POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master Thesis

Integration of Timed Metadata in the HTTP Streaming Architecture



Supervisor

prof. Antonio Servetti

Candidate

Gabriele Ghibaudo

ACADEMIC YEAR 2019 - 2020

Contents

Introduction	4
HTTP Streaming	5
1.1 HTTP Streaming Architecture	6
1.2 The HLS Protocol	8
1.3 Nginx HLS Media Server	13
1.4 Module Configuration	14
1.5 Low Latency HLS	16
1.6 Content Publishing	19
Nginx-rtmp Module	21
2.1 Mpeg TS Format	22
2.2 How to Store Metadata in MPTS files	27
2.3 How nginx-rtmp Handles Audio Packets	30
2.4 Integration of Timed Metadata	36
Parsing of AMF packets	37
Generation of ID3 Tag	40
Generation of PES packet	41
2.5 Integration of Packed Audio Segments	43
HLS Playlist Files	43
Integration in the nginx-rtmp Module	44
Configuration	46
3.1 id3v2lib library modifications	47
3.2 Building the server	48
3.3 GStreamer Updates	49
3.4 Testing	51
3.5 HLS Development Resources	52
Future Improvements	53
4.1 Transcoding with FFmpeg	54
4.2 Low Latency Extension	54
Conclusions	57
Bibliography	59

List of Figures

1	HTTP Streaming Architecture	6
2	Streaming Protocols Comparison	8
3	Playlist Files Structure	9
4	Bitrate Switch During Stream	12
5	Nginx-rtmp configuration file	14
6	Cmaf File Components	18
7	Gstreaner Pipeline Example	19
8	MPEG TS Header	22
9	TS Packets Sequence	23
10	PSI Header Fields	24
11	PAT Table Fields	24
12	PMT Table Fields	25
13	AMF Metadata Packet Capture	36

Introduction

The nginx-rtmp module is a popular extension of the Nginx web server used for HLS streaming, a last-mile distribution protocol for multimedia content based on HTTP. While proprietary solutions usually come with native support for timed metadata describing the audiovisual data or signaling the presence of advertisements, open-source libraries lack the integration of this feature. This work aims at filling this gap by implementing the Apple metadata specification in the mentioned module.

The thesis will start with a general overview of HTTP based streaming protocols, followed by a detailed description of the HLS standard and the latest trends in the industry. The focus will then move to the internals of nginx-rtmp and the structure of the MPEG TS container format, used for the encapsulation of the encoded content. With this knowledge, we will look at the requirements for the integration of timed metadata in the module and what modifications to the code base were necessary to achieve our goal.

We will go through the steps of configuration and testing of the improved streaming chain and conclude with possible future improvements that could benefit the module and the open source ecosystem.

Chapter 1

HTTP Streaming

1.1 HTTP Streaming Architecture

It is clear that the way we consume content today has shifted to the streaming model. Despite the need for a persistent internet connection, faster play times and less required storage have led to the widespread adoption of this technology. The variety of protocols available covers the most disparate use cases and scenarios, from live event coverage to near real-time ultra low-latency streaming, and is subject to continuous improvements both by the ideators and the open-source community.

Unreliable communication infrastructure and an immense assortment of digital devices spurred the development of the Adaptive Bitrate (ABR) streaming model. The name of this family of protocols comes from their ability to dynamically adapt to variable network conditions and change on the fly to a more convenient version, encoded at a different bitrate. This operation is carried automatically by the client software and can be done in the middle of a stream, without playout delays or stallings.

To set up a Live Streaming service, the elements to be taken in consideration are multiple, starting from the content production up until the distribution to end users. Different protocols work together at the intersection of the various components and must be selected with care between the pool of developed solutions. This is the architecture of a full HTTP streaming chain.

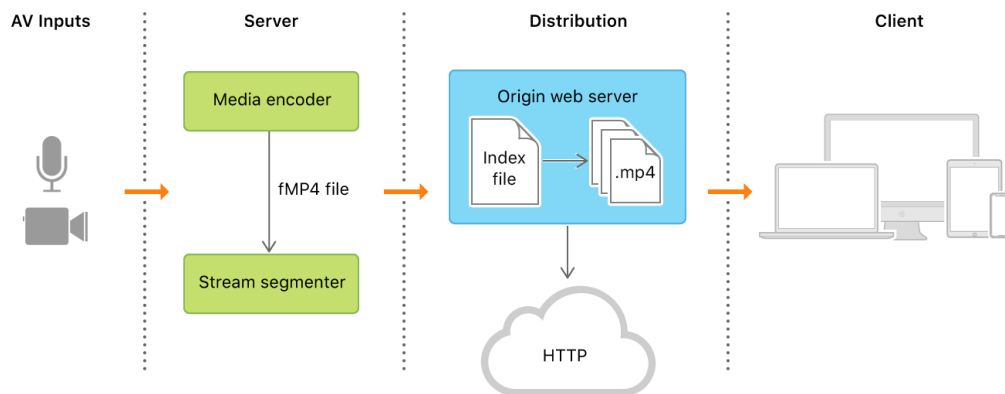


Figure 1: HTTP Streaming Architecture

On the left side, where the production of the actual content occurs, there is a first encoding step, usually done locally. Once ready to be uploaded, the

data is encapsulated in packets and pushed to the media server.

The most common protocol for this part is the TCP based RTMP (Real Time Messaging Protocol). It was once used for the full streaming stack, thanks to the wide diffusion of the FLASH Player on client devices. With the deprecation of this technology, RTMP has left room for newer standards, but still remains the predominant solution for content publishing.

On the server, two main operations take place:

- **Transcoding**, the process of uncompressing encoded data, audio or video, and reencoding it with different parameters, for example to change its bitrate. Also, a completely distinct codec could be used to recompress the data, for example when reformatting an H.264 video to HEVC.
- **Transmuxing**, which refers instead to the restructuring of the bitstream without modification to the media itself, but only on the wrapping container format, as in the case of the move from RTMP to HLS.

The high level components in charge of these two operations are the **Media Encoder** and the **Stream Segmenter**.

For an adaptive bitrate protocol to work effectively, the content needs to be encoded at different bitrates so that the client will have the option to select the more appropriate for the situation. The **Encoder** transcodes the single input bitstream into more renditions, the number of which depends on the circumstances and the server configuration.

This output is then fed to the **Segmenter**, that will take care of splitting the continuous stream in smaller chunks and save them to file. The fragmentation allows a player to move between the available media variants at precise points in time, with a high level of granularity.

The audiovisual content, processed and ready to be distributed, appears as a common HTTP resource and can be exposed to the internet with a web server like Nginx or Apache. Moreover, it can also take advantage of consolidated technologies developed for the underlying protocol, like CDNs, or Content Delivery Networks, which allow the caching of the media to a location closer to the final user and a consequently improved experience.

1.2 The HLS Protocol

From its inception at Apple, HLS, or HTTP Live Streaming, has taken the lead in the pool of the so-called adaptive bitrate protocols, shining for adoption and implementations in respect to DASH and Microsoft Smooth Streaming. As shown in the Wowza 2019 Video Streaming Latency Report [13], this is the industry adoption percentage (only ABR protocols displayed here)

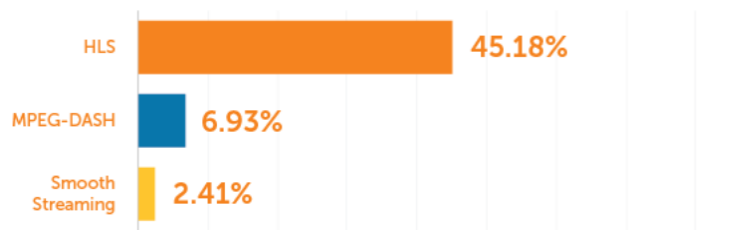


Figure 2: Streaming Protocols Comparison

Its advantage comes from the spread support on the client side: with native reproduction in almost all web browsers on Android and iOS devices, it can be easily implemented on Windows and Linux with the help of libraries like Hls.js or through the use of proprietary solutions.

The main drawback of the protocol is the generally high latency, falling in the 10-45 seconds range, which can be further reduced with carefully thought out optimizations. HLS is not the best option when the application requires low delays, like in video conferencing or highly interactive software. However, low-latency variations have been developed both by private entities and open-source communities, and Apple has announced their official extension for the base protocol in 2019.

As previously stated, the continuous flow of data needs to be separated in chunks or media segments, each of specific duration. Apple recommends the usage of segments of 6 seconds. The Media segment formats supported by the HLS standard include MPEG-2 Transport Streams and Fragmented MPEG-4, with the option of Packed Audio in the case of audio only streams.

To know the types and locations of the various segments, a client must make use of the HLS playlist files, which come in two flavors

- the **Media Playlist** indexes the media segments for a specific bitrate variant, through a list of URIs mixed with protocol tags defined in the specification
- the **Master Playlist** contains a reference to all the available variant streams, and each URI will point to a media playlist file instead of a fragment

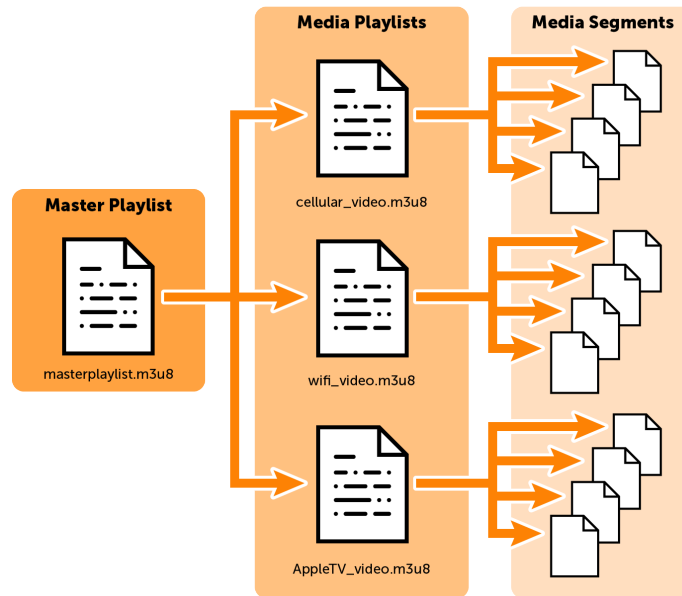


Figure 3: Playlist Files Structure

The master file is the first that must be retrieved and can be read only once since the advertised variants are not expected to change. The client can then proceed to open the desired Media Playlist and start retrieving segments with HTTP requests.

To give better context to a player parsing a playlist, HLS specifies a set of descriptive tags that must be placed inside these files, some specific to one type or even mutually exclusive between the two (if defined in the master playlist, cannot be present in the media one). The followings are general tags that can appear everywhere:

- **EXTM3U** sits at the beginning of the file and indicates that the format used is an extension of the one deployed with the previous HLS version. Its

presence is mandatory in every m3u8 file.

- **EXT-X-VERSION** is use to specify which version of the protocol is implemented by the server, and consequently the list of supported features.

Specific to master m3u8 files is instead the tag **EXT-X-STREAM-INF**, which delineates the parameters for a specific encoded variant and the URI of the media playlist. For each one of those, a line of this type has to be defined. The variant characteristics are described through a list of attributes, some of them recommended:

- **BANDWIDTH** is the maximum possible value of the bitrate, considering all the segments composing a media list. It's an integer with bps (bits per second) as a measure of unit.
- **AVERAGE-BANDWIDTH** is the mean value for the bitrate, counting again all the fragments
- **RESOLUTION** is the dimension in pixels of a video frame, obviously not present for audio only streams
- **CODECS** is a string of comma separated values, containing the audio and video encoders used in the related variant, in the ISO Base Media File format. Useful for the client to know the required codecs before downloading the content.
- **FRAME-RATE** is the highest value of the homonymous parameter

Of all the listed fields, only **BANDWIDTH** is required by the specification, although **CODECS** is marked as highly suggested for a better user experience. To illustrate the tags in action, this is a sample Master Playlist file.

```
#EXTM3U
#EXT-X-STREAM-INF:BANDWIDTH=150000,RESOLUTION=416x234,
                    CODECS="avc1.42e00a,mp4a.40.2"
http://example.com/low/index.m3u8
```

```
#EXT-X-STREAM-INF:BANDWIDTH=240000,RESOLUTION=416x234,
CODECS="avc1.42e00a,mp4a.40.2"
http://example.com/lo_mid/index.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=440000,RESOLUTION=416x234,
CODECS="avc1.42e00a,mp4a.40.2"
http://example.com/hi_mid/index.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=640000,RESOLUTION=640x360,
CODECS="avc1.42e00a,mp4a.40.2"
http://example.com/high/index.m3u8
#EXT-X-STREAM-INF:BANDWIDTH=64000,CODECS="mp4a.40.5"
http://example.com/audio/index.m3u8
```

Media Playlists usually follow a simpler structure and employ three fundamental tags:

- **EXT-X-TARGETDURATION** is the duration in seconds of media files and, by Apple recommendation, has to be around 6 seconds. No segment should last more than this value.
- **EXT-X-MEDIA-SEQUENCE** is the starting value for the incremental index included in the segments filename. When not specified, the client should assume an initial value of 0.
- **EXT-INF** is the location of a fragment, with respective duration in seconds

This example shows the case of a variant with MPEG TS segments of 10 seconds length and relative paths.

```
#EXTM3U
#EXT-X-TARGETDURATION:10
#EXT-X-VERSION:4
#EXT-X-MEDIA-SEQUENCE:1
#EXTINF:10.0,
fileSequence1.ts
#EXTINF:10.0,
fileSequence2.ts
#EXTINF:10.0,
```

```
fileSequence3.ts
#EXTINF:10.0,
fileSequence4.ts
```

During a live stream, this file is being continuously updated by the media server, with new segments appended at the tail of the list and older ones removed from the head. The media sequence tag must also be set accordingly. This concept goes by the name of Sliding Window, with the number of elements falling in the window range specified in the configuration.

At playback time, a client will ask for the Master Manifest and discover the encoded versions that the server makes available. Based on the quality of its connection, it will select the most appropriate rendition and request the related Media Playlist, starting then to stream the content. In case the network, previously congested in this example, suddenly offers more bandwidth, the player will move to a higher resolution variant, downloading the 1080p Manifest and the respective segments.

At any time, if needed, it will be possible to roll back to the low bitrate

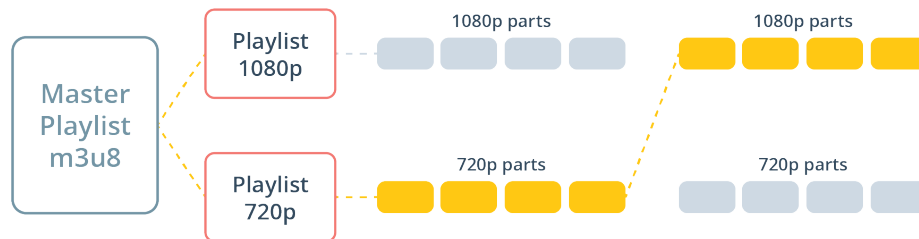


Figure 4: Bitrate Switch During Stream

version. Note that the switch happens from one fragment to the other, since that is the basic logical block of the stream. A smaller segment duration will offer a more granular control, but it could not be worth the cost of the added protocol overhead.

1.3 Nginx HLS Media Server

The `nginx-rtmp` module is a popular open-source solution built upon Nginx, integrating in the powerful web server the functionalities needed to operate as a media server. It supports the RTMP protocol for content ingestion, HLS and MPEG-DASH for last-mile distribution, while the carried audio and video content must be encoded with AAC and H.264. It can be used in a live streaming scenario, but also as a VOD (Video On Demand) server, when the media is not produced on the fly but statically stored in the file system, as in the case of Youtube and Netflix content.

To rearrange the incoming data to a format conforming the HLS standard and prepare it to be streamed to end users, a media server needs to work as a media encoder and segmenter.

- The **encoder** is responsible for the transcoding of the received data, possibly in multiple versions with various qualities and bitrates. This differentiation will allow clients to adapt to variable network conditions by requesting the best stream for their situation.
- The **segmenter** has to divide the continuous stream (or better, the streams leaving the encoder) into chunks of specified length. It's also its responsibility to create and update the playlist files indexing the generated fragments.

While the RTMP module natively implements the segmentation code, it relies on the FFmpeg tool for the transcoding of the incoming RTMP stream. FFmpeg is a versatile software component for working with audio and video files, with support for a huge list of protocols and formats. It is used in this instance to prepare multiple versions of an input stream, encoded at different bitrates.

The missing piece to complete the streaming architecture is the media distributor, which takes care of delivering the content to the final users. In the Adaptive Bitrate protocols family, since the resources can be accessed with HTTP requests, a common web server is responsible for this task. Obviously, this will be taken care of by Nginx itself.

1.4 Module Configuration

All the protocol options can be set in the nginx configuration file, where custom directives have been added alongside the ones for the basic web server. To enable HLS streaming, with three different encoded versions of the content, this is a sample nginx.config with all the required parameters.

```
rtmp {
    server {
        listen 1935;

        application src {
            live on;

            exec ffmpeg -i rtmp://localhost/src/$name
                -c:a aac -b:a 32k -c:v libx264 -b:v 128K -f flv rtmp://localhost/hls/$name_low
                -c:a aac -b:a 64k -c:v libx264 -b:v 256k -f flv rtmp://localhost/hls/$name_mid
                -c:a aac -b:a 128k -c:v libx264 -b:v 512K -f flv rtmp://localhost/hls/$name_hi;
        }

        application hls {
            live on;

            hls on;
            hls_path /tmp/hls;
            hls_nested on;

            hls_variant _low BANDWIDTH=1600000;
            hls_variant _mid BANDWIDTH=3200000;
            hls_variant _hi BANDWIDTH=6400000;
        }
    }
}
```

Figure 5: Nginx-rtmp configuration file

We can see that two applications are defined:

- **src** is the RTMP endpoint for the transcoding of the stream. With the **exec** directive, we set up the FFmpeg command that will take care of the operation, defining the expected bitrates for both audio and video, and the destination URLs for the output. Note that these paths include the name variable of the incoming stream and add a suffix string for each variant (**_low**, **_mid** and **_hi**), and have as destination the **hls** application instead of **src**.

- `hls` will take care of the segmentation, placing the media fragments inside three distinct subfolders of `/tmp/hls`, since specified with the `hls_nested` directive. For each variant the attributes to be inserted in the `EXT-X-STREAM-INF` playlist tag need to be specified, as for `BANDWIDTH` in this case (which corresponds to the summed bitrates for audio and video outputted by FFmpeg) .

Other notable directives are

- `hls_fragment` sets the duration for a media segment, with a default value of 5 seconds if not specified.
- `hls_type` is used to specify the insertion in a media manifest of the line `EXT-X-PLAYLIST-TYPE:EVENT`. With this tag the server should not modify the playlist file, other than when inserting new segments at the end of it, overriding the sliding window functionality.
- `hls_continuous` allows to restart the sequence number from where it was left the stream before
- `hls_playlist_length` is the size of the sliding window, so the number of segments written in the playlist at any moment, expressed in seconds

1.5 Low Latency HLS

Given the advantages that come with an HTTP based streaming model, it is a natural step trying to overcome the associated limitations. As noted before, Apple proposed an official solution to lower the latency of HLS streams down to two seconds or less, while keeping the high level of reliability bound to the protocol. With this goal, a set of single improvements has been combined together to achieve better performance:

- Partial Segments
- Playlist Delta Updates
- Block Playlist Reload

A big source of delay comes from the necessity to encode a full segment before making it available for clients, and when its duration is in the order of different seconds (generally six) the time wasted is considerable. Moreover, a player has to wait for the reception of three segments (or three times the target duration) before it can start the reproduction.

The proposed idea was then to reduce the size of a segment, or rather have the possibility to split it up into different smaller ones, through the use of incomplete .ts files or CMAF chunks, which take the name of **Partial Segments**. To advertise this incomplete parts, the **EXT-X-PART** tag has been added to the specification and can be used with the following syntax inside a media playlist:

```
#EXT-X-PART:DURATION=0.3333,URI="segmentPart-14.0.ts",  
      INDEPENDENT=YES  
#EXT-X-PART:DURATION=0.3333,URI="segmentPart-14.1.ts"  
#EXT-X-PART:DURATION=0.3333,URI="segmentPart-14.2.ts"
```

The **INDEPENDENT** attribute notifies the presence of an I-Frame inside the segment, since given its small size, a single partial fragment could not be enough to decode the media content. We see that in this case a duration of 0.33 seconds has been chosen, but it could reach a value as low as 200 ms.

With an increasing amount of segments, the manifest file will end up being updated more often, requiring a client to make more HTTP requests to stay

up to date. With **Delta Updates**, the server can respond with only the modified sections of the file, saving on bandwidth and latency.

Related to this feature, the polling mechanism employed for downloading the latest media playlist can be upgraded, giving a player the ability to ask the server to wait to return the file until a specific future segment (maybe not yet ready at request time) is available.

As mentioned, the CMAF (Common Media Application Format) file format can be used to represent a partial segment instead of the usual MPEG TS. Still in an early phase of adoption, it was born to tackle the problem of the streaming industry standards fragmentation, with its multitude of codecs and container formats in use. Aside from the advantages in terms of compatibility between HLS and other protocols like DASH, the latency improvements of this format could lower the delay from production to display.

To understand its structure, let's introduce some terms:

- a **track** is a sequence of samples of one format, either encoded audio, video or extra information like metadata, composed by a CMAF header and a list of fragments (more fragments grouped together form a **segment**)
- a **switching set** indexes tracks encoding different renditions of the same data, for example audio content at low, mid and high quality. It's used to dynamically adapt the stream bitrate
- a **selection set** bounds together switching sets of a media, which differ for example by the language utilized (think to a movie with different localizations), or even for the employed encoder
- a **presentation** is the union of multiple selection sets, where video, audio e related data finally converge, similar to the concept of Program in the MPEG TS standard

The basic idea is to split a segment in smaller units, the **chunks**, that are made available for download as soon as they are encoded, even if the related

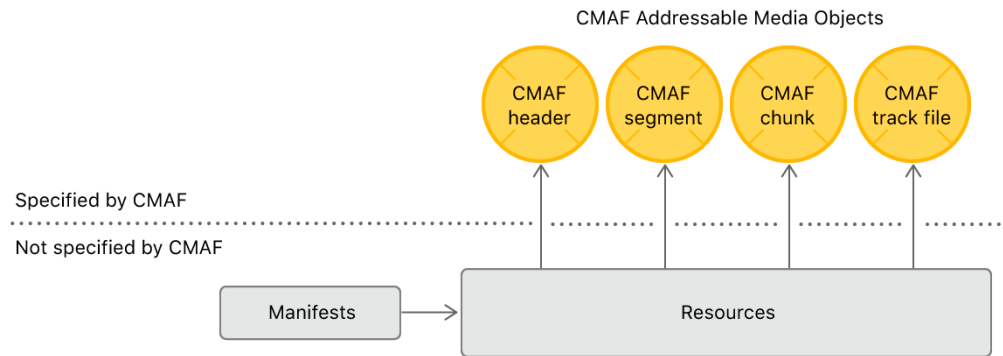


Figure 6: Cmaf File Components

segment has still to be fully completed. A client will be able to retrieve, through a manifest, one of the CMAF objects listed above, which could be as small as a single chunk, and instead of waiting for the buffering of three complete segments, it will start the reproduction after the reception of three chunks.

1.6 Content Publishing

So far we looked at the central component of an HTTP streaming chain, but we should also make an overview of the publishing side. As stated, the RTMP protocol is still the main solution applied to the transmission of the created audiovisual content to the media server. Since it would be impractical to send an uncompressed file, with a considerable waste on network bandwidth and noticeable delays, there is the need to encode it before the encapsulation inside RTMP packets.

The components that take care of this operation are many and differ based on the context. In the case of a live stream on a social platform like Youtube or Twitch a tool like OBS Studio would be more than enough, while in a professional environment it is common to employ a more complex solution like GStreamer or even hardware encoders.

GStreamer is an open source library for audio and video manipulation based on the concept of a pipeline, a sequence of linked modules each executing some specific manipulation on the media stream. Its support of a wide array of codecs and container formats, as well as its extensibility through the use of custom plugins, make it a popular choice not only in the streaming architecture, but also in applications that work with multimedia files as media players and video editors.

In the context of this work, the insertion of metadata in a live stream can

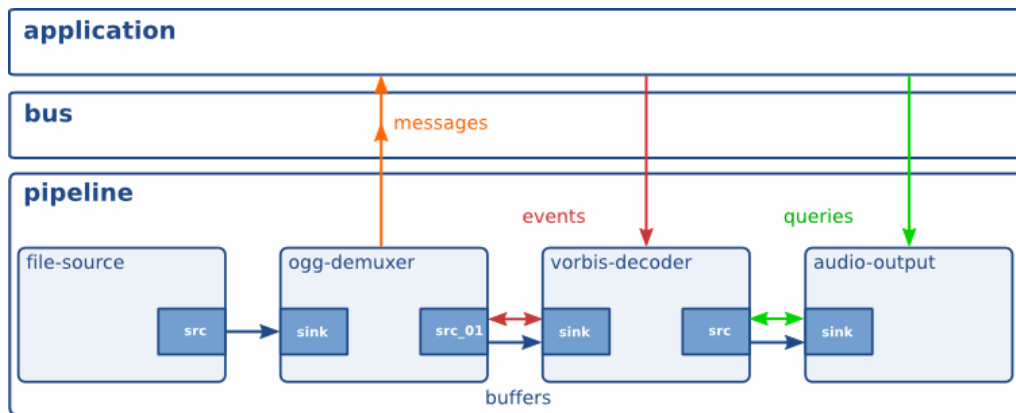


Figure 7: Gstreamer Pipeline Example

be achieved with the `taginject` plugin, extended with additional code to handle the injection not only at the launch of the encoder command but at

every moment of its life, through the use of a TCP server constantly listening for new input.

For compatibility with the Wowza Streaming Engine, the component was also modified to avoid the transmission of the `setDataFrame` string as the first field in the body of AMF packets, which begin instead with the `onMetadata` line. Alternatively, FFmpeg can be used to stream a media file to the nginx server, but although it supports the insertions of metadata packets, it cannot work in the same manner of GStreamer, accepting TCP connections while running. Further work could be aimed at investigating this missing feature.

Chapter 2

Nginx-rtmp Module

2.1 Mpeg TS Format

One of the ways media segments can be packaged inside an HLS stream is with the use of the MPEG TS container format. While originally conceived to be used for digital television broadcasting, it was later integrated in other technologies like streaming over IP networks. This format introduces the concept of Elementary Stream (ES), a sequence of bytes representing either audio, video or a related set of data. Different Elementary Streams can then be composed together to form a Program, and even separate programs can be assembled in a unified stream. The HLS standard allows only the use of single program streams.

The format is based on packets of a fixed length of 188 bytes. Each packet could either carry a chunk of the media data, or some system information for the correct handling of the bitstream at the receiver side. In both cases, a common header is placed in the first 4 bytes of the packet.

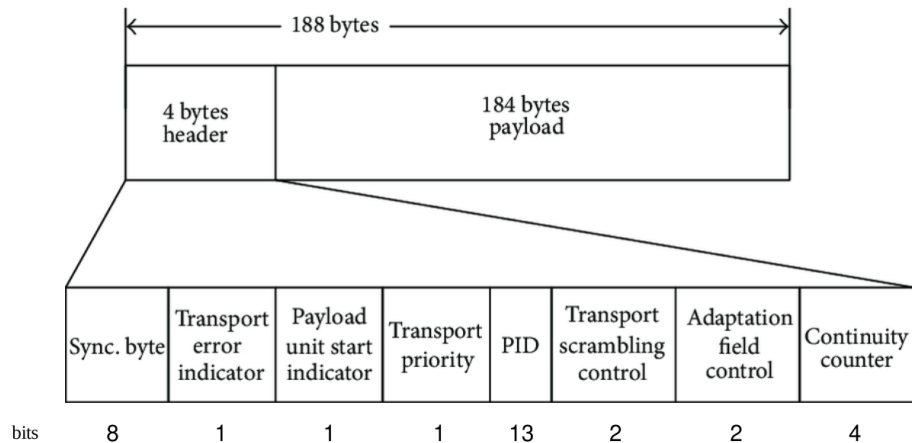


Figure 8: MPEG TS Header

The most relevant field in this sequence of bytes is the PID, or Packet Identifier, a 13 bits value which ties a packet to a specific Elementary Stream. It allows the recognition and separation of the different streams traveling together.

The PID can either be a number defined in the specification or a custom value dependent on the implementation, chosen by the developers. In case the payload of a packet is some kind of system data, the PID will be a defined

value listed in the standard.

The Adaptation Field Control is a two bit field signaling the presence or absence of an additional header, unsurprisingly named Adaptation Field. It is optional since it contains information that doesn't have to be sent in each TS packet.

After the header section, either Program Specific Information (PSI) or a Packetized Elementary Stream (PES) could constitute the payload. The most relevant types of PSI data are the Program Association Table (PAT) and the Program Map Table (PMT).

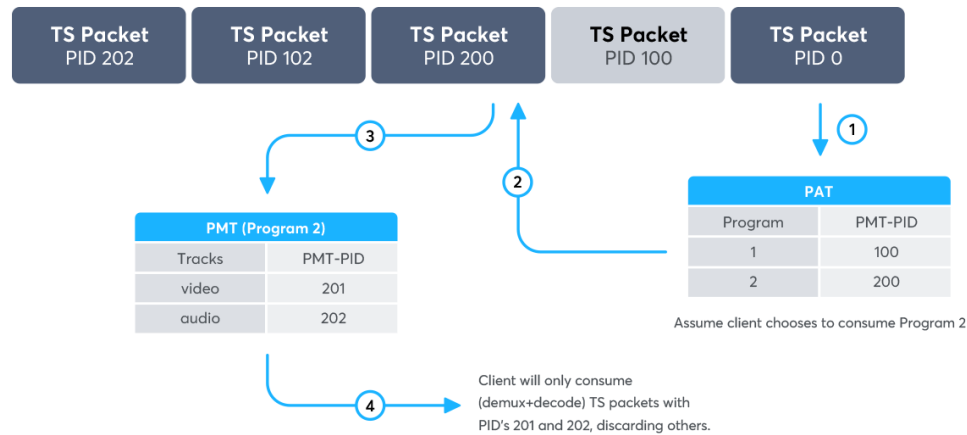


Figure 9: TS Packets Sequence

A PAT contains the information about all the programs available in a stream, which, as said before, are limited to one in the case of HLS. Its identifier must always take the 0 value. Inside this packet, there will be a list of PIDs corresponding to different PMTs, one for each program. A PMT lists the Elementary Streams composing a single program, such as the audio and video streams, each identified by their own PID. These two tables are the first structures that a receiver needs to correctly decode the incoming bitstream. A PSI header precedes the actual table, with a list of fields describing the format of the tabular data and its total length. Its most relevant fields for the aim of this work are the **Table ID**, which defines the type of system table contained in the packet, and the **Section Length**, the size in bytes of the packet after this field, excluding the padding.

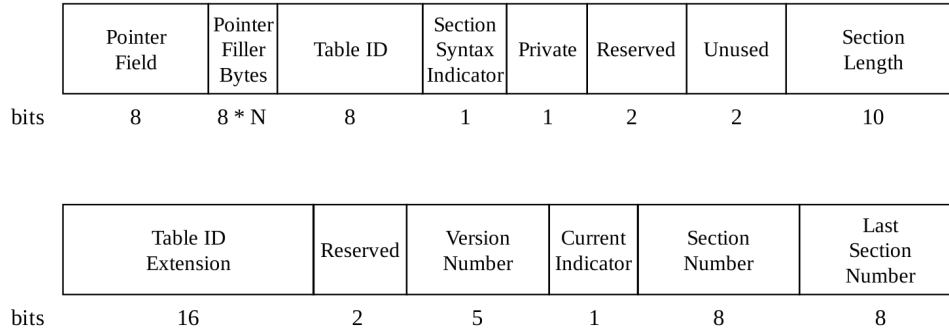


Figure 10: PSI Header Fields

After the PSI header, the actual table follows. For the PAT, the fields include the **Program Number**, a 16 bits value which identifies the program, and the identifier of the respective PMT.

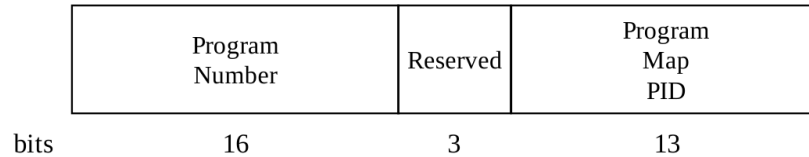


Figure 11: PAT Table Fields

The PMT is more complex and contains 4 initial bytes, followed by some optional descriptors and the list of Elementary Streams composing the Program. The **Program Info Length** field keeps the size in bytes of the descriptors and is set to 0 if none is present. For each ES, the table must contain its type, the PID and optional descriptors, in the same format as the ones used before. The **ES Info Length** field stores their length in bytes.

A descriptor is made of a unique **tag**, a field for its size and a body with predefined values. All the descriptor types and values can be found in the ISO 13818 [\[1\]](#) under section 2.6.

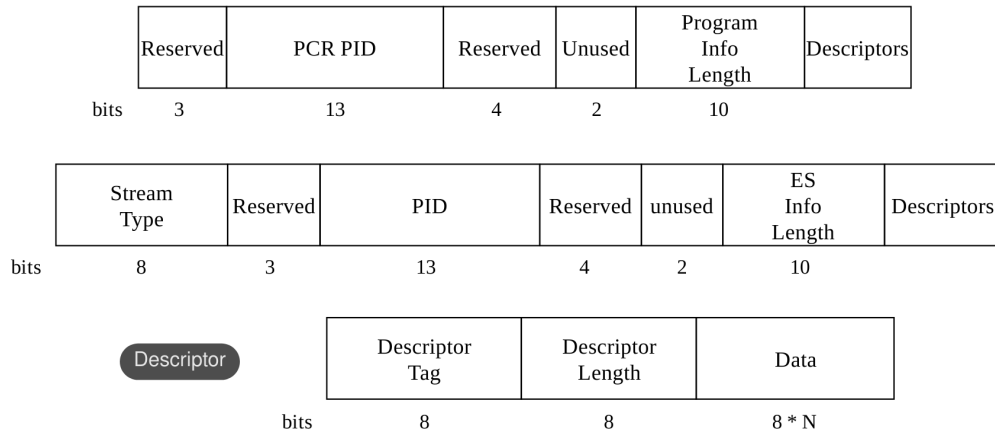


Figure 12: PMT Table Fields

In the nginx-rtmp module, this information is saved in an array of unsigned characters, inside the `ngx_rtmp_mpegts.c` file. It can be statically defined since the tables preserve the same values from one stream to another.

```
static u_char ngx_rtmp_mpegts_header[] = {
    /* TS header */
    0x47, 0x40, 0x00, 0x10, 0x00,
    /* PSI header */
    0x00, 0xb0, 0x0d, 0x00, 0x01, 0xc1, 0x00, 0x00,
    /* PAT table*/
    0x00, 0x01, 0xf0, 0x01,
    /* CRC */
    0x2e, 0x70, 0x19, 0x05,
    /* stuffing 167 bytes */
    ...
}
```

Looking closer at the PAT table we see:

- first 16 bits for the Program Number
- 3 reserved bits set to 0x7

- 13 bits for the PMT packet Identifier, of value 0x1001

We then have the PMT packet.

```
/* TS header*/
0x47, 0x50, 0x01, 0x10, 0x00,
/* PSI header*/
0x02, 0xb0, 0x17, 0x00, 0x01, 0xc1, 0x00, 0x00,
/* PMT table*/
0xe1, 0x00,
0xf0, 0x00,
0x1b, 0xe1, 0x00, 0xf0, 0x00, /* h264 */
0x0f, 0xe1, 0x01, 0xf0, 0x00, /* aac */
/* CRC */
0x2f, 0x44, 0xb9, 0x9b,
/* stuffing 157 bytes */
...
```

The 0x1001 PID can be seen in the TS header, starting from the 4th bit in the second byte. Inside the PMT table, we notice the video and audio Elementary Streams data, at the third and fourth rows respectively. One byte for the Stream Type (0x1b and 0x0f), 3 reserved bits and 13 for the PIDs (0x100 and 0x101). The remaining space could contain a list of stream descriptors but is in this case left empty.

A client reading an MPEG TS bitstream will have to look for the packet with an identifier equal to 0, the PAT table. Parsing its content, it will now know the PID of the PMT and how to recognize the respective packet, from which derive the Elementary Streams corresponding to audio, video and any other related information, like metadata or subtitles.

When considering the introduction of Timed Metadata, this structure will need to be modified accordingly.

2.2 How to Store Metadata in MPTS files

The Apple specification on Timed Metadata [7] details the procedure to follow when including metadata in an HLS stream. In the same way as with audio and video, the metadata must be place inside its own Elementary Stream, which will belong to one of the programs advertised in the PAT. The data for this new stream needs to be added to the Program Map Table, which with the proper fields will signal the presence of metadata in the program. The PAT can be left unmodified since HLS requires the use of single program streams. First, a descriptor of type **Metadata Pointer Descriptor** must be put in the previously empty **descriptors** section of the PMT header, with the values specified in the following table.

SYNTAX	VALUE
descriptor_tag	0x26
descriptor_length	– the length of the descriptor
metadata_application_format	0xFFFF
metadata_format_identifier	‘ID3 ’ (0x49 0x44 0x33 0x20)
metadata_format	0xFF
metadata_format_identifier	‘ID3 ’ (0x49 0x44 0x33 0x20)
metadata_service_id	– any ID, typically 0
metadata_locator_record_flag	0
MPEG_carriage_flags	0
reserved	0x1f
program_number	– identifies the program with the metadata descriptor

This gives a client consuming the bitstream the necessary details about the structure of possible metadata, which have to be, when used for HLS, in the ID3v2 format. ID3 tags were ideated to be used in mp3 files, carrying media related data such as title, artist and many other useful informations. They compose, in this context, the payload of TS metadata packets.

The newly created stream must be then included in the Elementary Streams list section of the PMT, adding the following fields:

SYNTAX	VALUE
stream_type	0x15
reserved	0x7
elementary_PID	- pid of the Elementary Stream
reserved	0xf
ES_info_length	- length of the ES info descriptor loop, also counting the metadata descriptor

Differently from audio and video, the definition of this Elementary Stream must include a descriptor:

SYNTAX	VALUE
descriptor_tag	0x25 - Metadata_descriptor tag
descriptor_lenth	- the length of the descriptor
metadata_application_format	0xFFFF
metadata_format_identifier	'ID3' (0x49 0x44 0x33 0x20)
metadata_format	0xFF
metadata_format_identifier	'ID3'
metadata_service_id	- any ID, typically 0
metadata_locator_record_flag	0
MPEG_carriage_flags	0
reserved	0x1f
program_number	- identifies the program with the metadata descriptor

All this information, which must be placed in the `ngx_rtmp_mpegts.c` array containing the PMT packet, translates to the sequence of bytes

```

/* descriptor list */
0x25, 0x0f, 0xff, 0xff, 0x49, 0x44, 0x33, 0x20, 0xff, 0x49,
0x44, 0x33, 0x20, 0x00, 0x1f, 0x00, 0x01,

/* metadata ES */
0x15, 0xe1, 0x02, 0xf0, 0x0f,

/* ES descriptors */
0x26, 0x0d, 0xff, 0xff, 0x49, 0x44, 0x33, 0x20, 0xff, 0x49,
0x44, 0x33, 0x20, 0x00, 0x0f

```

Note that a PID of value 0x102 has been assigned to this ES. The Stream Type describes the kind of data transported: 0x15 corresponds to "Metadata carried in PES packets".

Both the `section length` in the PSI header and the `program info length` in the PMT table, containing the size of parts of the packet, must be updated taking into account the added sections. The CRC32 checksum has to be recomputed excluding the TS header and the stuffing bytes, which must be reduced to reach a packet size of 188 bytes.

Every time a new TS fragment is opened by the module, the PAT and the PMT packets are immediately written on file, being the first information needed by the receiver to understand the following bytes in the bitstream. The duration of the media segment will consequently determine the frequency of the transmission of these system tables.

Understood how to configure the system tables for the integration of timed metadata, the next step is to figure out how the module should behave when an AMF packet, containing the fields of interest, is actually received. The metadata will be carried inside a PES packet, the same type used for the encoded audiovisual content.

Before implementing the code handling this process, it's useful to understand how the module works when audio and video come in.

2.3 How nginx-rtmp Handles Audio Packets

The nginx module takes care of the RTMP protocol message exchange between the publisher and the server, offering an useful abstraction that saves us from dealing with the underlying transport layer. We have instead direct access to the packets payload and can listen for module defined events triggered by the reception of specific data, such as audio and video frames.

The list of possible events, with the respective handler functions that execute when triggered, is defined in a `ngx_rtmp_core_main_conf_t` global structure, where the module stores two arrays named `events` and `amf`. While the former has a more general purpose, the latter is specific to AMF packets reception. In fact, it's important to point out that this format is not only employed for the encapsulation of metadata, but is also the medium for RTMP sender and receiver to exchange control directives through dedicated commands.

The base handlers for events common to all the module components are initialized in the `ngx_rtmp_init_event_handlers` function, inside the `ngx_rtmp.c` file. In its post_configuration at the end of the `ngx_rtmp_hls_module.c` file, the HLS submodule defines its own event handlers for rtmp audio and video events.

In the case of audio, the following function gets called:

```
ngx_rtmp_hls_audio(ngx_rtmp_session_t *s, ngx_rtmp_header_t *h,
                  ngx_chain_t *in)
```

The pointer to the rtmp session allows to retrieve the configuration parameters and the global context where to store data that must persist through the life of the server, while the `in` argument points to the sequence of bytes corresponding to the payload of the RTMP packet.

Retrieving the global variables, the hls context and the app config, is the first operation carried inside the handler. These are the fields that make up the context:

```
typedef struct {
    unsigned                                opened:1;
    ngx_rtmp_mpegts_file_t                  file;
    ngx_str_t                               playlist;
    ngx_str_t                               playlist_bak;
```

```

    ngx_str_t          var_playlist;
    ngx_str_t          var_playlist_bak;
    ngx_str_t          stream;
    ngx_str_t          keyfile;
    ngx_str_t          name;
    u_char             key[16];

    uint64_t           frag;
    uint64_t           frag_ts;
    uint64_t           key_id;
    ngx_uint_t         nfrags;
    ngx_rtmp_hls_frag_t *frags;

    ngx_uint_t         audio_cc;
    ngx_uint_t         video_cc;
    ngx_uint_t         key_frag;

    uint64_t           aframe_base;
    uint64_t           aframe_num;

    ngx_buf_t          *aframe;
    uint64_t           aframe_pts;
    ngx_rtmp_hls_variant_t *var;
} ngx_rtmp_hls_ctx_t;

```

The most relevant fields of this structure, from top to bottom, include:

- **file** stores the information for the currently open media segment, like its file descriptor and the size, as well as some parameter for the eventual encryption (specified in the configuration)
- **playlist** and **var_playlist** are strings containing the file path for the m3u8 files, respectively the media and the master playlist.
- the **bak** variables save the paths for the backups of the playlist files. When a new fragment is added to the available ones, the media playlist must be updated. The module does so by first modifying a copy stored

in the backup path file, and later replacing the original with the up to date version.

- **stream** is the location in the file system of the fragment files, comprising the path to the folder plus the string specified by the content publisher via the rtmp address (the location flag in the gstreamer launch command). For example, when connecting to `rtmp://localhost/live/test`, the string will be `test`.
- **frag** is the incremental counter appended at the tail of the filename of a fragment, which, depending on a config parameter, could either start at 0 or preserve the previous stream value.
- **frag_ts** keeps the timestamp of the first frame in the file and its set when opening a new fragment. It is used to check if a segment has reached its target duration inside the `update_fragment` function, subtracting it to the last inserted frame timestamp.
- **nfrags** is the total number of ts files
- **frags** is a circular buffer of `ngx_rtmp_hls_frag_t` types, a struct storing data about the fragment files such as the id and the duration. When writing or updating the media playlist, it is used to correctly fill the HLS tags. The size of the buffer is derived from the `winfrags` configuration parameter and computed as `winfrags * 2 + 1`.

```
.      typedef struct {
          uint64_t id;
          uint64_t key_id;
          double duration;
          unsigned active:1;
          unsigned discontinuity:1;
      } ngx_rtmp_hls_frag_t;
```


The discontinuity flag is set in case of timestamp discrepancies in respect to the previous segment, signaled with the insertion in the media playlist of the `#EXT-X-DISCONTINUITY` tag.

- `audio_cc` and `video_cc` are used to fill the Continuity Counter field in the PES header, which keeps track of the sequence of packets in a stream and helps detecting losses in the transmission.
- `aframe` is a buffer for audio frames, whose count is kept in the `aframe_num` variable. When the allocated memory is exhausted, the `flush_audio` function is called and the content saved on file.
- `aframe_pts` is the presentation timestamp of the last inserted frame in the audio buffer.

The configuration object contains instead all the possible parameters that can go inside the nginx config file specific to the HLS application, and differently from the context, its fields are well documented in the module wiki available on the Github repository.

The handler function goes on to retrieve the Codec context, which stores the parameters for the audio and video encoders extracted from the RTMP stream by the `ngx_rtmp_codec_module.c` submodule. This information is needed to check the correspondence between the data format in use and the ones supported by the HLS module (AAC in the case of audio in MPEG TS containers).

After that, a series of operations specific for audio is executed, which is not that useful for the handling of metadata . What is relevant to our use case, is the management of the media segment files (or fragments, using the module naming convention).

```
. static void ngx_rtmp_update_fragment(ngx_rtmp_session_t *s,  
                                     uint64_t ts,  
                                     ngx_int_t boundary,  
                                     ngx_uint_t flush_rate)
```

This function takes care of checking if a fragment has already been opened, looking at the context flag `opened`, and creating an empty one if needed. It makes sure the target duration has not yet been reached, otherwise closing the current file and opening a new one with the use of the `force` variable. The `boundary` argument gives the caller the ability to manually force this operation. Before returning, it also takes care of emptying the audio buffer when the delay of the most recent audio frame from the current timestamp exceeds a defined limit.

This avoids the unavailability of audio at play time when too many video packets are placed in a sequence.

Back to `ngx_rtmp_hls_audio`, if the new audio frame doesn't fit inside the buffer stored in the context, the flush audio function gets called.

```
.        if (b->last + size > b->end) {  
            ngx_rtmp_hls_flush_audio(s);  
        }
```

The `b` variable is a struct of type `ngx_buf_t`, which offers a convenient abstraction for working with dynamically allocated memory. It holds two unsigned character pointers, `start` and `end`, referencing the lower and upper boundaries of the buffer, which size is specified in the variable `audio_buffer_size` and can be set in the configuration, with a default value of 1024*1024 bytes. It also makes use of other two pointers, `pos` and `last`. When writing to the structure, we incrementally increase the `last` pointer one byte at a time. When instead we want to read from it, we begin from `pos` and move towards the former, knowing that when the two correspond (two identical addresses pointing to the same memory location), the buffer has been emptied.

Understanding how this works will come in handy later on, when it will be used to write on the segment file.

If the size of the frame, extracted from the RTMP header, added to the last written address exceeds the upper limit of the buffer, its content is saved on file with `ngx_rtmp_hls_flush_audio`.

In any case, an ADTS header is added before the audio, the inbuffer frame counter increased (`aframe_num`) and the most recent timestamp updated (`aframe_pts`).

The flush function has to prepare the data for the PES header that will sit in front of the payload, and to this end makes use of the following structure.

```
.
    typedef struct {
        uint64_t pts;
        uint64_t dts;
        ngx_uint_t pid;
        ngx_uint_t sid;
        ngx_uint_t cc;
        unsigned key:1;
    } ngx_rtmp_mpegts_frame_t;
```

It contains all the relevant and stream dependent fields, like timestamps, PID, Stream Type and Continuity Counter.

As declared in the static array for the PAT and PMT tables, the audio Elementary Stream is assigned the 0x101 identifier, while the CC, which must be preserved between different packets, is read from the context and later updated. Finally, with all the data ready to be saved in MPEG TS format, the writer function is invoked.

```
.
    ngx_int_t ngx_rtmp_mpegts_write_frame (
        ngx_rtmp_mpegts_file_t *file,
        ngx_rtmp_mpegts_frame_t *f,
        ngx_buf_t *b )
```

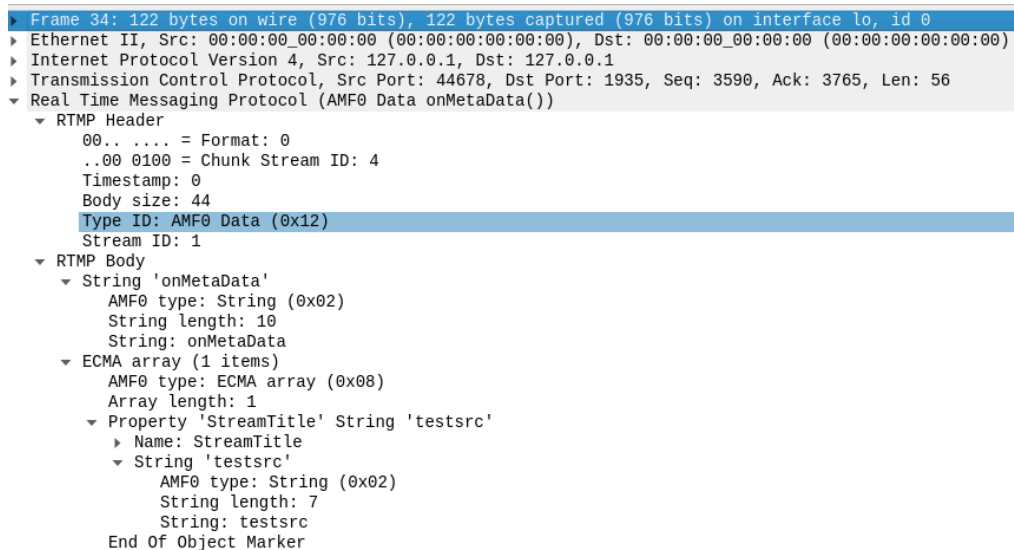
Combining the variable fields with the remaining header flags, it assembles a complete PES packet, ready to be saved on file alongside the video stream.

2.4 Integration of Timed Metadata

We already saw how to update the PMT containing a reference to the Elementary Streams of a single program, adding to the existing ones for audio and video the newly created dedicated to the transport of metadata. The next task is the handling of incoming AMF packets, which requires the manipulation of the data into a structure suitable for the MPEG TS container format.

This is a sample packet captured with the wireshark tool.

The RTMP body will include the `onMetaData` string followed by an array



```

Frame 34: 122 bytes on wire (976 bits), 122 bytes captured (976 bits) on interface lo, id 0
  Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00), Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
  Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
  Transmission Control Protocol, Src Port: 44678, Dst Port: 1935, Seq: 3590, Ack: 3765, Len: 56
  Real Time Messaging Protocol (AMF0 Data onMetaData())
    RTMP Header
      00... .... = Format: 0
      ..00 0100 = Chunk Stream ID: 4
      Timestamp: 0
      Body size: 44
      Type ID: AMF0 Data (0x12)
      Stream ID: 1
    RTMP Body
      String 'onMetaData'
        AMF0 type: String (0x02)
        String length: 10
        String: onMetaData
      ECMA array (1 items)
        AMF0 type: ECMA array (0x08)
        Array length: 1
        Property 'StreamTitle' String 'testsrc'
          Name: StreamTitle
          String 'testsrc'
            AMF0 type: String (0x02)
            String length: 7
            String: testsrc
        End Of Object Marker

```

Figure 13: AMF Metadata Packet Capture

of properties. Our goal is to send a single string field with key `StreamTitle` containing title and artist of the currently streamed media content.

To manage the reception of this kind of packets, we can define a custom handler function that will run at the reception of the aforementioned `onMetaData` string, executing the following operations:

- parse the AMF packet and save the content to memory
- create an ID3 tag with title and artist fields

- place the tag inside a PES packet
- write the packet to the MPTS file

Parsing of AMF packets

The module makes use of a global structure of type `ngx_rtmp_core_main_conf_t` where, together with other configuration parameters, stores two arrays named `events` and `amf`. With a call to `ngx_rtmp_init_event_handlers` at the start of the server, the `events` array is filled with the functions that need to be executed when one of the following types of event occurs:

- standard protocol event
- AMF event
- user protocol event
- audio / video event

The `amf` array instead, is filled in the post-configuration section of each sub-module (like the HLS one) and contains AMF specific handlers.

So when an AMF message is received, first the `ngx_rtmp_amf_message_handler` function is invoked. Then, based on the `type` field of the RTMP header and the command string carried inside the packet, an handler of the `amf` array will be called.

The AMF event that we want to listen to for intercepting metadata packets is `onMetadata`. In the HLS module post-configuration we add the lines:

```
ch = ngx_array_push(&cmcf->amf);
if (ch == NULL) {
    return NGX_ERROR;
```

```

}
ngx_str_set(&ch->name, "onMetaData");
ch->handler = ngx_rtmp_hls_meta;

```

which adds to the `amf` array our custom handler function, `ngx_rtmp_hls_meta`, that will run when a metadata packet is received. It takes as parameters pointers to the rtmp session, packet header and payload.

The first operation is the extraction of the metadata content from the packet payload. The server expects an AMF element of Object type (a sequence of key-value pairs), that will in turn contain a string field named `StreamTitle`, where two different strings separated by a pipe character are stored, one for the Title and the second for the Artist.

In the same way as when parsing an AMF command, we make use of the function

```

ngx_rtmp_receive_amf(ngx_rtmp_session_t *s, ngx_chain_t *in,
                    ngx_rtmp_amf_elt_t *elts, size_t nelts)

```

which from the incoming payload bytes `in` will extract the metadata and place them in the `elts` array.

This data structure contains a sequence of structs representing AMF elements (strings, numbers, arrays, etc.), that needs to be correctly initialized before being handled to the parsing function.

In the code we define:

```

static ngx_rtmp_amf_elt_t      in_inf[] = {
    { NGX_RTMP_AMF_STRING,
      ngx_string("StreamTitle"),
      &v.streamTitle, 64 },
};
static ngx_rtmp_amf_elt_t      in_elts[] = {
    { NGX_RTMP_AMF_STRING,
      ngx_string("first_string"),
      NULL, 0 },
    { NGX_RTMP_AMF_OBJECT,
      ngx_string("mix_array"),
      in_inf, sizeof(in_inf) },
};

```

We can see inside the `in_elts` array the expected AMF Object, with a reference to the `StreamTitle` string `in_inf`.

The first field in this structures is the type of the element, a string in our case, followed by an identifier that must correspond to the field key of the AMF packet. We then have a pointer to the variable where the string will actually reside (the `v` struct) and the size of allocated memory for the element. If the AMF object will contain a field named `StreamTitle`, its value will be put inside `v.streamTitle`.

The function can then be called passing the `in_elts` array as argument. The `skip` variable is used to handle the string placed before the metadata by FFmpeg. If the first element read is of type `NGX_RTMP_AMF_STRING`, it is set to 0 and `in_elts` will include the `first_string` field. Otherwise, `in_elts` will contain only `mix_array`.

```
skip = !(in->buf->last > in->buf->pos
        && *in->buf->pos == NGX_RTMP_AMF_STRING);
if (ngx_rtmp_receive_amf(s, in, in_elts + skip,
        sizeof(in_elts) / sizeof(in_elts[0]) - skip))
{
    ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
        "codec: error parsing data frame");
    return NGX_OK;
}
```

When the `ngx_rtmp_receive_amf` will return, we'll be able to manipulate the Title and Artist strings stored in the `streamTitle` field of the `v` struct.

Knowing the format of the `StreamTitle` field, we want to split the two strings, separated by the pipe character, and store them separately, also trimming any leading or trailing white spaces.

```
token = strtok(v.streamTitle, "|");
token = trim_spaces(token);
snprintf(v.title, sizeof(v.title), "%s", token);

token = strtok(NULL, "|");
token = trim_spaces(token);
snprintf(v.artist, sizeof(v.artist), "%s", token);
```

The `strtok` function of the `string.h` library is used for the splitting, while the custom `trim_spaces` takes care of the spaces. The cleaned strings are placed in the `title` and `artist` variables of the `v` struct and are now conveniently stored for the successive operations.

Generation of ID3 Tag

Since HLS expects metadata to be structured in the ID3 format (the same used to store meta information in mp3 files), we'll make use of an appropriate external C library.

Among the vast offering of modules, we opted for the open sourced `id3v2lib`, since listed in the ID3 standard website, recently updated and of small size. An ID3 tag is composed of a 10 bytes header and a list of frames containing each a specific type of data about the media they refer to.

We define and create an empty tag with the line

```
Id3v2_tag* tag = new tag();
```

With a simple call to the library, we can set title and artist for the tag, copying the strings from the `v` struct saved in memory.

Since the library is conceived to be used with mp3 files, there is no real interface for interacting with the mpeg ts container format. We manually need to write the bytes composing the tag container to a memory buffer, and pass this buffer to a function writing on the TS file. This buffer is an ngx defined structure

```
ngx_buf_t          out;
static u_char      buffer[132];

out.start = buffer;
out.end = buffer + sizeof(buffer);
out.pos = out.start;
out.last = out.pos;
```

As explained in a previous section, this buffer makes use of `start` and `end` for the allocated memory boundaries, the `last` pointer for writing in new data and `pos` when it's time to read it back.

Before calling the write function though, we must still take care of the PES container.

Generation of PES packet

As previously stated, the metadata will be inserted inside a PES packet and belong to a separate Elementary Stream, which will be multiplexed with the audio and video tracks in the same transport stream. This packet will need an appropriate header with the essential informations that a receiver has to know to correctly extract and interpret the payload:

- the PID of the Metadata ES
- a Timestamp for the alignment with the rest of the media content
- a Stream Identifier defining the kind of payload
- a dedicated Continuity Counter for packet loss detection

The rtmp module defines the struct `ngx_rtmp_mpegts_frame_t` with the necessary fields used for audio and video PES packets, described in section 2.3. The same can be done for metadata packets. So in our handler function we fill an instance of this structure.

```
frame.cc = ctx->meta_cc;
frame.dts = (uint64_t) h->timestamp * 90 + 100;
frame.pts = frame.dts;
frame.pid = 0x102;
frame.sid = 0xbd;
```

The PID 0x102 corresponds to the higher ES identifier value already assigned (0x101 for audio) incremented by one.

The Stream Type must be set to 0xbd as for the Apple Timed Metadata specification [7] and indicates a private stream.

The Continuity Counter (cc) contains some status information that must be preserved from one packet to another, for this reason it has to be saved in the global context. As for the video and audio cc, we add the `meta_cc` variable inside the `ngx_rtmp_hls_ctx_t` structure and increment it at every metadata packet reception. Being a 4 bits value, it will cycle in the 0 to 15 range as we would expect for this field.

Once the header and the payload are ready, `ngx_rtmp_hls_update_fragment` is invoked. This function takes care of the fragment files, creating an empty one at startup and making sure that when the target duration for a segment is reached, a new one with the right identifier is opened. The descriptor of the currently open fragment file is saved in the global context variable `file`. Finally, with `ngx_rtmp_mpegts_write_frame`, PES header and payload are concatenated together and saved on file. A small modification has been done to this code since the metadata packet requires the `Data Alignment` flag in the header to be set to 1, to indicate that the payload starts with the ID3 tag [7]. In case the tag doesn't fit in one single packet, the successive packets need to have the `Data Alignment` bit set to 0.

```
if (f->pid == 0x102) {
    *p++ = 0x84; /* set data alignment bit for metadata */
} else {
    *p++ = 0x80;
}
```

Here `p` is a pointer to the packet buffer and before the if statement points to the seventh byte, where the data alignment field is located, precisely in the sixth bit. When the PID corresponds to the metadata Elementary Stream identifier, we set the corresponding bit to 1.

Internally, this function makes a call to `ngx_rtmp_mpegts_write_file`, which takes care of encrypting the file if specified in the configuration.

2.5 Integration of Packed Audio Segments

In the case of audio only streams, the media segments can also be composed of the bare audio frames, without the encapsulation offered by the MPEG TS container. The HLS specification defines this segment format as Packed Audio.

In the case of AAC each frame should be prepended by an ADTS header, a sequence of 7 bytes (or 9 if CRC protection is used) describing data information like the codec profile and the sampling frequency. It is suggested to use one ADTS header for each AAC frame.

When using raw AAC, the timestamp of each sample is not carried in the stream. Instead, an ID3 private frame containing the timestamp of the first sample must be added at the start of the segment.

From the HLS RFC document:

The ID3 PRIV owner identifier MUST be "com.apple.streaming.transportStreamTimestamp". The ID3 payload MUST be a 33-bit MPEG-2 Program Elementary Stream timestamp expressed as a big-endian eight-octet number, with the upper 31 bits set to zero.

HLS Playlist Files

To notify the clients of the segment format for the stream the master playlist will simply need to define the audio codec in the EXT-X-STREAM-INF tag. Taking as example the Radio Norba stream:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-STREAM-INF:BANDWIDTH=64457,CODECS="mp4a.40.2"
chunklist_w2052014452.m3u8
```

The media playlist will include the usual tags, listing aac files instead of ts ones:

```
#EXTM3U
#EXT-X-VERSION:3
```

```
#EXT-X-TARGETDURATION:6
#EXT-X-MEDIA-SEQUENCE:4790
#EXTINF:5.015,
media_w2052014452_4790.aac
#EXTINF:5.016,
media_w2052014452_4791.aac
...
```

Integration in the nginx-rtmp Module

To manage this segment format, two new files are added to the hls folder alongside the ones used for interfacing with MPEG TS containers. Inside the header file, we define

```
typedef struct {
    ngx_fd_t    fd;
    ngx_log_t    *log;
    unsigned    encrypt:1;
    unsigned    size:4;
    u_char      buf[16];
    u_char      iv[16];
    AES_KEY     key;
} ngx_rtmp_aac_file_t;
```

A variable of this type needs to be added in the hls context to keep track of the opened AAC fragment information, with which IO operations can be carried out through the helper functions

```
ngx_int_t ngx_rtmp_aac_open_file(ngx_rtmp_aac_file_t *file,
                                u_char *path,
                                ngx_log_t *log);
ngx_int_t ngx_rtmp_aac_close_file(ngx_rtmp_aac_file_t *file);
ngx_int_t ngx_rtmp_aac_write_file(ngx_rtmp_aac_file_t *file,
                                u_char *in, size_t in_size);
```

When calling `ngx_rtmp_hls_update_fragment`, instead of the usual functions we take care of correctly renaming the segment with an AAC extension and writing the expected ID3 private tag.

```

if (hacf->packed_audio) {
    id = ngx_rtmp_hls_get_fragment_id(s, ts);
    ngx_sprintf(ctx->stream.data + ctx->stream.len,
                "%uL.aac%Z", id);
    ngx_rtmp_aac_open_file(&ctx->aac_file, ctx->stream.data,
                          s->connection->log);
    ngx_rtmp_aac_write_header(&ctx->aac_file, ts);
}

```

The `packed_audio` flag has been placed inside the module configuration and can be set with the use of the homonymous directive in the `nginx.config` file. Next, at the time of saving the audio frames with a `flush audio` call, we pass the buffer to the IO handler, avoiding to fill the PES header information.

```

if (hacf->packed_audio) {
    rc = ngx_rtmp_aac_write_file(&ctx->aac_file, b->pos, bsize);
    if (rc != NGX_OK) {
        ngx_log_error(NGX_LOG_ERR, s->connection->log, 0,
                      "hls: audio flush failed");
    }
}

```

Chapter 3

Configuration

The setup needed to compile the Nginx server with the RTMP module relies on external packages and the library for ID3 tags. The software chain has been developed and tested on a Linux environment with the use of Docker, starting from an Ubuntu base image. The following dependencies need to be installed (with the package manager):

- ca-certificates
- openssl
- libssl-dev
- build-essential
- zlib1g-dev
- ffmpeg
- cmake

Considering we are including our improvements to the code, the next components have to be installed from source:

- nginx (downloaded later with wget)
- nginx-rtmp module, <https://github.com/arut/nginx-rtmp-module>
- id3v2lib, <https://github.com/larsbs/id3v2lib>

3.1 id3v2lib library modifications

Since ID3 tags of private type were not supported by default, some modifications have been done on the library to correctly handle this format, required when Packed Audio segments are used instead of .ts files. In the `src/id3v2lib.c` file, the following functions were added:

```

ID3v2_frame* tag_get_private_data(ID3v2_tag* tag)

void set_private_frame(char* data, char* owner_identifier,
                      ID3v2_frame* frame)

void tag_set_private_data(char* private_data,
                          char* owner_identifier,
                          ID3v2_tag* tag)

```

The new frame identifier has to be defined in `include/id3v2lib/constants.h`:

```
#define PRIVATE_FRAME_ID "PRIV"
```

The `tag_set_private_data` function was added to the external interface, inside `include/id3v2lib.h`, and is the one that gets called each time an AAC segment file is opened.

3.2 Building the server

Place the source code folders for `nginx-rtmp` and the `id3` library inside the `/tmp/build` directory. To build `id3v2lib`, from `/tmp/build/id3v2lib` run:

```

$ mkdir build && cd build
$ cmake /tmp/build/id3v2lib
$ make && make install

```

To install `nginx` with the `rtmp` module run:

```

$ mkdir -p /tmp/build/nginx
$ cd /tmp/build/nginx
$ wget -O nginx-1.19.1.tar.gz
    https://nginx.org/download/nginx-1.19.1.tar.gz
$ tar -zxvf nginx-1.19.1.tar.gz
$ cd /tmp/build/nginx/nginx-1.19.1
$ ./configure \
    --sbin-path=/usr/local/sbin/nginx \
    --conf-path=/etc/nginx/nginx.conf \

```



```

--error-log-path=/var/log/nginx/error.log \
--pid-path=/var/run/nginx/nginx.pid \
--lock-path=/var/lock/nginx/nginx.lock \
--http-log-path=/var/log/nginx/access.log \
--http-client-body-temp-path=/tmp/nginx-client-body \
--with-http_ssl_module \
--with-threads \
--with-ipv6 \
--without-http_rewrite_module \
--with-cc-opt="-Wimplicit-fallthrough=0" \
--with-ld-opt="-lid3v2" \
--with-debug \
--add-module=/tmp/build/nginx-rtmp-module-1.2.1
$ make
$ make install
$ mkdir /var/lock/nginx

```

Place the nginx configuration file (nginx.conf) in the `/etc/nginx` directory.
To run the nginx server use:

```
$ nginx -g 'daemon off;'
```

or, to run it in the background, simply

```
$ nginx
```

Alternatively, the Dockerfile can be used:

```
$ docker build -t nginx .
$ docker run --net=host --name nginx nginx
```

3.3 GStreamer Updates

Starting from the previous work on the GStreamer library for metadata injection in an RTMP stream, the setup has been updated to support version 1.16.2. Instead of downloading the original repository for rtmpdump, there is a clone with the necessary patch already applied, which can be installed with

```
git clone https://github.com/JudgeZarbi/RTMPDump-OpenSSL-1.1.git
rtmpdump
```

Modify the librtmp/rtmp.c file, commenting or deleting the lines which insert the SetDataFrame string in AMF packets, inside the RTMP_Write function

```
/* if (pkt->m_packetType == RTMP_PACKET_TYPE_INFO)
pkt->m_nBodySize += 16; */
...
/* if (pkt->m_packetType == RTMP_PACKET_TYPE_INFO) {
//enc = AMF_EncodeString(enc, pend, &av_onMetaData);
pkt->m_nBytesRead = enc - pkt->m_body;
} */
```

and run the following:

```
$ cd rtmpdump/librtmp
$ make
$ cp librtmp.so.1 /lib
$ cd ..
$ make SYS=posix
$ make install
$ cd /gststreamer/gst-plugins-bad-1.16.2/ext/rtmp
$ touch gstrtmpsink.c
$ make
$ make install
$ ldconfig
```

The changes on the GStreamer source remain pretty much the same, taking into account that the library code has been slightly modified but without affecting the necessary additions.

The component can be compiled by hand or using the provided Dockerfile, placing it in the same folder where gstreamer (with respective plugins) and rtmpdump are located.

3.4 Testing

Once the nginx server is up and running, we can test the full streaming chain, from content publishing to livestream reproduction inside a media player. To talk with nginx through the RTMP protocol, we can resort to Gstreamer or FFmpeg.

With a version of Gstreamer supporting the insertion of the `StreamTitle` metadata field [1], we can stream a mp4 file with the following command:

```
gst-launch-1.0 filesrc location=input.mp4 ! qtdemux name=demux
! flvmux name=mux streamable=true ! queue ! rtmpsink
location='rtmp://localhost/live/test' demux. ! multiqueue
name=mq ! h264parse ! mux. demux. ! mq. mq. ! aacparse
! taginject tags="StreamTitle=\"Title|Artist\"" ! mux.
```

where `input.mp4` is the content we want to send to the media server located at the `rtmp://localhost/live/test` URL. Note that we connect to the `live` application with the `test` stream name. The `StreamTitle` metadata packet is set with the `!taginject` parameter. To include the pipe operator the string must be placed inside double quotes, escaped by the backslash character. The same applies for whitespaces.

Alternatively, we can make use of FFmpeg:

```
ffmpeg -re -i input.mp4 -metadata StreamTitle="Title | Artist"
-c copy -f flv rtmp://localhost:1935/live/test
```

making sure to use the right arguments for input and metadata. In this case though, the AMF packet will not start with the `OnMetadata` string but with `SetDataFrame`.

The server will take care of segmenting the content, and we can now reproduce the HLS stream in a player like VLC, though it doesn't display our metadata, or iTunes. In this case, the resource URL will be `itals://localhost/tv/test.m3u8`

If the setup has been done correctly, the metadata Title field will appear in the top bar of the player, next to the media controls panel.

3.5 HLS Development Resources

During development, testing the latest modifications to the code was often time consuming. Other than the Wireshark network analyzer, I made use of some tools to facilitate the process.

<http://thumb.co.il/> is an online MPEG TS file visualizer, which decodes the container format displaying system information like PAT and PMT tables, alongside audio and video packets. It also supports metadata, which was useful for quickly checking the correctness of the protocol headers when modifying the server module.

<https://github.com/bengarney/list-of-streams> is a list of public HLS streams of various types and quality, which extends the limited testing resources published by Apple. It contains some video resources with included timed metadata, which apparently are quite difficult to find, since they are usually relegated to audio only streams.

<https://github.com/dusterio/hlsinjector/> is a PHP library for metadata injection in hls streams, useful to get a first understanding on how the specifications can be translated in a real-world implementation.

Chapter 4

Future Improvements

4.1 Transcoding with FFmpeg

The work done on the Nginx-rtmp module brings timed metadata support only for a subset of the desired functionalities of a media server. In fact, the component handles correctly the segmentation phase of the process, when the continuous stream is chunked in smaller units and encapsulated inside MPEG TS packets. This could be easily achieved since the code for this operation is part of the module itself and not delegated to external libraries.

For the transcoding, on the other hand, there isn't a custom implementation, since the original author decided to rely instead on the FFmpeg tool.

While there are some options to configure the handling of metadata, I have not been able to obtain the desired outcome of letting through all the AMF packets.

When the RTMP stream reaches the server, based on the nginx configuration file, the bitstream is splitted in different quality versions, each at a specific bitrate. This operation, as for the current implementation, causes the loss of the metadata AMF packets, which can be seen with the wireshark network analyzer. From this point, the new RTMP streams will hit the segmenter endpoint, but the information of our interest has already been lost.

If for some reason transcoding is not required by the application, then the whole chain will work as expected, since FFmpeg will not manipulate the audiovisual data. The official documentation states that metadata should be preserved from input to output and describes some parameter that modify the behaviour of the command, such as

- `-movflags use_metadata_tags`
- `-map_metadata 0`

but both did not produce the required results.

4.2 Low Latency Extension

As mentioned before, latency has been a major concern in recent years and the focus of development efforts both by the industry and the open source community. While there are commercial options available like the Wowza

implementation, free and open alternatives keep lagging behind.

To begin with, the ability to separate TS files in smaller parts could be added, giving clients the possibility to download an incomplete fragment and consequently start the reproduction before the reception of three complete segments. CMAF files definitely introduce more complexity, but seem to be the solution for the intercompatibility between HTTP streaming protocols, DASH and HLS in particular.

To enable the delivery of Partial Segments, the module will need to support the new **EXT-X-PART** tag in a media playlist, which indicates the location of each chunk with the respective duration. Its syntax closely resembles the one used for the **EXT-INF** tag, with segment duration and URI defined through specific attributes.

```
#EXT-X-PART:DURATION=0.3333,URI="segmentPart-14.1.ts"
```

This modification has to be placed inside the `ngx_rtmp_hls_write_playlist` function, where instead of simply writing **EXTINF** lines, the server has to check if writing a complete or a partial fragment, since both are allowed in the Apple Low Latency specification. This decision could be based on an additional flag saved in the global hls context, or in the ts file reference directly, alongside its file descriptor and size. Additional care has to be taken for the **INDEPENDENT** attribute, indicating the presence of a key frame inside a segment, necessary for the decoding process.

It's important to note that these smaller segments are a partial representation of a complete fragment, which is considered the parent object and effectively named parent segment. Both can exist at the same time: while a standard segment is being produced by the server, its components incrementally accumulate until two instances of the same content are present.

With this mode of operation, the playlist will rapidly increase in size, especially when selecting a very small length for the parts. The recommendation is to delete partial segments older than three times a complete target duration.

To integrate this behaviour, the rtmp-nginx module will have to manage:

- the writing on file of partial segments while they are created, preserving the original operations on the complete segment

- the handling of old segments falling outside the window
- the tracking of I video frames in the encoded content, in order to mark the respective partial segment with the `INDEPENDENT` label

The integration of the CMAF format will require extensive work, with a need for additional functions that, much like for MPEG TS, will handle the I/O operations and expose an interface to the hls module.

Aside incomplete segments, the low latency specification resorts to additional strategies:

- delta updates: when asking for a playlist, a client can be served with only the lines that changed in respect to the previous request
- block playlist reload: if asked, the server can wait until a particular segment has been produced and added to the media playlist before responding to the HTTP request

The support for these non trivial features will require further extended analysis and careful considerations.

Conclusions

The streaming model has become the standard way to consume content online, taking up a considerable amount of bandwidth in communication networks. The family of HTTP based protocols has seen the most widespread adoption, thanks to the advantages that come with the underlying layer, like support on almost all end-user devices, lower delays due to caching on CDN infrastructures and ability to bypass firewalls. HTTP Live Streaming, abbreviated HLS, is in particular the solution which gained most popularity in the industry, with server and client implementations offered both by private companies and the open source community. In this work, a popular Nginx module turning the web component into a media server has been extended to support timed metadata traveling alongside the audio and video content. Such additional information, synchronized with the multimedia stream, can be used to improve the user experience with content related descriptions, but it can also enable the automated processing of the media stream, such as the marking of beginning and end of commercials.

The `nginx-rtmp` module ingests an RTMP stream from the content publisher and performs transcoding and stream segmentation, preparing the data to be distributed through the HLS or DASH protocols. The focus of this thesis has been on the HLS part.

The encoded information exiting the transcoder is encapsulated in MPEG TS packets, a container structure multiplexing together the audio and video components, as well as any other information related to the data, which individually take the name of Packetized Elementary Streams. In order to describe the format of the packet stream and the included audiovisual data, it makes use of system tables of various types, also sent over the network to allow the parsing of the bitstream by the receiver.

Moreover, the `nginx` module takes care of assembling the files required by the HLS specification, indexing the segments and the bit-rate variants in media and master playlists.

The integration of metadata has been achieved with the addition of a new Elementary Stream dedicated to this type of information, advertised to end-users with a proper header in the Program Map Table.

While receiving an RTMP stream, the server listens for the interception of AMF packets with the `OnMetadata` string, which is the metadata format expected to be used by the publisher.

The code proceeds to parse and sanitize the relevant data, in our case title

and artist of the currently streamed media, contained in the StreamTitle field. Through the use of a third party library, it generates an ID3v4 tag with the received strings. After assembling a ts packet with a complete header and the tag as payload, it proceeds to write the result to the open segment file.

The proposed solution suffers from limitations owing to the design choices made by the module developer. While the segmenter is hardcoded into the component, the transcoding operation relies on the use of the FFmpeg tool, which drops the AMF packets of our interest.

At the moment, the metadata integration works correctly when the audiovisual content is distributed in the format in which it was received by the publisher, not utilizing at its fullest the advantages of adaptive bitrate streaming that come with HLS.

Concluding, this thesis laid the groundwork for a full HTTP streaming chain with support for timed metadata based on open-source libraries, which will be hopefully extended with future additions and improvements in the effort to fill the gap with proprietary solutions.

Bibliography

- [0] Leonardo Chiariglione - The MPEG Representation of Digital Media
- [1] BS ISO-IEC 13818-7-2003 - Generic Coding of Moving Pictures and Associated Audio Information: Systems
- [2] Live Video Transmuxing/Transcoding: FFmpeg vs TwitchTranscoder.
<https://blog.twitch.tv/en/2017/10/10/live-video-transmuxing-transcoding-f-ffmpeg-vs-twitch-transcoder-part-i-489c1c125f28/>
- [3] hlsinjector
<https://github.com/dusterio/hlsinjector>
- [4] Requirements for Media Timed Events $\#+\text{latex}$:
<https://www.w3.org/TR/media-timed-events/>
- [5] ID3 tag version 2.4.0 - Main Structure
<https://id3.org/id3v2.4.0-structure>
- [6] HTTP Live Streaming draft-pantos-http-live-streaming-23
<https://tools.ietf.org/html/draft-pantos-http-live-streaming-23>
- [7] Timed Metadata for HTTP Live Streaming
<https://developer.apple.com/library/archive/documentation/...>
- [8] Fun with Container Formats
<https://bitmovin.com/fun-with-container-formats-3/>
- [9] Understanding the HTTP Live Streaming Architecture
https://developer.apple.com/documentation/http_live_streaming/...
- [10] MPEG TS Program Specific Information
https://en.wikipedia.org/wiki/Program-specific_information
- [11] Enabling Low-Latency HLS
https://developer.apple.com/documentation/http_live...

- [12] Live Low Latency Streaming
<https://bitmovin.com/live-low-latency-hls/>
- [13] About the Common Media Application Format with HTTP Live Streaming https://developer.apple.com/documentation/http_live_streaming/...
- [14] FFmpeg Documentation <https://ffmpeg.org/ffmpeg.html>
- [15] GStreamer Documentation <https://gstreamer.freedesktop.org/>
- [16] Guide to Mobile Video Streaming with HLS <https://mux.com/blog/mobile-hls-guide/>