Delft University of Technology
Master's Thesis in Embedded Systems MSC

# Improving the video streaming backend with on-the-fly format conversion and cloud storage

**Christina Kylili**

embedded software

TU Delft Delft University of Technology

# Improving the video streaming backend with on-the-fly format conversion and cloud storage

Master's Thesis in Embedded Systems MSC

Embedded Software Section
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Christina Kylili
ckylili@student.tudelft.nl

2nd January 2018

**Author**
Christina Kylili (ckylili@student.tudelft.nl)
**Title**
Improving the video streaming backend with on-the-fly format conversion and cloud storage
**MSc presentation**
10th January 2018

**Graduation Committee**

| | |
|---|---|
| Fernando A. Kuipers | Delft University of Technology |
| Rufael Mekuria | Unified Streaming |
| Pablo Cesar | Delft University of Technology |

**Abstract**

Online advanced media streaming services using HTTP adaptive streaming are increasingly popular. However in practice, the multi-protocol, multi-format nature of adaptive streaming creates a lot of engineering effort and costs for the operators, in the storage and preparation of the different formats. In this work, we acknowledge these issues and we study a streaming setup that can address these. Such streaming setup consist of a backend cloud storage and a processing node that generates streaming presentations for different devices on-the-fly. We analyze the streaming setup and its performance by testing in cloud deployments. Through this evaluation, we identify the performance limitations of the setup, imposed by the transfer of data between the object storage and the processing node. We propose a new backend storage caching scheme, based on rarely used existing feature of dref in the specification of the MPEG-4 standard. Experimental results show that the proposed scheme can improve the streaming performance, such as reduced latency and increased outgoing traffic volume towards the clients.

# Preface

This thesis is the result of my master's study in Embedded Systems at the Delft University of Technology. In my master's studies I obtained knowledge by attending courses related to Embedded Software and Networking. Thanks to the kind suggesting of Fernando Kuipers, I was able to join Unified Streaming for my master's thesis project and work on the interesting area of Video Streaming.

Foremost, I would like to express my very great appreciation in my daily supervisor Rufael Mekuria, for his steady and precious support. Rufael was always available for a lot of valuable advises and guide me through the research, development and writing of the thesis work with his patience, motivation and knowledge. He always tried to steered me in the right the direction whenever he thought I needed it and he was willing to give his time despite his busy schedule.

Moreover, I would like to thank Fernando Kuipers for his useful critiques on the academic research and constructive suggestions for improving this work.

Finally, I wish to thank my family and friends for their support and encouragement throughout my study.

Christina Kylili
Delft, The Netherlands
2nd January 2018

# Contents

# Chapter 1

# Introduction

On-line video distribution has become very popular today. Video traffic is currently consuming a large share of network resources and by 2020 the total traffic due to video services is expected to increase up to 80% of all internet traffic [1] .

Hosting and deploying video streaming services, requires large amounts of network bandwidth and storage, in order to serve millions of Internet users simultaneously. In addition, based on the number of users connected, the server infrastructure needs varies. For example in the evenings more users are connected, so more bandwidth and servers are needed to serve content. To deal with this varying demand, cloud-based deployment is beneficial as it allows to match server and network resources based on demand, using the infrastructure as a service paradigm (IaaS). Further cloud storage is one of the key cloud services provided under the Iass, as it offers seemingly unlimited storage space. This storage is also persistent and protected form device failures.

Today, HTTP adaptive streaming (HAS) is the most popular method of content distribution for Video on Demand (VoD) and live streaming (Live). In HAS, the video content is usually divided into smaller video segments, coded at multiple different bit-rates each. The client can decide which bit-rate segment it requests via an HTTP GET request, adapting thus to the network conditions.

There are different HAS protocols developed by different vendors, each targeting a different subset of devices and media players. Most of these specify the format of video segments and the structure of the manifest file, which signals the available video segments and their respective locations. But still, within a specific HAS protocol scheme there are several degrees of freedom for deployment such as different codecs and encryption solutions.

In practice, this means that a VoD provider needs to prepare and store multiple versions of a content. For large scale deployments, this will increase significantly the storage costs. Further, such statically coded asset

repositories are hard to maintain: in case a new format comes out, a new version of the file needs to be created, while it might still be necessary to keep the old version to support legacy devices. Last, the time to prepare all these contents and store them can be significant (introducing costly multiple encoding of content), increasing the time to deployment.

An alternative approach, is to store a single version of each packaged/encoded version and generate the HAS specific manifest and segment on-the-fly. This enables smaller repositories, less content preparation work and better future proofness when new formats are released and legacy support is still needed.

An interesting streaming architecture consist of a storage node backend that communicates with an on-the-fly conversion server, for serving incoming HTTP video requests. Such a setup has received little attention from academia, as most researches focus on the streaming frontend with a client and a server. However this setup is becoming more important as its compatible with future networks that employ NFV, edge computing, and network-based applications by distributing (media) processing units in networks.

In such a setup, the communication between the on-the-fly conversion server and the storage backend, could have a strong impact on the streaming performance. One could wonder if this setup is efficient, as the volume of the video traffic that needs to be transferred from the storage to the processing node, is quite big. Furthermore a delay between a centralized storage backend and on-the-fly conversion node deployed remotely, could degrade significantly the streaming performance .

This thesis introduces a backend streaming framework based on an on-the-fly conversion node and a cloud storage. Several experiments are performed in a realistic cloud deployment to study the behaviour of the setup using different streaming protocols. Further based on the behavior of the on-the-fly conversion node, we propose a scheme for (remote) storage access that can reduce the streaming latency and improve client throughput. The scheme is based on caching of server manifest file and MPEG-4 index files. The intuition behind the scheme can be used as a paradigm of improving the performance of more complex media processing nodes in the network. We test the setup in different (edge) cloud deployments and for large scale deployments with a realistic load generator for scalability.

The rest of this thesis is organized as follows. Chapter 2 gives an overview of the related technologies and related work. Chapter 3 presents the streaming setup and the performance of it through an experimental work with different media formats and protocols. The optimization scheme is presented in Chapter 4 together with the performance evaluation of this, in different cloud configurations (a centralized cloud and an edge cloud). Chapter 5 deals with configuring and testing the solution for highly concurrent video streaming traffic for high loads. Finally Chapter 6 concludes this thesis and outlines future research directions.

# Chapter 2

# Technical Background and Related Work

This chapter presents an overview of the technical background for video streaming and the related work. At the end of the chapter we present an outline of the research contributions of this thesis.

## 2.1 Live and On-Demand Streaming Model

Video Streaming is an increasingly popular method of delivering video content. Streaming enables the viewer to start video playback while the content is being downloaded, differing from normal file download where the entire content needs to be downloaded first.

Figure 2.1 shows a typical end-to-end delivery model for streaming video on-demand and live, with on-the-fly packaging. In on-demand streaming, the video files are in the storage and they are already compressed, encoded and packaged to a chosen file format. In live streaming, the video data are captured by a live source and then they are compressed and encoded in real-time by an encoder. The streaming server packages the media data to the different protocols and serves the videos to the client. Over-the-Top (OTT) content providers use Content Distribution Networks (CDN) to distribute content on a large scale. CDN chooses the best available streaming server to serve the videos, based on the geographically proximity of the server and the client. On the client-side, the receiving content is decoded by a video player and is rendered on a display. In this research we focus only on-demand video streaming instead of live streaming. This is because live streaming does not use the backend storage and does not need large scale content storage.

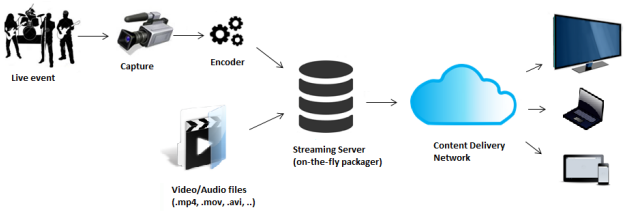



Figure 2.1: Live and On-demand Video Streaming.

## 2.2 Technical Background

### 2.2.1 Video Codecs

Video compression or decompression is used in all video streaming protocols, to reduce the size of a media presentation. There are different codecs available and similarly with adaptive bitrate protocols, not all codecs are supported by all media players. The most widely used video codecs are based on standards such as H.264 or also known as Advanced Video Coding(MPEG-4 Part 10, AVC)[6]. Using AVC, a video should be encoded multiple times in order to obtain different representations that have different quality(bitrate). Other popular video codec are MPEG's High Efficiency Video Coding (HEVC/H.265)[7], VP9 from On2(now Google)[8]. There are also audio codecs such MPEG-1 Audio Layer 3[9], also known as MP3, and Advance Audio Coding(AAC)[10].

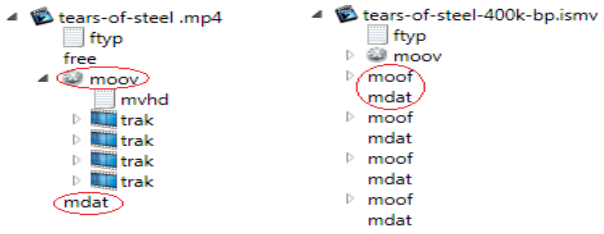
### 2.2.2 Media File Formats

Media data streams are wrapped in a container format. The container includes the physical data of the media but also metadata that are necessary for playback. For example it signals to the video player the codec used, subtitles tracks etc. In video streaming there are two main formats that are used for storage and presentation of multimedia content: MPEG-2 Transport Streams (MPEG-2 TS)[25] and ISO Base Media File Formats (ISOBMFF)[24](MP4 and fragmented MP4).

MPEG-2 Transport Streams are specified by [25] and are designed for broadcasting video through satellite networks. However, Apple adopted it for its adaptive streaming protocol making it an important format. In MPEG-2 TS audio, video and subtitle streams are multiplexed together.

MP4 and fragmented MP4 (fMP4), are both part of the MPEG-4, Part 12 standard that covers the ISOBMFF. MP4 is the most known multimedia container format and it's widely supported in different operating systems and devices. The structure of an MP4 video file, is shown in figure 2.2a. As shown, MP4 consist of different boxes, each with a different functionality. These boxes are the basic building block of every container in MP4.

For example the file type box ('ftyp'), specifies the compatible brands (specifications) of the file. MP4 files have a Movie Box ('moov') that contains metadata of the media file and sample tables that are important for timing and indexing the media samples ('stbl'). Also there is a Media Data Box ('mdat') that contains the corresponding samples. In the fragmented container, shown in figure 2.2b, media samples are interleaved by using Movie Fragment boxes ('moof') which contain the sample table for the specific fragment(mdat box).




(a) MP4 Container (b) Fragmented MP4 Container

Figure 2.2: Structure of Media Containers using MP4Explorer.

### 2.2.3 HTTP Adaptive Streaming

Video streaming today is based on HTTP adaptive streaming (HAS). The use of adaptive streaming requires the content to be encoded at multiple bitrates streams to support different qualities. This enables adaptation to the network conditions. Further each bitrate stream is partitioned to small video segments with duration 2-10 seconds. When content is requested from a server, the client first receives a manifest file. Through this manifest file, the video player becomes aware of the available bitrates and other relevant information. Then the video segments are requested by the video player. The bitrate requested for each segment is determined by a rate-adaption logic integrated to the client's video player.

There are different HAS protocols that differ in the manifest file type and the format that they use for the video segments[3][2]. Among these, the most used protocols are Apple's HTTP Live Streaming (HLS)[4] and Dynamic Adaptive Streaming over HTTP (DASH)[5].

#### HTTP Live Streaming

HLS is the media streaming protocol that is designed by Apple, and can be used both for VoD and LIVE. HLS uses MPEG-2 TS as their main segment file format. The manifest file that is used by HLS, is a traditional playlist (m3u8), which lists the media segments that are available in a timed-order. When multiple bitrates are available, there is a root playlist that referenced other lower-level playlists, with each representing an alternate encoding.

#### MPEG-DASH

DASH is the first international standard for HAS developed by MPEG and other standard groups (e.g 3GPP[11]). DASH is media codec agnostic and defines segment container formats for both ISOBMFF and MPEG-2TS. Yet in practice, mostly fragmented MP4 is used. This helps with the seamless segment switching between different bitrates, since segments are aligned and different bitrate representations share a common timeline. The manifest file used is xml-based and is called Media Presentation Description (mpd). The mpd contains information required by a DASH client to construct HTTP-URLs of the segments.

### 2.2.4 Cloud Object Storage

For maintaining large video databases, cloud storage is often preferred by content providers. For example Netflix has recently migrated to the Amazon Cloud for their computing and storage needs[28]. In particular, object based storage is usually preferred for content repositories. Object storage is designed to be used at application level and it offers an HTTP based API for storing and retrieving unstructured objects. Objects are addressable by a unique identifier and not by their location, offering a flat object addressing. Its key features include big amount of storage, durability, scalability and security.

### 2.2.5 Media Processing Operations in the Network

With the multi-protocol nature of video streaming and the static repositories that are used, multiple versions of the same content are distributed through CDN, leading to redundant CDN caching and transport of media data. This approach is clearly unsustainable and poses a lot of limitations for the media domain to evolve and adapt in next generations network environments.

A more efficient approach that exploits the common source of media segments and considers the challenges of emerging network environments, is given with the latest emerging MPEG streaming standard: Network-Based Media Processing (NBMP)[29]. This standard allows media delivery with media processing services/functions embedded in the network. The main aim of NBMP is to define the reference framework and interfaces between media sources and media processing functions in the cloud or the network (the NBMP format). The reference architecture diagram is shown in figure 2.3. It shows how a media source connects to a media processing function that takes care of the delivery to a client using different published formats. NBMP has a good potential to handle many of the problems with video streaming. Besides, is already proven by previous work[21] that moving a media processing node in the edge can address some of the current limitations. Further, the definition of standardized interfaces in NBMP will

help interoperable deployments in the cloud or in the network. For more information about NBMP, we refer to the document of [29].

The research in this work is aligned with state of art streaming systems, conforming to the NBMP reference architecture by using a media processing node that in this case is an on-the-fly format conversion server. On-the-fly format conversion deals with the existence of different formats and protocols. An example of on-the-fly conversion is dynamic packaging and manifest generation for multi-protocol video delivery from a single source.

Further, as this is an emerging standard, there is room for introducing new concepts, insights and related metrics that describe its performance and behaviour. This work takes a small first step in doing this, by implementing a simple VoD streaming scenario that suites the NBMP architecture.

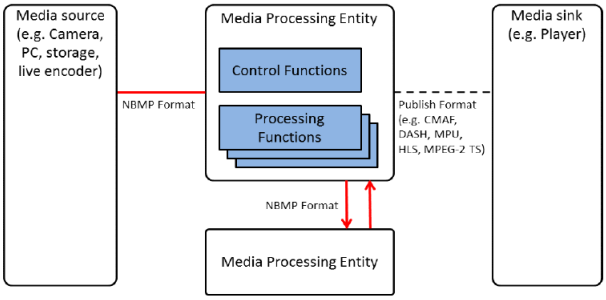



Figure 2.3: Framework of Network-Based Media Processing system [29].

## 2.3 Related Work

The emergence of DASH and the need for adaptive bitrate selection methods, has trigger a lot of research on the rate adaption algorithms. Several studies have exposed the performance problems of adaptive streaming in shared networks [12][13]. There is quite a lot of work that focus in assisting the bitrate selection in a way that will overcome these problems with the goal of maximizing the quality of experience (QoE) of the end-user[15][14][16][18]. On the contrary, our work relates to the server side architecture and aims to improve the communication interface between an object storage and a dynamic packaging server. Moreover our setup is confirming the emerging NBMP streaming architecture of MPEG.

Other work investigates the server-side deployment of adaptive streaming which is more closely related to this work. Specifically there is a some work that focuses on video transcoding and different strategies of trading transcoding cost and storage(for different bitrate encoded versions of a content)[19][20]. Although transcoding is important and relates to the storage problem, in this research, we focus on more lightweight conversions

like container conversion, manifest generation etc. Therefore, instead of the trade-off between compute and storage, we focus on the communication interface between the media source (object storage) and the on-the-fly converter.

A more related work that deals with dynamic packaging is presented in [22] and [21]. [22] evaluates the performance of two different implementations of on-the-fly packaging integrated in the client side. The results show that when the dynamic packaging happens in the client side, the performance (duration of conversion) can be slow. Mekuria *et al.*[21] present a multi-protocol video delivery architecture, where protocol specific media segments are generated on the fly. The conversion node is moved into the edge, and caches both protocol-specific segments and raw media data retrieved from the centralized storage. This way, the authors aim to reduce redundant traffic and caching that happens in the CDN. The work that is presented in that paper has a similar setup with the setup that we focus on, with a dynamic packaging node and a storage node. However that paper focuses on creating a smart edge cache and improving the efficiency of caching, while this work aims at reducing the latency and overhead related to object storage access.

## 2.4 Our Contribution

For practical video streaming that targets multiple protocols and large asset repositories, the combination of on-the-fly conversion, such as dynamic packaging, with object based cloud storage is a powerful one. Furthermore such a setup fits well with emerging future networks by decomposing media services, into flexibly configured network-based media processing nodes. However, prior work on video streaming does not consider this architecture and the possible performance bottleneck between the object storage and the dynamic packaging node. This thesis studies this problem and presents some improvement for such a setup.

In summary, this thesis presents the following contributions:

1. Performance evaluation of a state-of-the-art streaming framework with on-the-fly conversion and object storage on a realistic workload, using different file formats.

2. Analysis of the communication between object storage and dynamic packager based on Unified Origin. This reveals parts of the media data and its metadata, that are critical for the specific media processing node.

3. Optimization scheme for back-end using the current HTTP infrastructure and existing features of the MPEG-4 ISOBMFF technology, based

on analysis mentioned in 2. The scheme has shown reduced latency and improved throughput.

In addition, the setup that we focus on, fits well with the NBMP framework. We believe that insights on this emerging standard can be derived from this work, by studying this setup and introducing a metric related with the conversion efficiency of a media processing node, that could be useful in assessing a future NBMP format.

# Chapter 3

# Evaluating Video Streaming with Media Processing node and backend cloud Storage

This chapter presents the video streaming setup with on-the-fly format conversion and backend cloud storage. It presents the setup and related research aspects and an implementation of an experimental testbed that is analyzed for its functionality and performance.

## 3.1 Overview & Research Question

In this thesis we focus on a powerful video streaming setup that combines object based cloud storage with a media processing compute node, such as on-the-fly conversion. The streaming setup is shown in diagram 3.1. In this setup, VoD providers can store a single source but stream in different protocols. Such a setup enables smaller, more maintainable repositories and better future proofness when new formats are released and legacy support is still needed. Also is fully cloud based. Furthermore, more advanced versions of such a setup can support other desirable media processing operations for VoD providers, such as ad-insertion, dynamic content encryption etc.

This setup corresponds with the architecture of NBMP defined in MPEG as the new emerging streaming standard for immersive media that is well suited for future networks and address the needs of network-based emerging applications (section 2.2.5). As the NBMP aims to define a file format between the media processing and storage node, there is the need to study the communication interface of the two and understand the role of the storage format. This work takes a first step in doing this, by studying a simple VoD streaming scenario that suites the NBMP architecture.

A key question regarding the above setup, is the actual performance of it in practical deployments. For example, the communication between the

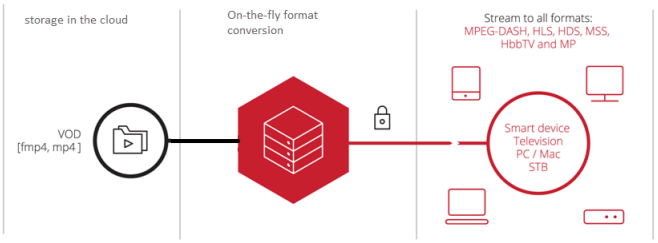



Figure 3.1: Setup with cloud storage and on-the-fly format conversion.

storage and the compute node may raise some performance limitations. This could be because of limited features and functionality of the cloud storage system (e.g. not supporting byte range request, limit on request rate, limited access, limited API, policies). Also the connection to the object storage could be limited in bandwidth or introduce some latency when the compute node is deployed remote from the cloud storage. Another limitation could be a low efficiency of the media processing node (more data is needed from the storage than that is finally produced for the user).

As far as one can tell from the literature, these issues and in general such a streaming architecture have not been studied by other work so far. In this work we study the performance and information exchanges between the cloud storage and a compute node responsible for dynamic packaging. The main research question that we want to answer in this chapter is:

"*What is the performance and information exchange of the setup when different file formats are used, under heavy load? Specifically how is the performance influenced by using MP4 and fMP4 as a storage format? Also what is the information needed from the media files for generating HLS and DASH presentations. Can some of the concepts and insights of the evaluation be used for optimizing the media processing node, thus assisting the NBMP framework?* "

In the next section of this chapter we will describe the experimental testbed that will be used to answer the above research question. Then a performance evaluation of this setup is presented and the results are discussed.

## 3.2 Experimental Testbed: Video Streaming setup with Origin and backend cloud storage

To address the research question, an experimental testbed was built, shown in figure 3.2. Starting from the right side we have the cloud object-based storage, which stores the content in two different file formats. The storage that is used, is Amazon Simple Storage Solution (S3). Next, we have the Origin Server which is the on-the-fly format conversion server. The on-the-fly format conversion software that is used in the setup is Unified Origin [26]

and is described in section 3.2.1. Finally, on the left side we have the load generator tool, that takes the role of simulating the client. More on this, in section 3.2.3. The deployment details of the setup are given in section 3.2.4.

Below we give an overview of the high-level behaviour of this streaming setup: When a client wants to stream video, it will initially request a manifest file (mpd, m3u8, etc.) from the Origin Server. The Origin, then fetches a special manifest file, called server manifest file (ism) in order to generate the client manifest file. Then, the client will start requesting video segments in a HAS protocol. For each segment request, the Origin will fetch the necessary video files from the storage and will package on-the-fly the video segment in the requested HAS protocol. Finally the protocol-specific segment will be send to the client.


Figure 3.2: Experimental testbed deployed in Amazon cloud.

### 3.2.1 Unified Origin

Unified Origin[26] is the on-the-fly format conversion software that was chosen in the experimental testebed. Unified Origin is a software plug-in for popular web-servers and it supports dynamic packaging, manifest generation, on-the-fly encryption, different DRM systems and more.

In this setup Origin needs to communicate with the backend storage and fetch the information that is needed to satisfy each client request. Most of this information can be found in the actual video files (MP4 or fMP4). Besides that, Origin uses a special 'server manifest' file, with extension .ism, that has important metadata information of the content (e.g available bitrate representations, codec etc.) needed to generate the protocol-specific segments and manifests.

### 3.2.2 S3 Storage and File Preparations

The cloud based object storage used in this work, is Amazon's S3 [31], offered by Amazon Web Services (AWS) [30]. S3 is chosen as its simple and its used from popular organizations in video streaming such as Netflix[28]. S3 has all the benefits mention in section 2.2.4 such as HTTP interface, durability etc. Special care is given in populating the content database of S3, as the content repository should reflect the challenges of a realistic repository.

### Choosing a Media Container

To populate the content database we choose to use as storage format MP4 and fMP4. FMP4 was chosen since it is already enabled for adaptive streaming and is used in DASH and other protocols (HDS, HSS and recently HLS). MP4 was chosen because it has been used as a storage format a lot in the past and is still being used, since converting all the content to fMP4 is time consuming and costly.

### Choosing and preparing the content

Three short videos are stored in S3: sintel[1], tears of steel[2] and elephants dream[3]. These videos were chosen because they are open videos with no copyright issues. The videos are available online as MP4 files, encoded in a single bitrate. To enable adaptive streaming, each video is encoded into multiple bitrates and multiple resolutions as shown in table 3.1, using the ffmpeg tool[36]. Tears of steel is available only in 400 kpbs, 800 kbps, 1200 kpbs, 1900 kpbs and 3000 kpbs, because it was already available in an encoded version. The chosen resolutions reflect the capabilities of typical user devices such as smart-phones, tablets and HD TV.

| **Resolutions** | **Video Bitrate(kbps)** |
|---|---|
| 480p: 640x480 | 200, 400, 750, 850, 1000 |
| 720p: 1280x 720 | 200, 400, 750, 900, 1000, 1200, 1500 ,2000 |
| 1080p: 1920 x1080 | 200, 600, 900, 1000, 2000, 2400, 2900, 3300 |

Table 3.1: Resolutions and Video Bitrates for Sintel and Elephants dream.

Each encoded video was then packaged to fragmented mp4 video using Unified Packager [37] and finally the server manifest file is created using the command shown in table 3.2.

```
sudo mp4split -o sintel_480p_fmp4.ism sintel_audio.ismv
sitnel_480p_200k.ismv sitnel_480p_400k.ismv sitnel_480p_750k.ismv
sitnel_480p_850k.ismv sitnel_480p_1M.ismv
```

Table 3.2: Creating the server manifest file(ism) for a fragmented video: The manifest file references the .ismv files(fMP4) for each bitrate representation.

[1]https://durian.blender.org/
[2]https://mango.blender.org/
[3]https://orange.blender.org/

### 3.2.3 Load Generator Tools

The role of the load generator tool is to simulate video streaming clients which request protocol specific manifests and video segments of different bitrates. In this work we used two different benchmark tools: Tensor [35] and Apache benchmark(AB) [32].

#### Tensor

Tensor is a tool for testing adaptive bitrate streaming by generating a realistic video streaming workload based on collected traces from a real video player. Tensor is able to simulate a large number of concurrent connections in order to measure the server behavior under peak load conditions. Tensor is based on two open source programs, WRK[33] and Performance Co-Pilot(PCP)[34]. WRK is the actual load generator. With WRK, the number of concurrent connections is increased every 5 seconds while the number of requests in each connection is maximized in order to identify the maximum throughout that the server is capable of. WRK gives the throughput and latency metrics reported in the client side. Furthermore Tensor collects hardware and software statistics from the origin server (throughput, CPU usage, memory utilization) using PCP. These metrics are displayed in a web-interface with graphs. Starting an experiment with Tensor is simply done by supplying Tensor with the URL of a client manifest file of a specific video. The web interface of Tensor is shown in figure 3.3. Adaptive bitrate streaming is possible because WRK gives the option of generating a workload of HTTP requests with different URL request thus requesting video segments of different bitrates (different URL) simultaneously.

#### Apache Benchmark

AB is a popular tool that can generate various HTTP workloads to measure the server performance, by specifying the total number of requests and the number of multiple requests to perform at a time. Unlike Tensor, only one URL request can be selected for the whole workload. For this reason AB is used to measure the performance of the streaming setup when delivering either a manifest file or just a specific segment. AB tool reports application layer statistics on HTTP behaviours (e.g reply rate, response time), providing just a user-level view of the performance.

We choose to use two benchmarks, since each has a different testing range. Tensor is used for testing large scale adaptive streaming and the workload is more realistic. On the other hand, testing with AB is better for understanding what is the performance in lowest-level operations of video streaming by just doing individual requests either for a manifest or a video segment.

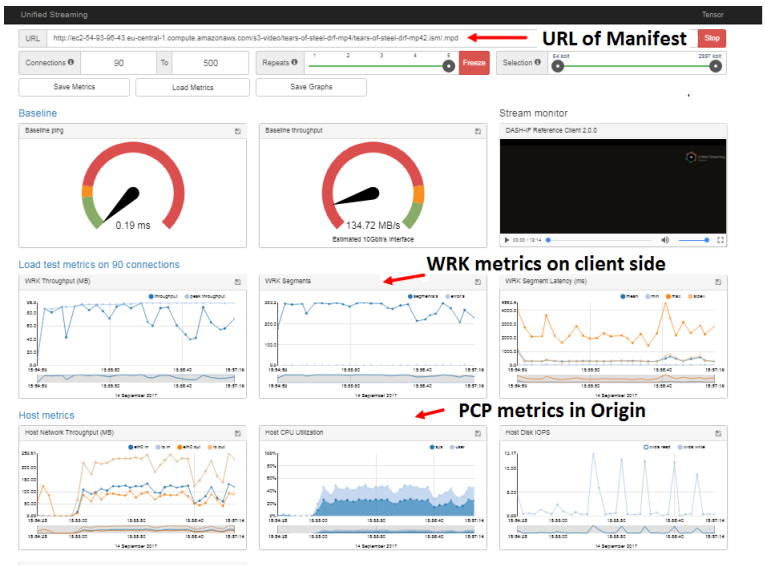



Figure 3.3: Tensor web interface.

### 3.2.4 Setup Deployment

The test framework of figure 3.2, is deployed in a cloud environment, in Frankfurt using AWS. In this preliminary performance evaluation all the components of the setup run on the same cloud environment. In practice the client should be further away but due to cost limitations, is placed in the same cloud (data transfer fees for sending data outside AWS are higher). The Origin server and the load generator tool, run on virtual servers in Amazon's Elastic Compute Cloud (EC2). The Origin runs on an Apache web-server installed on an Amazon m3.xlarge instance[43] and the load generator runs on a compute-optimize instance (c4 instance) in order to be able to handle a big amount of connections.

### 3.2.5 Key Performance Indicators

The performance of the setup is measured by using three key performance indicators (KPI). The first KPI is the average latency for a handled request and the second is the achieved throughput in the server(or in the client in case of using AB). The achieved throughput is measured with different ways in AB and Tensor, as shown in table 3.3. The last KPI's is a specific metric for this setup, that we define later. We name this metric the 'conversion efficiency' for each file format and is the ratio of the outgoing traffic and the incoming traffic in the server. This ratio can indicate the amount of more data that the Origin needs to fetch from S3, related to what is actually produced for the client.

| KPI | Description |
|---|---|
| latency(ms) | The average time it takes for a request to return a response |
| throughput | Tensor: Megabytes received/send per second(MB/s)<br>AB: amount of requests handled per second (request/s) |
| conversion efficiency | Ratio of outgoing traffic and incoming traffic in the server |

Table 3.3: Summary of the three KPI's.

These metrics are suitable for testing a streaming setup in large scale with a big number of requests. Also they can pinpoint network impairments which makes them suitable when focusing on the communication interface of the Storage and the Origin. Even thought these metrics are QoS-based and not QoE-based ([39],) they can still give an indication on the QoE of the client. For example, lower latency indicates faster video playback and lower start-up time.

### 3.2.6 Evaluation Methodology and Tested Workload

To identify what is the overhead of using a backend storage to the setup, a comparable study is performed: using local storage and backend cloud storage. When local storage is used, the original video files are stored in the Origin, allowing dynamic packaging. Furthermore, some more experiments are done with the backend storage in order to give some insights on how different file formats influence the streaming performance.

Each test that is done (local storage, backend storage, different videos, DASH and HLS) with Tensor, is repeated three times for cross-validation. The throughput and latency metrics that we get for each repeated test have a low standard deviation (4.5 for throughput and 6.9 for latency) so they can be considered reliable. Any statistical information that is mentioned in the following sections is derived by using the averages of each of the three experiments.

## 3.3 Experimental Findings on Streaming Performance

### 3.3.1 Local Storage vs Cloud Storage

Figure 3.4 shows the incoming and outgoing traffic measured by Tensor, for both local and backend storage (video stored as MP4 and fMP4). In all cases, dynamic packaging is used and the video is requested in DASH. Incoming traffic reflects on the backend data that the Origin fetches from S3. The outgoing traffic is the traffic between the client and the Origin. The figure reveals the important information regarding the performance of the

setup that separates the media source from the Origin :

- Backend traffic is higher than the front end traffic: When backend storage is used, in addition to the outgoing traffic there is also incoming traffic. This is due to the raw media data (MP4/fMP4 data and ism) that are send from S3 to the Origin. The backend traffic is actually higher than the front end traffic.

- File format in remote storage matters: While with local storage the outgoing throughput that is achieved for each video is similar for both MP4 and fMP4, for the cloud storage this is not the case. The outgoing throughput is different when the same video is stored as MP4 and as fMP4.

- Throughput and conversion performance decreases: When backend cloud storage is used the outgoing traffic is decreasing compared to using local storage, even though the traffic between the client and the Origin hasn't changed. This is because the Origin needs to fetch some more data from S3 than what it actually produces for the dynamic packaging operation. Combining this with the previous point, we conclude that the conversion performance of Origin changes when backend storage is added and this is also influenced by the file format. This is why we introduce the 'conversion efficiency' KPI that we discuss later.

- Latency increases: Latency is increased when using the backend storage, because to satisfy every segment request, the Origin needs to make additional requests to the remote storage. The figures of latency are not shown here for space reasons.

- Rate Limit by virtual compute node: In all the tests, the maximum throughput achieved in the server in a single interface(incoming) is around 120MB/s(=1Gbit/sec). Thus we can conclude that Amazon caps the in/out traffic on each instance. In the instance that is used in these tests, this limit is 1Gbit/sec for each interface.

### 3.3.2 Conversion Coefficient and Differences between file formats

As said above, when the media storage is separated from Origin, the outgoing throughput is decreased. Specifically the decrease in the case of fMP4 was found to by around 30% while for MP4 the decrease is 59% [4]. Thus the streaming performance is influenced by the file format that is used in the storage in the case of remote storage. This is more clear by observing

[4]The numbers are calculated by using the average of the decrease noted for all three videos

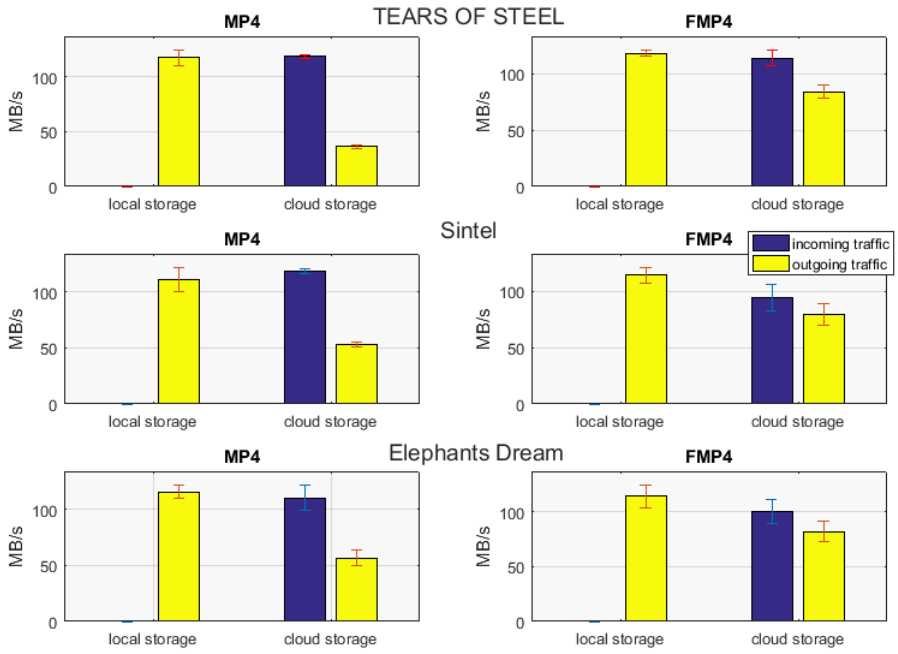



Figure 3.4: Incoming and outgoing traffic for local and backend storage.

the saturated throughput measurements obtained by Tensor in table 3.4. In all cases, storing the content as fMP4 results to higher outgoing throughput than storing as MP4. This might be due to the fact that MP4 contains one big 'moov' box that contains not only the metadata but also the information on accessing the media data. Further the saturated throughput in the outgoing link does not reach the maximum throughput of the Amazon instance that the operator pays for, therefore resources go to waste.

This undesirable behaviour is due to the separation of the Origin and the storage. The Origin needs to fetch more data from S3, than what it produces. In fact the amount of data that it needs for packaging a segment, differs per file format. To measure this amount, we define a coefficient that we call 'conversion efficiency'. This coefficient is the ratio of the outgoing traffic and incoming traffic and is an indication on the amount of more data that the Origin needs to fetch from S3 related to what is actually produced for the client after the conversion. This metric is specific for the on-the-fly conversion node but it could be defined for any media processing function. Table 3.4 shows that converting DASH segments from an fMP4 video source does not create significant overhead (coefficient is almost 1) while when the video is stored as MP4, the Origin needs almost double data (low efficiency of 0.5).

| file format | mp4 | | | fmp4 | | |
|---|---|---|---|---|---|---|
| source video | ed | sintel | tears | ed | sintel | tears |
| in throughput(MB/s) | 110,57 | 118,17 | 118,12 | 100,12 | 94,45 | 104,82 |
| out throughput(MB/s) | 56,80 | 53,08 | 35,7 | 82,22 | 79,53 | 76,31 |
| conversion efficiency(out/in) | 0,51 | 0,45 | 0,31 | 0,82 | 0,84 | 0,74 |

Table 3.4: Backend Storage results (video requested in DASH)

### Testing with AB

As Tensor test the setup in large scale, we want to verify that using fMP4 in the storage gives better performance than storing MP4, also in terms of latency. Thus AB is used to measure the latency when requesting just a single segment. This way we can make a more clear comparison between the two file formats. Using AB we generate a large number of DASH and HLS requests (30000 requests) for a low bitrate segment (200kbps) and a high bitrate segment (1Mbps) of sintel, 480p. The results are shown in the following figure.

In summary, the results show the efficiency of using fMP4 in the storage rather than mp4. The percentage difference of f-mp4 and mp4 in DASH case is 78% for the low bitrate segment and 50% for the high bitrate segment. In HLS case this difference is reduced by 20 (48% low bitrate, 30% high bitrate), because in HLS more data are needed due to the multiplexed form of segment(contains both video and audio). For a similar reason, the difference between mp4 and f-mp4 is less for the high bitrate segment, compared to the difference that they have when a lower bitrate segment is requested.

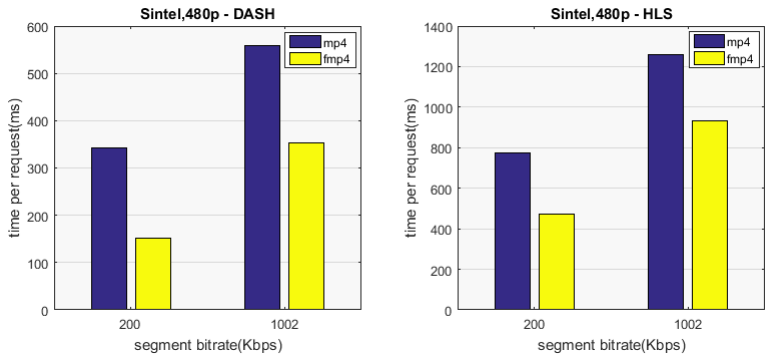



Figure 3.5: Difference between MP4 and fMP4 for time per request when segment is requested in DASH and HLS format.

### 3.3.3 Discussion

The performance evaluation of the setup shown in figure 3.2, reveals that the communication between the Origin and the backend storage creates an overhead on the video streaming performance. The latency is increased, the throughput between the client and the Origin is decreased and backend traffic is higher than the front end traffic. This is an undesirable behaviour, because the maximum throughput of the instance that the costumer pays, cannot be achieved, therefore resources go to waste. This will also have a negative impact on the QoE of the end user since it receives less MB/s.

The most interesting part of the performance evaluation is that the findings reveal the fact that in order for the Origin to create a manifest file or a video segment it needs some more data from S3 than what the client will actually receive. Further this behaviour is influenced on the file format used in the storage. One could say that this is due to the limitations of the Origin but the good performance with local storage show that this is not the case and is the result of separating the storage and the conversion node. To understand why this happens we need to look deeper into the communication between the Origin and the Storage and further understand the functionality of the Origin. This is done in the next section. In any case, this behaviour could be expected from any media processing node and not just an on-the-fly conversion node. Therefore the conversion efficiency metric that we introduce, although simple, is useful for assessing a future NBMP format as this metric relates to the format used in the storage and other important factors such as the processing node, the video content, the client request and the protocol used for the communication of the Origin and the storage. A future work that could model this metric as a solid function on these factors, would be really useful for optimizing the processing node.

## 3.4 Communication Analysis of Origin and S3

Analyzing the communication between the Origin and the Storage is essential in the streaming setup that we focus, as the communication of the two causes performance degradation and is affected by the media container used in the storage. For this we inspect the HTTP traffic between the Origin and S3, when a client requests either a manifest file or a segment (HLS and DASH) while content is stored in S3 as MP4 and fMP4. This is done using packet inspecting tools like tcp dump and Wireshark. Figure 3.6 depicts the setup in this case. Packets are captured in the Origin using tcpdump and saved in a file that is later used in Wireshark for analysis.

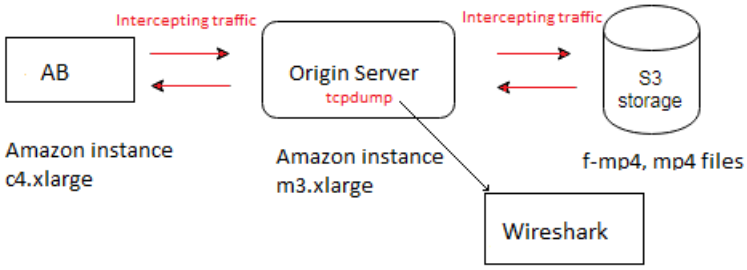



Figure 3.6: Setup for capturing traffic between the Origin and the Storage.

### Generating a client manifest file

A manifest file (mpd or m3u8) contains metadata information of the content and the available bitrate representations of the video. On the server side, this information is stored in the server manifest file (ism) that is in the storage. Thus this is fetched from the storage. Further, a client manifest file contains information for constructing the segments URLs. In order to construct this information, the Origin will need to access the timing and indexing information of the samples, of each bitrate representation. This is done by a number of byte range requests sent from the Origin to the storage, for each file referenced in the server manifest file (audio file, available bitrate representations). Since MP4 and fMP4 have a different structure, the number of byte range requests are different for each format.

Table 3.5 shows the information that is requested from the Storage, upon a manifest request from the client. Each cell of the table indicates one byte range request sent. When the content is stored as MP4, 3 byte ranges requests are done in order to fetch the 'ftyp' box and the 'moov' box for each bitrate representation ('mvhd' is the header of moov box). The moov atom is needed because it contains metadata and the sample tables boxes that give indexing information about the media data ('stbl' box).

When the source video is stored as fMP4, an additional box named Movie Fragment Random Access Box ('mfra'), which is part of the fMP4 container is needed. This box contains the information of locating the moof box of each sample. The mfra box is fetched with 2 byte range requests: first for determining the size of the box and then retrieving the mfra box. Also for DASH, the last moof atom is necessary to know where the presentation finishes. When an HLS manifest is generated, there are just four byte range requests since the last moof atom is not needed.

### Generating a video segment

When the client requests a video segment of a specific bitrate, the Origin needs again the server manifest file to locate the stream that contains the

| MP4 | fMP4 |
|---|---|
| ftyp | ftyp |
| mvhd | moov (hundreds of bytes) |
| moov (hundreds of KB) | mfra size (16 bytes) |
| - | mfra (a few KB) |
| - | last moof header (16 bytes) |
| - | last moof (few KB or less) |

Table 3.5: Byte range requests to the storage for generating a manifest file.

| Source video | Protocol | # HTTP requests/responses |
|---|---|---|
| MP4 | DASH/HLS | (num_bitrate_files*3 + 1 request for ism)*2 |
| fMP4 | DASH | (num_bitrate_files*6 + 1 request for ism)*2 |
| | HLS | (num_bitrate_files*4 + 1 request for ism)*2 |

Table 3.6: Number of HTTP requests and responses exchange between the Origin and backend storage when generating a manifest file (requests and responses)

requested bitrate. Then it has to fetch metadata and media samples from that bitrate representation (audio/video). Fetching this information is done with a number of byte range requests.

For MP4, there are 4 byte range requests as shown in table 3.7. As mentioned, the moov atom is needed for locating the samples that are requested. In the fMP4 case the mfra box needs to be fetched as well since with this box the Origin can locate the related segment(moof and mdat box). The segment, in this case is a pair of a moof box and mdat box which are both retrieved by one byte range request. In an HLS segment, audio and video are multiplexed, thus the number of byte range requests will be doubled for both MP4 and fMP4 cases.

Note that Origin uses byte range request in order to fetch the necessary data from S3, instead of fetching the whole video file which would be huge. Using multi-byte range requests could decrease the amount of requests send to S3, but multi-byte range requests are still not supported by S3 and some other cloud storage.

### 3.4.1 Discussion

The analysis done in this section has served in a various ways the understanding of the performance of the streaming setup and in general gives some important insights for network-based media applications:

First regarding the performance of our setup, we have seen that using fMP4 in the storage is more efficient (throughput is higher, transport coefficient close to 1) than MP4. This is caused by the size of the moov atom

| **MP4** | **fMP4** |
|---|---|
| ftyp | ftyp |
| mvhd (16 bytes) | moov (hundreds of bytes) |
| moov (hundreds of KB) | mfra size (16 bytes) |
| mdat (just the related bytes) | mfra (few KB) |
| - | moof,mdat |

Table 3.7: Byte range requests to the storage for generating a DASH segment.

| **Source video** | **Protocol** | **# HTTP requests/responses** |
|---|---|---|
| MP4 | DASH | (4 + 1 request for ism)*2=10 |
| | HLS | ( 8 + 1 request for ism)*2=18 |
| fMP4 | DASH | (5 + 1 request for ism)*2 =12 |
| | HLS | (10 + 1 request for ism)*2=22 |

Table 3.8: HTTP traffic exchange between the Origin and backend storage when generating a segment(requests and responses)

that is needed for both manifest and segment generation. The size of the moov atom in MP4 files is bigger than in fMP4 (since it contains sample boxes) and therefore the responses of S3 to Origin will have a bigger size.

Next, although this work is Origin-specific, we have shown that Origin requests in an optimal way (byte range requests) only the necessary information required by DASH and HLS, for generating a segment or a manifest file. Therefore we can consider that any dynamic packaging software that behaves reasonably as Origin does, could be used in this work.

Also, through the communication analysis, we see that there are parts of the media data, that are critical for the on-the-fly conversion server. This can be true for any media processing function. In our case this part is the moov atom and the sample tables. This information is very important for devising a scheme that improves the communication of the Origin and the Storage and solve the issues mention in section 3.3.3. This scheme is presented in the next chapter.

Finally, this work shows that file formats influence the media processing node and its conversion efficiency, as some parts of the container are critical for the processing. In this regard, we can conclude that it make sense to have a NBMP format that will be designed to optimize the processing node. For assessing such a format, the conversion efficiency metric could be used.

# Chapter 4

# Proposed Optimization using dref MPEG-4 files

This chapter describes the optimization scheme for the backend storage access that is proposed in this research. This scheme was first introduced by Unified Streaming in [40]. The design and implementation of the scheme is presented. Finally the experimental evaluation of the proposed optimization is given.

## 4.1 Optimization Scheme

As we have seen, the streaming setup that combines an on-the-fly format conversion server and a cloud storage, has some performance limitations due to the communication of the two components. To address this, we propose a simple optimization scheme that can reduce the number of requests that are send from the Origin to the Storage. This is done by placing a smart cache between the Origin and the Storage, as shown in figure 4.1. This cache does not store media data, as this would eventually expand the storage volume of the cache. Instead, it caches only the metadata of the content and the server manifest file.

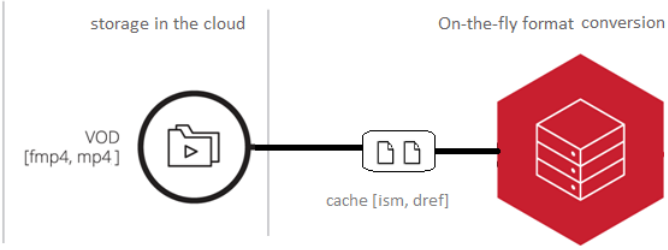



Figure 4.1: Adding a cache between the backend cloud storage and the Origin.

Caching just the metadata, is possible by leveraging an existing technology of the ISOBMFF standard in a novel way. This technology refers to the dref MPEG-4 box which is part of the specification of ISOBMFF but is not commonly used in practice. The dref is basically a data reference 'box' defined in ISOBMFF as follows: "*The data reference object contains a table of data references (normally URLs) that declare the location(s) of the media data used within the presentation.*".

In other words, the dref box, which is part of the moov box, specifies the location of the related media data. Containers, that hold only metadata of a content and no physical media data, will be called dref files. In such a container, shown in figure 4.2.b, the media data samples are located in a separate file, that is pointed by the URL of the dref box. In contrast, a normal MP4 container will contain both moov box and an mdat box, thus the URL of the dref box points to the same file (figure 4.2.a). We propose caching the lightweight dref file, together with the server manifest file (ism). This will help the Origin to obtain the necessary metadata without the need to contact the Storage.

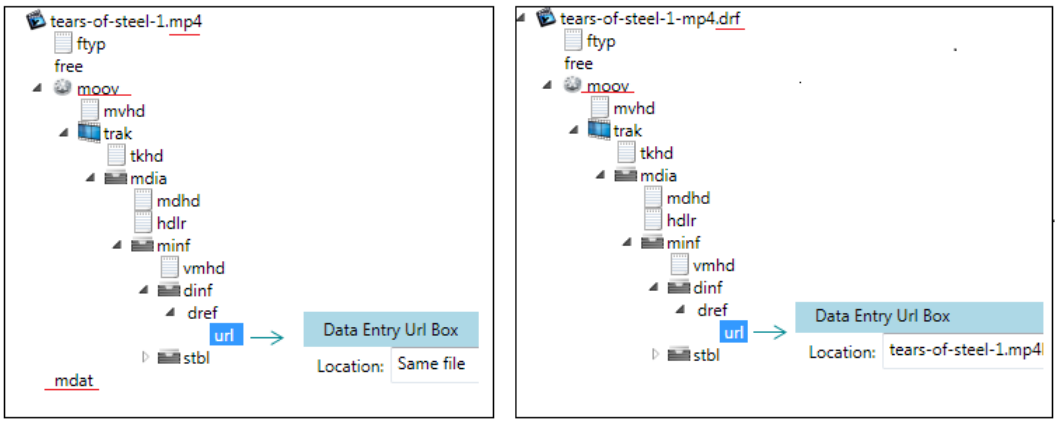



Figure 4.2: (a)Dref points to the same file, because the media data are in the same file. (b)Dref points to a different file, which contains the media data.

### 4.1.1 Motivation of caching the dref

The communication analysis of the Origin and S3 (section 3.4), revealed critical data needed for dynamic packaging, that also influence the communication. These were the server manifest file (ism) and the moov box of each bitrate representation. The dref file of representation contains only the moov box. Therefore caching it, will cause less requests to the backend storage. Specifically there will be a single byte range request for accessing

the media data for each requested segment.

Figure 4.3 shows the sequence diagram and message exchange between the Origin and the Storage, when a client request a video segment. The cache stores only the dref and the ism file, while the media data are still served from the back-end storage. The cache is implemented on the streaming server for quick access to the data.

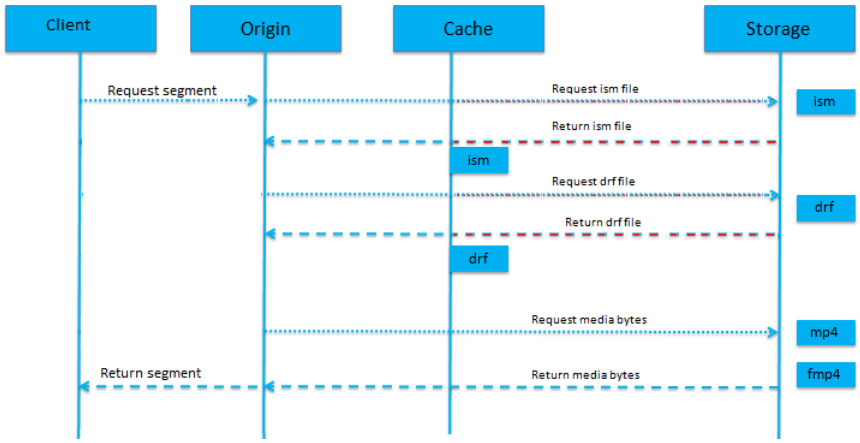



Figure 4.3: Sequence diagram of the optimization scheme. The communication indicated with red, will happen just once.

## 4.2 Implementation of the proposed scheme

The implementation of the proposed scheme consist of creating the dref MPEG-4 files for the video contents and setting up a cache in the Origin server that stores the dref and the ism files.

### 4.2.1 Creating the dref MPEG-4 file

The dref file should be created for every available bitrate representation of a content. In this work we use Unified Packager to create the dref MPEG-4 files using the command shown in 4.4 .Further the server manifest file for each content is created again, to reference the dref's of each bitrate representation, instead of the whole file(MP4 or fMP4) for each representation.

```
sudo mp4split -o sintel_720p_1000k_mp4.drf –use_dref sintel_720p_1000k.mp4
```

Figure 4.4: Creating the dref MPEG-4 file for an mp4 file encoded to 1000kbps

### 4.2.2 Implementation of the Cache

To implement the cache, we use Apache cache [41] for simplicity reasons, since the Origin runs in an Apache server. Apache cache is controlled by the modules mod_cache_disk and mod_cache of the Apache server. Mod_cache controls the cacheability of an HTTP response by implementing the RFC 2616 compliant HTTP content caching filter. Mod_cache_disk is used to store the cached responses on a disk, which is useful when proxying from a remote location.

To integrate the cache in the setup with the Origin and the Storage, a new virtual host is added to the configuration file of Apache. The Origin sits in the default virtual host(port 80) and uses the special custom directive *IsmProxyPass* to generate the byte range requests. When cache is used, these requests are redirected to the new virtual host that the cache is sitting (8080). If the request are for a dref file or the server manifest file (.ism) then the range header is removed, before the request is redirected to the Storage, with the Apache directive *ProxyPass*. This is done because Apache cache does not support caching of partial content and the Origin uses byte range requests to fetch data from the storage. Therefore with removing the range header from the dref requests, the dref is requested with one request instead of 2 range requests. This in not a problem since the Origin needs the whole dref anyway. The procedure that is described here is written in the configuration file of Apache. A part of the configuration file of is shown in figure 4.5. As shown, cache is enabled just for the dref files and the server manifest file (.ism).

Internally, for each byte range request generated by the Origin the following takes place:

1. Origin sends request to Cache with *IsmProxyPass*.

2. Request is forwarded to Storage with *ProxyPass* .

3. Storage sends an HTTP response to the cache with the requested file (ism or dref or fMP4/MP4 data).

4. Response is cached if is contains ism or dref. Then, in any case, the response is forwarded to the Origin (port 80).

When there is a cache hit, the steps 2 and 3 are eliminated. The correct behaviour of cache was tested using using Wireshark. The testing is not shown here for space considerations.

```
<VirtualHost *:80>
  ServerName usp-Origin ..
  <Directory /var/www/s3-video>
    IsmProxyPass http://localhost:8080/
  </Directory>
  ..
<VirtualHost *:8080>
  ServerName usp-cache  ..
  <LocationMatch .ism>
    CacheEnable disk
  </LocationMatch>
  <LocationMatch .drf>
    RequestHeader unset Range
    CacheEnable disk
  </LocationMatch>
  ..
```

Figure 4.5: *IsmProxyPass* redirects Origin's request to the cache (8080) and then these are sent to S3.

## 4.3 Performance Evaluation: Experimental Testbed and Methodology

### 4.3.1 Testbed Deployment and Configurations

To evaluate the proposed setup, we carried out a set of experiments over a testbed, deployed in virtual servers in the cloud using AWS as illustrated in figure 4.6. In this setup we have added Apache cache and the dref files in the storage. The Origin runs on Apache web-server on an Amazon m3.xlarge instance and the client on an m3.medium instance[43].

As a first step in evaluating the optimized setup, we choose to use Apache Benchmark (AB) instead of Tensor. This is because Tensor gives a bigger overview of the streaming performance while testing in large scale and the results are more difficult to interpret. On the contrary, AB test the lowest-level operation of video streaming and thus is easier to understand and evaluate the effect of caching.

The proposed optimization is tested with three different configurations. Each configuration differs on the location of each component of the setup . These configurations are the following:

1. Setup 1: Client & Origin & Storage in the same cloud environment.
2. Setup 2: Origin & Storage in one cloud. Client in a different cloud.
3. Setup 3: Origin and Client in one cloud. Storage in a different cloud.

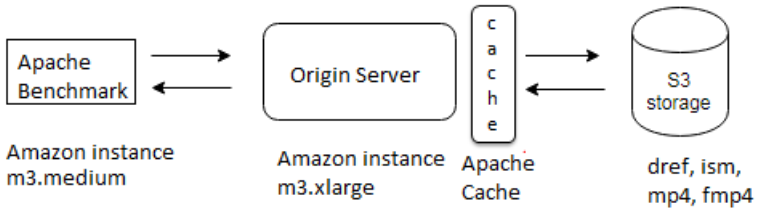



Figure 4.6: Experimental testbed for evaluating the proposed scheme.

In Setup 1, all the components are in the same cloud environment located in Frankfurt, just like the experiments described in section 3.3.2. In Setup 2, the client is moved in Ireland, far from the Origin, to simulate a more realistic scenario where client is in the edge. In the third configuration, the Origin is moved together with the client to the same cloud, to simulate the scenario of moving the Origin to the edge. Such a scenario can minimize the traffic in the CDN (generation of segments is done near the client) and increase the efficiency of CDN caching. In this case the Origin and the client instances are located in Frankfurt while the storage is moved in Ireland.

Note that the proposed scheme can have 2 caching flavours. The flavour that only caches the server manifest file (.ism) and the flavour that caches both the manifest file and the dref. Caching just the server manifest file is a known backend caching that might seem trivial and might be already implemented by the industry. However caching the dref is a smart caching that we propose and we belive that this can achieve greater performance. To show the effect of each flavour, both are shown in the results.

### 4.3.2 Tested Workload

The setups are tested with AB, by generating a significant amount of requests in a short period of time, with a specific concurrency level. The requests can be of two types: request for a manifest file and for a video segment (DASH and HLS). Each test is performed for two cases: storing the video content as MP4 an as fMP4. Each configuration is tested with different workloads, varying in a number of parameters (video, type of request, concurrency level and requests) to see how the setup performs under different loads. An overview on the tests that were performed can be found in table 4.1. For space considerations some of these results are given in the Appendix.

### 4.3.3 Key Performance Indicators

The performance of each setup is measured by the two KPI's collected by AB in the client side: average latency and the amount of requests that can

| Setup | Type of Request | Video | Concurrency | Requests |
|---|---|---|---|---|
| Setup 1 | Segment(low & high bitrate) | sintel | 1,10,20,35<br>10,20<br>90<br>150 | 1000<br>500<br>2700<br>30000 |
| Setup 2 | Segment(low & high bitrate) | sintel, elephants dream | 10 | 1000 |
| Setup 3 | Segment(low and high bitrate) | sintel, elephants dream | 10 | 1000 |
| | Manifest | sintel, elephants dream | 10<br>1 | 1000<br>100 |

Table 4.1: Overview of experiments that are performed for each configuration.

be handled per second. The second KPI reflects on the client throughput.

## 4.4 Performance Evaluation: Results

The performance evaluation of the proposed scheme is presented in two parts: First we test with performing video segment requests with AB, testing the first three configurations and second with performing manifest requests, in the same manner.

As table 4.1 indicates, different tests were done for setups 1 to 3. Here we only present the tests with 1000 requests and 10 level of concurrency. The rest of the tests are given in the Appendix. The information and conclusions that are derived from the results are discussed while presenting the results. Note that in the rest of the text the term server manifest file and ism are used interchangeably.

### 4.4.1 Segment Request (AB)

Figures 4.7 to 4.10, show the first two KPI's obtained from AB, for the setups 1 to 3, when requesting video segments of sintel, 480p and elephant's dream, 720p. The blue bars of the graphs, indicate the results when only the ism is cached, and the rest show the results when the dref's are also cached. Further the numbers in the bars, indicate the average percentage increase(or decrease) that is achieved when the dref is used on the results that are obtained from caching just the ism file (average of MP4 and fMP4 increase).

As shown, caching the dref and ism file performs best, for all setups. The highest gain is achieved when testing Setup 3 by requesting segments of sintel, 480p (figure 4.7). There, the amount of requests that are handled is tripled in all cases (different segments of sintel) and the latency is decreased by 70%. However, when requesting a big segment of elephants

dream, encoded into 2Mbps there is little or none improvement when the dref is cached. This is because a segment of elephant's dream, 720 resolution, encoded into 2Mbps has a size 11 times bigger than the same segment encoded into 200Kbps. Even when the dref is cached, still the client has to handle a big size of responses, thus the average requests/second will be low.

For a similar reason, we notice that DASH performs better (higher request/sec and lower time per request) than HLS (almost double the performance in most cases), since an HLS segments contains both video and audio data. Thus the responses will be almost twice as large as the responses for DASH segments.

Another interesting observation is that when dref is cached the file format of the source video does not matter and MP4 and fMP4 have almost similar performance, as anticipated by this approach. This allows one to avoid repackaging media collections stored using non fragmented MP4 file format.

**Comparing the Setups:**

Among the three setups, setup 1 performs best since everything is closer to the client, thus the client is served more quickly. When the client is moved far from the Origin (setup 2), more time is needed to fetch segments, thus the latency of request increases and the amount of requests per second decreases. When the Origin moves in the same cloud with the client (setup 3) and the cache only stores the manifest file, even more time is needed per request. This is because the Origin will be further away from the Storage, and there will be additional delay in fetching the media and meta data from the Storage. However, when the cache is enabled to store also the metadata, the requests to the Storage will be decreased and the fact that the Storage is further away will have less impact on the performance compared to when caching just the ism. Thus we see, that for Setup 3, when dref is cached (yellow and green bars), the time per request is less than Setup 2 and the amount of requests/second is higher (in most of the cases). Therefore, comparing Setup 2 and 3, we can conclude that Setup 3 is more beneficial when using dref, because of the results shown here and the other benefits that it offers (less traffic and caching in CDN).

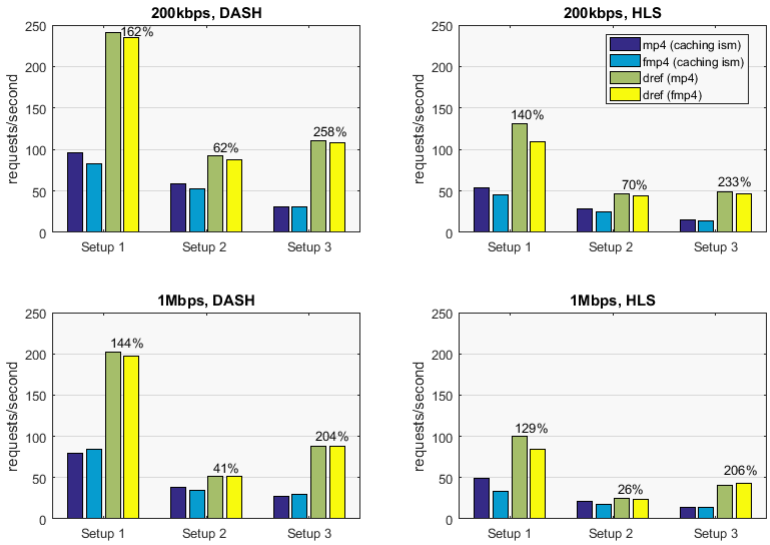



Figure 4.7: Requests/second when requesting a DASH and HLS segment from sintel,480p in two different bitrates.

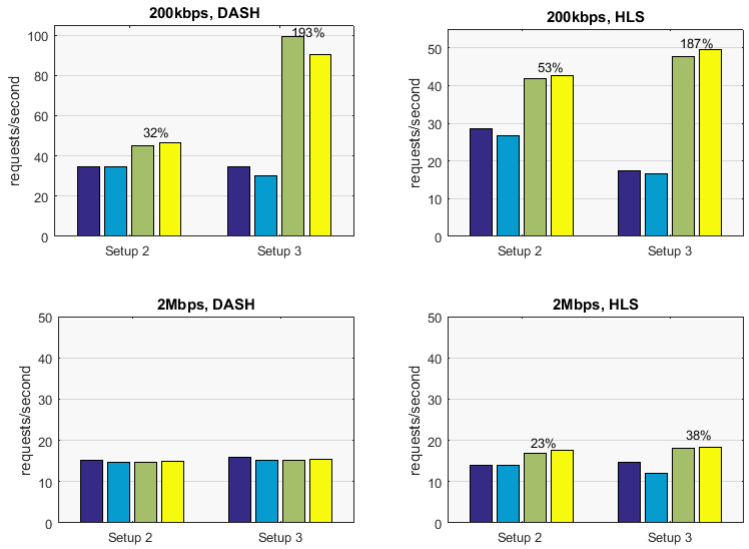



Figure 4.8: Requests/second when requesting a DASH and HLS segment from elephants dream,720p in two different bitrates.

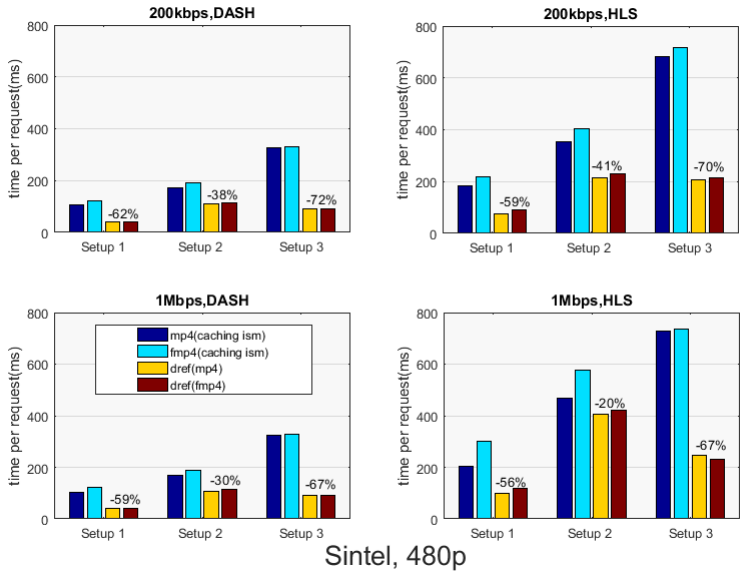



Figure 4.9: Time per request when requesting a DASH and HLS segment from sintel,480p in two different bitrates.

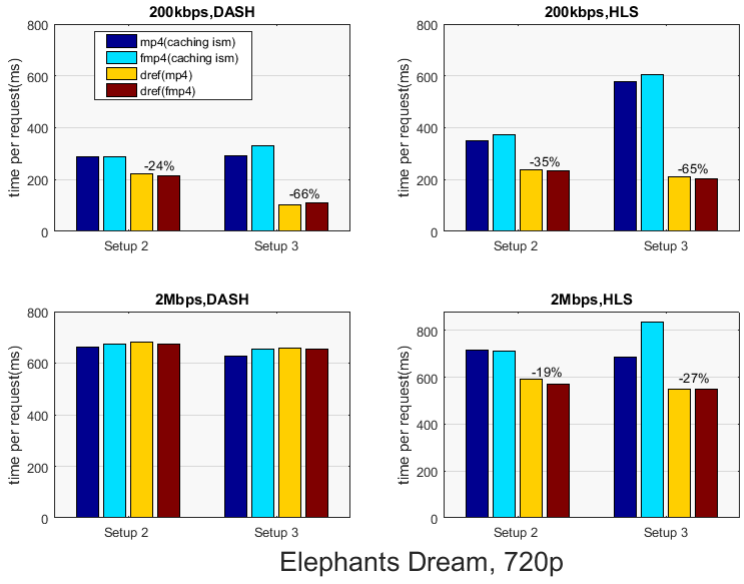



Figure 4.10: Time per request when requesting a DASH and HLS segment from elephants dream,720p in two different bitrates.

### 4.4.2 Manifest Request (AB)

Testing the performance when a client request a manifest file, is important as these results can indicate what is the impact of the proposed scheme in the start-up delay which is a key QoE metric. This is because before video playback, the video player needs the client manifest file and the initialization segment, which contains metadata. Origin generate those by using the ism file and dref, thus when these are stored in the cache, the client requests will be served faster, reducing thus the start-up latency. For space considerations we only show the latency results for the Setup 3, in figure 4.12 when requesting the DASH and HLS manifest for sintel and elephants dream.

Figure 4.12 shows the results when the DASH and HLS manifest file is requested 1000 times. The results show that the time per request is decreased up to 97% (for both DASH and HLS) when the dref is cached. This is because to generate the manifest file the information needed by Origin are the ism file and the dref files. Thus, when these are cached there will be no requests to the remote Storage. This is a promising result that indicates the reduction of the start-up delay. For this, we have make some simple experiments to measure the start-up delay. These are shown in the next subsection

Further, there are also other information that can be extracted from the figure and explained having in mind the communication analysis done in section 3.4:

- When the content is stored as MP4 and only the ism is cached, the HLS and DASH tests show similar performance.

  This is because, when the content is stored as MP4, the number of requests that are sent from the Origin to the Storage is the same for both HLS and DASH (table 3.6).

- When the content is stored as fMP4 and only the ism is cached, the latency in the DASH case is bigger than in the HLS case.

  This is because, for the fMP4 case the number of requests that the Origin does to the storage to generate a manifest file, is more in the DASH case than in the HLS case (table 3.6).

- Requesting the manifest file of Sintel,480p gives better performance than when requesting the manifest file of elephants dream, 720p.

  This is because the amount of request to the centralized Storage will be more when more bitrate representations are available, thus more time will be needed per request.

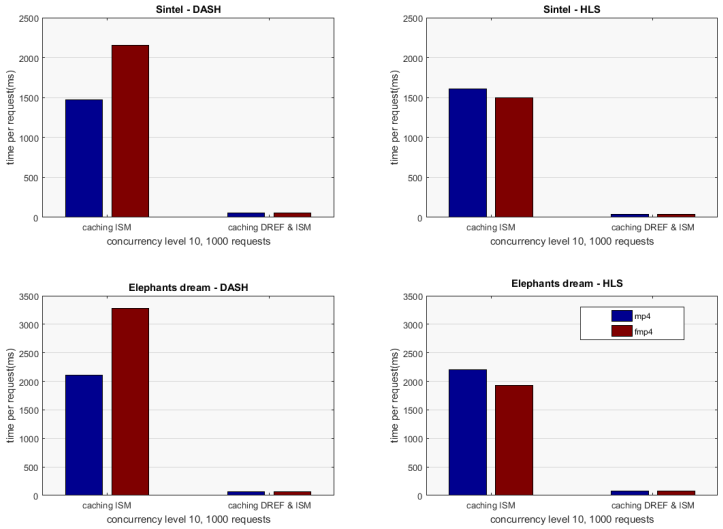



Figure 4.11: Setup 3: Time per request for 1000 requests when client request the manifest file of sintel,480p and elephant's dream, 720p.

### Measuring the startup delay

As said, the previous show that the start-up delay of video playback could decrease. For this, we have performed an experiment to measure the startup delay when streaming video.

The startup delay that the user observes when streaming a video, can be found by measuring the time from when the viewer intends the video to play (selects play), to when the first frame of the video is displayed (according to the viewer). As this can be a short interval (few seconds) measuring the time manually with a timer would be a very rough approximation. Therefore we have measured this time with a different way: First we use an online video player to stream a video and at the same time we record the whole procedure with a screen recorder software. Later, once the streaming is done, we used a video player with high precision timing (showing milliseconds) and we replayed the whole procedure in the slower speed. Then we simply note the time that the viewer selects the play button, and the time that the first frame is being noticeable. In this measurement there might be some margin of error due to the ambiguity around when exactly to note the time as this depends on the user(e.g when does the user observer the first frame can be different among users). Still it is a simple and reproducible experiment that can be considered valid even though more sophisticated approaches exist (take measurements from video player source code).

The online video player that is used in these experiments is the DASH reference video player dash.js 2.5.0 player [47], with auto-play (video stats as soon as the video is loaded -no preload). This video player is developed

by the DASH Industry Forum (DASH-IF) which develops an open source reference client implementation for DASH-AVC/264. The stream recording software that was used is Icecream Screen Recorder [45] and the video player with high precision timing is the Media Player Classic Home Cinema [46]. Further we have used two videos sintel, and tears of steel, with just one bitrate enabled for simplicity. We have performed the above procedure 10 times to measure the startup delay for each video in two streaming setups: the baseline and the proposed setup with dref caching. In both cases, we have used the configuration of setup 3, where the storage is remote from the Origin, since the distance can make the differences between the two setups more noticeable.

The bar graph below, gives the average start up delay taken by the 10 experiments. As shown, the startup delay in the proposed setup is decreased by half a second for tears of steel and for sintel almost 2 seconds. We consider these results a significant achievement, considering that a long startup delay is one of the main reasons that causes user abandonment of a video and according to Akamai's reports [44], the tolerance duration of viewers waiting the startup of a video is close to 2 seconds.

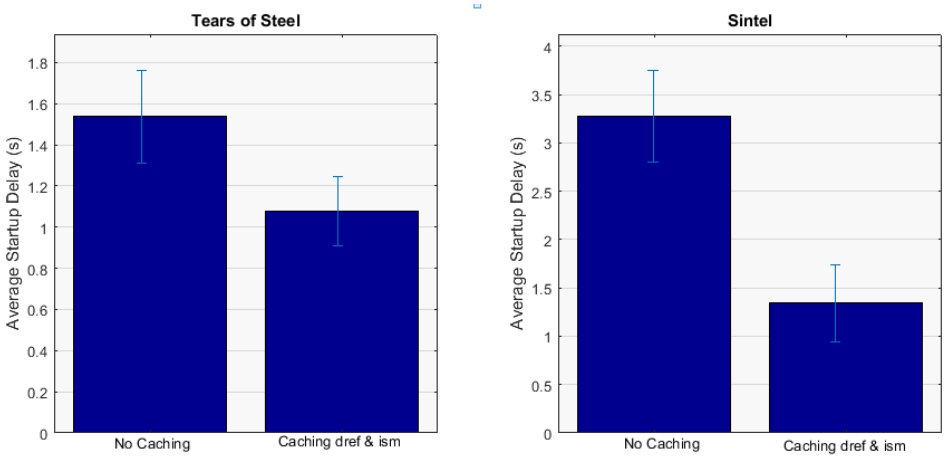



Figure 4.12: Average startup delays.

## 4.5 Conclusions

In this chapter, we evaluated the proposed scheme by testing the performance that is achieved for the lowest level operations in video streaming which consist of generating a video segment and manifest file for the client. For this, three different variations of the setup were used (Setups 1 to 3, section 4.3.1). Through this we have seen that caching the dref files and the server manifest file achieves the best performance in terms of throughput

and latency. More importantly, we have seen a significant reduction in the time of receiving a client manifest file. This could be an indication of reducing the start-up delay of the video playback. By performing a few simple experiments we have seen that indeed there is a reduction of the startup delay for at least 30%. Having in mind that a the startup delay is a very influential metric on the end user satisfaction, we consider this an important achievement.

Note that the results obtained here are not testing a streaming service as a whole, but just separate request of segments and manifest. A more realistic workload is requests with different ULRs that are recorded from a real video player. In fact, this is what Tensor does, in large scale. This is tested in the next chapter, by first tuning the streaming server to be suitable for high concurrent video streaming traffic.

# Chapter 5

# Testing Origin with Highly Concurrent Video Streaming Traffic

This Chapter completes the experimental evaluation of the proposed setup by testing it with high concurrent video traffic with a realistic workload, using Tensor. In order to do this the Apache server needs to be tuned to be able to withstand high concurrency and traffic. For this an experimental work is performed to find an Apache configuration suitable for high concurrency.

## 5.1 High concurrency in Web Servers

Web applications have to cope with increasing concurrent demand and scale to larger user bases. These applications are based on web-servers that need to handle concurrent connections and use the available resources such as CPU, RAM, and network interface capabilities. However handling high concurrent traffic in web servers while maintaining high throughput and low latencies, is not trivial. This problem has been studied a lot in the past, with the famous famous C10K problem [49] that has shown that the hardware is no longer the bottleneck for high connection concurrency, but handling large number of TCP connections given some OS constrains and limitations. For this, different I/O models(synchronous/asynchronous, blocking/non blocking) and concurrency strategies have been developed and have been implemented in the web servers. These has lead to two main competitive server architectures: thread-based architectures and event-based server architectures, each handling the incoming HTTP connections in a different way. For thread-based server architectures each incoming connection is associated with a separate thread (for mutli-threading architectures) or process (multi-process architecture) with a blocking way. In this case concurrency

is achieved by employing multiple threads at the same time. Webservers like Apache have employed this strategy. On the other hand, event-driven server architectures map a single thread to multiple connections by using an non-blocking I/O model and queuing the I/O events of each connection while handling others. The web server Nginx has adopted this architecture.

Still, using these concurrency models is not as simple as plug-and-play. Some customization on the webservers concurrent models might be needed. Consider for example that the multi-threading concurrency model, might have some limitations with utilizing efficiently the hardware resources such as limiting the number of simultaneous connections to the number of threads, high CPU and memory consumption due to context switching and the mapping of thread stack to each connection. For this, servers like Apache, have developed different strategies in overcoming these issues by offering modules that can be configured according to the application needs. For example Apache offers an event module that handles the connections in an asynchronous way.

Therefore, the webservers solve the C10K problem by offering software that can be configured for deploying web applications with high concurrent connections. Considering that video streaming applications have a client-intensive nature and high data volume, these aspects need to be considered and the webservers might need tuning to meet these needs. Our proposed setup is no exception and as we want it to be functional for large scale deployments, we test how it behaves in such high concurrency scenarios.

## 5.2 Tuning Apache for highly concurrent video streaming traffic

As the server performance and the achieved concurrency can be limited by the connections handling, when the server is subjected under heavy load, proper tuning of the server could take place to optimize the performance for scenarios of high concurrency and traffic. This is important in our setup, where multiple activities need to be handled at the same time (client connections, Storage connections, data transfer between cache and Origin) and should be able to serve large scaled deployments.

In fact, this is evident by testing the proposed setup in large scale using Tensor. When a high number of concurrent connections was used (around 90) the Apache server would crash, with the log file indicating that the maximum number of threads is reached. This can be possible be considering the following: For each client request, the Origin spawns multiple range requests in a blocking way. Thus, the threads that handle those requests, will be blocked until they receive a response. These requests are sent to the cache and then the cache will need to redirect some of these requests to S3. However if all threads are already blocked, then the cache that is

implemented as part of the Apache server, will not be able to handle the requests to S3, thus all threads could remain in waiting state, resulting to deadlock. Under heavy load, this scenario could be more frequent. Simply allowing more threads, wouldn't solve the problem as then other issues could appear (e.g memory swapping). Thereby, if the server is not carefully tuned for high concurrency, it can become the bottleneck in the streaming service and have a negative impact on the throughput and the latency.

To overcome these problems and avoid failures, we have performed an experimental work to tune the Apache server for highly concurrent video traffic. Towards this goal, first we study the architecture of Apache to understand how the connections are handled and how we can configure this in a different way, to achieve high concurrency.

### 5.2.1 Apache Architecture

Apache has a modular architecture comprised by a small core and a number of modules. The Apache core is responsible for the basic functionality of an HTTP server (listening connections, serving requests) while the different modules extend the functionality of the core. Modules can be developed separately and loaded to the server. Each module is concerned with handling one or more phases of the HTTP request processing (parsing request, map URL to file system, authentication etc). In fact the Origin that is used in the set-up, is a module of the Apache server. Figure 5.1 shows the basic architecture of Apache.

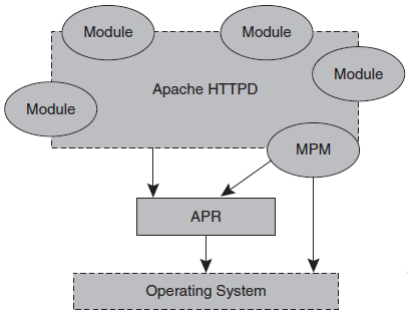



Figure 5.1: Apache Architecture from [48].

The module that can be used for tuning the server for scenarios of high concurrency and traffic, is a special-propose module called the Multi-processing module (MPM). The MPM module is the only module that can access the operating system directly, since it manages Apache operations at system level such as maintaining and managing a pool of thread and processes. Apache offers three MPM modules, differing in the way of handling threads and processes so one can use the module that best suits the application:

- **Prefork Module**: Based on process pre-forking where each process contains just one thread, handling one connection per time. Thus, more processes are needed to handle a lot of request resulting to a large memory-footprint compared to the other two multi-threaded MPMs.

- **Worker Module**: A multi-threaded module where each child process manages a pool of threads. Each thread can handle one connection and thus more connections can be handled simultaneously with fewer resources (fewer processes). This module is recommended for high traffic servers.

- **Event Module**: Similar with the worker module but keep alive connections are handled more efficiently. When a keep alive connection is established, the thread that handles this connection will not be blocked waiting for more requests from that connection, but instead it passes the control to a listener thread, so it will be available to server other requests. The event module can handle high loads more easily and is recommended to servers with extremely high hit rates. However the event MPM does not work with secure HTTP (HTTPS) and the Origin and requires a thread-safe polling function available in the OS (epoll).

In our work we have use the default module in our server which is the worker module. The MPM worker module is controlled by a number of directives such as total number of threads, initial number of processes etc. These directives can have a big impact on the streaming performance and they can be changed in order to select the configuration that best suits to the application and the desired characteristics of the server. For example the directive that controls the maximum number of total threads that can be lunched in the server, indicates the maximum number of request that can be served simultaneously. More information on these directives and their functionality can be found in Appendix B.1.

### 5.2.2 Experimental work and final configuration

To tune the MPM worker module of Apache for highly concurrency, we have performed an experimental work. Through the experimental work we have shown how the worker's directives affect the performance and we have found a configuration of the worker module that gives the best results in terms of latency, throughput and CPU usage for the tested videos. This configuration is given in figure B.9. For details on the experimental work for Apache tuning, we refer an interested reader to the Appendix B.

```
<IfModule mpm_worker_module>
ServerLimit 32
StartServers 4
MinSpareThreads 25
MaxSpareThreads 75
ThreadsPerChild 8
MaxRequestWorkers 256
MaxConnectionsPerChild 1500
KeepAlive Off
<IfModule>
```

Figure 5.2: Final Configuration

## 5.3 Large Scale Testing of Proposed setup

This section presents a large scale testing of the proposed setup with Tensor, by testing the whole streaming procedure with requests to the Origin, that are based on HTTP request sequences obtained from the respective DASH manifest. In this experiments, we have used the final tuned configuration of Apache that was found in section B.2.3.

### 5.3.1 Experimental Testbed and Tested workload

For this evaluation, the experimental setup used, is the same as the one in figure 4.6. For this test, the client, Origin and the Storage are all located in the same Amazon cloud in Frankfurt. Tensor is used as the benchmarking tool by using 90 concurrent connections and maximizing the number of requests for approximately 2 minutes. The tests are done by requesting only the DASH manifest file of the three videos in all available resolutions. Tensor is running on an c4.xlarge instance and Origin m3.xlarge. These results can be compared against the results obtained in section 3.3.

### 5.3.2 Key Performance Indicators

The KPI's used in this evaluation are the ones described in section 3.2.5: average latency, throughput on the server and conversion efficiency.

### 5.3.3 Results

#### Throughput

Figure 5.3 and table 5.1[1] shows the average saturated incoming and outgoing traffic measured in Origin, when the dref and the ism are cached.

[1] Average results for all resolutions

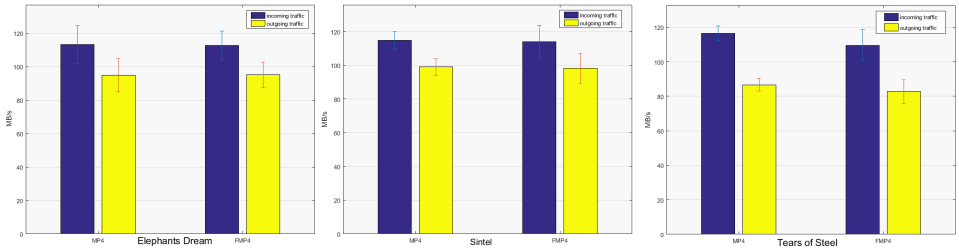



Figure 5.3: Large scale Testing: Incoming and outgoing traffic

| **file format** | **MP4** | | | **fMP4** | | |
|---|---|---|---|---|---|---|
| **source video** | ed | sintel | tears | ed | sintel | tears |
| in throughput(MB/s) | 113,14 | 114,72 | 116,36 | 112,63 | 113,90 | 109,56 |
| out throughput(MB/s) | 94,89 | 99,07 | 86,65 | 95,10 | 98,21 | 82,66 |
| conversion efficiency(out/in) | 0,84 | 0,86 | 0,74 | 0,84 | 0,86 | 0.75 |

Table 5.1: Backend Storage results with cache (video requested in DASH).

The results show that backend traffic is still higher than the front end traffic, since media data still need to be fetched from S3. However the outgoing traffic has increased in all cases. Figure 5.4 shows the gain for different resolutions of sintel and elephant's dream, comparing with the baseline results obtained in section 3.3. For the MP4 case, the increase is higher (throughput on outgoing interface is doubled in sintel, 480p) since the MP4 case was really bad when caching was disabled. For tears of steel, the proposed scheme achieves an increase of 143% and 8% for the case of storing as MP4 and fMP4, respectively. In addition, the 'conversion efficiency' for the MP4 case has improved a lot for all the videos, reaching almost 1. This is an indication that using and caching the dref, is a suitable way for optimizing the on-the-fly conversion node. Further, there is no performance variation between storing the content as MP4 or as fMP4.

### Latency

Figure 5.5 shows the average latency obtained by Tensor when the cache is disabled, when the ism is cached and when both the ism and the dref are cached. Two different resolutions are shown for both the case of storing the content as MP4 and fMP4. The average latency depends on the content and resolution. The numbers in the bars indicate the percentage decrease that is achieved related to the results when no caching takes place.

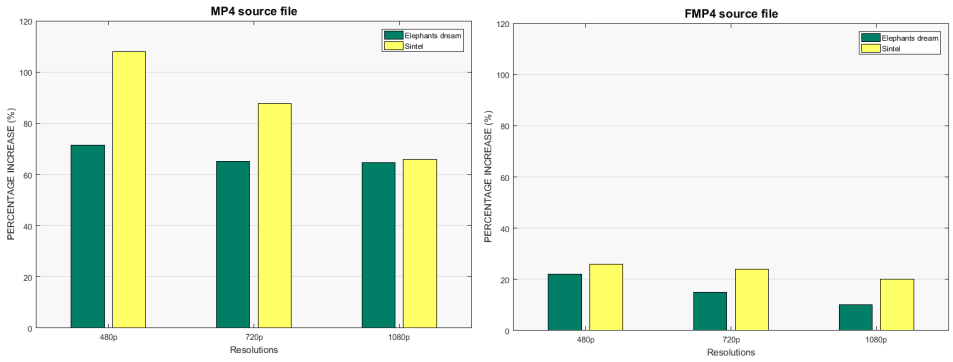



Figure 5.4: Percentage increase of the outgoing throughput when the dref is cached, for different resolutions.

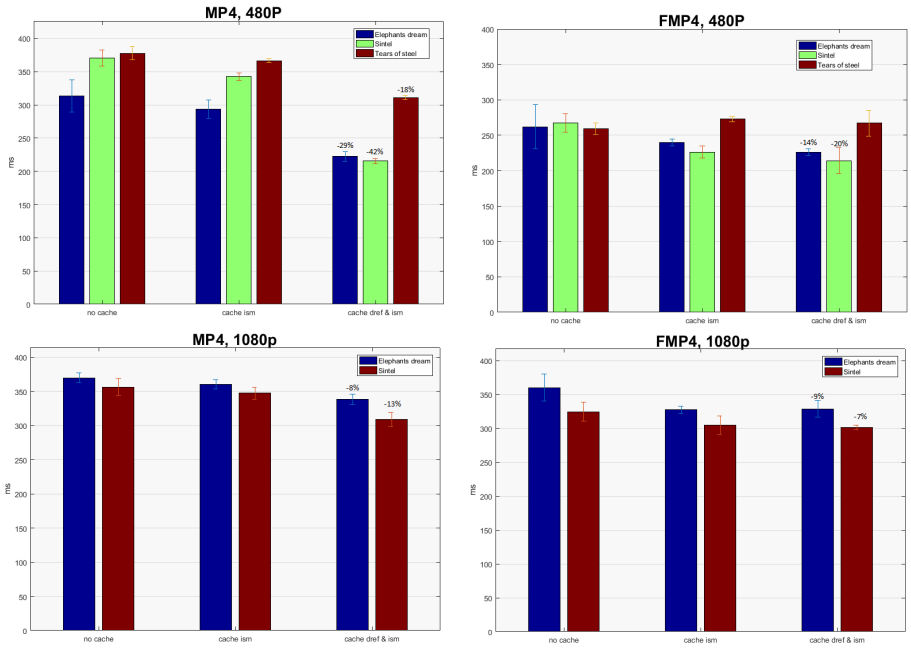



Figure 5.5: Large scale Testing: Latency obtained from Tensor

## 5.4 Conclusions

In this section, we have tested the proposed scheme in large scale. For this we had to perform an experimental work to tuned the Apache server for high concurrent traffic. Through this evaluation we have seen that using the dref has improved the conversion efficiency metric and solve the conversion overhead that is created when using backend storage. With the use of this metric we can conclude two things: First the proposed scheme indeed improves the video streaming performance and is a container-agnostic solution, since the variations in the performance when using as storage format MP4 and fMP4 are diminished. This gives great benefits for the content owners that use traditional MP4 media files. Second, this metric can be useful and valid for assessing the use of a new storage format (in our case is not really new since we use the dref) that can be used to optimized a media processing node in the network, which is the goal of the NBMP emerging standard.

# Chapter 6

# Conclusions and Future Work

In this thesis, we have looked into a streaming setup that consist of the an on-the-fly format conversion server and a cloud object-based storage. Such a setup deals with the multi-protocol nature of video streaming in an efficient way by reducing the storage costs and the content preparation effort. A performance evaluation of the setup was performed using different file formats and we were able to identify the performance limitations that are imposed by the communication of the object storage and the conversion server. Further we have looked into the data that are exchanged between the storage and the conversion server. This was useful for understating the evaluation findings. Also through the communication analysis, we were able to identify critical data for the on-the-fly format conversion server, that are imposed by the different standards. Based on this, a simple optimization caching scheme based on the dref box of the ISOBMFF standard was used. This scheme can reduce the number of requests to the object storage by caching the critical data for the on-the-fly conversion operation. We have evaluated the proposed scheme in realistic cloud configurations with realistic workloads. The results have shown that this approach has improved network performance with increasing the throughput and decreasing the latency. Furthermore, simple experiments have shown that the starup delay of the video playback can be reduced, with a maximum reduction of 60% for sintel.

An important research aspect of this work is the fact that the setup that we have looked into, is aligned with the architecture of the NBMP emerging standard of MPEG, where the media processing node is separated from the media source. We consider this as an important feature of the thesis, as this standard can help the media domain to evolve and adapt in the next generations network environments. We believe that through the work of this thesis, some important insights can be derived that can be useful for the NBMP standard. For example, we have shown that parts of the media containers are

critical for a media processing operation and they way these are stored, can influence the efficiency of the media processing node. This shows two things: First, it makes sense to have a common NBMP format that will be designed in a way that the critical data will be efficiently stored to optimize the media processing node. Also considering the fact that the critical data for a media processing node can be imposed by the media protocols e.g DASH, these can be used to indicate inefficiencies in the existing protocols that can be improved. Second, the conversion efficiency metric that we introduce, could be useful for assessing a future NBMP format, since we have seen that this is related to the format used in the storage and it can be defined for any media processing node. In this case, the metric should by studied more in order to define in a solid way how this affected by the storage format. Through our work, we have seen that this metric is not only affected by the storage format but is a function of different factors: format in the storage (MP4/fMP4), client request and protocol (DASH/HLS), communication protocol between storage and processing node (HTTP/file access), the media processing unit and last the requested content. Future work could focus on defining this function that will fully describe the conversion efficiency.

In addition, in the future, the overall implementation of the scheme could be improved by using a different approach to implement the cache of the proposed scheme, as mention in B.2.4. Furthermore this work could be extended by using a more complex media processing function . For example the media processing function could be an on-the-fly conversion server that is also responsible for stitching content from different sources e.g for personalization streams and advertisements. Then the setup differs as there is an additional media source entity. In this case, it would be interesting to see how the dref approach performs.

# Bibliography

[1] Cisco Global Cloud Index:Forecast and Methodology, 20152020, Cisco technical report

[2] Microsoft. Smooth Streaming Protocol. [MS-SSTR] - v20150630, June 2015. http://download.microsoft.com/download/9/5/E/95EF66AF-9026-4BB0-A41D-A4F81802D92C/[MS-SSTR].pdf.(visited on 20/12/2017.)

[3] Adobe Systems Incorporated. HTTP dynamic streaming specification. Version 3.0, August 2013. http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/hds/pdfs/adobe-hds-specification.pdf (visited on 20/12/2017.)

[4] Pantos, R., May, W. 2010. HTTP Live Streaming. IETF draft, November 2015.https://tools.ietf.org/pdf/draft-pantos-http-live-streaming-18.pdf

[5] I. Sodagar, "The MPEG-DASH Standard for Multimedia Streaming Over the Internet," in IEEE MultiMedia, vol. 18, no. 4, pp. 62-67, April 2011.

[6] H.264/14496-10 AVC Reference Software Manual (revised for JM 19.0)

[7] G. J. Sullivan, J. R. Ohm, W. J. Han and T. Wiegand, "Overview of the High Efficiency Video Coding (HEVC) Standard," in IEEE Transactions on Circuits and Systems for Video Technology, vol. 22, no. 12, pp. 1649-1668, Dec. 2012.

[8] VP9 [Online] Available: https://www.webmproject.org/vp9/ (visited on 20/12/2017.)

[9] MP3 [Online] Available: https://en.wikipedia.org/wiki/MP3 (visited on 20/12/2017.)

[10] ISO (1997). "ISO/IEC 13818-7:1997, Information technology – Generic coding of moving pictures and associated audio information – Part 7: Advanced Audio Coding (AAC)".

[11] 3rd Generation Partnership Project (3GPP) [Online] Available : http://www.3gpp.org/about-3gpp (visited on 20/12/2017.)

[12] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen, "What happens when HTTP adaptive streaming players compete for bandwidth?," in Proc. NOSSDAV, 2012, .

[13] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari, "Confused, timid, and unstable: picking a video streaming rate is hard," in IMC '12: Proceedings of the 2012 ACM conference on Internet measurement conference. New York, New York, USA

[14] J. W. Kleinrouweler, S. Cabrero, R. van der Mei and P. Cesar, "Modeling Stability and Bitrate of Network-Assisted HTTP Adaptive Streaming Players," 2015 27th International Teletraffic Congress, Ghent, 2015,

[15] Ricky K. P. Mok, Xiapu Luo, Edmond W. W. Chan, and Rocky K. C. Chang. 2012. QDASH: a QoE-aware DASH system. In Proceedings of the 3rd Multimedia Systems Conference (MMSys '12). ACM, New York, NY, USA.

[16] Abdelhak Bentaleb, Ali C. Begen, and Roger Zimmermann. 2016. SDN-DASH: Improving QoE of HTTP Adaptive Streaming Using Software Defined Networking. In Proceedings of the 2016 ACM on Multimedia Conference (MM '16). ACM, New York, NY, USA.

[17] Panagiotis Georgopoulos, Yehia Elkhatib, Matthew Broadbent, Mu Mu, and Nicholas Race. 2013. Towards network-wide QoE fairness using openflow-assisted adaptive video streaming. In Proceedings of the 2013 ACM SIGCOMM workshop on Future human-centric multimedia networking (FhMN '13). ACM, New York, NY, USA.

[18] Jan Willem Kleinrouweler, Sergio Cabrero, and Pablo Cesar. 2016. Delivering stable high-quality video: an SDN architecture with DASH assisting network elements. In Proceedings of the 7th International Conference on Multimedia Systems (MMSys '16). ACM, New York, NY, USA.

[19] F. Jokhio, A. Ashraf, S. Lafond and J. Lilius, "A Computation and Storage Trade-off Strategy for Cost-Efficient Video Transcoding in the Cloud," 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, Santander, 2013.

[20] H. Zhao, Q. Zheng, W. Zhang, B. Du and H. Li, "A Segment-Based Storage and Transcoding Trade-off Strategy for Multi-version VoD Systems in the Cloud," in IEEE Transactions on Multimedia, vol. 19, no. 1, pp. 149-159, Jan. 2017.

[21] Rufael Mekuria, Jelte Fennema, and Dirk Griffoen. 2016. Multi-Protocol Video Delivery with Late Trans-Muxing. In Proceedings of the 2016 ACM on Multimedia Conference (MM '16). ACM, New York, NY, USA.

[22] Daniel Silhavy, Stefan Pham, and Stefan Arbanowski. 2017. Performance considerations of HTML5-based dynamic packaging for media streaming. In Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'17). ACM, New York, NY, USA, 7-12.

[23] Marina Kalkanis, Video Factory: Transcoding Video for BBC iPlayer. [Online] Available: `http://www.bbc.co.uk/blogs/internet/entries/eb9d3ca8-56bb-39a0-b990-07e14c5996f4` (visited on 20/12/2017)

[24] ISO/IEC 14496-12:2005 Information technology – Coding of audio-visual objects – Part 12: ISO base media file format

[25] International Organization for Standardization, "ISO/IEC International Standard 13818: Generic coding of moving pictures and associated audio information", October 2007,

[26] CodeShop B.V., "Unified Streaming," 5 4 2016. [Online]. Available: `www.unified-streaming.com.` (visited on 20/12/2017)

[27] Wowza [Online] Available: `https://www.wowza.com` (visited on 20/12/2017)

[28] Yury Izrailevsky, Stevan Vlaovic and Ruslan Meshenberg, Completing the Netflix Cloud Migration [Online] Available: `https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration` (visited on 20/12/2017)

[29] ISO/IEC JTC1/SC29/WG11 MPEG2017/N17262, Network Based Media Processing, Use cases and draft requirement for NBMP (v2).

[30] Amazon Web Services [Online] Available: `https://aws.amazon.com/` (visited on 20/12/2017)

[31] Amazon Simple Storage Solution [Online] Available: `https://aws.amazon.com/s3/` (visited on 20/12/2017)

[32] Apache HTTP server benchmarking tool[Online] Available: `http://httpd.apache.org/docs/current/programs/ab.html`

[33] W. Glozer, "WRK Modern HTTP benchmarking tool," 8 4 2016. [Online]. Available: `https://github.com/wg/wrk`. (visited on 20/12/2017)

[34] Performance Co-Pilot [Online] Available: http://pcp.io/ (visited on 20/12/2017)

[35] Abe Wiersma. 2016. Determining meaningful metrics for Adaptive Bitrate Streaming HTTP video delivery Bachelor thesis. University of Amsterdam (UVA), Amsterdam, The Netherlands

[36] FFMPEG tool. [Online]. Available: https://www.ffmpeg.org/ (visited on 20/12/2017)

[37] Unified Packager. [Online]. Available: http://docs.unified-streaming.com/documentation/package/index.html (visited on 20/12/2017).

[38] Unified Capture. [Online]. Available: http://docs.unified-streaming.com/documentation/capture/index.html (visited on 20/12/2017).

[39] M Seufert, S Egger, M Slanina, T Zinner, A survey on quality of experience of HTTP adaptive streaming

[40] Remote Storage Reducing Latency [Online]. Available: http://docs.unified-streaming.com/documentation/vod/optimizing-storage-caching/creating-the-index-mp4-files (visited on 20/12/2017)

[41] Apache Caching Guide: https://httpd.apache.org/docs/2.4/caching.html

[42] Hypertext Transfer Protocol – HTTP/1.1 Section 13 o fRFC2616-HTTP caching

[43] Amazon EC2 Instance Types [Online] Available: https://aws.amazon.com/ec2/instance-types/

[44] S. Shunmuga Krishnan and Ramesh K. Sitaraman. 2012. Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs. In Proceedings of the 2012 Internet Measurement Conference (IMC '12). ACM, New York.

[45] Icecream Screen Recorder tool [Online] Available: https://icecreamapps.com/Screen-Recorder/

[46] Media Player Classic-HS [Online] Available: https://mpc-hc.org/

[47] DASH Industry Forum- DASH Reference Client 2.5.0 [Online] Available: http://dashif.org/reference/players/javascript/v2.5.0/samples/dash-if-reference-player/index.html

[48] Nick Kew, The Apache Modules Book: Application Development with Apache

[49] Kegel,Dan: The C10k Problem, Tech. Rep., Kegel.com

# Appendix A

# Additional Results when Evaluating the proposed setup using AB

In this section we present some additional results of the experiments that were done as part of the evaluation of the proposed setup in Chapter 4. Specifically we show some of the experiments described in table 4.1, mostly for Setup 1.

## A.1 Experiments with high concurrency level (Setup 1)

The tests with high concurrency (90 and 150)[1] are mainly used to compare the proposed setup with the baseline setup (Chapter 3), when no caching takes place. This is because the same experiment is performed with the baseline setup, so its fair to compare them. The following figures (A.1-A.4), show the two KPI'S collected by AB, when requesting a segment of sintel (specific bitrate and protocol) in three different cases: the baseline configuration and the two flavors of the proposed optimization: caching ism and caching ism and dref. Figures A.1 and A.3 show the results when performing 2700 requests and figures A.2 and A.4, the results for 30000 requests. From the figures, we make the following observations:

- Caching only the ism file, does not give significant gain in most cases, compared to not using a cache, probably due to the small size of it (a few KB). The highest gain achieved when caching the ism, is observed in the case of requesting 2700 DASH fmp4 segments encoded into 1Mbps (increase number of requests/second by 74% and decrease

[1] Tuned configuration is used(chapter 5)

time per request by 43%). There is a case when the setup reacts very irregular and caching the ism performs worst. This is when requesting a dash segment of 200kbps. This might be caused by an overhead created by sending everything (metadata and video files) through the additional proxy host of the cache, while the cache is not storing anything expect the ism.

- Over the three cases (no cache, cache the ism, cache ism and dref), caching the dref and ism file performs best or as good as the baseline results. The gain that is achieved depends in the concurrency level, the bitrate of the segment and the protocol used. The percentage increase (or decrease for time per second) of using dref, over the baseline results are indicated in the figures.

- The results verify the fact that when dref is cached, mp4 and fmp4 perform similar, in contrast with the case of not using cache, where fmp4 and mp4 have different performance.

- When delivering a segment of higher bitrate, a lower requests/second amount is achieved (and higher latency) compared to delivering a lower bitrate segment. This is because a segment encoded into higher bitrate has a bigger size than a lower bitrate segment.

- DASH performs better(higher request/sec and lower time per request) than HLS (almost double the performance). This can be explained by the fact that an HLS segment includes both video data and audio data.

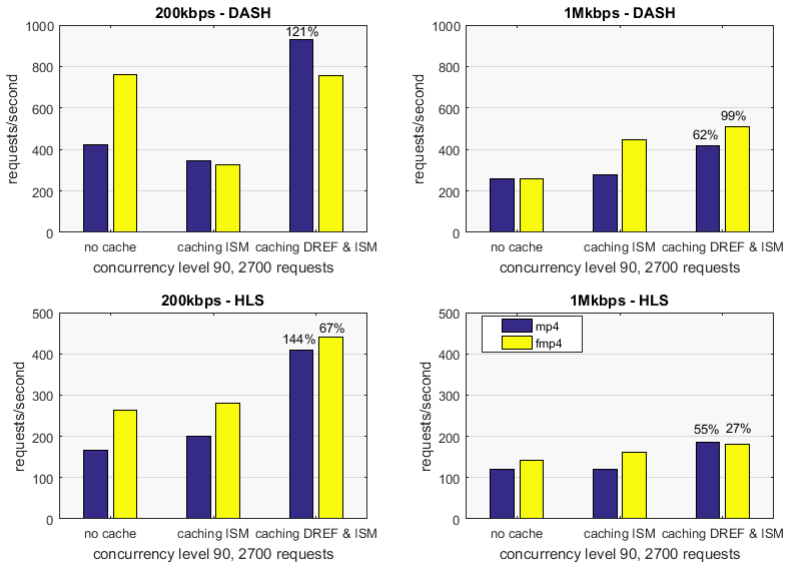



Figure A.1: Requests/second when requesting a DASH and HLS segment from sintel, in two different bitrates. Concurrency level=90, number of requests=2700.

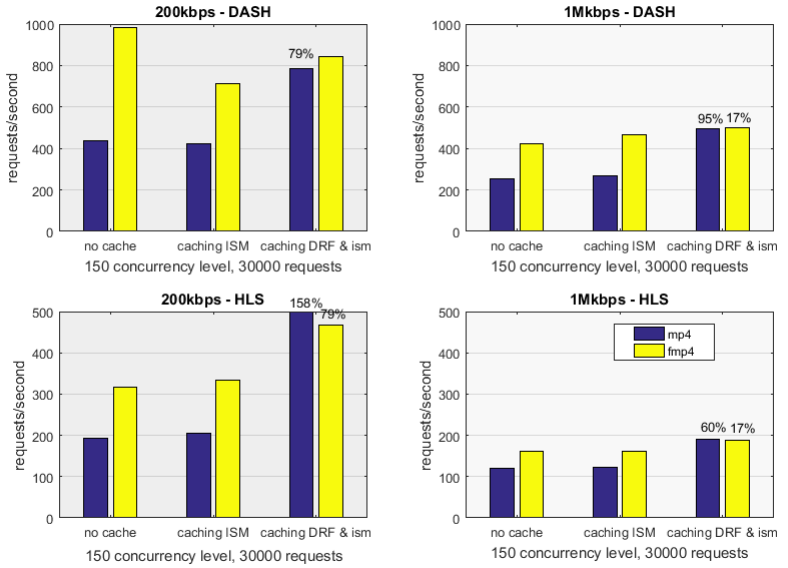



Figure A.2: Requests/second when requesting a DASH and HLS segment from sintel in two different bitrates. Concurrency level=150, number of requests=30000.

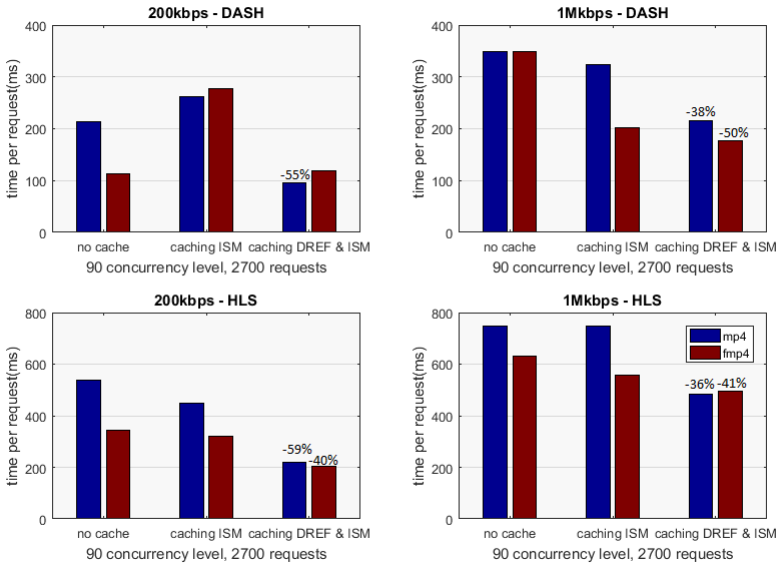



Figure A.3: Time per requests(ms) when requesting a DASH and HLS segment from sintel, in two different bitrates. Concurrency level=90, number of requests=2700.

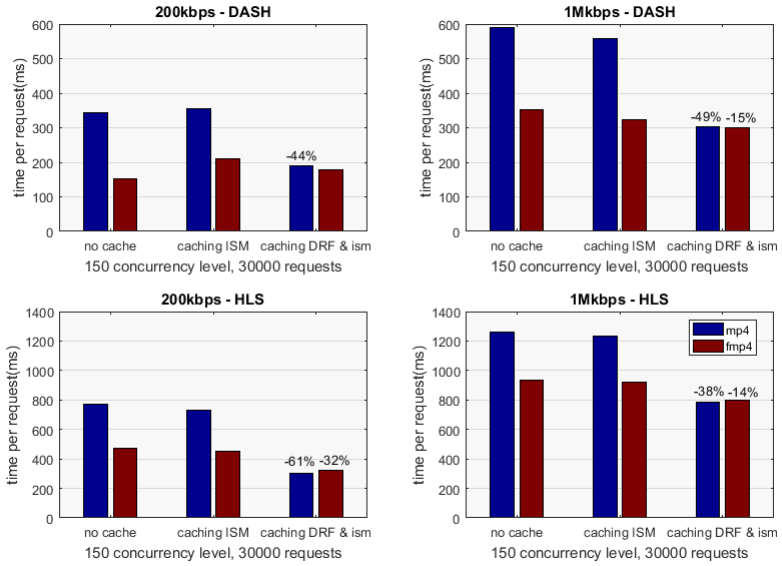



Figure A.4: Time per requests(ms) when requesting a DASH and HLS segment from sintel, in two different bitrates. Concurrency level=150, number of requests=30000.

## A.2 Experiments with increasing concurrency level (Setup 1)

Figures A.5 to A.8 show how the two KPI's of section 4.3.3, are influenced by increasing the concurrency level. Again, the two cache versions are shown here: caching just the ism, caching ism and the dref. The results of caching only the ism file are indicated with the bars denoted as mp4 and fmp4. Figures A.5 and A.6 give the results when the number of request to be performed at a time are 1, 10, 20 and 35 with 1000 requests and figures A.5 and A.8 show the results when the concurrency level is increased from 10 to 20 for 500 requests.

The most important observation in these results, is that in all cases, using the dref performs better than just caching the ism file. Also when the concurrency increases, the requests per second increases as more requests can be send at once.

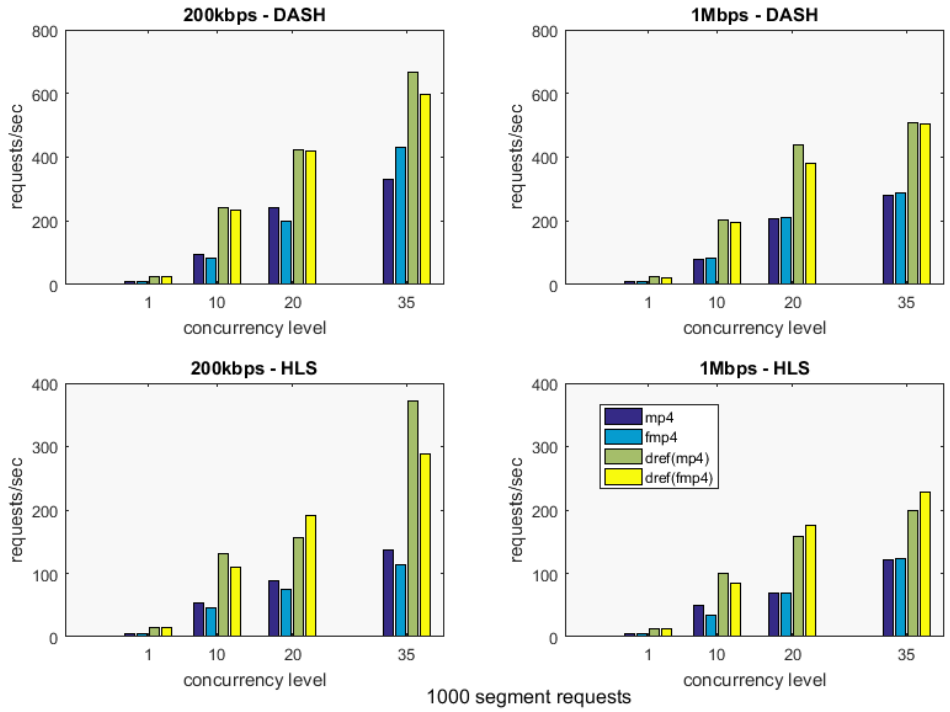



Figure A.5: Requests per second is plotted against concurrency level. For each case there are 1000 requests for a sintel segment in two different bitrates and protocols.

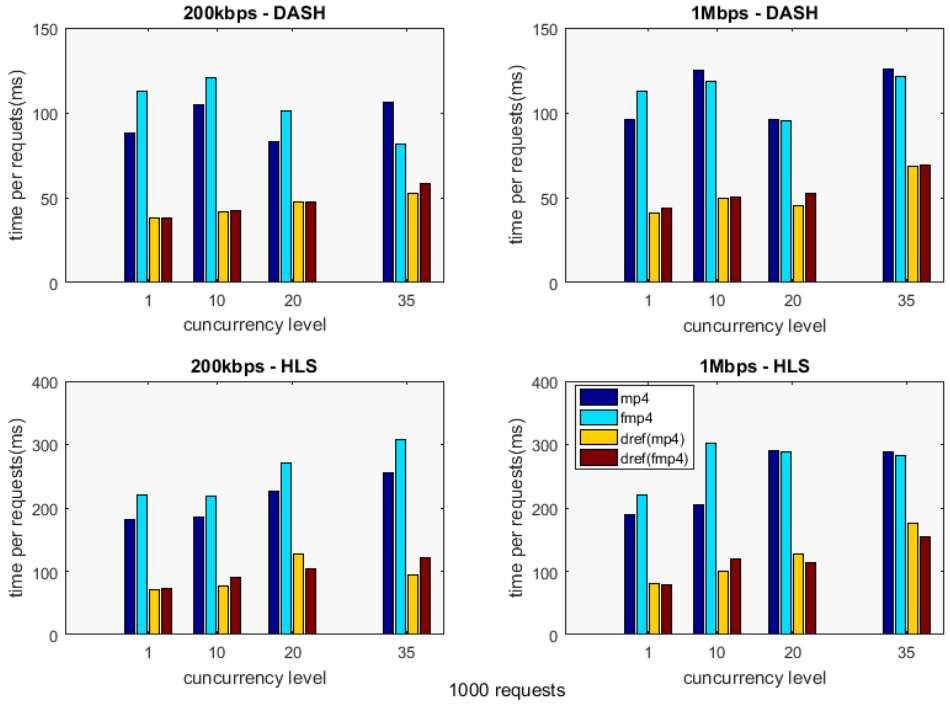



Figure A.6: Average time per request against concurrency level (1000 requests).

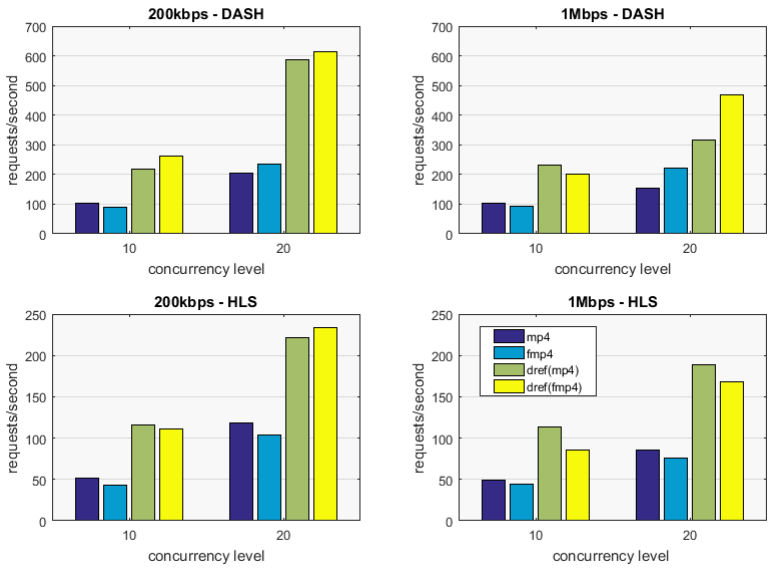



Figure A.7: Requests per second against concurrency level. (500 requests for sintel segments).

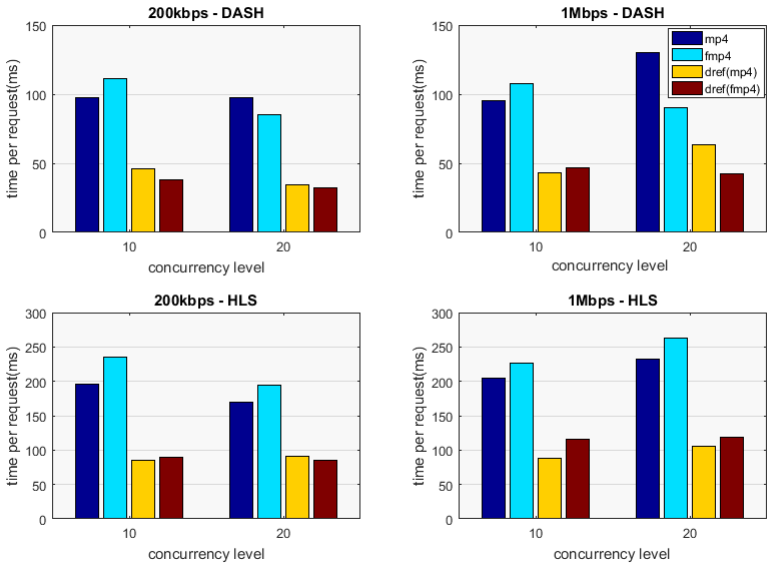



Figure A.8: Average time per request againest concurrency level (500 requests for sintel segments).

## A.3 Requesting the manifest

Figure A.9 show the results when the DASH and HLS manifest file is requested 1000 times, with 1 connection, for Setup 3. As discussed in section 4.4.2 the latency for a manifest has a huge decrease when the server manifest file and the dref are cached. Further figure A.10 show the results when requesting the manifest file from sintel, for the three setups.

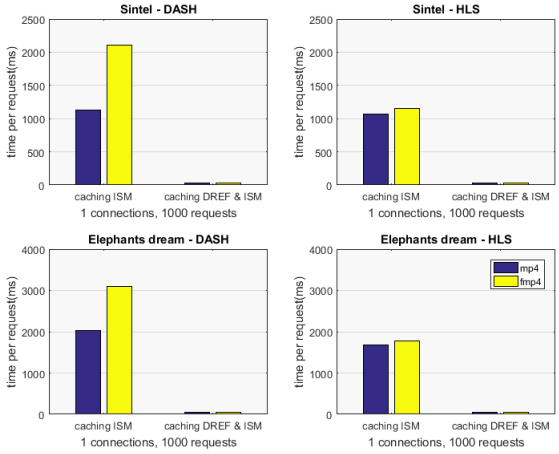



Figure A.9: Setup 3: Time per request for 1 connections and 1000 requests. The client is requesting a manifest file (DASH and HLS) for sintel,480p and elephant's dream, 720p.

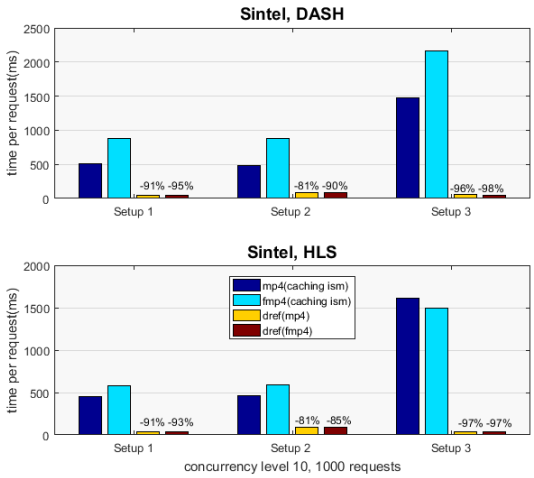



Figure A.10: Requesting manifest file from the three setups.

# Appendix B

# Tuning Origin for Highly Concurrent Video Streaming Traffic

In this Chapter we show the experimental work that is performed in order to tune the Origin server for high concurrent video streaming traffic. The experimental work is done by generating a high load on the server using Tensor and monitoring the server behaviour while twiddle specific knobs of the worker multi-threaded module of Apache. Through this we specify a configuration of the module that is suitable for high concurrency in our setup.

Note that analyzing concurrency concepts and techniques is a huge topic by itself and is out of the scope of this thesis. In this work we just give some basic insights on how different parameters affect the streaming performance and how is better to change them when using such a setup. This experimental work can also be useful for others that use different setups and want to tuned their streaming server for high concurrency.

## B.1 Multi-processing Worker module

In our work we have use the default module in our server which is the worker module. The MPM worker module is controlled by a number of directives. These directives can be changed in order to select the configuration that best suits to the application and the desired characteristics of the server. The way the module works and the control of each directive is described below (figure B.1):

Initially the parent process, lunches a number of child process specified by ***StartServers***. Each child process, maintains a pool of a fixed number of threads that is specified by ***ThreadsPerChild***, and a listener thread. The listener thread listens for new incoming requests, and once a request

arrives it pass the request to an idle thread within the process. When all the threads of a process are busy with handling requests, the listener thread will not longer listen to incoming requests. The maximum number of total threads that can be lunched in the server is specified by ***MaxRequestWorkers*** which indicates the maximum number of request that can be served simultaneously. The maximum number of child processes is found by dividing the total number of threads to the number of threads per child and is set using the directive ***ServerLimit***. The server always maintains a pool of idle threads that can serve new incoming requests. The number of idle threads is controlled by ***MinSpareThreads*** and ***MaxSpareThreads***. Depending on the number of idle threads, the server will either kill or spawn new threads, to meet the minimum and maximum values. Finally ***MaxConnectionsPerChild*** specifies the number of connections that a process can serve before it dies.

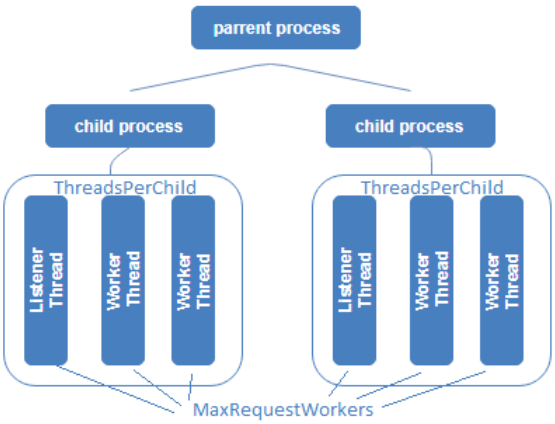



Figure B.1: Apache MPM module.

## B.2 Tuning the Apache Server

### B.2.1 Methodology

To tune Apache for high traffic load, an experimental work is performed, where incremental changes are done to the worker module while monitoring the performance.

The most important parameters of the MPM worker module are *ThreadsPerChild*, *MaxConnectionsPerChild* and *MaxRequestWorkers*. For each parameter we perform a number of tests while varying the value of it and monitoring the effect of each value, in order to find the best setting. The experimental testbed that is used, is the same setup as figure 4.6, with using Tensor as the benchmark tool. Tensor is used since it can generate a

heavy load and gives server-side characteristics which are more helpful for these experiments. Tensor runs an Amazon instance c4.xlarge to be able to handle the heavy load. Each test runs for approximately 2 minutes with 90 connections. The number of connections remains constant since with 90 connections the throughput has already reached the saturation point.

### B.2.2 Experimental Work

As mentioned the experimental work consist of incremental changes on three parameters of MPM worker module:

#### Varying *ThreadsPerChild*:

First we vary the parameter *ThreadsPerChild*, while keeping constant the other parameters of the module. We choose to decrease this value from 64 to 32, 16 and eventually to 8 threads. This is because threads from the same process can be dependent, as they can handle request that depend on each other (request from client, causes multiple byte range requests to cache). A high number of dependent threads under heavy load, can cause a slow processing of the requests and can result to low throughput. Further, the synchronization needed between these threads, could increase CPU usage. Also, if the number of *ThreadsPerChild* is high when a process crashes, more client connections will be affected. These facts, indicate that decreasing the number of threads per process might be better.

The configuration that is used for this experiment is shown in figure B.2. This will is the starting configuration for all the experiments and its changed during the experimental work.

```
<IfModule mpm_worker_module>
ServerLimit 250
StartServers 4
MinSpareThreads 5
MaxSpareThreads 10
ThreadsPerChild 64
MaxRequestWorkers 256
MaxConnectionsPerChild 500
KeepAlive off
<IfModule>
```

Figure B.2: Configuration used when changing ThreadsPerChild

Table B.1 shows the results for each experiment when testing with the video tears of steel stored as MP4, and when dref is cached.The results show that when *ThreadsPerChild* decreases, the outgoing throughput is increasing with the performance of the sever becoming more stable, indicated by a

decrease in the standard deviation of the throughput samples obtained by Tensor. However the low throughput that we get for *ThreadsPerChild*=64 and 32, could be due to the low value of *MaxConnectionsPerChild*. Using 64 or 32 threads, are too many when the process handles less connections (500 in this case) and most of them will probably be unused. This will be discussed later in the section dedicated for *MaxConnectionsPerChild*. Still, even when *MaxConnectionsPerChild* is higher, the throughput for these two cases(64 and 32) is still lower than the other cases. Figure B.3 summarized in a graphic way, the average results for throughput, indicating that using 8 threads per process gives higher throughput.

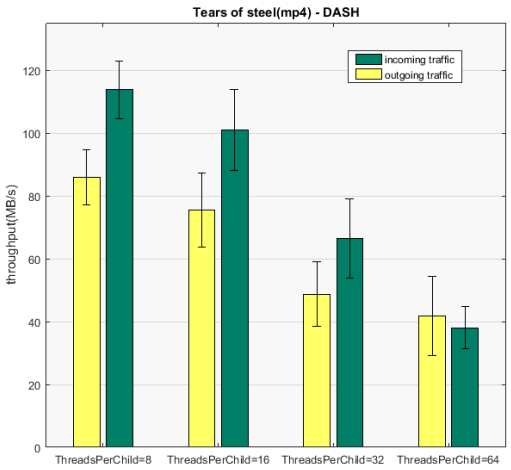



Figure B.3: Average incoming and outgoing throughput for different *ThreadsPerChild* values. (tears of steel, stored as mp4, requested in DASH).

| **ThreadsPerChild** | **64** | **32** | **16** | **8** |
|---|---|---|---|---|
| incoming throughput (MB/s) | 39,2 | 66,44 | 101,03 | 113,75 |
| outgoing throughput (MB/s) | 39,2 | 47,35 | 83,41 | 88,64 |
| outgoing traffic standard deviation | 25,11 | 20,46 | 23,74 | 17,69 |
| CPU system (mean) | 16% | 22% | 22% | 23% |
| CPU user (mean) | 10% | 14% | 17% | 18% |
| mean latency(ms) | 484 | 383,87 | 364,30 | 316,39 |

Table B.1: Results when requesting DASH streaming of with tears of steel, which is stored as mp4.Dref is cached.

**Changing *MaxConnectionsPerChild*:**

To find the most the most appropriate value for *MaxConnectionsPerChild* different values were tested. The values that are tested are 500, 1000, 1500, 3000 and unlimited, which means processes never expire. We choose to increase this value since with a low value, processes will be often restarted. The disadvantage of this is that restarting processes will interrupt on-processing connections and requests will need to be dropped. This can cause instability on the throughput of the Origin and increase the latency. On the other hand, a high value will cause long running processes that will accumulate memory and increase the probability of memory leaks.

Interesting results were observed when testing with different videos. For low bitrate videos (480p) the throughput increases when *MaxConnectionsPerChild* increases. On the other hand, for higher bitrate videos (e.g tears of steel), there is not significant difference in the throughput when changing the parameter. (figure B.4[1]).

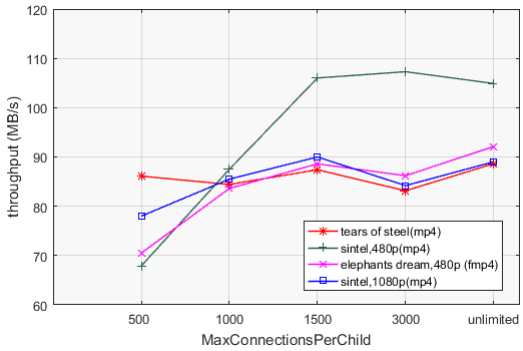



Figure B.4: Average saturated outgoing throughput versus *MaxConnectionsPerChild* for different videos.

Moreover, the parameter *MaxConnectionsPerChild* affects the stability on the server performance. This is indicated by the amount of standard deviation on the throughput samples, since Tensor saturates the throughput, so ideally it should remain constant. For example when *MaxConnectionsPerChild* is set to a relatively low value (500), the throughput measurements have a big standard deviation for all videos (figure B.5) and the performance in not stable indicated by high and low peaks in the throughput graphs obtained from Tensor. This non stability can be caused by the frequent restart of a process. Also, different videos have different optimal setting of this parameter. For example, tears of steel has the lowest standard deviation at 1500 while sintel at 3000 (B.6 and B.7). This might be due to the different

[1]The line graph does not imply linear relationship. Is just used for visualization purposes

encoding process or that the two videos have a different bitrate range.

From these tests we concluded that the setting of *MaxConnectionsPerChild*=1500, gives satisfactory results in terms of throughput and stability for all videos.

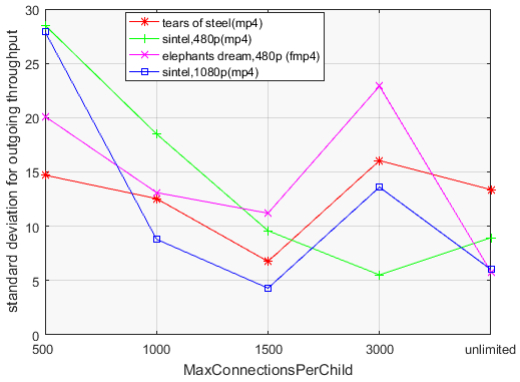
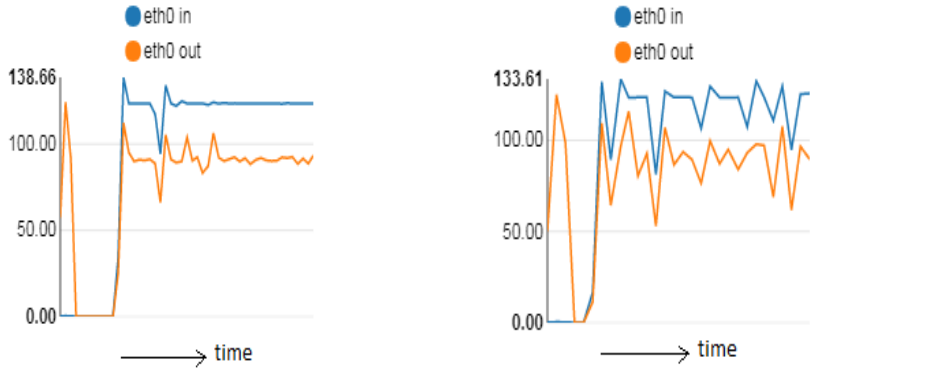



Figure B.5: Standard deviation of throughput samples versus *MaxConnectionsPerChild*. The samples have less dispersion around the value 1500.

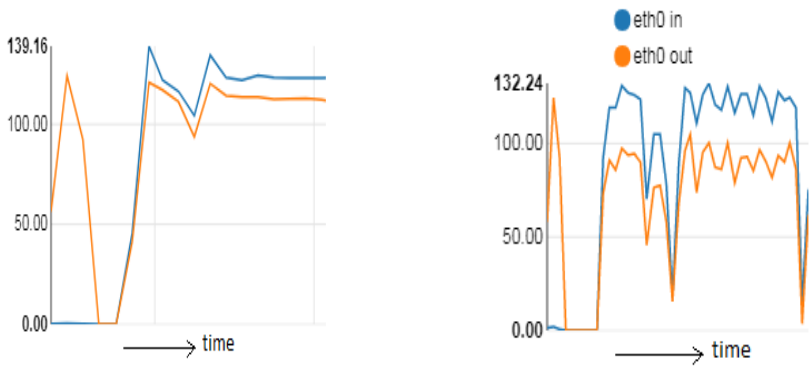



(a) *MaxConnectionsPerChild*=1500 (b) *MaxConnectionsPerChild*=500

Figure B.6: Throughput for tears of steel (MP4)

### Changing *MaxRequestWorkers*:

Last, we deal with the *MaxRequestWorkers* parameter. Setting correctly this parameter is very important so the server will have enough threads in order to serve all the requests on time, while keeping the total number of threads low. If a higher number of threads is allowed than what the available RAM can handle, memory swapping is possible to happen. Memory swapping should be avoided as it can increase significantly the latency of requests.




(a) *MaxConnectionsPerChild*=3000 (b) *MaxConnectionsPerChild*=500

Figure B.7: Throughput for sintel,480p (mp4).

A good indication of the appropriate value for *MaxRequestWorkers* is found by dividing the total available RAM, with the average size of an Apache process, while leaving some memory for the other processes on the server: $MaxRequestWorkers = \frac{(TotalMemory - CriticalServicesMemory)}{ApacheProcessSize}$

The available RAM in the origin is 14308MB. The memory needed by other services is best determined by observing the memory usage, without the webserver running. The size of an Apache Process can be found by using the *ps* command and looking at the residence size of each process. Still the size of each process is not constant and depends on the scripts that are running and the load that is being served. For example when there is no load on the server, the average size of an Apache process is 5 MB. When a user streams video the average size can be 10 MB and under heavy load it can reach 22 MB. Therefore a rough approximation for the total number of threads is $(1430 - 300)/22 = 636$. This high number indicates that the server has enough memory to support a high number of threads. Still we want to use a lower value to avoid excessive context switching and memory consumption. Under heavy load a multithreaded server consumes a lot of memory due to the fact that each connection is associated to a thread resulting to a thread stack per connection. The default value was 150, but this has proven to be too little as there were not enough threads to serve all requests.

To find the appropriate value for *MaxRequestWorkers*, we followed a trial-and-error method, where different values were tested while monitoring the server performance. Four different values were tested : 192, 248, 256 and 400. In all the cases the number of threads per process is kept 8 and *MaxConnectionsPerChild* is set to 1500.

By observing the server status during the experiments, we see that the maximum number of simultaneous requests is never bigger than 256. Even when the number of total threads is allowed to reach 400 and origin handles

a big load the total number of threads is never reached. On the other hand when the number of total threads is decreased to 192, the number of total threads is not enough to serve the clients and incoming request will be queued and eventually rejected. Thus we use choose to use 256 workers.

The measurements obtain from Tensor for the values 400, 248 and 256 for videos tears of steel and sintel,480p are shown in tables B.2 and B.3 respectively. The results show that there are not significant differences in throughput and CPU usage, when 248 and 256 workers are used.

| **MaxRequestWorkers** | **400** | **248** | **256** |
|---|---|---|---|
| incoming traffic (MB/s) | 111,55 | 118,6 | 116,2 |
| outgoing traffic (MB/s) | 81.8 | 87,3 | 84,4 |
| outgoing throughput s.d. | 7,8 | 8,2 | 12,5 |
| CPU system (mean) | 22% | 25% | 25% |
| CPU user (mean) | 18% | 20% | 19% |
| mean latency(ms) | 325,86 | 332,74 | 338,9 |

Table B.2: Results for tears-of-steel when changing total number of threads

| **MaxRequestWorkers** | **400** | **248** | **256** |
|---|---|---|---|
| incoming traffic (MB/s) | 104,19 | 113,6 | 114,7 |
| outgoing traffic (MB/s) | 95,05 | 103,3 | 106 |
| outgoing throughput s.d. | 21,7 | 10,9 | 9,59 |
| CPU system (mean) | 32% | 36% | 33% |
| CPU user (mean) | 27% | 30% | 28% |
| mean latency(ms) | 320,4 | 252,47 | 227,12 |

Table B.3: Results for sintel when changing total number of threads

### Handling KeepAlive Connections

KeepAlive enables persistent connections allowing multiple requests to be sent from the same TCP connection. This parameter is outside of the MPM module but its very important for the server performance and it can tie in closely with the MPM choices.

In high traffic servers, persistence connections can degrade the performance of the server because threads might be blocked for a long time waiting more requests from one connection. While the threads are blocked they occupy RAM that could be used to service other requests. In the worst case scenario, all threads might be blocked in the KeepAlive state, limiting concurrency the resulting to low throughput.

Indeed, in our case when KeepAlive is on and the server is under heavy load, the maximum number of workers is reached immediately and Apache

crashes. As shown by figure B.8, the scoreboard is full and a lot of threads are in the K state( keepalive). Thus we choose to turn KeepAlive off.

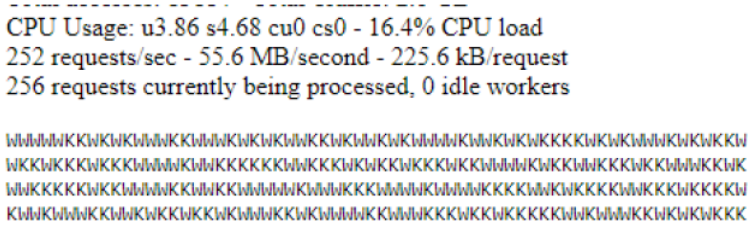



Figure B.8: A lot of workers are stuck in the KeepAlive state.

### B.2.3 Final Configuration

From the above experiments, we concluded that the configuration shown below will give the best results in terms of latency, throughput and CPU usage for the tested videos. Note that this configuration is not unique. There might be different configurations that can give the same performance that is achieved using this one. Given more time and a broader design space exploration, more solutions can be found.

```
<IfModule mpm_worker_module>
ServerLimit 32
StartServers 4
MinSpareThreads 25
MaxSpareThreads 75
ThreadsPerChild 8
MaxRequestWorkers 256
MaxConnectionsPerChild 1500
KeepAlive Off
<IfModule>
```

Figure B.9: Final Configuration

### B.2.4 Conclusions

In this chapter, we acknowledge the problems that can be caused in the streaming performance when the webserver has to handle large concurrent connections We have used the different modules offered by the server for this propose, to tune the server for high concurrent video traffic.

Also, through the experimental work we have notice that different videos, have different optimal setting points of the MPM_worker module. For instance, the three videos had different optimal values for the directive Max-

ConnectionsPerChild. This is an interesting point that could be further studied to understand what aspect of the content create these differences e.g the bitrate (could be useful for content-aware streaming). Still, even thought such different optimal point exists, a generic configuration can be found that performs well with all the content.

A different approach that we could follow instead of tuning the Apache server would be to use a different web-server. Although Apache is good as a media application server, using an event-driven web server, like Nginx, is probably more suitable for implementing a server for highly concurrent usage. Nginx handles multiple requests in a single event loop thus less resources are used. Alternatively, we could even use Apache as the application server and Nginx as the cache server. This would limit the amount of concurrency needed in Apache, since the cache's connections would be processed by a different server, removing this load from the Origin server. Note that in demanding situations where the Origin cannot handle the load, scaling horizontally with more servers and using a load balancer is also an option. However, for this work, we assume that existing infrastructure will have to do the job.

# Appendix C

# (Co)-authored submitted paper

This work was been submitted and is under review for the ACM Multimedia Systems Conference (MMSys 2018) held from June 12 - 15, 2018 in Amsterdam, The Netherlands. Below you can find the submitted paper.

# Improving the Video Streaming Backend with object storage and on-the-fly conversion

Christina Kylili
TU Delft
C.Kylili@student.tudelf.nl

Rufael Mekuria
Unified Streaming B.V
rufael@unified-streaming.com

Arjen Wagenaar
Unified Streaming B.V
arjen@unified-streaming.com

## Abstract

Online advanced media streaming services using HTTP adaptive streaming are increasingly popular. In practice, the multi-protocol, multi-format nature of adaptive streaming creates a lot of engineering effort and costs for the operators, in the storage and preparation of the different formats. In this work, we acknowledge these issues and we study a streaming setup that can address these. Such streaming setup consist of a backend cloud storage and a processing node that generates streaming presentations for different devices on-the-fly. We analyze the streaming setup and its performance by testing in cloud deployments. Through this evaluation we identify the performance limitations of the setup imposed by the transfer of data between the object storage and the processing node. We propose a new backend storage caching scheme based on rarely used existing feature of dref in the specification of the MPEG-4 standard. Experimental results show that the proposed scheme can improve the streaming performance such as reduced latency and increased outgoing traffic volume towards the clients.

***CCS Concepts*** • **Networks** → *In-network processing*;

***Keywords*** HTTP adaptive video streaming, on-the-fly conversion, media processing distributed in the network, storage node

**ACM Reference Format:**
Christina Kylili, Rufael Mekuria, and Arjen Wagenaar. 2018. Improving the Video Streaming Backend with object storage and on-the-fly conversion. In *Proceedings of ACM Multimedia Systems Conference (MMSys 2018).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Video streaming has evolved rapidly in the last years, with the emergence of an international standard [3] and some large scale deployments [1]. Video providers rely on HTTP adaptive streaming (HAS) to deliver their content to different client devices. The use of adaptive streaming itself requires the content to be encoded at multiple bitrates. In the context of video on demand (VoD), this implies storing the content in multiple bitrate versions. Considering the existence of the different HAS protocols such as Apple's HTTP Live Streaming (HLS) [2] and Dynamic Adaptive Streaming over HTTP (DASH) [3], this means that the storage requirements for a VoD provider are very high.

One of the most important trends in video streaming is using object-based cloud storage for storing the content assets. Cloud storage offers seamingly unlimited storage space. In addition, this storage is persistent and protected form device failures and it provides other features such as HTTP interface. Moreover, in video streaming, cloud infrastructure offers compute power as well. This compute power can be used for media processing operations in the cloud such as transcoding, dynamic packaging or any other processing or conversion operation.

By combining object based cloud storage with a compute node with media processing, such as on-the fly conversion, a powerful video streaming setup is achieved. In this setup, VoD providers can store a single source, but stream using different protocols or encryption schemes by using on the fly-conversion in the compute node. This makes it easier for VoD providers to support newer and legacy formats. In addition this solution reduces the storage needs and is fully cloud based. Furthermore, more advanced versions of such a setup can support other desirable media processing operations for VoD providers, such as ad-insertion, dynamic content encryption etc.

A key question regarding the above setup, is the actual performance of it in practical deployments. For example, the communication between the storage and the compute node may raise some performance limitations. This could be because of limited features and functionality of the cloud storage system (e.g. not supporting byte range request, limit on request rate, limited access, limited API, policies). Alternatively, the efficiency of the media conversion could be low and the connection to the object storage could be limited in bandwidth or introduce some latency. Latency and bandwidth limitation could for example be a problem when the compute node is deployed remote from the cloud storage (e.g. in an edge computing scenario). These aspects could reduce



*MMSys 2018, June 2018, Amsterdam, The Netherlands*

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the video streaming performance and the final quality at the end users (QoE).

In this work we want to study these issues and thus we focus in a setup with a realistic on-the-fly conversion and cloud storage. This work performs several experiments in a realistic cloud deployment to study the behavior using different streaming protocols. Based on the behavior of the on-the-fly conversion server, we propose a scheme for (remote) storage access that can reduce the streaming latency and improve client throughput. We test the proposed setup in different (edge) cloud deployments.

The rest of this paper is organized as follows. Section 2 gives an overview of the related technologies and related work. Section 3 presents the streaming setup and the performance of it, through an experimental work with different media formats and protocols. The optimization scheme is presented in section 4. The performance evaluation of the scheme is given in section 5. Finally, section 6 concludes this paper and outlines future research directions.

## 2 Background and Related Work

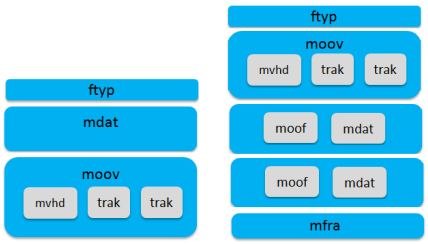
### 2.1 Media file formats

In VoD there are two main formats used for storage and presentation of multimedia content: MPEG-2 Transport Streams (MPEG-2 TS)[5] and ISO Base Media File Format (ISOBMFF) [6].

MPEG-2 TS were initially designed for broadcasting video through satellite networks and physical media delivery solutions. However, Apple adopted it for it's adaptive streaming protocol making it an important format. In MPEG-2 TS, audio, video and subtitle streams are multiplexed together. Mp4 and fragmented mp4 (fmp4), are both part of the MPEG-4, Part 12 standard that covers the ISOBMFF.

Figure 1 demonstrates the structure of an mp4 and f-mp4 file. As shown, each container consist of different 'boxes', each with a different functionality. The traditional mp4 file has a Movie Box ('moov') that contains all the metadata of the media presentation file and sample tables that are important for timing and indexing the media samples ('stbl'). There is also a Media Data Box ('mdat') that contains the corresponding samples. In the fragmented container, media samples are interleaved by using Movie Fragment boxes ('moof') which contain the sample table for the specific fragment(mdat box). Further fmp4, has an additional box named Movie Fragment Random Access Box ('mfra') which is used for indexing movie fragments.

Fragmented mp4 was chosen in the recent specification of common media application format (CMAF)[7]. This standard is aiming to create a subset of container formats and codec settings to be supported by the different HAS protocols.




(a) MP4 Container (b) f-MP4 container

**Figure 1.** Structure of media containers.

### 2.2 On-the-fly conversion

On-the-fly format conversion is important in video streaming systems, to deal with the existence of different formats, encryption schemes and protocols and to reduce storage requirements. Examples of on-the-fly conversion are dynamic packaging and manifest generation for multi-protocol. With this there is no need to prepare and store the video segments and manifests multiple times, hence saving costs and storage capacity. There are different on-the-fly format conversion solutions in the industry offered by Amazon (elemental delta), Azure, Wowza [18], Unified Origin [19] etc. In addition to dynamic packaging they may offer other media processing functions like on-the-fly encryption, transcoding or subtitle conversion.

On the-fly format conversion fits well with the latest emerging MPEG streaming standard: Network-Based Media Processing (NBMP) [20]. This is an emerging standard that allows media delivery with media processing services/functions embedded in the network, such as on-the-fly format conversion. In such framework the media source is connected to media processing functions that takes care of the delivery to a client using different published formats (DASH, HLS, CMAF). NBMP has a good potential to handle many of the problems with video streaming by deploying on-the-fly conversion functions. Further, the definition of standardized interfaces in NBMP will help interoperable deployments in the cloud or in the network. The research in this work is aligned with state of art streaming systems, conforming the NBMP reference architecture.

### 2.3 Related Work

The emergence of DASH and the need for adaptive bitrate selection methods, has triggered a lot of research on the rate adaption algorithms. Several studies have exposed the performance problems of adaptive streaming [8][9]. There is quite a lot of work that focus on assisting the bitrate selection and overcoming these problems [11][10][12][13]. Our work relates to the server side architecture and aims to improve the communication interface between an object storage and a dynamic packaging server.

Other work investigates the server-side deployment of adaptive streaming which is more closely related to this work. Specifically there is a some work that focuses on video transcoding and different strategies of trading transcoding cost and storage (for different bitrate encoded versions of a content) [14] [15]. Although transcoding is important and relates to the storage problem, in this research, we focus on more lightweight conversions like container conversion, manifest generation etc. As these conversions are lightweight, instead of the trade-off between compute and storage, the main focus is on the communication between the media source (object storage) and the on-the-fly converter.

A more related work that deals with dynamic packaging is presented in [17] and [16]. [17] evaluates the performance of two different implementations of on-the-fly packaging that is integrated in the client side. The results show that when the dynamic packaging happens in the client side, the performance (duration of conversion) can be slow. Mekuria *et al.* [16] present a multi-protocol video delivery architecture, where protocol specific media segments are generated on the fly. The conversion node is moved into the edge and caches both protocol-specific segments and raw media data retrieved from the centralized storage. This way, the authors aim to reduce redundant traffic and caching that happens in the CDN. The work that is presented in that paper has a similar setup with the setup that we focus on, with a dynamic packaging node and a storage node. However that paper focuses on creating a smart edge cache and improving the efficiency of caching, while this work aims at reducing the latency and overhead related to object storage access.

### 2.4 Contribution

For practical video streaming that targets multiple protocols and large asset repositories, the combination of on-the-fly conversion, such as dynamic packaging, with object based cloud storage is a powerful one. However, prior work on video streaming does not consider this architecture and the possible performance bottleneck between the object storage and the dynamic packaging node. This work studies this problem and presents some improvement for such a setup, based on media streaming aware backend storage caching. In summary, this paper presents the following contributions:

1. Performance evaluation of a state-of-the-art streaming framework with on-the-fly conversion and object storage on a realistic workload, using different file formats. The communication of the two is analyzed to reveal part of the file formats, critical for the specific media processing node.
2. Optimization scheme for back-end using the current HTTP infrastructure and rarely used existing features of the MPEG-4 part 12 specification in ISOBMFF, showing reduced latency and improved throughput.

## 3 Video Streaming with on-the-fly conversion and backend storage

The setup that we focus in this work is shown in figure 2. This setup consist of an object based cloud storage and a media processing node, responsible for dynamic packaging. In this setup, a single version of each encoded content version is stored in the storage. The on-the-fly conversion server communicates with the backend node to retrieve the information needed in order to generate the HAS specific manifest and segments formats, on-the-fly, targeting all user devices.

Such a setup enables smaller and more maintainable repositories, less content preparation work and better future proofness when new formats are released and legacy support is still needed. Further, the separation of the storage and the conversion server, protects the content asset repository from possible corruptions when the streaming server crashes due to request load. This setup corresponds with the architecture of NBMP [20] defined in MPEG as the new emerging streaming standard for immersive media, where media source is separated by media processing operations distributed in the network/cloud.

A key question regarding the above setup, is the actual performance of it in practical deployments. For example, the communication between the storage and the compute node may raise some performance limitations. This could be because of limited features and functionality of the cloud storage system (e.g. not supporting byte range request, limit on request rate, limited access, limited API, policies). Also the connection to the object storage could be limited in bandwidth or introduce some latency when the compute node is deployed remote from the cloud storage. Another limitation could be a low efficiency of the media processing node (more data is needed from the storage than that is finally produced for the user). As preliminary results showed that typically incoming and outgoing link bandwidth is capped in processing nodes in the cloud, this reduces the outgoing data rate.

In this section we perform a performance evaluation of the setup by experimenting with different file formats in the storage in order to specify the bottlenecks of such a setup. Next we describe the experimental testbed that we used and the results.

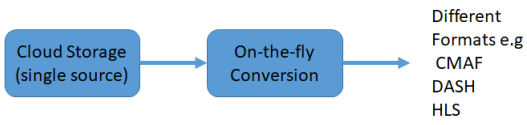



**Figure 2.** Setup with cloud storage and on-the-fly format conversion

### 3.1 Experimental Testbed

We describe the components and implementation of the experimental testbed in the following sections.

#### 3.1.1 Unified Origin

In this work we experiment with the Unified Origin server [19] which is the on-the-fly format conversion software. Unified Origin is a software plug-in for popular web-servers and it supports dynamic packaging, manifest generation, on-the-fly encryption, different DRM systems and more. In this setup Origin needs to communicate with the backend storage and fetch the information that is needed to satisfy each request. The Origin is used as an on-the-fly conversion node for generating manifest and protocol segments. It is deployed on a compute node in the cloud.

#### 3.1.2 Cloud Storage and Content Preparation

The object cloud storage that is used in this work, is Amazon Simple Storage Solution (S3) [22]. To populate the content database we choose to use as storage format mp4 and f-mp4. Fmp4 was chosen since it is already enabled for adaptive streaming and is used in DASH and other protocols (CMAF and recently HLS). On the other hand, mp4 has been as a storage format a lot in the past and is still being used, since converting all the content to fmp4 is time consuming and costly.

Three videos are stored in S3: sintel [24], tears of steel [23] and elephants dream [25]. To enable adaptive streaming, each video is encoded into multiple bitrates and multiple resolutions as shown in table 1. Tears of steel is available only in 400 kpbs, 800 kbps, 1200 kpbs, 1900 kpbs and 3000 kpbs, because it was already available in an encoded version. The chosen resolutions reflect the capabilities of typical user devices such as smart-phones, tablets and HD TV. Then each encoded video was packaged to fragmented mp4 and finally a server manifest file is created that can contains the track information.

| Resolutions | Video Bitrate(kbps) |
|---|---|
| 480p: 640x480 | 200, 400, 750, 850, 1000 |
| 720p: 1280x 720 | 200, 400, 750, 900, 1000, 1200, 1500 ,2000 |
| 1080p: 1920 x1080 | 200, 600, 900, 1000, 2000, 2400, 2900, 3300 |

**Table 1.** Resolutions and Video Bitrates for Sintel and Elephants dream.

#### 3.1.3 Load Generator Tools

The role of the load generator tool is to simulate the video streaming clients of the figure 2, which request protocol specific manifests and video segments of different bitrates. In this work we used two different benchmark tools: Tensor [28] and Apache benchmark(AB) [26].

Tensor is a tool for testing adaptive bitrate streaming by generating a realistic video streaming workload based on collected traces from a real video player. Tensor is able to simulate a large number of concurrent connections in order to measure the server behavior under peak load conditions. Tensor is based on two open source programs, WRK[31] and Performance Co-Pilot(PCP)[32]. WRK is the actual load generator that Tensor uses. With WRK, the number of concurrent connections is increased every 5 seconds while the number of requests in each connection is maximized in order to identify the maximum throughout that the server is capable of. Then WRK gives the throughput and latency metrics reported in the client side. Furthermore Tensor collects hardware and software statistics from the origin server (throughput, CPU usage, memory utilization) using PCP. These metrics are displayed in a web-interface with graphs. Starting an experiment with Tensor is simply done by supplying Tensor with the URL of a client manifest file of a specific video. The requests are based on HTTP request sequences obtained from the respective DASH or HLS manifests.

AB can generate various HTTP workloads to measure the server performance, by specifying the total number of requests and the number of multiple requests to perform at a time. Unlike Tensor, only one URL request can be selected for the whole workload. For this reason AB is used to measure the performance of the streaming setup when delivering either a manifest file or just a specific segment. AB tool reports application layer statistics on HTTP behaviours (e.g reply rate, response time), providing just a user-level view of the performance.

We choose to use two benchmarks, since each has a different testing range. Tensor is used for testing large scale adaptive streaming and the workload is more realistic. On the other hand, testing with AB is better for understanding what is the performance in lowest-level operations of video streaming by just doing individual requests either for a manifest or a video segment.

#### 3.1.4 Cloud deployment

Figure 3 shows the experimental tesbed that is used for evaluation. The testbed is deployed in a cloud environment, in Frankfurt using Amazon Web Services [21]. The Origin server and the load generator tool, run on virtual servers in Amazon's Elastic Compute Cloud (EC2). The Origin runs on an Apache web-server installed on an Amazon m3.xlarge instance [27] and the load generator runs on a compute-optimize instance (c4 instance) in order to be able to handle a large amount of connections.

### 3.2 Key Performance Indicators

The performance of the setup is measured by using three key performance indicators (KPI). The first KPI is the average latency for a handled request and the second is the achieved throughput in the server(or in the client in case of using AB).


**Figure 3.** Experimental testbed deployed in Amazon cloud.

In AB the throughput is measured by the number of requests that are handled per second in the client. Tensor measures the amount of Megabytes received or send per second, in the server. We use the throughput saturation point given by Tensor, where throughput cannot increase more, even if the connections are increasing, due to the physical limits of the network interface. The last KPI's is a specific metric for this setup, that we define later. We name this metric the 'conversion efficiency' for each file format and is the ratio of the outgoing traffic and the incoming traffic in the server.

These metrics are suitable for testing a streaming setup in large scale with a big number of requests. Also they can pinpoint network impairments which makes them suitable when focusing on the communication interface of the Storage and the Origin. Even thought these metrics are QoS-based and not QoE-based ([29],) they can still give an indication on the QoE of the client. For example, lower latency indicates faster video playback and lower start-up time. Higher data-rate will result in higher quality representations at the end users (especially when the number of total users is low).

#### 3.2.1 Methodology and Tested Workload

To identify what is the overhead of using a backend storage to the setup, a comparable study is performed: using local storage and using cloud object storage. When local storage is used, the original video files are stored in the Origin, allowing dynamic packaging. Furthermore, some more experiments are done with the backend storage in order to give some insights on how different file formats influence the streaming performance.

Each test that is done (local storage, backend storage, different videos, DASH and HLS) with Tensor, is repeated three times for cross-validation. The throughput and latency metrics that we get for each repeated test have a low standard deviation (4.5 for throughput and 6.9 for latency) so they can be considered reliable. Any statistical information that is mention in the following sections is derived by using the averages of each of the three experiments.

### 3.3 Performance Evaluation and Discussion

#### 3.3.1 Local Storage vs Cloud Storage

Figure 4[1]. shows the incoming and outgoing traffic measured by Tensor, for both local and backend storage (video stored as mp4 and fmp4). In all cases, dynamic packaging is used and the video is requested in DASH. Incoming traffic reflects on the backend data that the Origin fetches from S3. The outgoing traffic is the traffic between the client and the Origin. The figure reveals important information regarding the performance of the setup that separates the media source from the Origin:

- While with local storage the outgoing throughput that is achieved for each video is similar for both mp4 and fmp4, for the cloud storage this is not the case. The outgoing throughput is different when the same video is stored as mp4 and as fmp4.
- When backend cloud storage is used the outgoing traffic is decreasing compared to using local storage, even though the traffic between the client and the Origin hasn't changed. This is because the Origin needs to fetch some more data from S3 than what it actually produces for the dynamic packaging operation. Combining this with the previous point, we conclude that the conversion performance of Origin changes when backend storage is added and this is also influenced by the file format.
- Latency is increased when using the backend storage, because to satisfy every segment request, the processing node needs to make additional requests to the remote storage. The figures of latency are not shown here for space constraints.

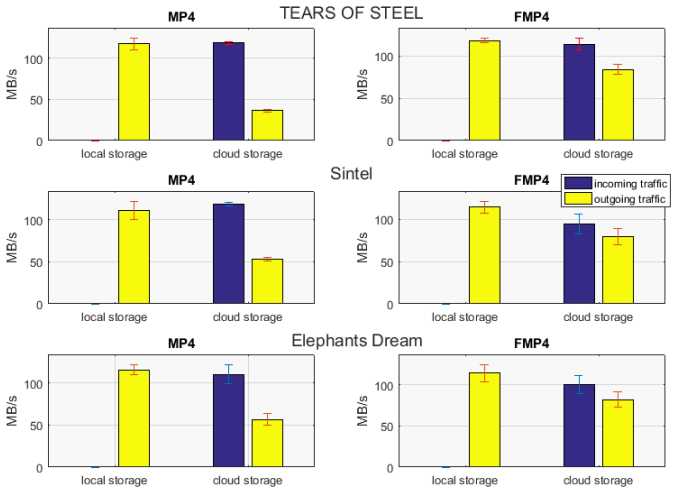



**Figure 4.** Local Storage vs Backend Storage: Incoming and outgoing traffic.

#### 3.3.2 Conversion Coefficient and Differences between file formats

Figure 4 shows that when we use backend storage, the performance that we get in terms of throughput depends on the file format used in the storage. Specifically, when mp4 is

[1] Average results for all resolutions

| file format | mp4 | | | fmp4 | | |
|---|---|---|---|---|---|---|
| source video | ed | sintel | tears | ed | sintel | tears |
| in throughput(MB/s) | 110,57 | 118,17 | 118,20 | 100,12 | 94,45 | 113,67 |
| out throughput(MB/s) | 56,80 | 53,08 | 36,30 | 82,22 | 79,53 | 83,91 |
| conversion efficiency(out/in) | 0,51 | 0,45 | 0,31 | 0,82 | 0,84 | 0,74 |

**Table 2.** Backend Storage results (video requested in DASH).

used, the outgoing throughput is lower than in case of f-mp4. This is more clear by observing the saturated throughput measurements obtained by Tensor in table 2.

The table shows that storing the content as fmp4 results to higher outgoing throughput than storing as mp4. This might be due to the fact that mp4 contains one big 'moov' box that contains not only the metadata but also the information on accessing the media data. As said before, the Origin needs to fetch more data from S3, than what it actually produces. In fact the amount of data that it needs differs per file format. To measure this amount of data per file format, we define a metric that we call 'conversion efficiency' for each file format. This metric is the ratio of the outgoing traffic and incoming traffic and is an indication on the amount of more data that the media processing needs to fetch from S3, related to what is actually produced for the client. The table shows that transporting data from an f-mp4 video source has a higher conversion efficiency than the case of mp4, where at least double data are needed to be fetched from S3, than what will actually produced for the client.

As Tensor test the setup in large scale, we want to verify that using fmp4 in the storage gives better performance than storing mp4, also in terms of latency. Thus AB is used to measure the latency when requesting just a single segment. This way we can make a more clear comparison between the two file formats. Using AB we generate a large number of DASH and HLS requests (30000 requests) for a low bitrate segment (200kbps) and a high bitrate segment (1Mbps) of sintel, 480p. The results are shown in figure 5.

In summary, the results show the efficiency of using fmp4 in the storage rather than mp4. The percentage difference of f-mp4 and mp4 in DASH case is 78% for the low bitrate segment and 50% for the high bitrate segment. In HLS case this difference is reduced by 20 (48% low bitrate, 30% high bitrate), because in HLS more data are needed due to the multiplexed form of segment (contains both video and audio). For a similar reason, the difference between mp4 and f-mp4 is less for the high bitrate segment, compared to the difference that they have when a lower bitrate segment is requested.

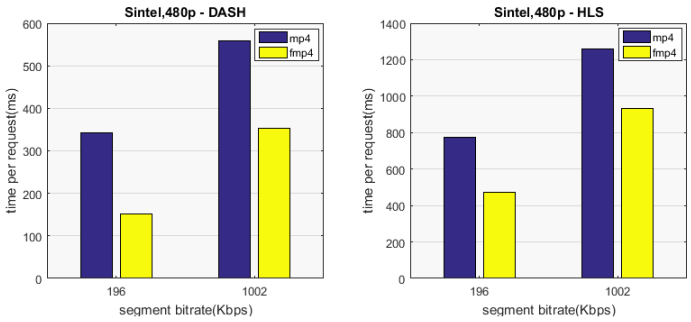



**Figure 5.** Difference between mp4 and fmp4 for time per request when segment is requested in DASH and HLS format.

## 4 Communication Analysis & Proposed Improvement Scheme

### 4.1 Analysis of communication between Origin and Storage

The previous results revealed that the setup of figure 3 has two performance limitations: low outgoing throughput and increased latency. These are mainly caused by the communication between the Origin and the Storage, thus we present here an analysis of this communication. To do this we use packet inspecting tools like tcp dump and Wireshark, to identify the information exchanged when a client request either a manifest file or a segment (HLS or DASH).

#### 4.1.1 Generating a client manifest file

A manifest file (DASH or HLS) contains metadata information of the content and the available bitrate representations of the video. On the server side, this information is stored in the server manifest file (ism). Thus this is fetched from the storage. Further, a manifest file contains information for constructing the segments URL. In order to construct this information, the Origin will need to access the timing and indexing information of the samples. This is done by a number of byte range requests sent from the Origin to the storage. Since mp4 and fmp4 have a different structure, the number of byte range requests are different for each format.

Table 3 shows the information that is requested from the Storage, upon a manifest request from the client. Each cell of the table indicates one byte range request. When the content is stored as mp4, 3 byte ranges requests are done in order to fetch the 'ftyp' box and the 'moov' box for each bitrate

representation ('mvhd' is the header of moov box). The 'ftyp' box specifies the compatible brands of the files. The moov box is needed because it contains metadata and the sample tables boxes that give indexing information about the media data ('stbl' box).

In the case of storing the source video as fmp4, the 'mfra' box is also requested as it contains the information of locating the moof box for each sample. The mfra box is fetched with 2 byte range requests: first for determining the size of the box and then retrieving the mfra box. Also for DASH, the last moof box is necessary to know where the presentation finishes. When an HLS manifest is generated, there are just four byte range requests. The last moof box is not needed.

| **mp4** | | **fmp4** | |
|---|---|---|---|
| **manifest** | **segment** | **manifest** | **segment** |
| ftyp | ftyp | ftyp | ftyp |
| mvhd | mvhd | moov | moov |
| moov | moov | mfra size | mfra size |
| - | mdat | mfra box | mfra |
| - | - | moof header | moof mdat |
| - | - | moof | |

**Table 3.** Byte range requests to the storage for generating a DASH manifest file and a DASH segment.

#### 4.1.2 Generating a video segment

When the client requests a video segment of a specific bitrate, the Origin needs again the server manifest file to locate the stream that contains the requested bitrate. Then it has to fetch metadata and media samples from that bitrate representation (audio/video). Fetching this information is done with a number of byte range requests.

For mp4, there are 4 byte range requests as shown in table 3. As mentioned before, the moov box is needed for locating the samples that are requested. In the fmp4 case the mfra box needs to be fetched as well since with this box the Origin can locate the related segment. The segment, in this case is a pair of a moof box and mdat box which are both retrieved with one byte range request. In an HLS segment, audio and video are multiplexed, thus the number of byte range requests will be doubled.

Note that Origin uses byte range request in order to fetch the necessary data from S3, instead of fetching the whole video file which would be huge. Using multi-byte range requests could decrease the amount of requests send to S3, but multi-byte range requests are still not supported by S3 and some other cloud storage.

#### 4.1.3 Discussion

Through the communication analysis we have understood that using fmp4 in the storage is more efficient than mp4 because the size of the moov box in mp4 files is bigger than in fmp4 files. Further we have seen that there are parts of the media data, that are critical for the on-the-fly conversion to DASH and HLS. In our case this part is the moov box and the sample tables. This information is very important for devising a scheme that improves the communication of the Origin and the Storage. This scheme is presented in the next section.

### 4.2 Proposed scheme using dref MPEG-4 files

In this section we propose a simple optimization scheme that can reduce the number of requests that are send from the Origin to the Storage. This is done by placing a smart cache between the Origin and the Storage. This cache does not store media data, as this would eventually expand the storage volume of the cache. Instead, it caches only the metadata of the content and the server manifest file (.ism).

Caching just the metadata, is possible by leveraging an existing technology of the ISOBMFF standard in a novel way. This technology refers to the dref MPEG-4 box. The dref MPEG-4 box is part of the specification of ISOBMFF[6] but is not commonly used in practice. The dref is basically a data reference 'box' defined in ISOBMFF as follows: *The data reference object contains a table of data references (normally URLs) that declare the location(s) of the media data used within the presentation.*[6]. In other words, the dref box which is part of the moov box, specifies the location of the media data. Containers, that hold only the metadata of a content inside a moov box and no physical media data, will be called dref files. We propose caching this lightweight dref file, together with the server manifest file (ism). This will help the Origin to obtain the necessary metadata without the need to contact the Storage. Furthermore, when dref is used for indexing our media content, it doesn't matter what is the original format of the media data (mp4 or f-mp4). The performance should be the same. Thus this container-independent solution will diminish the variation in performance between mp4 and fmp4, that we saw in section 3.3.2.

Figure 6 shows the sequence diagram and message exchange between the Origin and the Storage, when a client request a video segment. The cache stores only the dref and the ism file, while the media data are still served from the back-end storage.

#### 4.2.1 Implementation

The implementation of the proposed scheme consist of creating the dref MPEG-4 files for the video contents and setting up a cache in the Origin server that stores the dref and the ism files.

**Creation of the dref MPEG-4 file:** The dref file should be created for every available bitrate representation of a content. Unified Packager[4] is used for this purpose. Further the server manifest file for each content is created again,

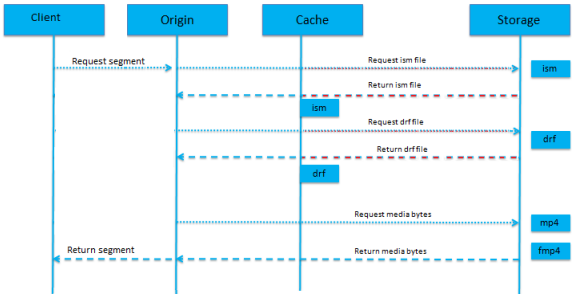



**Figure 6.** Sequence diagram of the optimization scheme. The communication indicated with red, will happen just once.

to reference the dref files generated for each bitrate representation, instead of the whole file(mp4 or fmp4) for each representation.

**Implementation of the cache:** To implement the cache, we use Apache cache [30] for simplicity reasons, since the Origin runs in an Apache server. To integrate the cache in the setup with the Origin and the Storage, a new virtual host is added to the configuration file of Apache. The Origin sits in the default virtual host (port 80) and uses the special custom directive *IsmProxyPass* to generate the byte range requests. When cache is used, these requests are redirected to the new virtual host that the cache is sitting. If the request are for a dref file or the server manifest file (.ism) then the range header is removed, before the request is sent to the storage using the Apache directive *ProxyPass*. This is done because Apache cache does not support caching of partial content, so the dref will be fetched with just one request. The response is then forwarded to Origin.

# 5 Performance evaluation

## 5.1 Experimental Conditions

To evaluate the proposed setup, we used the testbed of figure 3, but in this case the cache is added to the Origin server node.

The proposed optimization is tested with four different configurations. Each configuration differs on the location of each component of the setup . These configurations are the following:

1. Setup 1: Client & Origin & Storage in the same cloud environment.
2. Setup 2: Origin & Storage in one cloud. Client in a different cloud.
3. Setup 3: Origin and Client in one cloud. Storage in a different cloud.
4. Setup 4: Large scale testing.

In Setup 1, all the components are in the same cloud environment, just like the experiments described in section 3. In Setup 2, the client is moved in Ireland, far from the Origin, to simulate a more realistic scenario where client is in the edge. In the third configuration the Origin is moved to the same cloud as the client, to simulate the scenario of moving the Origin to the edge. Such a scenario can minimize the traffic in the CDN (generation of segments is done near the client) and increase the efficiency of CDN caching. These three setups are tested using AB with single requests (for video segment and manifest) because the results are easier to interpret and to understand the effect of caching. Setup 4 is the large scale testing of the streaming procedure, using Tensor. For this test, the three components are in the same cloud and the results can be compared against the results obtained in section 3.3.

Note that the proposed scheme can have 2 caching flavours. The flavour that only caches the server manifest file (a generic manifest containing simple information of the media presentation in SMIL format) and the flavour that caches both the manifest file and the dref. Caching the dref is a smart caching that we propose and we claim that this can achieve greater performance. To show the effect of each flavour, both are shown in the results

## 5.2 Tested Workload

The first three setups are tested with AB, by generating a significant amount of requests in a short period of time. Specifically 1000 requests are generated in total. The requests can be of two types: request for a manifest file and for a video segment (DASH and HLS). Each test is performed for two cases: storing the video content as mp4 an as fmp4. The final setup, is tested with Tensor, by opening 90 concurrent connections and maximizing the number of requests. The tests are done by requesting only the DASH manifest file of the three videos in all available resolutions.

## 5.3 Key Performance Indicators

The KPI's used in this evaluation are the ones described 3.2: average latency, throughput on the server and conversion efficiency (for the large scale testing).

## 5.4 Results and Discussion

First the results for segment and manifest request are given, for the first three setups. Large scale testing is presented last.

### 5.4.1 Segment Request (AB)

Figures 7 and 8, show the first two KPI's for the three setups, when testing with 2 different video segments of sintel, 480p and elephant's dream, 720p. The blue bars of the graphs, indicate the results when only the ism is cached, and the rest show the results when the dref's are also cached. Further the numbers in the bars, indicate the average percentage increase(or decrease) that is achieved when the dref is used, on the results that are obtained from caching just the ism file. As shown, caching the dref and ism file performs best, for all setups. The highest gain is achieved when testing Setup 3 by requesting segments of sintel, 480p (figure 7.a).

There, the amount of requests that are handled is tripled in all cases (different segments of sintel) and the latency is decreased by 70%. However, when requesting a big segment of elephants dream, encoded into 2Mbps there is little on none improvement when the dref is cached. This is because a segment of elephant's dream, 720 resolution, encoded into 2Mbps has a size 11 times bigger than the same segment encoded into 200Kbps. Even when the dref is cached, still the client has to handle a big size of responses, thus the average requests/second will be low.

Another interesting observation is that when dref is cached the file format of the source video does not matter and mp4 and fmp4 have almost similar performance, as anticipated by this approach. This allows one to avoid repackaging media collections stored using non fragmented mp4 file format.

Among the three setups, setup 1 performs best since everything is closer to the client. When the client is moved far from the Origin (setup 2), more time is needed to fetch segments, thus the latency of request increases and the amount of requests per second decreases. When the Origin moves in the same cloud with the client (setup 3), and the cache only stores the manifest file, even more time is needed per request. This is because the Origin will be further away from the Storage, and there will be additional delay in fetching the media and meta data from the Storage. However, when the cache is enabled to store also the metadata, the requests to the Storage will be decreased and the fact that the Storage is further away will have less impact on the performance compared to when caching just the ism. Thus we see, that for Setup 3, when dref is cached (yellow and green bars), the time per request is less than Setup 2 and the amount of requests/second is higher (in most of the cases). Therefore, comparing Setup 2 and 3, we can conclude that Setup 3 is more beneficial when using dref, because of the results shown here and the other benefits that it offers (less traffic and caching in CDN).

#### 5.4.2 Manifest Request (AB)

Testing the performance when a client request a manifest file, is important as these results can indicate what is the impact of the proposed scheme in the start-up delay which is a key QoE metric. This is because before video playback, the video player needs the client manifest file and the initialization segment, which contains metadata. Origin generate those by using the ism file and dref, thus when these are stored in the cache, the client requests will be served faster, reducing thus the start-up latency. For space considerations we only show the latency results for the Setup 3, in figure 9 when requesting the DASH and HLS manifest for sintel and elephants dream.

The results show that the time per request is decreased up to 97% (for both DASH and HLS) when the dref is cached. This is because to generate the manifest file the information needed by Origin are the ism file and the dref files. Thus, when these are cached, there will be no requests to the remote Storage. This is a promising result that indicates the reduction of the start-up delay but further experiments should be performed to measure this accurately.

#### 5.4.3 Large Scale Testing (Tensor)

Large scale testing is done using Tensor. Figure 10 and table 4 shows the average incoming and outgoing traffic measured in Origin, when dref and ism are cached [2].

Backend traffic is still higher than the front end traffic, since media data still need to be fetched from S3. However the outgoing traffic has increased in all cases. Figure 11 shows the gain for different resolutions of sintel and elephant's dream, comparing with the baseline results. For the mp4 case, the increase is higher (throughput on outgoing interface is doubled in sintel, 480p) since the mp4 case was really bad when caching was disabled. For tears of steel, the proposed scheme achieves an increase of 143% and 8% for the case of storing as mp4 and fmp4, respectively. In addition, the 'conversion efficiency' for the mp4 case has improved a lot and there is no performance variation between storing the content as mp4 or as fmp4. Figure 12 shows the average latency obtained by Tensor when the cache is disabled, when the ism is cached and when both the ism and the dref are cached. Two different resolutions are shown for both the case of storing the content as mp4 and fmp4. From this figure we see that even with just caching the manifest file, there is a decrease in the average latency. As we see, the average latency depends on the content and resolution.

## 6 Conclusions

In this work we deal with a video streaming setup that consist of a cloud object-based storage and an on-the-fly format conversion server. This setup offers flexibility and a number of advantages such as reducing the storage requirements. A performance evaluation of the setup reveals that the setup suffer from two bottlenecks that are imposed by the communication of the storage and the server: throughput is decreased and the latency is increased. Throughout an analysis of the communication we were able to understand why this happens and how the different file formats used in the storage, influence these bottlenecks. To overcome this we proposed the use of a smart cache that stores the server manifest file and metadata of the content, and that would reduce the number of requests send to the storage. The proposed scheme is evaluated under different cloud configurations. The results have shown that this approach has improved network performance with increasing the throughput and decreasing the latency in most of the cases. Furthermore this solution is container-independent making it ideal for content owners. As future work we would like to perform an experimental

[2] Average results for all resolutions

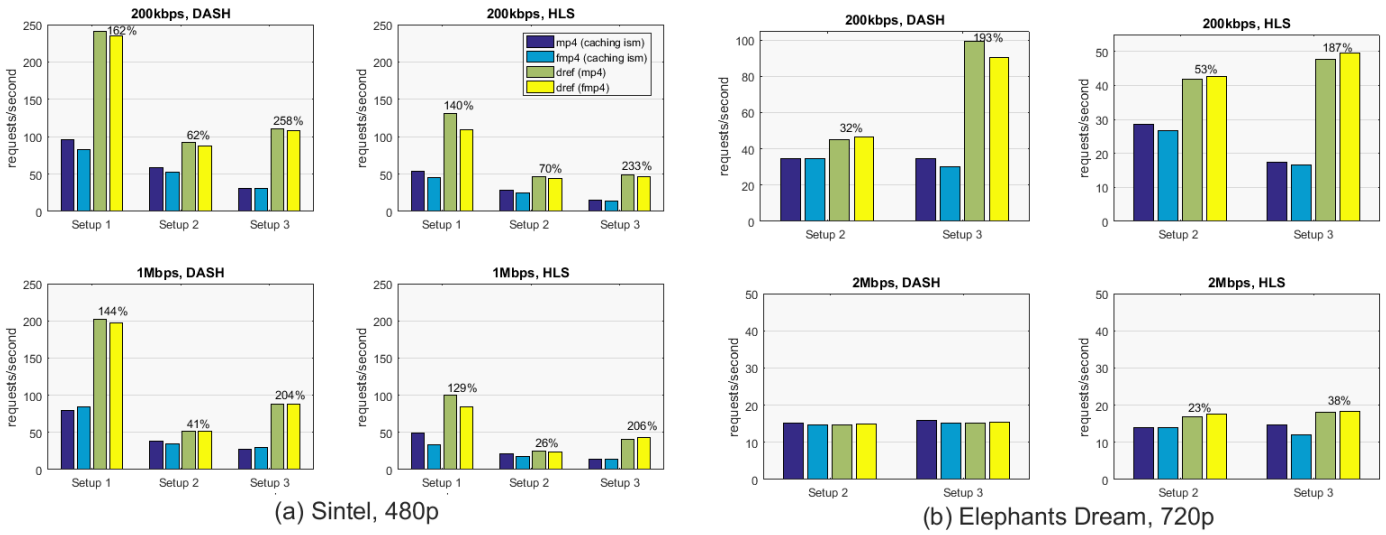



**Figure 7.** Requests/second for Sintel, 480p and Elephants dream, 720p

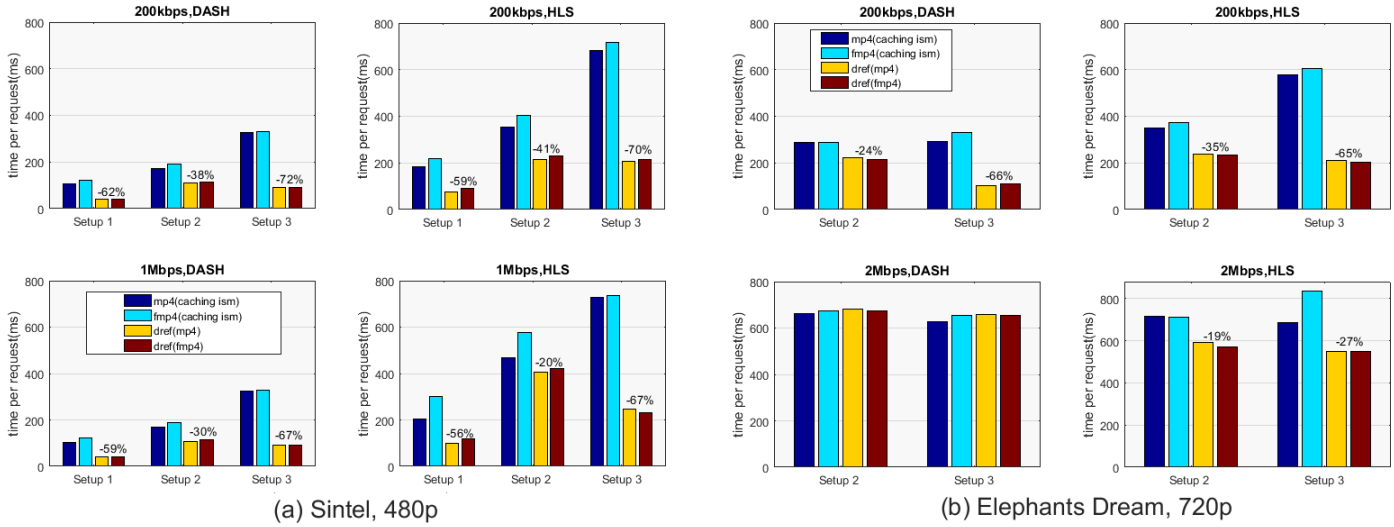



**Figure 8.** Time Per Request for Sintel, 480p and Elephants dream, 720p

| **file format** | **mp4** | | | **fmp4** | | |
|---|---|---|---|---|---|---|
| **source video** | ed | sintel | tears | ed | sintel | tears |
| in throughput(MB/s) | 113,14 | 114,72 | 116,36 | 112,63 | 113,90 | 109,56 |
| out throughput(MB/s) | 94,89 | 99,07 | 86,65 | 95,10 | 98,21 | 82,66 |
| conversion efficiency(in/out) | 0,84 | 0,86 | 0,74 | 0,84 | 0,86 | 0.75 |

**Table 4.** Backend Storage results with cache

analysis using a set of QoE metrics that will be used to calculate the user satisfaction in order to see what is the impact of the proposed setup in the user experience.

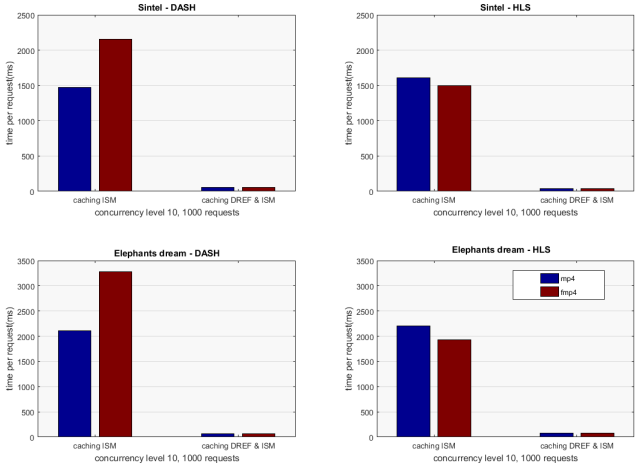



**Figure 9.** Setup 3: Time per request for 1000 requests when client request the manifest file of sintel,480p and elephants dream, 720p.

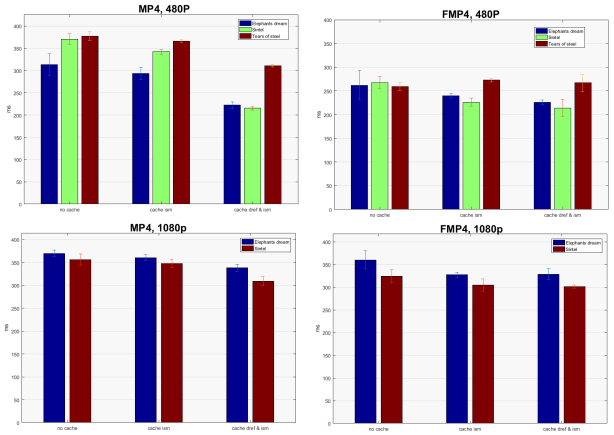



**Figure 12.** Large scale Testing: Latency obtained from Tensor

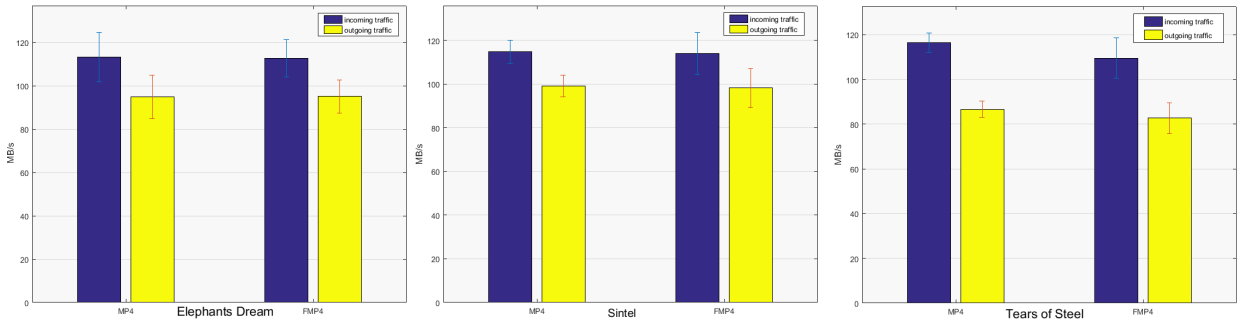



**Figure 10.** Large scale Testing: Incoming and outgoing traffic

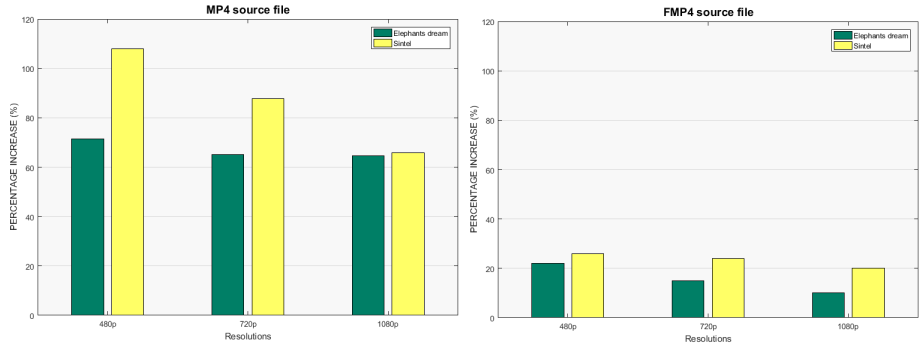



**Figure 11.** Percentage increase of the outgoing throughput when the dref is cached, for different resolutions.

## References


[1] Marina Kalkanis, Video Factory: Transcoding Video for BBC iPlayer [Online]. Available: http://www.bbc.co.uk/blogs/internet/entries/eb9d3ca8-56bb-39a0-b990-07e14c5996f4 (visited on 20/12/2017)

[2] Pantos, R., May, W. 2010. HTTP Live Streaming. IETF draft, November 2015. https://tools.ietf.org/pdf/draft-pantos-http-live-streaming-18.pdf.

[3] ISO/IEC JCT1/SC29 MPEG, ISO/IEC 23009-1:2014: Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats' ISO, 2014.

[4] Unified Packager. [Online]. Available: http://docs.unified-streaming.com/documentation/package/index.html (visited on 20/12/2017).

[5] ISO/IEC standard 13818-1 Generic coding of moving pictures and associated audio information – Part 1: Systems

[6] ISO/IEC 14496-12:2005 Information technology – Coding of audio-visual objects – Part 12: ISO base media file format

[7] ISO/IEC 23000-19 Information technology – Multimedia application format (MPEG-A) – Part 19: Common media application format (CMAF) for segmented media

[8] S. Akhshabi, L. Anantakrishnan, C. Dovrolis, and A. C. Begen, 'What happens when HTTP adaptive streaming players compete for bandwidth?' in Proc. NOSSDAV, 2012, pp. 9-14.

[9] T.-Y. Huang, N. Handigol, B. Heller, N. McKeown, and R. Johari, 'Confused, timid, and unstable: picking a video streaming rate is hard', in IMC'12: Proceedings of the 2012 ACM conference on Internet measurement conference. New York, New York, USA: ACM Request Permissions, Nov. 2012, pp. 225-238.

[10] J. W. Kleinrouweler, S. Cabrero, R. van der Mei and P. Cesar, 'Modeling Stability and Bitrate of Network-Assisted HTTP Adaptive Streaming Players', 2015 27th International Teletraffic Congress, Ghent, 2015, pp. 177-184

[11] Ricky K. P. Mok, Xiapu Luo, Edmond W. W. Chan, and Rocky K. C. Chang. 2012. QDASH: a QoE-aware DASH system. In Proceedings of the 3rd Multimedia Systems Conference (MMSys '12). ACM, New York, NY, USA, 11-22.

[12] Abdelhak Bentaleb, Ali C. Begen, and Roger Zimmermann. 2016. SDNDASH: Improving QoE of HTTP Adaptive Streaming Using Software Defined Networking. In Proceedings of the 2016 ACM on Multimedia Conference (MM '16). ACM, New York, NY, USA, 1296-1305.

[13] Jan Willem Kleinrouweler, Sergio Cabrero, and Pablo Cesar. 2016. Delivering stable high-quality video: an SDN architecture with DASH assisting network elements. In Proceedings of the 7th International Conference on Multimedia Systems (MMSys '16). ACM, New York, NY, USA, Article 4, 10 pages.

[14] F. Jokhio, A. Ashraf, S. Lafond and J. Lilius, "A Computation and Storage Trade-off Strategy for Cost-Efficient Video Transcoding in the Cloud," 2013 39th Euromicro Conference on Software Engineering and Advanced Applications, Santander, 2013, pp. 365-372.

[15] H. Zhao, Q. Zheng, W. Zhang, B. Du and H. Li, "A Segment-Based Storage and Transcoding Trade-off Strategy for Multi-version VoD Systems in the Cloud," in IEEE Transactions on Multimedia, vol. 19, no. 1, pp. 149-159, Jan. 2017.

[16] Rufael Mekuria, Jelte Fennema, and Dirk Griffoen. 2016. Multi-Protocol Video Delivery with Late Trans-Muxing. In Proceedings of the 2016 ACM on Multimedia Conference (MM âĂŹ16). ACM, New York, NY, USA.

[17] Daniel Silhavy, Stefan Pham, and Stefan Arbanowski. 2017. Performance considerations of HTML5-based dynamic packaging for media streaming. In Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'17). ACM, New York, NY, USA, 7-12.

[18] Wowza Media Systems [Online] Available: https://www.wowza.com

[19] CodeShop B.V., "Unified Streaming," 5 4 2016. [Online]. Available: www.unified-streaming.com.

[20] ISO/IEC JTC1/SC29/WG11 MPEG2017/N17262, Network Based Media Processing, Use cases and draft requirement for NBMP (v2).

[21] Amazon Web Services [Online] Available: https://aws.amazon.com/

[22] Amazon Simple Storage Solution (S3)[Online] Available :https://aws.amazon.com/s3/

[23] Tears of Steel [Online] Available: https://mango.blender.org/about/

[24] Sintel [Online] Available: https://durian.blender.org/

[25] Elpehands Dream [Online] Available: https://orange.blender.org/

[26] Apache HTTP server benchmarking tool [Online] Available:http://httpd.apache.org/docs/current/programs/ab.html

[27] Amazon EC2 Instance Types [Online] Available: https://aws.amazon.com/ec2/instance-types/

[28] Abe Wiersma. 2016. Determining meaningful metrics for Adaptive Bit-rate Streaming HTTP video delivery Bachelor thesis. University of Amsterdam (UVA), Amsterdam, The Netherlands

[29] M Seufert, S Egger, M Slanina, T Zinner, A survey on quality of experience of HTTP adaptive streaming

[30] Apache Caching Guide [Online] Available: https://httpd.apache.org/docs/2.4/caching.html

[31] W. Glozer, "WRK Modern HTTP benchmarking tool," 8 4 2016. [Online]. Available: https://github.com/wg/wrk.

[32] Performance Co-Pilot [Online] Available: http://pcp.io/