Hasselt University
Master of Computer Science

Improving DASH Streaming with Server and Network Assistance (SAND)

Author
Joris Herbots

Promotor
Prof. dr. Peter Quax

Co-promotor
Prof. dr. Wim Lamotte

Mentor
dr. Maarten Wijnants

2018-2019

Acknowledgements

I would like to start by thanking all the people who stood by my side while realizing this thesis, this work would not have been possible without them.

Firstly, I would to thank my mentor Dr. Maarten Wijnants for his continuous guidance, motivation and support of my master thesis. His insightful feedback and knowledge have been of great value to me and have helped me a lot during the writing and development of this thesis. I could not have imagined having a better mentor for my master thesis.

I would also like to express my gratitude to my promotors professor Peter Quax and professor Wim Lamotte for providing this unique opportunity. Without their support, it would not have been possible to research this thesis subject.

Last, but definitely not least, I would like to thank my girlfriend, my family and my friends for their unconditional support and patience.

List of Abbreviations

Abbreviation	Meaning
ABNF	Augmented Backus–Naur Form
ABR	Adaptive Bitrate
BOLA	Buffer Occupancy based Lyapunov Algorithm
CDN	Content Delivery Networks
CE	Core Experiment
CORS	Cross-Origin Resource Sharing
DANE	DASH aware network element
DASH-IF	DASH Industry Forum
DNS	Domain Name System
EWMA	exponentially weighted moving average
FQDN	Fully Qualified Domain Name
HAS	HTTP Adaptive Streaming
HDS	HTTP Dynamic Streaming
HEVC	High Efficiency Video Coding
HLS	HTTP Live Streaming
HTTP	Hypertext Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
ID	Unique Identifier
IP	Internet Protocol
ISP	Internet Service Provider
JSON	JavaScript Object Notation
MITM	man-in-the-middle
MPD	Media Presentation Description
MPEG	Moving Picture Expert Group
MPEG-DASH	Dynamic Adaptive Streaming over HTTP
MSS	Microsoft Smooth Streaming
Mbit	Megabit
NAT	Network Address Translation
OF	OpenFlow
PED	Parameters Enhancing Delivery
PER	Parameters Enhancing Reception
POC	Proof of Concept
PQDN	Partially Qualified Domain Name
QoE	Quality of Experience
REST	Representational State Transfer
RNE	Regular network element

RTMP	Real-Time Messaging Protocol
RTMPT	Real-Time Messaging Protocol Tunneled
RTP	Realtime Transport Protocol
RTSP	Realtime Streaming Protocol
SAND	Server and Network Assisted DASH
SDN	Software-defined networking
SIP	Session Initiation Protocol
TBF	Token Bucket Filter
TC	Traffic Control
TCP	Transmission Control Protocol
UDP	User Datagram Protcol
UUID	Universally unique identifier
VO	Variable Oject
XML	Extensible Markup Language
XSD	XML Schema Definition

Contents

Abstract

Video streaming over the Internet has increasingly become more popular in the last decade due to advancements in technology and hardware. We have seen a shift towards the adaptive streaming paradigm, with implementations such as Dynamic Adaptive Streaming over HTTP (MPEG-DASH) becoming the defacto standard. DASH introduces an effective and scalable way for major content providers to deliver media streaming services over the Internet. Media origin servers host media segments which are compatible with CDNs and firewalls, clients proceed by streaming each segment individually and in an adaptive way based on, for example, network conditions. MPEG-DASH's adaptation behaviour, nevertheless, suffers from performance problems when a large number of clients are deployed on a shared network connection. This shared network connection, known as the last mile, typically forms a bottleneck when a large number of clients compete for bandwidth, in turn leading to many video quality changes and stalling behaviour both of which are detrimental to the Quality of Experience (QoE) of a user.

In 2017, the Moving Picture Expert Group (MPEG) released its Server and Network Assisted DASH (SAND) specification, which defines a protocol for exchanging realtime operational characteristics in the form of messages between entities involved in the DASH streaming process. This enables servers and middleboxes who understand DASH content passing through them, to provide more accurate information such that clients can make better informed choices concerning their adaptation logic. In this thesis, the SAND specification in a shared network connection use case is explored. More specifically, we will create a bandwidth guidance entity as well as a smart web cache entity which will provide an overall better QoE towards users on the shared network condition, as well as reducing the combined bandwidth consumption by all these clients. As an example, our evaluation shows that deploying a bandwidth guidance entity within a shared network setting, MPEG-DASH clients end up using a fair bandwidth share, in turn leading to a better QoE for each individual user.

Chapter 1

Introduction

Multimedia delivery forms the main bulk of Internet traffic anno 2019 of which video streaming is the biggest. The sharing of media, in particular video and audio, is rapidly rising in popularity. Social media platforms such as Facebook[1] and Instagram[2] allow users to share and express their experiences through the use of (live) video streaming. Amateur content creators get to express their creativity through platforms such as Youtube[3] or Vimeo[4]. Even professional companies like Netflix[5] and HBO[6] or local ISPs such as Telenet[7] offer movies and series on a paid streaming platform. All these services provide an easy to use platform through which users can watch content on demand. This can be noticed in the footprint IP video traffic has on a global scale. Projections by Cisco indicate that video traffic will rise up to 82% of all IP traffic by 2022 coming from 75% in 2017 [1].

As of lately a new paradigm has been adopted for streaming media over the Internet. Traditionally, streaming was done by connecting to a media streaming service (e.g., with the **Realtime Streaming Protocol** (RTSP) over the **Realtime Transport Protocol** (RTP)) in which a server-driven approach has control over the streaming logic and clients issue commands to control the media streaming (e.g., play, pause, stop). This however does not scale well and requires complex setups, which is why nowadays the majority of Internet streaming sessions happens through adaptive streaming protocols such as **MPEG-DASH**. These protocols shift the responsibility of the streaming logic to a client-driven approach; instead of the server deciding which quality to provide to a client based on network characteristics (e.g., throughput), the client now performs this logic.

These adaptive streaming protocols typically utilize HTTP as their transport protocol, hence they are also called **HTTP Adaptive Streaming** (HAS) protocols. HAS-based clients yield many advantages over traditional RTP-based clients; all devices capable of video playback that include access to the Internet (e.g., a smart TV) inherently have access to adaptive streaming. HAS also scales very well and works transparently with current web infrastructures such as web caches and CDNs. Despite these advantages, HAS-based clients suffer from several drawbacks. For example: if a web cache is available on the media delivery path between the client and server, cache misses cause extra delay in the media content being transferred, which might be misinterpreted by clients as a drop in throughput, causing it to change its streaming quality which in turn directly influences the Quality of Experience for a user [2, 3]. Another problem which is becoming more prevalent, is concurrent HAS clients in shared networks competing for bandwidth. Clients competing with each other for bandwidth will experience throughput fluctuations which in turn leads to bitrate switching and stalling which has a negative impact on QoE [4].

Previous examples show how a lack of network information can have detrimental effects to the streaming experience of a user. This is one of the reasons why the industry and standardization bodies recently have started looking into server and network assisted streaming. The MPEG-DASH protocol received an extra part in 2017 called **Server and Network Assisted DASH** (SAND). The goal of SAND is to

[1] https://www.facebook.com/
[2] https://www.instagram.com/
[3] https://www.youtube.com/
[4] https://vimeo.com/
[5] https://www.netflix.com/
[6] https://www.hbo.com/series
[7] https://www.yeloplay.be/

enhance delivery of DASH content by sharing real-time operational characteristics (e.g., cache availability, throughput, QoS information, ...) with DASH clients in order to achieve a more efficient streaming process.

1.1 Research Questions

The purpose of this thesis is to focus on the advantages that server and network assistance offer in adaptive streaming. For this we will look at the standardized MPEG-DASH protocol and its new SAND specification. We will look at use cases that involve clients consuming DASH streams whilst competing for bandwidth on a shared network connection. Examples of such environments include households, apartment complexes, airports, university buildings, etc... The limited throughput of the shared last mile in these networks is the bottleneck which makes it hard to support multiple concurrent users streaming (high quality) content.

This master thesis falls under the category of a **case study**, we will focus on previously mentioned problems with the following research questions:

1. What aspects are required to transition from the SAND specification to a deployable SAND implementation?
2. Is it possible to guide multiple users on a shared network connection into a fair bandwidth usage using only SAND such that they do not experience the detrimental effects caused by bandwidth competition?
3. Can we support multiple clients consuming the same DASH stream within the same network in such a way that:
   (a) Clients enjoy the same or a better overall Quality of Experience compared to a non-SAND scenario
   (b) The overall bandwidth consumption is lower than when the clients would individually compete for the same content at the quality provided by SAND

1.2 Thesis Overview

To answer previous research questions, we will analyze the SAND specification and come up with a proof of concept. This thesis is divided into seven chapters:

- Chapter 2: A brief introduction into streaming describing the older streaming protocols and the new HTTP adaptive streaming paradigm.
- Chapter 3: A walk-through of the SAND specification.
- Chapter 4: In this chapter we explain our proof of concept that will tackle our research questions.
- Chapter 5: An experimental evaluation of our proof of concept together with its findings.
- Chapter 6: This chapter will review related works on the topic of server and network assistance in adaptive streaming, more specifically with a focus on MPEG-DASH and SAND.
- Chapter 7: The conclusion of this thesis together with future work in the domain of network and server assistance for HTTP adaptive streaming which we identified while realizing this thesis.

Chapter 2

Streaming

In order to better understand the details and context of this thesis, this chapter will serve as a short introduction into the world of streaming over the Internet. Multimedia - such as video and audio - being streamed, also known as **streaming media**, is a process in which a server (the origin or provider) transmits multimedia content to a client (the receiver) which consumes it immediately, contrary to **downloading** where the client saves the multimedia content on a local device to be consumed after the download process has finished. Streaming enables users to enjoy multimedia content as if they had saved it locally without the actual space requirement and without having to wait until the full download if complete. In a typical setup over the Internet, the server presents its contents via a public address which the clients access using software, e.g. an HTTP web server with video files accessed through a web browser, or an RTSP (Realtime Streaming Protocol) live security camera feed consumed by a mediaplayer such as VLC[1].

In order to enable streaming of media and depending on the sort of content and quality, a minimal amount of **bandwidth** is required. During the early 2000s the growth in network capabilities and bandwidth in the last mile (i.e. the Internet connection between a household and its Internet Service Provider (ISP)) facilitated the growth of Internet media streaming. Suddenly it became feasible to stream and consume SD quality video in real time. At the time of writing, the global average Internet connection speed is 59.45 Mbit per second with most first world countries reaching speeds far above that average [5]. The required bandwidth for media typically scales up in bitrate as its quality rises. Netflix for example recommends the following bitrate capabilities: 3 Mbit per second for SD quality (i.e. 360p), 5 Mbit per second for HD quality (i.e. 720p) and 25 Mbit per second for Ultra HD quality (i.e. 4K) [6]. As such, media content providers typically provide their media in different quality levels to reach as many clients as possible.

2.1 Media Composition

All digital media - either saved locally or being streamed - is represented and stored in so called **container formats**, sometimes also called **wrappers**. A container format defines the overall structure of a file, including how its video, audio and metadata information are multiplexed together [7]. A container format however does not define how the video and/or audio is encoded (i.e. binary represented). Container formats are usually indicated by giving a media file a certain extension. For example, video can be identified by one of the following (but not limited to): AVI, MP4, WEBM, FLV or MOV. Every container format has its own traits and supported codecs which are all specified in its metadata structure such that a playback device can properly decode (i.e. interpret) the content. Simpler container formats are exclusive to one audio or video stream (e.g. WAV is a popular Windows audio only container format). More advanced container formats allow multiple video and audio streams to be contained in a single file (e.g. MKV is an open standard container format intended to serve as a universal format for storing multimedia [8]). Figure 2.1 shows the hierarchy of a container format and an example of the Matroska container format. The choice of a container and codec(s) depends, for example, on the needs of the media creator or on the hardware support of the target audience machines.

Decoding of video and audio is usually done by the CPU which is a taxing process; most modern devices however come equipped with hardware acceleration support (e.g. Intel Quic Sync [9]) for specific codecs

[1] https://www.videolan.org/

to unburden the CPU from the decoding task. Hardware acceleration plays a big role on mobile devices (e.g. smartphones or laptops) which are designed to be power efficient thus making them inherently not fast for media decoding when considering the CPU alone, they typically can decode video streams but this drains the battery fast and generates heat which will cause these passive cooled devices to throttle the CPU down. This heterogeneous ecosystem of devices with different capabilities is what **Adaptive Streaming** (also known as **HTTP Adaptive Streaming** (HAS)) tries to tackle (see Section 2.4).

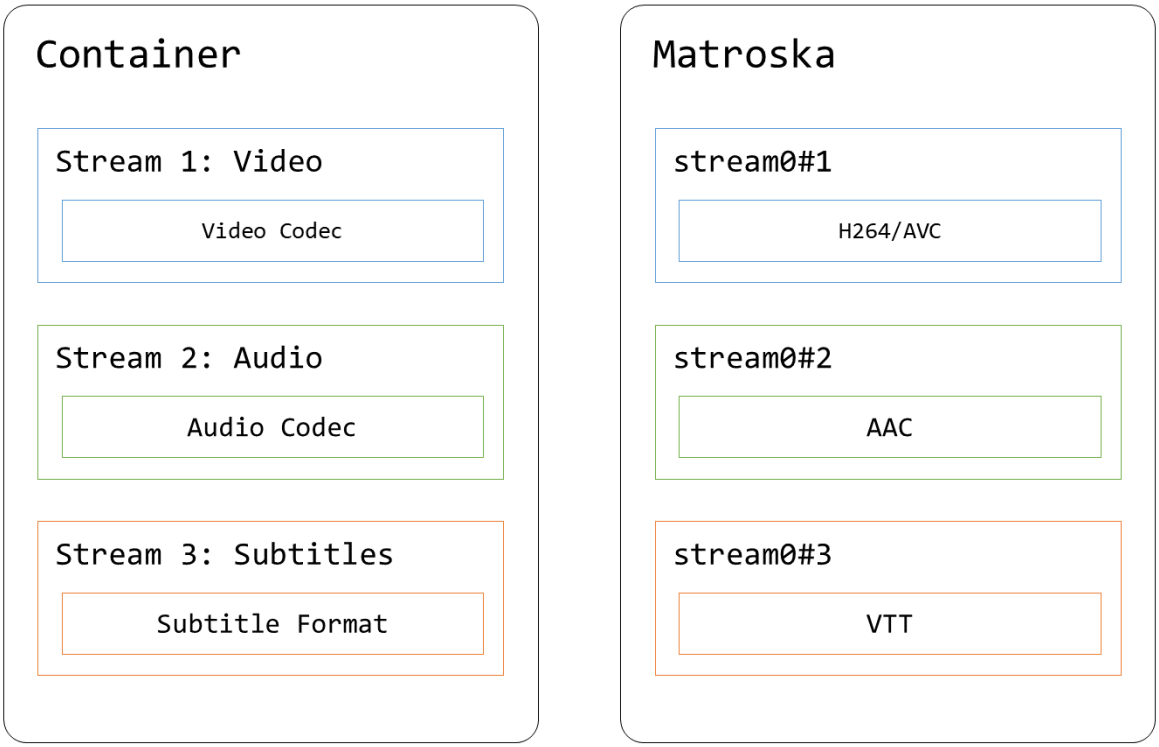

Figure 2.1: On the left a container format hierarchy; on the right an example of a Matroska container format containing three streams of which one video stream encoded with the H.264 codec, one audio stream encoded with the AAC codec and finally a subtitle stream in the WebVTT format. The *streamx#y* naming comes from how the Ffprobe[3] tool outputs metadata information about media files.

2.2 Early Streaming Protocols

Early streaming protocols had to invent the wheel when it comes to streaming media over IP networks, **Realtime Streaming Protocol** (RTSP) and **H.323** are examples of such endeavours. H.323 was designed as a real time media protocol for use cases such as video conferencing or livestreams [10]. RTSP on the other hand was designed to have Videocassette Recorder (VCR) like capabilities, e.g. play, pause and seek. Both protocols use the **Realtime Transport Protocol** (RTP) underneath for its end-to-end data transmissions. The end-to-end principle implies that only server and client require intelligence about the streaming process and that the systems that make up the network path only need to forward the data. The underlying architecture of RTP uses the **User Datagram Protocol** (UDP) as its transport protocol to avoid the overhead that the **Transmission Control Protocol** (TCP) introduces. The ultimate goal of RTP, contrary to the reliability goal of TCP, is to pursue timeliness [12, 11]. Peers that wish to instantiate an RTP connection usually need some knowledge about each other and each other's capabilities (e.g. in terms of codec support), i.e. the RTP session parameters which are exchanged via the **Session Initiation Protocol** (SIP) [13].

The actual content itself being transmitted, e.g. audo and video, are not defined by the protocol. In order to stay up to date, RTP enables future updates through its **profile** and **payload format** system. Each type of content being streamed is defined in a so called profile, popular profiles are the video and audio profile. Every profile is associated with multiple payload formats which define the supported codec.

Because of this system, the RTP protocol had the ability to evolve and adapt to the newest inventions in media codecs and network protocols. At the time of writing, RTP supports most latest generation codecs such as High Efficiency Video Coding (HEVC) and even modern network requirements such as encrypted traffic all through the addition and expansion of profiles [14, 15].

RTP is still deployed as the go-to protocol for real time data transport, e.g. Voice over IP or gaming [16], but has largely been superseded in the media streaming use case by more modern streaming protocols that make use of existing web infrastructures.

2.3 Modern Streaming Protocols

One of the biggest drawbacks of streaming protocols such as RTP is that all logic resides at the server-side; streaming heavily relies on server CPUs to process and distribute media to all clients, a resource which nowadays is rather not spent on such trivial tasks. RTP also inherits the unreliable transport layer properties tied to carrying its traffic over UDP, making it a stateless design which requires some kind of session bookkeeping. RTP is considered to be a transport protocol but in reality runs on the application layer of the OSI model [17]; RTP in other words runs in user space which is inherently slower than kernel space (also known as ring 0). Because of how modern networks are structured, many clients reside behind a **Network Address Translation** (NAT) device and/or a firewall which hinders incoming UDP connections; complex workarounds are usually required combined with a UDP hole punching technique in order for streaming sessions to work correctly [18].

It is because of the requirement for workarounds and/or complex server-side protocols that most streaming solution nowadays look at TCP [19]. TCP inherently consumes extra bandwidth compared to UDP due to its stateful design but is preferred because of its ease to use and widespread support across servers, clients and network devices. Another modern approach is to use the **Hypertext Transfer Protocol** (HTTP) [20]; this web based protocol nowadays comes with support for encrypted traffic, performs network address translations of domains through the use of the **Domain Name System** (DNS) and is built around a system to identify resources (i.e. media content) based on **Uniform Resource Identifiers** (URI) [21]. By design, the HTTP specification allows network elements to improve connections between servers and clients, e.g. HTTP web caches that improve network latency and reduce network traffic.

Real-Time Messaging Protocol (RTMP) was one of the first streaming protocols to utilize TCP connections and was sometimes encapsulated by HTTP (called **Real-Time Messaging Protocol Tunneled** (RTMPT)) to traverse firewalls; the benefit of traversing corporate firewalls outweighed the added penalty of HTTP headers. The proprietary protocol was originally developed by Macromedia and later made public after the company was acquired by Adobe. Its original use was to be the streaming protocol used between a Flash player and server [22]. RTMP enjoyed a long time user base because of the popularity Flash based application had on the web until 2017 Adobe decided to call Flash end of life [23].

The original[4] web envisioned clients downloading content from servers and then process these contents on the clients' machines, thus not straining servers in terms of processing power for rendering web pages. Instead all web clients (i.e. browsers) interpret web based content (e.g. HTML) and build up a visual representation tailored to the device on which the browser runs. By deploying media streaming over HTTP, the same paradigm for content processing was applied, shifting the responsibility of processing streams to the client side. The problem with this setup is that servers have no idea what media content a client can or wishes to handle; adaptive streaming was invented for this reason.

2.4 Adaptive Streaming

As mentioned in Section 2.3, (HTTP) adaptive streaming - as its name suggests - is a method to let a client dynamically adapt to changing conditions during a streaming process. The streaming logic contained at the client side decides during media playback what quality or version of the content suits the client best

[4] We nowadays notice a shift in this paradigm where origin servers and/or edgeworkers pre-generate web pages (e.g. React framework [https://reactjs.org/]) thus reducing the actual processing requirements for clients. This is especially helpful in cases where clients are not equipped with the right amount of processing power or when latency plays an important role (e.g. e-commerce websites).

based on its characteristics (e.g. video element size), device capabilities (e.g. support for H.264 hardware accelerated decoding) and the available network bandwidth which can fluctuate due to many reasons such as (but not limited to): mobile connection instability, oversaturated last mile, congested origin server or network path, ...

2.4.1 Adaptive Streaming Content Preparation

The way origin servers provide media content also changes because of this adaptive approach. Instead of only providing media in one type of quality, the origin now has to provide a wide range of qualities in order to satisfy a heterogeneous target audience. In practise, this is done through an **adaptive streaming preparation** step in which an input media is transcoded[5] into multiple representations, each with differing settings [24]. Typically, input media is transcoded into multiple videos each with a different resolution (240p, 360p, 720p, 1080p, 1440p and 4K are commonly used) and corresponding bandwidth budget[6] and audio qualities (all in seperate files); lower resolution videos and audio take up less bandwidth than their high resolution counterparts. All these transcoded video and audio files, also called the **intermediate formats**, are then sliced into small **segments** of specific short duration (typically around two to ten seconds long). In order to keep track and inform clients of all these different qualities and how they are segmented, the preparation step finally bundles all information into what is called a **manifest file** [24, 25]. Figure 2.2 gives a visual representation of the previously explained pipeline. Clients begin their streaming process by fetching the manifest file and deciding what representation to stream based on the manifest's contents.

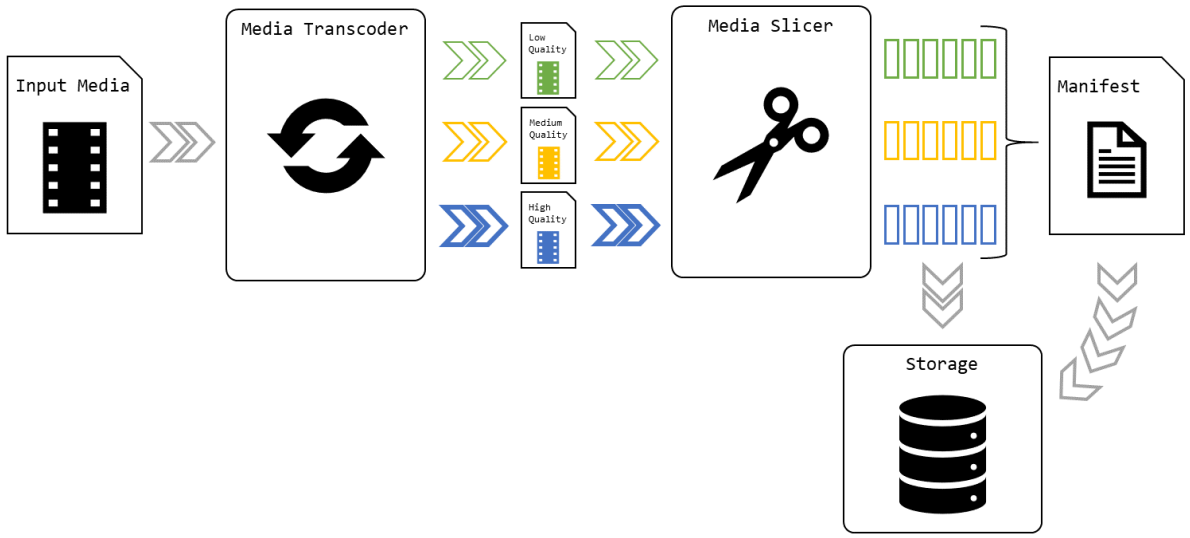

Figure 2.2: Adaptive streaming content preparation pipeline: Input fed to the media transcoder produces multiple qualities that get sliced into short temporal segments which are stored away and referenced by the manifest file. The different colors indicate the different media qualities. The temporal segments indicated by small rectangles differ in storage size depending on the media quality they belong to.

2.4.2 MPEG-DASH

During the early years of adaptive streaming, multiple protocols were developed, each similar to the above explained behaviour, but with its own twist of what exactly is supported. These protocols included **HTTP Live Streaming** (HLS) by Apple [26], **Microsoft Smooth Streaming** (MSS) [27] and **HTTP Dynamic Streaming** (HDS) by Adobe [28]. These adaptive streaming protocols were all non-standardized, proprietary in nature and (mostly) driven by individual companies all with their own idea about how the future of adaptive streaming should look like. Hence in 2010 development for a

[5] A process that takes an already encoded media input, typically decodes it into a raw representation and re-encodes it to a new format. During this process characteristics such as the bitrate budget, actual video resolution and/or used codecs can change. The net result is the same media but represented differently under the hood.

[6] It is not unusual to have videos with the same spatial and temporal resolutions but encoded at different bitrates. Codecs such as H.264 and H.265 allow for more advanced compression algorithms which require more processing power to decode but result in less bandwidth used.

general use specification design by the **Moving Picture Expert Group** (MPEG) was started called **Dynamic Adaptive Streaming over HTTP** (MPEG-DASH) which was internationally standardized in 2012. At the time of writing many big companies - among which Netflix and Youtube - have adopted the MPEG-DASH specification as their default for adaptive streaming.

An MPEG-DASH stream starts like any regular adaptive streaming protocol by fetching the manifest file, the manifest is referred to as the **Media Presentation Description** (MPD) and utilizes **Extensible Markup Language** (XML) as its data format. The MPD contains at least one **period** element which describes the actual content being played by its start time and duration; typically a manifest contains one period for the full media it represents. The period itself contains at least one **adaptation set** that represents a media stream (see Section 2.1), for example, a manifest representing media with video, audio and subtitles will have one period with three adaptation sets for video, audio and subtitles respectively. Each adaptation set typically contains multiple **representation** elements that represent the different qualities as explained in Section 2.4.1. A representation element offers three ways of referring to the segmented media files:

1. Individual segment URLs through the **SegmentList** and **SegmentURL** elements
2. Template-based system through the **SegmentTemplate** element
3. Range-based system; either on byte ranges or time ranges through the **SegmentTimeline** element

Of these three ways, the first two are the most simple and popular. Listing A.2 shows the difference a template makes in terms of required XML lines compared to specifying each segment individually. For examples of how a manifest file is represented, see Appendix A; for a high level overview of the MPEG-DASH streaming process see Figure 2.3.

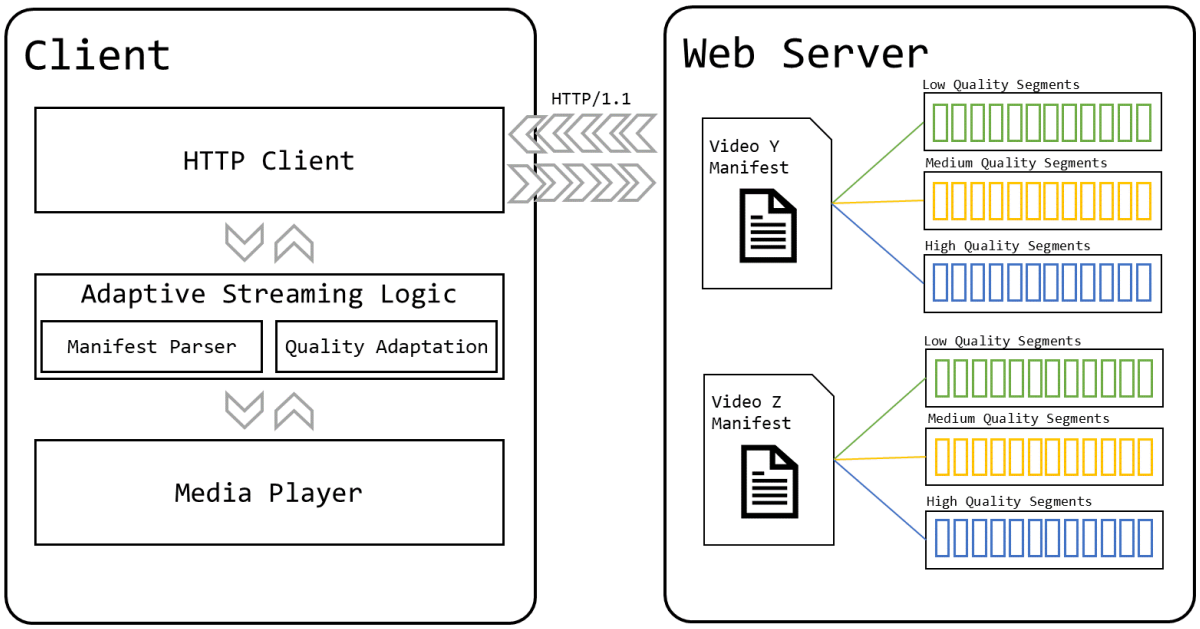

Figure 2.3: MPEG-DASH inner workings: A client capable of MPEG-DASH fetches a manifest, which gets parsed; based on its contents, the quality adaptation logic decides what quality segments to fetch from the origin server and to finally display to the user in the media player.

Chapter 3

Server and Network Assisted DASH

As explained in Chapter 2, HTTP-based adaptive streaming has risen to be the primary choice when it comes to media streaming over the Internet. The most popular protocol for streaming is the standardized MPEG-DASH, which enjoys the support of many large industry bodies including the DASH Industry Forum (DASH-IF). Because MPEG-DASH is designed to be a protocol which operates over HTTP in an end-to-end fashion where the streaming logic resides at the client, it inherently enjoys the benefits HTTP provides such as scalability through the use of **web caches** and **Content Delivery Networks** (CDNs).

These decentralized benefits and the one-sided client logic however also introduce some drawbacks for both the origin server providing the media content and the client fetching the media. The origin server has no control over client behaviour, which in certain scenarios is an unwanted side-effect. For example, an origin server with QoS support might have trouble streaming at a fixed (i.e, promised) quality when in-network elements - such as caches - interfere. On the other hand, the client can experience drawbacks too, for example, a cache miss will be interpreted as a bandwidth reduction to the origin server, which in turn will make the client drop the quality being streamed, or in a worst case scenario will cause the media playback to stall.

Previous drawbacks can be defined as one-to-one drawbacks, however, when multiple DASH clients are involved in the streaming process, they can indirectly influence each other by competing for the same shared bandwidth from the origin server, thus introducing oscillations which are detrimental to the **Quality of Experience** (QoE) [3].

When examining some of the previously mentioned drawbacks, it all boils down to a lack of information. When the origin server, network entities such as caches/CDNs and clients are able to exchange network- and streaming-related information, better informed choices can be made to avoid or minimize the aforementioned drawbacks (e.g. a cache entity detecting a cache miss could insert an extra piece of information indicating what happened upon which the client does not decide to lower its choice of streaming quality).

As such in 2013, the MPEG started a Core Experiment (CE) in which server and network assisted DASH was explored. By collecting and exploring use cases, experts came up with the **Server and Network Assisted DASH** (SAND) specification, officially referred to as ISO/IEC 23009-5 [29], officially published in 2017. The SAND architecture defines a set of messages between DASH clients and network entities for the purpose of sharing operational characteristics, for example server throughput metrics, which in turn enable these entities to make better informed choices such that potentially negative impacts on Quality of Experience can be mitigated.

This chapter will provide a better understanding of server and network assistance using SAND in DASH streaming.

3.1 Architecture

A normal DASH stream consists of a media origin (i.e., a web server) and a DASH client which communicate in an end-to-end fashion. Entities that wish to improve their streaming experience can upgrade

their service with the SAND protocol to enable in-band entities and out-of-band (OOB) entities to share operational characteristics (see Section 3.1.3 for more information). It is important to note that the (full) implementation of SAND is neither mandatory nor necessary to set up a DASH stream; the objective of SAND is to exchange streaming- and network-related information with SAND-capable entities to enhance the DASH streaming session. Depending on their function within the streaming process, the SAND architecture divides these entities into four broad categories:

1. **DASH clients**
2. **Regular network elements** (RNEs)
3. **DASH aware network elements** (DANEs)
4. **Metric servers**

3.1.1 Dash clients

Dash clients capable of SAND have to augment their internal processing model. A regular DASH client consists of a **Media engine** responsible for requesting and processing MPEG-DASH media segments, an **Application** responsible for user interaction and the **DASH access engine** which communicates with the network to coordinate the actual data communication in order to retrieve segments and manifests[1] [30]. The SAND augmentation takes place at the DASH access engine component which now has to have support for SAND delivery channels (see Section 3.3) and SAND messages (see Section 3.2). Figure 3.1 shows an abstract DASH client model augmented for SAND communication.

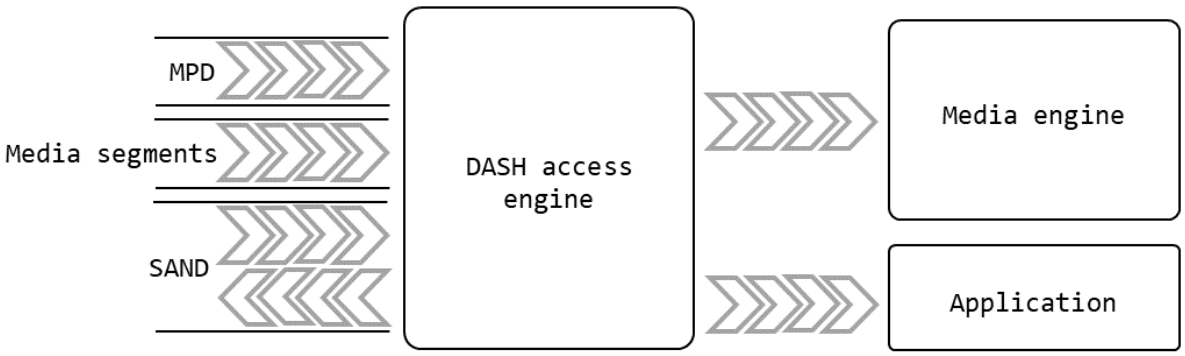

Figure 3.1: The ISO 23009-1 [30] DASH client model augmented with a SAND communication channel.

3.1.2 Regular Network Elements

RNEs are network entities on the end-to-end media delivery path between DASH clients and origin servers. They are neither DASH- nor SAND-aware and treat all traffic passing through them in an agnostic fashion (i.e., simple pass through). Examples of RNEs include, but are not limited to: routers, switches, gateway servers, CDNs, web caches, ... A web cache or CDN capable of caching/providing DASH streams is also considered a RNE since they blindly operate on web resources and do not question whether the requested content from a client is for a DASH streaming session. They can however be augmented to be DASH- and SAND-aware which then makes them a DANE (see Section 3.1.3). It is important to note that SAND messages exchanged over a non-encrypted HTTP connection between SAND-capable entities must be flagged with a no-caching directive to prevent RNEs from caching the messages. Figure 3.2 depicts an abstract network overview of a DASH stream with an in-band RNE that lies on the end-to-end delivery path. It is important to note that when we address a DANE as in-band, we target a DANE that is on the end-to-end delivery path; an out-of-band DANE targets a DANE which does not reside on the end-to-end media delivery path.

[1]The DASH specification officially states that manifest fetching is out of scope, in other words, it is left over to the actual implementation. However, typically DASH clients incorporate this behaviour in the DASH access engine (e.g., DASH.js[https://github.com/Dash-Industry-Forum/dash.js/] and Shakaplayer[https://github.com/google/shaka-player/].

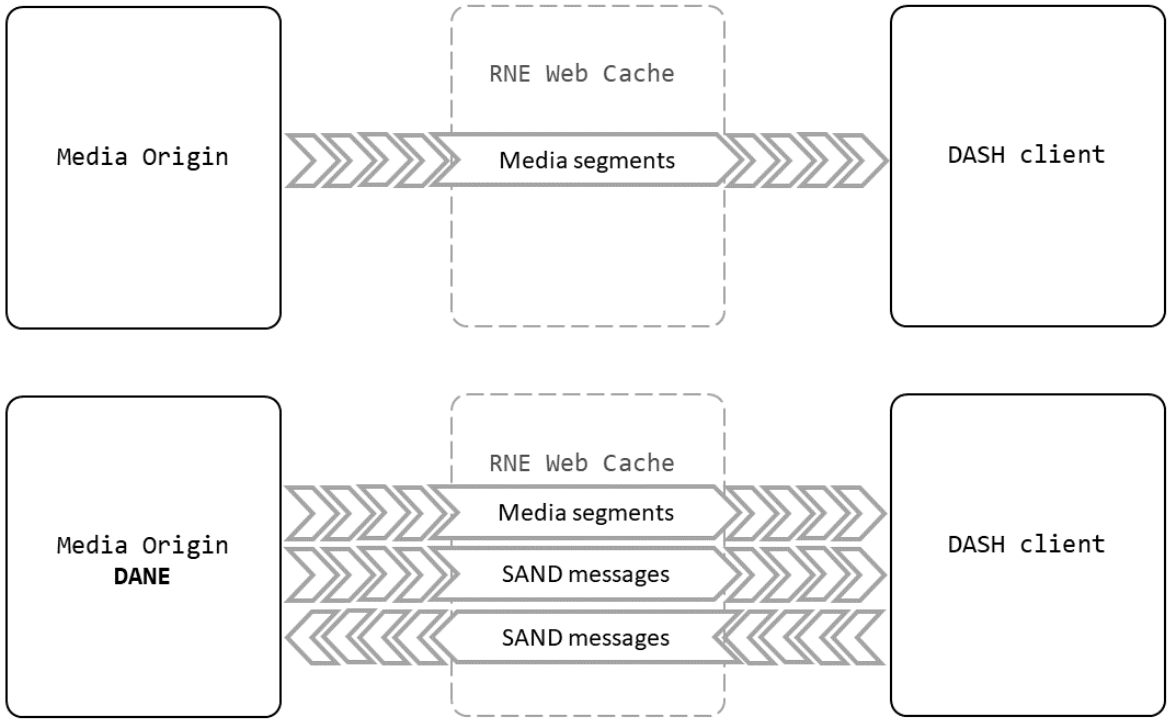

Figure 3.2: An abstract network overview showing a regular network element fit transparently into the SAND architecture. The top example depicts a RNE in a non-SAND DASH session whilst the bottom example depicts a RNE in a SAND-enabled DASH session.

3.1.3 DASH Aware Network Elements

Contrary to RNEs, entities that have a minimal knowledge about DASH streams taking place are called DANEs; such entities implement a subset or the full SAND specification. Their capabilities range from providing feedback to SAND-capable clients and other DANEs, to the prioritization or even alteration of DASH streaming behaviour depending on its goal (e.g., a resource allocation entity that coordinates multiple clients into sharing network bandwidth in a fair manner). DANEs are not required to be in-band entities and can be contacted by clients or other DANEs as an out-of-band entity. Even though an OOB DANE does not perceive DASH traffic flowing through it, it can augment a DASH stream via SAND messages sent from either other DANEs or the DASH client directly. Figure 3.3 depicts an abstract network overview incorporating DANEs.

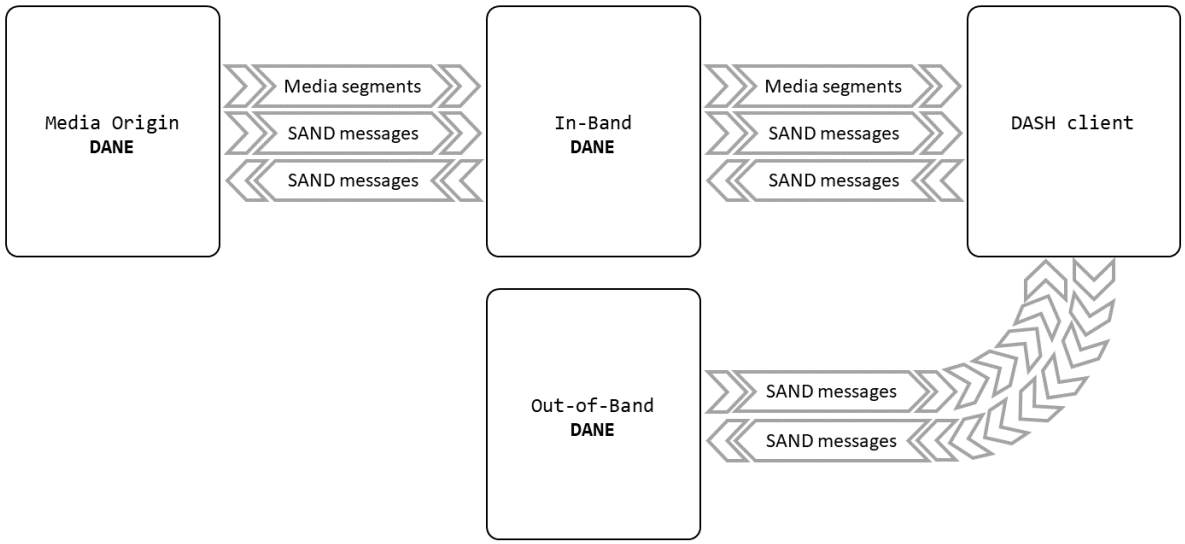

Figure 3.3: An abstract network overview depicting SAND communication between a media origin and client. An in-band network element capable of SAND participates in this communication and also sees media segments passing through itself. Another out-of-band DANE contacted by the client provides extra guidance during the DASH streaming process.

3.1.4 Metric servers

Metric servers implement the capability for aggregating and storing DASH metrics from DASH clients. Even though metric servers are defined as a separate category, in practise the SAND specification leaves open what an entity implements. In other words, it is possible for a DANE to also be a metric server. Figure 3.4 depicts a DASH session incorporating a metrics exchange with a metrics server.

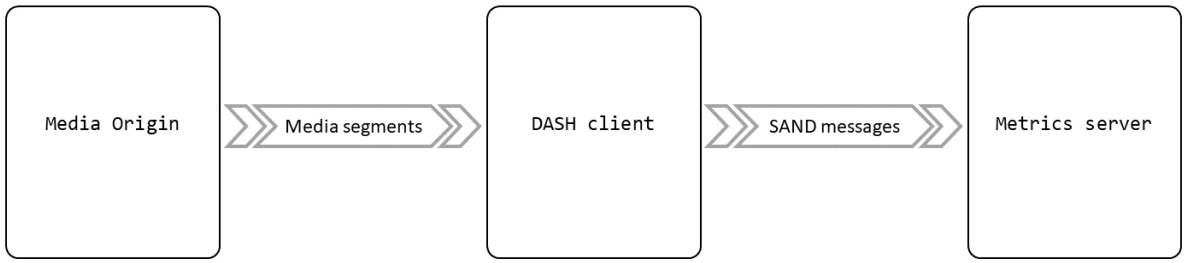

Figure 3.4: An abstract network overview depicting DASH metrics being exchanged via a SAND communication channel between a DASH client and an out-of-band metrics server. The media origin has no knowledge of SAND and does not know that the DASH client exchanges these messages with an external party.

3.2 Messages

Based on the architecture described in Section 3.1, the SAND specification came up with the following four categories of SAND messages, each with a specific goal in mind (see Sections 3.2.2, 3.2.3, 3.2.4 and 3.2.5 for a more detailed description):

1. **Status messages**
2. **Parameters Enhancing Reception** (PER)
3. **Parameters Enhancing Delivery** (PED)
4. **Metrics**

Based on these message categories and the SAND architecture described in Section 3.1, the SAND specification also considers the following interfaces for SAND message communication:

1. **Client-to-Metrics-server** interface: Carries metric messages
2. **Client-to-DANE** interface: Carries status messages
3. **DANE-to-DANE** interface: Carries PED messages
4. **DANE-to-Client** interface: Carries PER messages

Figure 3.5 depicts an abstract network overview including all types of SAND messages being exchanged.

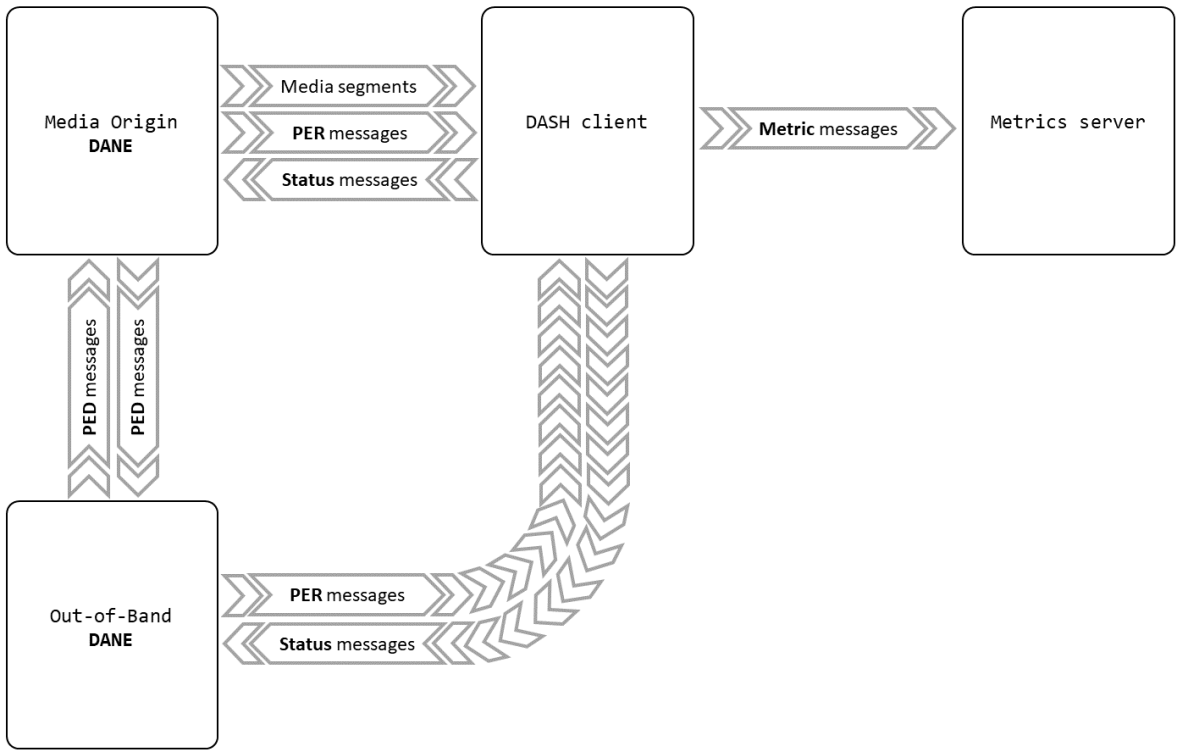

Figure 3.5: An abstract network overview depicting a DASH stream between a media origin and a client. The client communicates through status messages with the SAND capable media origin and an out-of-band DANE; both communicate back to the client through PER messages. Both DANEs also communicate with each other via PED messages. Finally, the client sends its metrics over to a metric server through metric messages.

3.2.1 Message Format

SAND messages are expected to be at least represented by the **Extensible Markup Language** (XML) data format [31] and HTTP header extension format. When exchanging SAND messages using XML, entities can signal this format by utilizing the formally registered MIME type `application/sand+xml`. SAND entities are however free to implement additional representation data formats (e.g., JavaScript Object Notation (JSON)), but support for XML and HTTP header extensions are mandatory.

The SAND specification defines a set of 21 general messages belonging to the categories explained in Sections 3.2.2, 3.2.3, 3.2.4 and 3.2.5 under the namespace `urn:mpeg:dash:sand:messageset:all:2016` (from now on referenced to as the **default SAND message set**). Implementations are free to implement the full default SAND message set or only a subset thereof; they are additionally not restricted to this message set and are free to create a new set of messages for their purposes under its own namespace. Each message is at the same time addressable by its message type (i.e. its identifier which is an integer in the range [1,255]); the default SAND message set occupies the range [0,21], the range [22,127] is reserved for future ISO additions and finally the range [128,255] is free for private use.

For efficiency reasons, the SAND specification opted for a **common message envelope** which allows for SAND message aggregation. In other words, DANEs and DASH clients do not require to send each SAND message individually, but can instead opt to aggregate them into one big message. In terms of XML, this is represented as one root tag named `SANDMessage` which contains two common attributes that apply to all SAND messages: `senderId` and `generationTime`, respectively the unique identifier of the sender and the time at which the message was generated (both are optional parameters). The SAND envelope contains at least one SAND message which in turn contains the data associated with it. All SAND messages have two parameters in common: `messageId` and `validityTime`, respectively a unique identifier that helps discriminate between multiple messages of the same type from the same sender and a time indicating at which point the validity of the message may not be guaranteed anymore by the sender (both are optional).

It is important to note that all SAND messages from the default SAND message set were designed to be transferred over an HTTP/1.1 connection either formatted in XML or as header extensions. Although the specification mentions that all SAND messages can be XML formatted, there exist exceptions on this rule, these are further explained in Section 3.3. Appendix B.1 shows the XML formatting in XSD form which the SAND messages utilize; the XSD can be used by implementations to validate SAND messages. Appendix B.2 represents the header extensions format and is presented in ABNF format.

3.2.2 Metric Messages

The DASH specification [30] defines a set of five metric messages that keep track of general QoE measurements during media playback. SAND provides a way to send these metrics to a metric server by encapsulating them in five different types of SAND messages. Table 3.1 provides an overview of all DASH metrics along with their description. The SAND metric messages which encapsulate the DASH metrics are described in Table 3.2.

Metric Key	Description
TcpList	A list of TCP connections describing all outgoing TCP connections and their respective time of opening, time of closing and handshake duration.
HttpList	A list of HTTP request and response transactions. Each entry can be linked to a TcpList entry via a tcpid attribute and describes a single HTTP transaction. Successful transactions will log a trace of throughput measurements. For non-progressive DASH streams, the type of transaction (i.e. MPD, segments, initialization segments, ...) will also be logged.
RepSwitchList	A list of representation switch events (i.e. quality changes). A representation switch occurs whenever the DASH media engine decides to switch to a lower or higher representation of the media being streamed. The timings of these events are logged as the time when the HTTP request for a different representation is sent.
BufferLevel	List of buffer occupancy level measurements during a normal (i.e. 1x speed) media playback. The interval of measurements is up to the implementation.
PlayList	A list of playback periods. A playback period is defined as the time between consecutive user actions (i.e. play, seek) and/or stop events (i.e. pause, media end, stalling or segment retrieve failure). For each entry the representation is logged together with the start, end, total playback time and stop event reason. From these entries a compact flow of information can be reconstructed.

Table 3.1: Overview of metrics collected by DASH clients.

Message Type	SAND Message Name	DASH Metric Key
1	TCPConnections	TcpList
2	HTTPRequestResponseTransactions	HttpList
3	RepresentationSwitchEvents	RepSwitchList
4	BufferLevel	BufferLevel
5	Playlist	PlayList

Table 3.2: Overview of SAND metric messages, which DASH metrics they encapsulate and their respective message type in the default SAND message set.

3.2.3 Status Messages

Even though status messages seem to carry similar information to metric messages, both types of messages are distinguished by their goals. Status messages convey real time operational information of a DASH client to one or multiple DANEs whereas metric messages provide a summary of the status of ongoing streaming sessions. Table 3.3 provides an overview of the default SAND message set status messages.

Message Type	SAND Message Name
6	AnticipatedRequests

Allows for a DASH client to signal interest in DASH media segments. The client does this by providing a list of segment uniform resource identifiers (URIs) and a respective optional byterange to signal interest in only a region of the resource. Clients with a sense of when they will request the respective resource can also supply a target time. The goal is to inform a DANE in advance of which segments with their corresponding quality level the DASH client is going to request from the origin server.

7	SharedResourceAllocation

DASH clients that wish to cooperate with other DASH clients can signal intent for sharing network resources. This is done by providing one or multiple DANEs, which are able to act as a resource allocation entity, with the SharedResourceAllocation message. The contents of this message includes a list of operation points which express quality levels that the DASH client would like to achieve. An operation point should at least include a bandwidth budget, which represents the summed bandwidth attributes of all representations being streamed. If, for example, a manifest file included two adaptation sets representing video and audio, each with three representations ranging from low over medium to high quality, a DASH client could, for example, generate three operations points for the SharedResourceAllocation message as follows:

$$OperationPoint1 = low\ quality\ audio\ bandwidth + low\ quality\ video\ bandwidth$$

$$OperationPoint2 = medium\ quality\ audio\ bandwidth + medium\ quality\ video\ bandwidth$$

$$OperationPoint3 = high\ quality\ audio\ bandwidth + high\ quality\ video\ bandwidth$$

The above however, is not by the SAND specification; other combinations are valid too. Together with the mandatory operation points, a DASH client could also signal a preferred resource allocation strategy and a weight for the DANE to use. It is however up to the DANE to decide which strategy to implement and apply. The weight is decided by the user and is directly used in certain strategies. The specification includes five resource allocation strategies:

1. **Basic**: A minimal strategy which divides resources equally among all DASH clients (i.e., $\frac{total\ bandwidth}{total\ clients}$).
2. **Premium privileged**: Utilizes the user-provided weight to provide a better quality service for privileged DASH clients.
3. **Everybody served**: The DANE will try to serve as many DASH clients by allocating them their smallest operation points (with potential upgrade if the total available bandwidth is not completely used).
4. **Weighted sessions**: Allocates DASH clients a bandwidth budget proportional to their set weight.
5. **Pricing**: A special allocation strategy involving the DANE calculating the price based on the amount of connected clients and their bandwidth requests.

8	AcceptedAlternatives

When a DASH client sends out an AcceptedAlternatives message, it signals to all receiving entities that it is willing to accept alternative representations indicated by the provided list of sources. If, for example, a caching DANE on the media delivery paths notices this message and has one of the alternatives cached for the current media request, it can choose to provide the alternative instead of forwarding the request to the origin server. Every alternative has an optional built-in hopcounter (cf. layer 3 routing of IP) called `deliveryScope`, which is decreased by all in-band DANEs when forwarded; DANEs should remove an alternative from the list once it reaches 0.

9	AbsoluteDeadline
A DASH client sends this message along with a segment request indicating the timestamp at which it expects the segment to be fully available at the client side.	

10	MaxRTT
Max round trip time signals the maximum time, in milliseconds, a request can take from the start until the requested resource is available at the DASH client.	

11	NextAlternatives
Contrary to AcceptedAlternatives, NextAlternatives allows a DASH client to signal that it is willing to accept alternative representations for the next request being issued.	

12	ClientCapabilities
This message allows a DASH client to announce to DANEs which types of SAND messages it supports. This is done by providing a list of message types or a message set namespace.	

Table 3.3: Overview of SAND status messages.

3.2.4 Parameters Enhancing Reception (PER) Messages

PER messages are sent by DANEs to DASH clients in order to improve or guide their streaming sessions. Typically these messages occur in the context of one of the following three scenarios:

- **Client assistance**: PER messages are sent along with requests (as described in Section 3.3) and provide auxiliary information which the client may or may not use to enhance its experience.
- **Client enforcement**: In a situation where a DANE decides to act without a DASH client specifically asking for it or when the DANE cannot comply with a user request, a PER message will be sent instead of the requested resource signalling alternative requests the DASH client can make.
- **Error cases**: When a DASH client's request is not valid, the DANE can provide a reason and/or solution in the form of a PER message.

Table 3.4 provides an overview of all PER messages from the default SAND message set.

Message Type	**SAND Message Name**
13	ResourceStatus

By sending this message, a DANE informs a DASH client about the availability of resources (i.e, DASH media segments). This is done by providing a list of resources identified by either a base URL or their respective representation ID together with a status. The status can be one of the following three:

1. **Available**: All resources indicated by this message are available at the DANE who sent the message.
2. **Unavailable**: Resources with this status applied to it are not available at the DANE who sent the message.
3. **Cached**: Resources indicated by this status are expected to be available at the time announced in the manifest .

14 DaneResourceStatus
This PER message is a complementary message to ResourceStatus; it provides the same information but allows to be more explicit in its resource description. Instead of grouping a status for one base URL or representation ID, this PER message allows to explicitly reference individual resources (including an optional byterange indicating that the status only applies to this specific byterange of the resource). In order to compress this PER message, it allows the use of a simplified POSIX regular expression. The DANE has to ability to assign resources to the following status types (at least one status type has to be present containing at least one resource): 1. **Cached**: All resources indicated by this message are cached at the DANE who sent the message. 2. **Unavailable**: Resources with this status applied to it are not available at the DANE who sent the message. 3. **Promised**: Resources indicated by this status are expected to be available at the time announced in the manifest.

15 SharedResourceAssignment
DANEs who act as a resource allocation entity typically sent our this PER message to indicate how much bandwidth DASH clients should use, depending on the resource allocation strategy implemented by the DANE (see SharedResourceAllocation from Table 3.3). This PER message is typically sent out to DASH clients as a response to the SharedResourceAllocation status message. The bandwidth allocated to a DASH client is not static and can increase or decrease over time. When this happens the DANE sends out an updated bandwidth budget to the respective DASH client in the form of a new SharedResourceAssignment SAND message.

16 MPDValidityEndTime
Sending this PER message signals a client that the MPD being used has an expiry date. DANEs can use this mechanism to advise clients to update their manifest information or indicate a faster update time in the case where the media presentation type is set to dynamic with a minimum update period set. The mechanism is mainly used in cases where a DANE notices operational problems (e.g., a DANE notifies clients to immediately retrieve a new MPD because the CDN hosting the DASH media segments has gone offline).

17 Throughput
Throughput allows a DANE to inform DASH clients about a guaranteed throughput in cases where QoS is provided on the link between the DASH client and DANE (e.g. an internal network). DASH clients can use this information to make better assumptions regarding quality adjustments. The DANE specifies a guaranteed throughput for a specific base URL or representation ID together with a confidence percentage that specifies the certainty of the provided throughput estimate.

18 AvailabilityTimeOffset
This PER message allows a DANE to signal an offset, in milliseconds, which a DASH client should apply to the segment availability start times as communicated in the MPD. This way, a DASH client can make better informed choices regarding quality adjustments or avoid buffer underflows or, in the worst, stalling. This message is typically used in a scenario where a DANE encounters unforseen network conditions and tries to adjust client behaviour accordingly during a streaming session. The offset is defined per base URL or representation ID as each resource quality can undergo a different path of processing before becoming available (e.g.,not enough bandwidth is available for a caching DANE to fetch higher quality segments which thus take longer to fetch when compared to lower quality segments).

19 QoSInformation
Allows a DANE to signal QoS information which a DASH client can take into consideration when applying its quality adaptation logic. This SAND message differs from the Throughput SAND message in that it contains more QoS related information; the guaranteed bitrate has the same meaning. The QoS information includes the following (optional) pieces of information: 1. Guaranteed bitrate between a DANE and DASH client 2. Maximum bitrate between a DANE and DASH client 3. Delay in milliseconds denoting the maximum packet delay with a 98 percent confidence 4. Packet loss parameter with the value set to $10^{\frac{-packet\ loss}{10}}$

20 DeliveredAlternative
Sent in response to an AcceptedAlternatives status message (see Section 3.2.3); if the DANE sends an alternative representation, it shall signal the DASH client by also sending a DeliveredAlternative.

21 DaneCapabilities
DaneCapabilities follows the same structure as the ClientCapabilities status message (see Table 3.3). By sending this PER message, the DANE indicates to receiving DASH clients which SAND messages it supports.

Table 3.4: Overview of SAND PER messages.

3.2.5 Parameters Enhancing Delivery (PED) Messages

At the time of writing, the SAND specification does not include any PED messages for the default SAND message set. Default message set additions however are possible in the future via the reserved ISO message types. Private implementations can also opt to implement their own PED messages with the reserved private message types.

3.3 Message Exchange

So far we have discussed the different types of entities involved in a SAND communication process, what types of messages exist and how they can be formatted. The SAND specification has additionally defined which transport protocols to use to carry the aforementioned messages. These are officially referred to as SAND communication channels.

3.3.1 SAND Communication Channels

The SAND specification defines the following three channels:

- `urn:mpeg:dasg:sand:channel:http:2016`: SAND messages are transferred as XML formatted messages through HTTP request and response bodies
- `urn:mpeg:dasg:sand:channel:header:2016`: SAND messages are transferred as HTTP header extensions for HTTP request and response objects concerning media segments
- `urn:mpeg:dasg:sand:channel:websocket:2016`: SAND messages are transferred as XML formatted messages through a websocket in data frame messages set to the `text` type

Of the aforementioned channels, SAND entities shall support both the HTTP and header channels; the websocket channel is an optional transport protocol. The SAND specification allows for implementations to utilize additional transport protocols.

When utilizing the HTTP and header communication channels, the following setup is mandatory:

- Metric message shall be sent using an HTTP POST request; headers are allowed for small metrics (it is up to the implementation to decide what small metrics entail)
- Status messages shall utilize headers unless a DANE is directly being contacted via URL/IP in which case HTTP POST shall be used
- PER messages are always retrieved using HTTP GET
- No specific rules are mentioned for PED messages

As mentioned in Section 3.2.1, some exceptions to this setup apply. The AcceptedAlternatives and AbsoluteDeadline status messages shall always be sent using HTTP header extensions. The DeliveredAlternative PER message shall not be retrieved as an XML formatted message via HTTP GET, but rather be delivered as an HTTP header extension together with the alternative representation.

3.3.2 Channel Signaling

There exist two official ways through which a DANE may make its presence known; it is however not limited to these methods. The first way is through adding a SAND channel to the manifest file with the `sand:Channel` element which includes the following three attributes:

- `id`: Specifies an identifier for the communication channel
- `schemeIdUri`: Specifies the communication channel protocol by urn (see Section 3.3.1)
- `endpoint`: Specifies the endpoint URI for the given communication channel protocol

Of previous three attributes, only the `schemeIdUri` and `endpoint` attributes are mandatory.

The second way for a DANE to make its presence known is by inserting the header extension `MPEG-DASH-SANDChannel`, providing the same information as if signalled via the MPD. This is typically used by DANEs on the media delivery path who wish to partake in the communication.

3.4 Discussion

3.4.1 Security Considerations

When utilizing the default recommendations for message exchange (i.e., HTTP and header extensions), no security considerations were taken into account when designing the specification. A DANE injecting SAND headers into an established connection could be seen as a positive form of a **man-in-the-middle** (MITM) attack [32]. This also means that entities are capable of changing messages as they see fit in an unregulated manner. SAND messages do not carry any executable content, they do however reference resources which could contain executable content. It is up to the implementation to actively take this into account when fetching resources.

It is possible to protect the confidentiality of a SAND message by encrypting it using the XML encryption syntax and processing rules [33]. Most traffic nowadays moves over **encrypted HTTP** (i.e., HTTPS) which inherently means that messages and media segments are encrypted in an end-to-end fashion. When HTTPS is used for SAND message delivery instead of HTTP, intermediate DANEs on the media delivery path will not be able to intercept the messages and will therefore not be able to enhance the streaming session.

Another security consideration to take into account when deploying SAND in a browser scenario utilizing the default recommendations, is **Cross-Origin Resource Sharing** (CORS). Modern web browsers nowadays have a built-in mechanism that restricts certain types of HTTP communication to other domains than the origin; browsers divide these requests in two categories: simple requests and preflighted requests. Simple requests are HTTP requests which are typically used during browsing (e.g., HTTP GET or HTTP POST), it is important that these requests do not specify any other headers apart from those set by the browser (e.g., connection, user-agent, ...). In the event a cross-site request is made which does not fall under the previous rules, the browser will check if the request can be made by first doing an HTTP OPTIONS request in order to determine whether the actual request to be made is safe to send.

The domain being contacted has to allow cross-site communication by specifying which domains, other than itself, are allowed. This is done via the `Access-Control-Allow-Origin` header. If the cross-site domain also sends back non-default headers (such as the SAND headers), it has to specify exactly which of those the client browser may access via the `Access-Control-Expose-Headers` header. Without this header, a DASH client implementation in the browser would not be able to retrieve the SAND header extension values [34].

3.4.2 Discrepancies

While working with SAND, a number of discrepancies from the specification were brought to light. Table 3.5 provides an overview of these, together with their respective page numbers as found in [29]. An attempt was made to contact the people involved with SAND standardization by submitting an issue on their SAND conformance test repositories on Github[2]; as of the time of writing, no response has been provided yet.

[2]https://github.com/Dash-Industry-Forum/SAND-Test-Vectors/issues/1

Page Number(s)	Description
8, 26, 33	**Common envelope → SandMessage → messageID**: XSD enforces the use of messageId by setting a `use="required"` flag whilst the documentation specifies a cardinality of `0..1`. The documentation on messageId in HTTP header extensions specifies that messageId is not required.
9, 10, 37, 38, 39	**Metric naming**: All status and PER messages use the same name as their subsection title in their respective XML or HTTP header extension format. All metrics use the DASH specification titles as their naming which does not follow the SAND XSD schema. Following metrics fall under previous description: `TCPConnections`, `HTTPRequestResponseTransactions`, `RepresentationSwitchEvents`.
11, 34	**AnticipatedRequests → targetTime**: Documentation requires targetTime to be of the date-time type whilst the XSD expects an unsignedLong.
13, 34	**AcceptedAlternatives → alternative**: Documentation mentions a cardinality of exactly one but the XSD mentions `1...N`
15, 34, 35	**NextAlternatives → alternative**: Documentation mentions a cardinality of exactly one but the XSD mentions `1...N`
16, 35	**ResourceStatus → resourceInfo**: Documentation specifies a resourceInfo element for both baseURL and representation ID cases whilst the XSD differentiates between the two by specifying following elements: `resourceURLInfo` and `resourceRepresentationInfo`.
17	**ResourceStatus → resourceInfo → repId**: `repId` represents the representation ID for which the caching status applies. The MPEG-DASH specification mentions the following about representation IDs: "specifies an identifier for this Representation. The identifier shall be unique within a Period unless the Representation is functionally identically to another Representation in the same Period." [30]. We can never know exactly which representation we target with the provided `repID`. If, for example, we were to send a ResourceStatus SAND message including the representation ID "1" and the manifest were to contain two periods, each containing a representation with the ID set to "1", our client would never be able to match the SAND message representation ID to the correct representation due to ambiguity.
24, 37	**DaneCapabilities → supportedMessage (→ messageType)**: Documentation specifies the element/attribute messageType whilst the XSD says nothing about this. The similar ClientCapabilities message with an almost equal message structure does define the messageType attribute in the XSD. *This discrepancy was addressed in the first amendment of the specification [35].*
26, 27	**Ignoring integer-list ABNF for ClientCapabilities example**: Header extension format `integer-list` does not match the example for `SAND-CientCapabilities`
29	**Sand channel endpoint attribute**: Required in XSD but not in specification description

Table 3.5: SAND specification discrepancies overview.

Chapter 4

POC Out of band smart resource allocation entity

In order to provide answers to the research questions asked in Section 1.1, we will create a proof of concept (POC) utilizing SAND and a MPEG-DASH player. SAND - as explained in Chapter 3 - is a recent specification for which DASH-IF interoperability guidelines only recently came out (21 december 2018) after our implementation work had already begun (see Section 6.2). For this POC we will therefore have to figure out the best way to deploy SAND (see Section 4.2).

As our use case in this POC, we will utilize the **shared network scenario** as depicted in Figure 4.1. A shared network scenario typically takes place in a household, apartment, airport, etc... where multiple clients concurrently consume DASH content. Within this shared network we will deploy a DANE that will be addressed as the **support DANE** from now on. This support DANE will provide two functions for all DASH clients communicating with it:

1. **Bandwidth guidance** for MPEG-DASH players within the shared network
2. **Smart caching** (i.e., prefetching DASH segments) of the same MPEG-DASH content being consumed by multiple MPEG-DASH players within the shared network

The support DANE will thus, take upon itself the roles of a **resource allocation entity** as well as a **smart caching entity**. The resource allocation role will divide the available network bandwidth according to one of the SAND allocation strategies (see the SharedResourceAllocation message type in Section 3.2.3) among the connected clients; the amount of bandwidth allocated to a client will henceforth be referred to as the **allocated bandwidth budget** expressed in bits per second. The smart caching role will make the DANE a web caching entity specifically aimed at DASH content being consumed within the shared network.

It is important to note that our POC will be based around guidance and not enforcement. The goal of this POC (see Section 1.1) is to see if SAND alone suffices to firstly reach fair play within a shared network and secondly to see if overall bandwidth reduction can be achieved in the scenario where multiple DASH players are consuming the same content over a shared network connection. As such, we assume that DASH clients within our shared network scenario are willing to play fairly by abiding to the recommendations the DANE sends them.

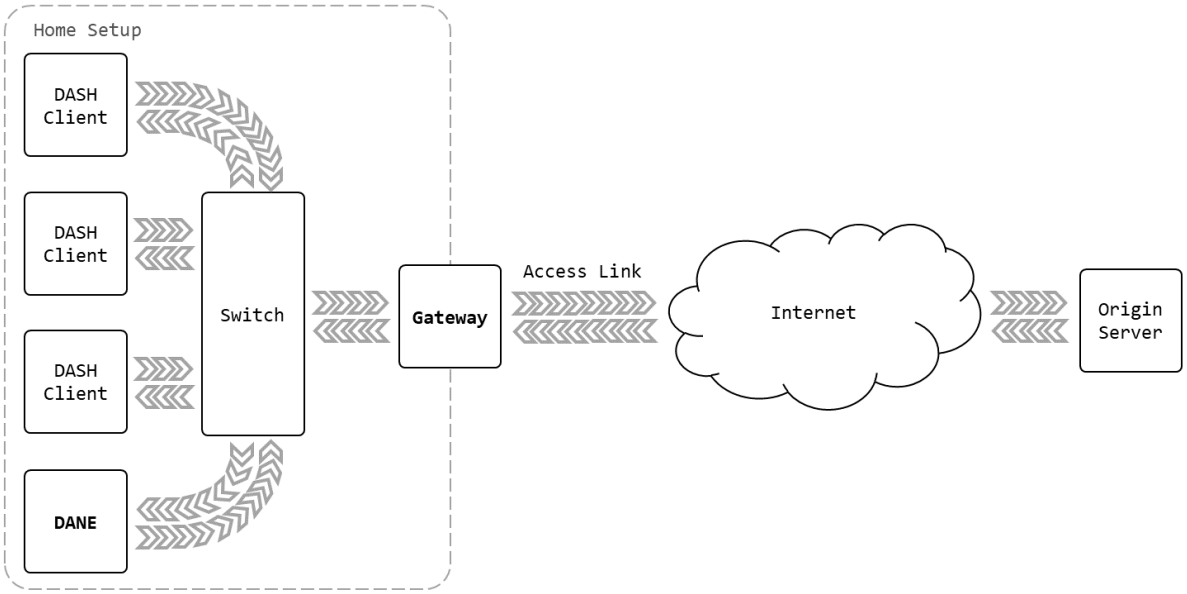

Figure 4.1: A network overview depicting a typical home setup with multiple DASH clients consuming DASH content from a single origin server on the Internet. The DANE depicted in this setup performs the roles of resource allocation entity as well as smart caching entity. The access link between the gateway and the Internet is typically referred to as the last mile.

4.1 Implementation Considerations

For our POC to work, we have to consider how to implement SAND in a DASH client and how to create a DANE. Sections 4.1.1 and 4.1.2 cover these topics.

4.1.1 DASH Client

Our POC requires DASH clients with the ability to process DASH streams as well as communicate with DANEs via SAND messages. Since MPEG-DASH is a standardized protocol as well as one of the most popular HAS protocols in use on the web (i.e., website media players playing DASH streams), we will focus on web based MPEG-DASH solutions that utilize the **Media Source Extensions** (MSE) offered by modern browsers[1] [36] to generate media streams using JavaScript. This gave us three popular choices[2]:

1. Implement **our own MPEG-DASH player** with SAND support (for our purposes this could be a headless client that only fetches DASH content but does not play it back)
2. Use the DASH-IF reference player: **Dash.js** V3.0.0[3]
3. Use the Google adaptive streaming player: **Shaka Player**[4]

Of these three options, option one would be the least desirable but the best fallback option in case options two or three do no work out. The reason for this is that adaptive streaming players such as Shaka Player and Dash.js have enjoyed many years of development and have long left their childhood stages of development and are considered to be production ready players at the time of writing.

Both the Dash.js and Shaka Player are considered to be good foundations upon which we can extend and implement SAND. Both players are implemented in JavaScript and offer a limited plugin system for extending their behaviour without the need for altering core components. As explained in Section 3.1.1,

[1] At the time of writing, MSE is supported by all browsers except iOS Safari [https://caniuse.com/#feat=mediasource]
[2] Other MPEG-DASH players such as the Bitmovin player exist[https://bitmovin.com/video-player/], but only the mentioned MPEG-DASH players are open source.
[3] https://github.com/Dash-Industry-Forum/dash.js
[4] https://github.com/google/shaka-player/

SAND however requires a more fundamental change in the way the client model works. Luckily, Dash.js and Shaka Player are released under the open source licenses of BSD-3[5] and Apache 2.0[6] respectively; in other words, we are allowed to change both projects in such a way that fits our needs. Table 4.1 sums up the biggest differences and most important features supported by both players.

	Dash.js	**Shaka Player**
Supported protocols	MPEG-DASH, MSS	MPEG-DASH, HLS
ABR logic	throughput[7], drop frames[8], bitrate switch history[9], BOLA[10] and abandonment[11]	throughput
Segment type support	index- and range-based	index-based
Multicodec support[12]	Yes	No
Fault-tolerant[13]	Yes	Yes
Preload functionality[14]	Yes	No
Livestream support	Yes	Yes[15]
Encrypted content support	Yes	Yes
Offline storage[16]	No	Yes

Table 4.1: Comparison of the biggest features and supported elements between Dash.js and Shaka player.

When considering Dash.js versus Shaka Player for our POC, we will go with Dash.js. The reason Dash.js is preferred above Shaka Player is because of its extensive and complex adaptive bitrate (ABR) system. The ABR logic summed up in Table 4.1 - also known as ABR rules in Dash.js terminology - is vastly more complex than the one implemented by Shaka Player. Instead of operating on only one type of ABR logic, Dash.js implements a system of primary and secondary ABR rules. Only one primary rule always executes, this is typically the rule which decides if a quality switch is needed in a normal scenario; these rules include the throughput rule and BOLA rule or a hybrid model which combines the strengths of both previous rules. The secondary rules only kick in in specific scenarios such as too many frame drops due to insufficient CPU processing power, too many quality switches due to detected oscillation or the imminent treat of stalling. Together, the primary and secondary ABR rules form a complex system which can easily handle many different situations (e.g., switching from a cabled network connection to a wireless network connection). The internal system of Dash.js also allows for extra rules to be added whilst keeping the other ABR rules intact and operational. In other words, the system is flexible enough to allow transparent changes to ABR logic without the need of modifying core components in such a way that could introduce incorrect playback behaviour.

[5] https://opensource.org/licenses/BSD-3-Clause

[6] https://opensource.org/licenses/Apache-2.0

[7] Throughput logic utilizes the exponentially weighted moving average (EWMA) of past segment throughputs to decide whether to change streaming quality.

[8] Drop frames utilizes dropped video frames as an indicator that the client does not have the required processing power to decode all video frames in real time, thus preferring lower quality representations. This rule kicks into action whenever more than 15% of frames are dropped.

[9] Bitrate switch history keeps track of all bitrate switches during playback. This ABR logic will prevent rapid oscillations if many bitrate switches are detected.

[10] Buffer Occupancy based Lyapunov Algorithm (BOLA) [37]: as buffer occupancy drops, BOLA will prefer lower bitrate segments; as buffer occupancy grows, it will prefer higher bitrate segments.

[11] Abandonment logic keeps track of segments being downloaded, whenever a stall is imminent, the rule will kick in and switch to a safer bitrate as well as cancel the original request.

[12] DASH manifests can contain multiple *periods* each with multiple *adaptationsets*. These *adaptationsets* however do not require to use the same codecs as the previous period. Dash.js allows *periods* to use different codecs whilst Shaka Player does not.

[13] Manifests can contain wrong information about timings, references, etc ...

[14] Ability to start processing DASH content before having access to the HTML video DOM element.

[15] Only index based livestreams are supported.

[16] Ability to store content on the host for later use (e.g., Youtube offline videos or Netflix offline videos).

4.1.2 DANE

At the time of writing, no public DANE implementations exist yet. Our POC requires a DANE that gets **contacted in an out-of-band fashion** by DASH clients within the same network. It should also **adhere to the SAND specification** concerning exchanged messages and the communication protocol. This means that HTTP GET, HTTP POST and HTTP headers will be used for communication between the DASH clients and the DANE. We will need to create a system which utilizes HTTP as its transport protocol and insert a mechanism to keep track of clients their state. Many such mechanisms exist nowadays, both simple and complex ones; the simplest method of keeping track is by letting clients identify themselves through the use of a unique identifier (ID), which could for example be inserted as an HTTP header. These are simple requirements that many modern programming languages can solve. As such, for our POC, we considered the following two languages: **JavaScript** (Node.js engine[17]) and **Python**.

Both options are nowadays very popular programming languages for purposes such as network applications. Both JavaScript and Python come with package managers - NPM[18] and PIP[19] respectively - that include an abundance of ready-to-use libraries. Out-of-the-box Node.js supports web oriented applications; Python on the other hand has many libraries for manipulating web traffic as well as mature web frameworks such as Flask[20] and Django[21]. Under the hood, JavaScript is a single threaded process using an asynchronous system to introduce concurrency. Python on the other hand offers programmers access to both threads as well as asynchronous behaviour. In terms of performance, both are suitable for handling incoming web traffic. When considering JavaScript versus Python, there is no real reason to pick one above the other for the purposes of this POC. We will however take the path of Python as we had previous experience with web applications run through Python web frameworks such as Flask.

4.2 Communication Workflow

The SAND specification provides a well defined set of messages which we are able to leverage in our POC without the need of defining new messages. However, the specification leaves its implementation open in terms of communication flow. In other words, there are no guidelines defining how SAND should be initiated, which messages should be exchanged and when, or at what frequency entities should communicate with each other.

As such, we shall define guidelines for our POC that are specifically made for SAND message exchange in shared network environments. It is important to note that other environments, such as, for example, cloud-based DANEs, might require a different approach in terms of message exchange frequency and authentication. We came up with two communication workflows, **bandwidth guidance** and **smart caching**, detailed in respectively Sections 4.2.3 and 4.2.4. The handshake workflow explained in Section 4.2.2 is used by both previously mentioned workflows to initiate a SAND session.

4.2.1 Polling-based Client Identification

Client identification plays an important role when considering the **resource allocation** and **smart caching** roles that the DANE will take upon itself. In order to allocate bandwidth to DASH clients or prefetch DASH content being consumed by multiple DASH players, the DANE has to have some notion of which DASH clients are in the shared network as well as which of them are active. If our setup were to use Websockets as indicated as a possible message channel in the specification (see Section 3.3.1), the DANE would be able to identify and discriminate between active and inactive DASH clients purely on the WebSocket connection from a DASH player to the DANE. An open connection would indicate an active DASH client willing to communicate. We however opted to not use WebSockets because we wanted to follow the specification which mandates minimal support for HTTP and HTTP headers; adding WebSocket support would mean an added layer of complexity which does not add anything overall to the project.

Our POC however, will solely be built upon the HTTP and header channels. Because these two channels

[17] https://nodejs.org/
[18] https://www.npmjs.com/
[19] https://packaging.python.org/guides/tool-recommendations/
[20] https://github.com/pallets/flask
[21] http://www.djangoproject.com/

utilize the HTTP transport protocol, they inherit a stateless design. Therefore we will introduce client identification by inserting an extra header in the communication process: `Sand-Client-ID: <unique identifier>`. It is worth noting that this unique identifier be unique within the shared network of DASH clients, which is why the following Universally Unique Identifier (UUID) versions are recommended for our POC: version 3, version 4 or version 5 [38].

With this ID alone the DANE is able to identify clients, but it can not yet discriminate between active and inactive clients. A solution for this problem is implementing a polling system; DASH clients will poll the DANE every two seconds. During this polling the DANE has the opportunity to exchange awaiting SAND messages; a poll by a client also indicates that it is in an active state. We define active state as follows: fetching media segments and not paused (i.e., playing DASH content and not having the remaining playback content in buffer). The support DANE shall prune all DASH clients who have not polled it in the last thirty seconds. By doing so, we achieve a best effort mechanism for active client identification.

4.2.2 Handshake

A handshake is defined as the first interaction between a DASH client and the support DANE. During this process the DASH client notifies the DANE of its presence by making a HTTP POST request containing its capabilities in the form of a *ClientCapabilities* SAND message. The DANE will proceed by adding the client to its collection of active clients and will respond with a HTTP 200 that also contains the `MPEG-DASH-SAND` header, a header defined by the SAND specification to let a DASH client know that the DANE has messages queued for it. The message in question is a *DaneCapabilities* SAND message to notify the DASH client of the DANE's abilities, which is retrieved by performing an HTTP GET request to the URL provided in the respective header.

Figure 4.2 depicts a sequence diagram for a handshake between a DASH client and the support DANE. The provided HTTP request and response bodies contain simplified examples to make it more clear what is being transmitted. In our POC, all support DANEs will be known at the beginning of a DASH stream. If, however, in a future work a support DANE joins the communication process at a later stage, a handshake will be initiated at that point. It is worth noting that in a realistic scenario, the SAND message exchange and DASH content fetching - depicted as the two loop groups in Figure 4.2 - can occur in an interwoven fashion. It is also worth mentioning, that the polling loop does not fully follow the SAND specification. Because our implementation implements polling and message announcements on the same URI, the DANE cannot differentiate between a client polling and a client fetching a SAND messages announced via the `MPEG-DASH-SAND` header. As such, in case an awaiting message is available during a polling event, the implementation will avoid the extra round-trip-time required by fetching a message announced by previous header. A fix to this would be to use different URIs.

Table 4.2 provides an overview of all SAND messages that are required to be supported by both the DASH client and the DANE in order for a handshake to succeed.

Message Type	**SAND Message Name**
12	ClientCapabilities
21	DaneCapabilities

Table 4.2: Overview of the required SAND messages for a handshake to work.

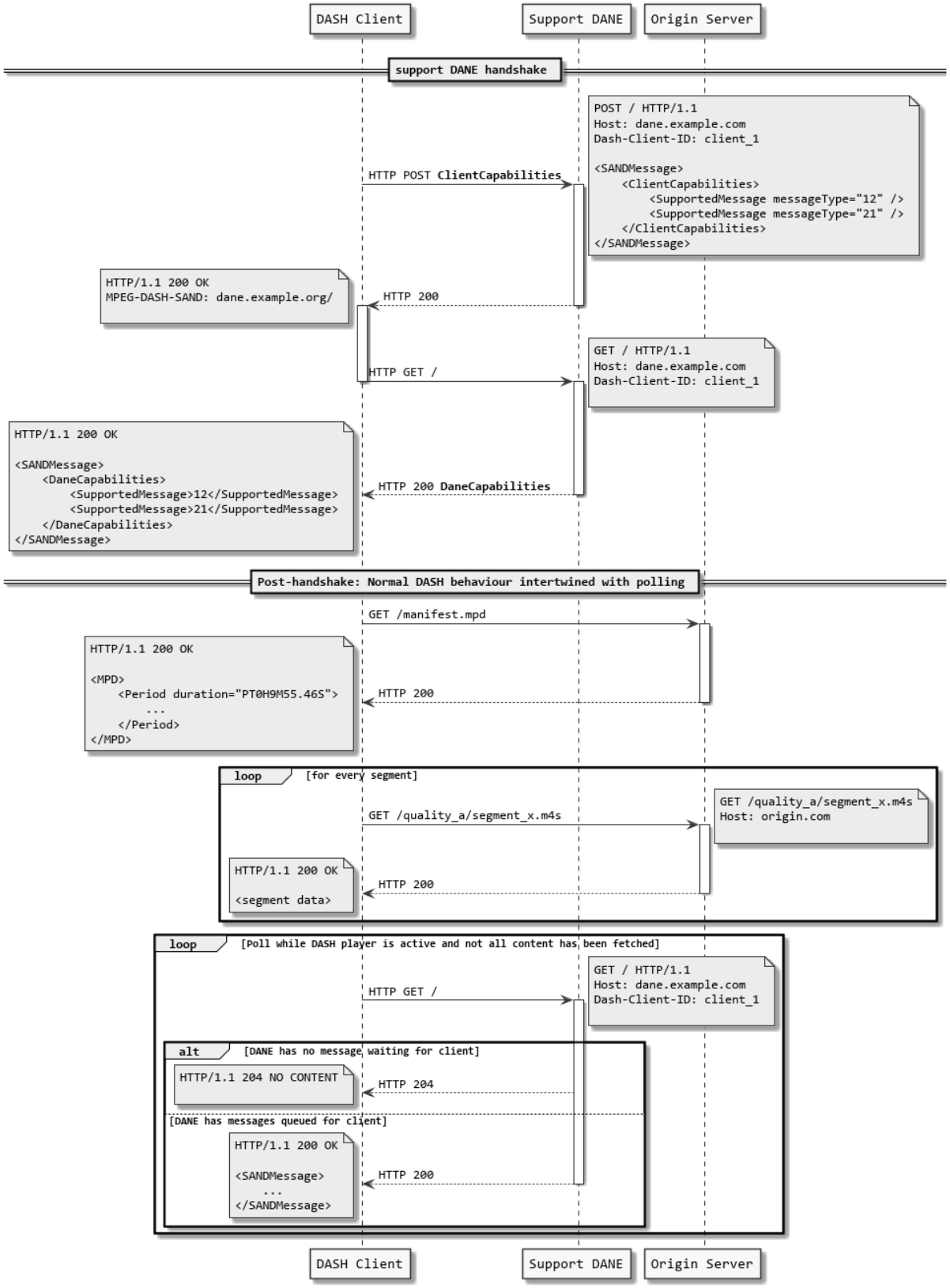
DASH Client
Support DANE
Origin Server
support DANE handshake
POST / HTTP/1.1
Host: dane.example.com
Dash-Client-ID: client_1
<SANDMessage>
<ClientCapabilities>
<SupportedMessage messageType="12" />
<SupportedMessage messageType="21" />
</ClientCapabilities>
</SANDMessage>
HTTP POST ClientCapabilities
HTTP/1.1 200 OK
MPEG-DASH-SAND: dane.example.org/
HTTP 200
GET / HTTP/1.1
Host: dane.example.com
Dash-Client-ID: client_1
HTTP GET /
HTTP/1.1 200 OK
<SANDMessage>
<DaneCapabilities>
<SupportedMessage>12</SupportedMessage>
<SupportedMessage>21</SupportedMessage>
</DaneCapabilities>
</SANDMessage>
HTTP 200 DaneCapabilities
Post-handshake: Normal DASH behaviour intertwined with polling
GET /manifest.mpd
HTTP/1.1 200 OK
<MPD>
<Period duration="PT0H9M55.46S">
...
</Period>
</MPD>
HTTP 200
loop
[for every segment]
GET /quality_a/segment_x.m4s
Host: origin.com
GET /quality_a/segment_x.m4s
HTTP/1.1 200 OK
<segment data>
HTTP 200
loop
[Poll while DASH player is active and not all content has been fetched]
GET / HTTP/1.1
Host: dane.example.com
Dash-Client-ID: client_1
HTTP GET /
alt
[DANE has no message waiting for client]
HTTP/1.1 204 NO CONTENT
HTTP 204
[DANE has messages queued for client]
HTTP/1.1 200 OK
<SANDMessage>
...
</SANDMessage>
HTTP 200
DASH Client
Support DANE
Origin Server

Figure 4.2: Network sequence diagram depicting the flow of messages during and after a handshake between a DASH client and the DANE. Actual HTTP headers and bodies contains simplified examples.

4.2.3 Bandwidth consumption guidance

Bandwidth guidance is provided by the DANE which keeps track of all DASH clients within the shared network. The sequence diagram in Figure 4.3 depicts how this process works. A DASH client shows intent to receive bandwidth guidance by sending the *SharedResourceAllocation* SAND message to the DANE. This message shall contain the operation points at which the DASH client wishes to operate. The DANE will respond with a *SharedResourceAssignment* SAND message containing the maximum allowed bandwidth budget for that client. With the polling system explained in Section 4.2.1, a DASH client notifies the DANE it is still in an active state and thus requires to be part of its bandwidth guidance program. Subsequently, all DASH clients participating in the bandwidth guidance program will be notified of changes via the same polling mechanism through which the DANE can send an updated bandwidth budget.

Table 4.3 provides an overview of all required SAND messages that are to be implemented by the DANE and all DASH clients participating in the bandwidth guidance program.

Message Type	**SAND Message Name**
7	SharedResourceAllocation
12	ClientCapabilities
15	SharedResourceAssignment
21	DaneCapabilities

Table 4.3: Overview of the required SAND messages for bandwidth guidance to work.

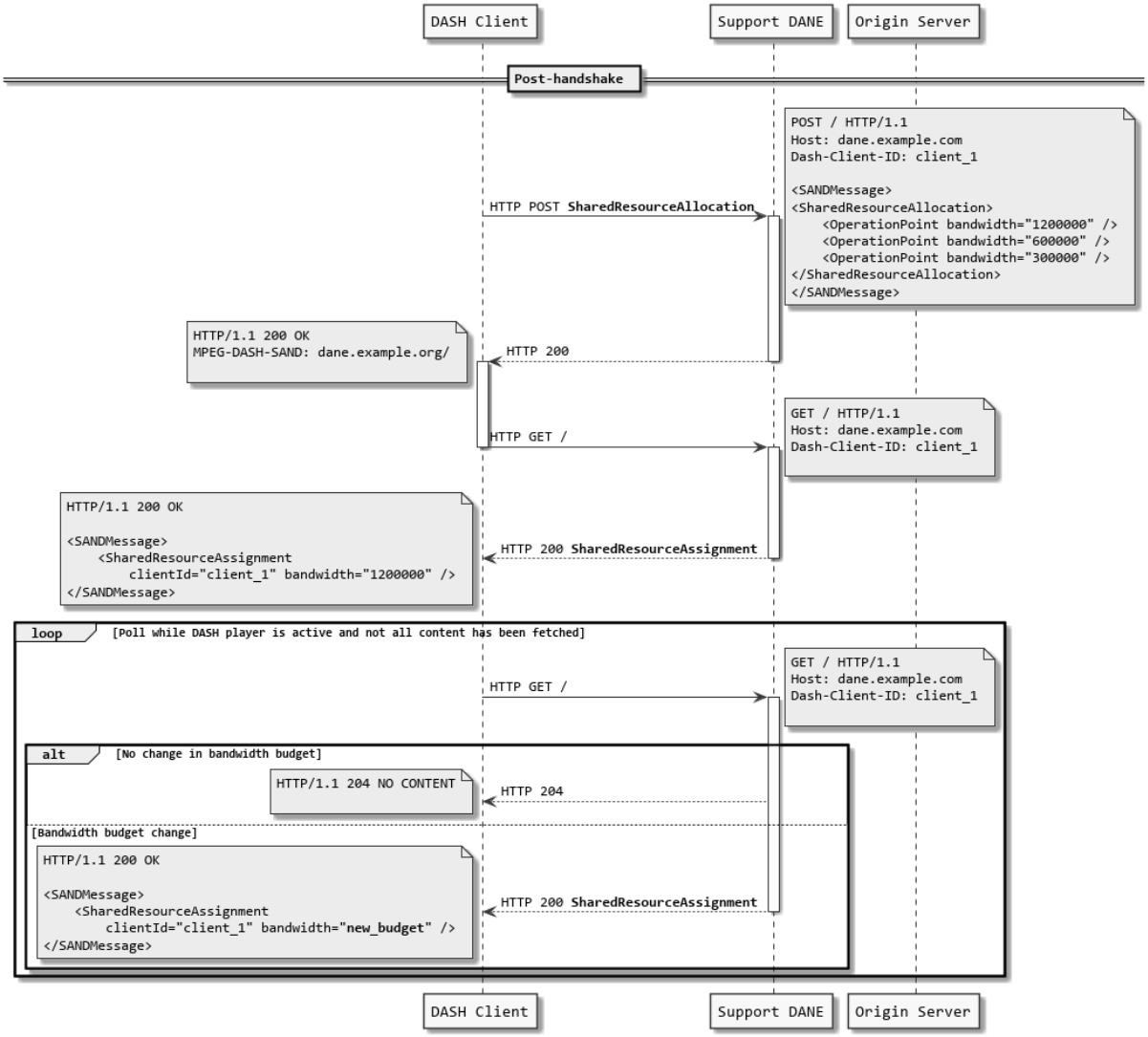

Figure 4.3: Network sequence diagram depicting the flow of messages during the bandwidth guidance of a DASH client by the DANE. Actual HTTP headers and bodies contain simplified examples.

4.2.4 Smart Caching

Smart caching builds further upon the bandwidth guidance role. A DASH client provides one extra piece of information in the *SharedResourceAllocation* SAND message, namely the MPD URL for the DASH content being consumed. Due to this, the DANE is capable of knowing what a particular DASH client is consuming. In the event multiple (i.e., at least two) DASH clients consume the same DASH content, the DANE will see this as a trigger to start prefetching and locally caching the highest quality segments for the respective DASH content.

During playback, DASH clients notify the DANE of anticipated segments using the *AnticipatedRequests* SAND message. In our POC, DASH clients shall notify the support DANE of the next 20 anticipated segment requests. DASH clients shall do this by keeping track of the following parameters: **current index** (i.e., the last downloaded segment available in the buffer) and the **highest sent anticipated request index**. A new *AnticipatedRequests* SAND message shall be sent every time:

$$current\ index > (highest\ anticipated\ request\ index - \frac{total\ anticipated\ segments}{2})$$

The highest anticipated request index shall also be reset every time a seek event happens such that the DASH client can report new anticipated requests and thus allow the DANE to support seeking.

Whenever the DANE detects that one of the DASH clients' anticipated request resources is available, it will answer the respective DASH client with a *DaneResourceStatus* SAND messsage which notifies the DASH client of these cached resources. A DASH client is then able to fetch the resource in the highest available quality from the DANE instead of from the origin server. In the event the DANE has all segments cached, it shall notify the client about this through the *ResourceStatus* SAND message by indicating that the full resource group is cached.

Figure 4.4 depicts previous process as a sequence diagram. It is worth noting that the two loops depicted in this diagram can occur interchangeably. Table 4.4 provides an overview of all messages involved in smart caching; the DANE and the DASH clients are expected to at least support these in order to participate in smart caching.

Message Type	**SAND Message Name**
6	AnticipatedRequests
7	SharedResourceAllocation
12	ClientCapabilities
13	ResourceStatus
14	DaneResourceStatus
15	SharedResourceAssignment
21	DaneCapabilities

Table 4.4: Overview of the required SAND messages for smart caching to work.

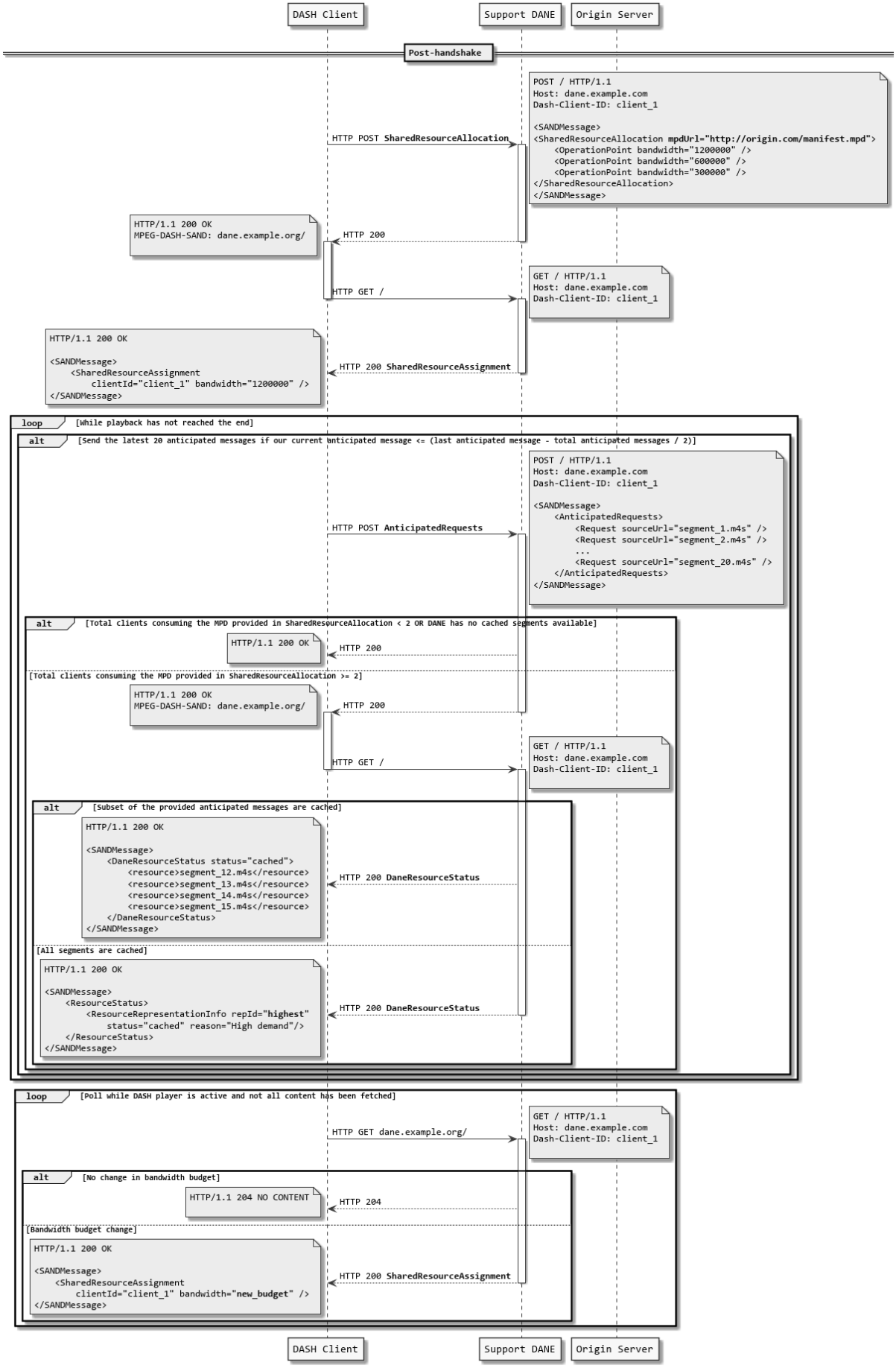

Figure 4.4: Network sequence diagram depicting the flow of messages during smart caching between a DASH client and the DANE. Actual HTTP headers and bodies contains simplified examples.

4.3 Dash.js SAND Implementation

As explained in Section 4.1.1, we shall expand upon Dash.js v3.0.0 to implement the SAND behaviour explained in Section 4.2. Dash.js is a vastly complex system consisting of multiple parts; unfortunately, there is no official documentation which helps in understanding how everything is built. Before we began implementing, we hence had to understand how Dash.js is constructed and where to integrate our POC functionality; this is further described in Sections 4.3.1, 4.3.2 and 4.3.3.

4.3.1 Architecture

When looking at the Dash.js source code, we notice that the code is structured in different systems as follows:

- **Core**
- **DASH**
- **MSS**
- **Streaming**

Core

The core provides global functionality to the Dash.js project. This includes a **factory subsystem** and an **event bus**. The factory subsystem is an implementation of the creational design pattern called **factory method** [39]; Dash.js uses the object-oriented paradigm for its implementation utilizing JavaScript functions as classes. The factory subsystem allows for creation of internal objects and subsequently introduces implementation hiding; classes are required to define an object that represents the function calls that may be used outside of the class (see pseudocode example in Listing 4.1).

```
function class() {
    function a(){}
    function b(){}
    function c(){}

    config = {
        a:a,
        b:b
    }

    return config;
}
factory.addNewClass("class_name", class);

classObject = factory.createClassInstance("class_name");
classObject.a(); // Works
classObject.b(); // Works
classObject.c(); // Undefined
```

Listing 4.1: Pseudocode example of the Dash.js factory subsystem.

The event bus allows an internal object to trigger an event coupled with data. By doing so, other systems active within Dash.js receive the opportunity to also act on the event or change the data sent with the event trigger (see pseudocode example in Listing 4.2). This implementation allows for a lot of flexibility, but also makes it harder to track the exact behaviour of Dash.js.

```
function classA() {
    function init() {
        console.log("Hello ");
        eventBus.trigger("example_trigger", {text:"world"});
    }

    setup();
    config = {
        init : init
    }
    return config;
}
```

```
13
14 function classB() {
15     function setup() {
16         eventBus.on("example_trigger", handler);
17     }
18
19     function handler(data) {
20         console.log(data.text);
21     }
22
23     setup();
24     return {};
25 }
26 factory.addNewClass("classA", classA);
27 factory.addNewClass("classB", classB);
28
29 classAObject = factory.createClassInstance("classA");
30 classBObject = factory.createClassInstance("classB"); // is required to be initiated so the eventBus has notion
       of the handler existing
31 classAObject.init(); // Will display "Hello world" in the console on two seperate lines
```

Listing 4.2: Pseudocode example of how to use the Dash.js factory subsystem.

DASH and MSS

The DASH and MSS systems provide all the required functionality to support both respective adaptive streaming formats. For the purposes of our POC, the MSS subsystem will be ignored as we are only interested in MPEG-DASH content and SAND.

The most important parts of the DASH system are the `DashAdapter`, `DashHandler` and `DashMetrics`. The `DashHandler` provides necessary information about a manifest, e.g., what segment index lies at timestamp X, what is the duration of the manifest, what qualities exist, how many adaptations exist for stream type video, audio or subtitles, etc... Internally it makes use of different parsing systems which utilize the XML data provided in the manifest file. The `DashAdapter` on the other hand is directly used by the Streaming system to provide the necessary information to make playback possible; it will provide the streaming system with VOs[22] containing the information needed to make HTTP requests to the right resources. Finally, `DashMetrics` is used to keep track of the metrics defined in the MPEG-DASH specification [30].

Streaming

The streaming system contains the main bulk of logic needed for adaptive streaming to take place. It contains the following subsystems: ABR logic, a protection system for DRM protected streams, a HTTP loader (based on XHR[23] or Fetch[24] depending on browser support) for downloading DASH/MSS content, a thumbnail system and a text system for subtitles. The streaming system hooks directly into the HTML `<video>` element provided to Dash.js during setup and couples internal logic to the possible DOM events, such as play, pause, seek, rate change, error, etc... The streaming subsystem also keeps track of the buffer which is overseen by the ABR logic and provides scheduling so that it can be filled. Scheduling consists out of a two step procedure which happens for each type of media (i.e., video, audio and subtitles):

1. ABR logic is activated which in turn fires off the primary and secondary ABR rules (see Section 4.1.1) which decide what adaptation quality is needed for future segments (only happens for video and audio media types)

2. The scheduler requests the appropriate VO objects from the DASH or MSS system, which it then passes along to the HTTP loader subsystem which will handle the actual content fetching

This scheduling steps are repeated every 500msec with the second scheduling step taking place if and only if the previous segment was successfully retrieved, was aborted by the ABR logic system or the

[22] Variable Objects (VOs) are Javascript objects which consist of keys and values, the keys are represented by a variable which is used to access its value. It can be compared to a C++ Map[http://www.cplusplus.com/reference/map/map/] or a Python Dictionary[https://docs.python.org/3.7/tutorial/datastructures.html#dictionaries].

[23] https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest

[24] https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API/Using$_F etch$

connection was aborted due to error. Dash.js, in other words, implements a back-to-back scheduler for its segments.

4.3.2 Execution flow

Dash.js' event based system, unfortunately, makes it difficult to trace execution logic. As such, we tried to examine what happens at initialization based on the methods called and events triggered. Listing 4.3 shows a simplified execution path from initialization until scheduling happens.

```
Dash.js Initialization
----------------------
player gets created with dashjs.Mediaplayer.create()
player is initialized with DOM element & MPD URL -> Mediaplayer.initialize
    -> calls attachView
        -> sets the DOM element and checks for Protection, Metric, MSS subsystems
        -> calls initializePlayback
            -> creates playback controllers: mediaController, streamController, playbackController, abrController, textController
        -> streamController load of MPD is called IF ''source'' is set, which it is not by this point
    -> calls attachSource
        -> sets the "source" object parameter in MediaPlayer (i.e., the MPD URL)
        -> calls initializePlayback
            -> streamController load called since ''source'' is set (steamController is Singleton)
                -> calls manifestLoader.load
                    -> performs httpLoader.load on a MPD TextRequest VO (contains request parameters for a MPD)
                        -> on success parses contents
                        -> passes parsed contents to xlinkController.resolveManifestOnLoad
                            -> performs XML xlinking
                            -> triggers XLINK_READY with ''manifest'' as payload

XLINK_READY event handlers
--------------------------
ManifestLoader.onXlinkReady
    -> triggers INTERNAL_MANIFEST_LOADED with ''manifest'' as payload

INTERNAL_MANIFEST_LOADED event handlers
---------------------------------------
ManifestUpdater.onManifestLoaded
    -> calls update with parsed ''manifest'' payload
        -> manifestModel.setValue called with manifest as param
            -> Sets internal ''manifest'' value with given manifest
            -> triggers MANIFEST_LOADED event with ''data'' containing the manifest
        -> checks and sets manifest refresh timers
        -> triggers MANIFEST_UPDATED event with ''manifest'' payload

MANIFEST_LOADED event handlers
------------------------------
No internal system utilizes this (i.e., the event appears to be defined for debugging or is no longer in use)

MANIFEST_UPDATED event handlers
-------------------------------
DVBErrorsTranslator.onManifestUpdate
    -> sets internal MPD value with payload

MetricsCollectionController.update
    -> Reloads/sets metrics related controllers
    -> Triggers METRICS_INITIALISATION_COMPLETE with no payload

StreamController.onManifestUpdated
    -> calls DashAdapter.updatePeriods with manifest as param
        -> sets voPeriods (manual parsing of available period data)
        -> gathers streamInfo
        -> gathers mediaInfo
        -> Tries to detect SegmentTimeLine functionality
        -> initializes BaseURLController with manifest
            -> calls baseURLTreeModel.update with the manifest
                -> calls getBaseURLCollectionsFromManifest with manifest as param
                    -> sets internally used baseURLs correctly
            -> calls baseURLSelector.chooseSelector
                -> selector is set to basicSelector
        -> TimeSyncController initialized with UTCTimings (only filled in if the MPD is live or availabilityStartTime is set)
```

```
            -> calls attemptSync (with the provided UTCTimings)
                -> tries to check what sort of synchronization feature can be used; will fail in simple cases and fallback to device time (i.e., when using VOD)
                -> calls onComplete
                    -> checks for Date HTTP header
                    -> completeTimeSyncSequence called
                        -> triggers TIME_SYNCHRONIZATION_COMPLETED

TIME_SYNCHRONIZATION_COMPLETED event handlers
-----------------------------------------------
TimelineConverter.onTimeSyncComplete
    -> Performs a sanity check on time related synchronization features (only needed for live MPDs)

StreamController.onTimeSyncCompleted
    -> checks for protection controller and triggers PROTECTION_CREATED (i.e., DRM-related)
    -> calls composeStreams
        -> gathers streamInfo DashAdaper.getStreamsInfo()
            -> gets all periods as a VO
        -> loops over all StreamInfo (= MPEG-DASH \textit{period} object) objects and creates "streams" or updates them
            -> calls Stream. initialize
                -> Sets up the stream with given streamInfo (to be used later)
            -> dashMetrics.addManifestUpdateStreamInfo called for current streamInfo
        -> searches for start time if no stream is active (which is the case at initialization )
            -> calls switchStream
                -> triggers PERIOD_SWITCH_STARTED with "fromStreamInfo" and "toStreamInfo" objects
                -> if an old stream exists; calls oldStream.deactivate
                -> calls playbackController. initialize with the active stream info
                    -> connects DOM event listeners to internal system
                -> call preloadstream
                    -> calls activateStream
                        -> calls stream.ativate
                            -> calls initializeMedia
                                -> calls checkIfInitializationCompleted
                                    -> triggers STREAM_INITIALIZED
        -> triggers STREAMS_COMPOSED

STREAM_INITIALIZED event handlers
-----------------------------------
ScheduleController.onStreamInitialized
    -> starts scheduling logic
```

Listing 4.3: High level overview of a Dash.js execution flow after being initialized for VOD DASH content.

4.3.3 SAND Architecture and Hooks

Based on the findings from Sections 4.3.1 and 4.3.2, we came up with an architecture for our SAND functionality as well as the places within the Dash.js software architecture where we should hook in our SAND logic.

Architecture

For our POC we shall introduce a new Dash.js system next to the already existing core, DASH, MSS and streaming systems; this system shall be named **sand** and is depicted in Figure 4.5 and further explained in Table 4.5.

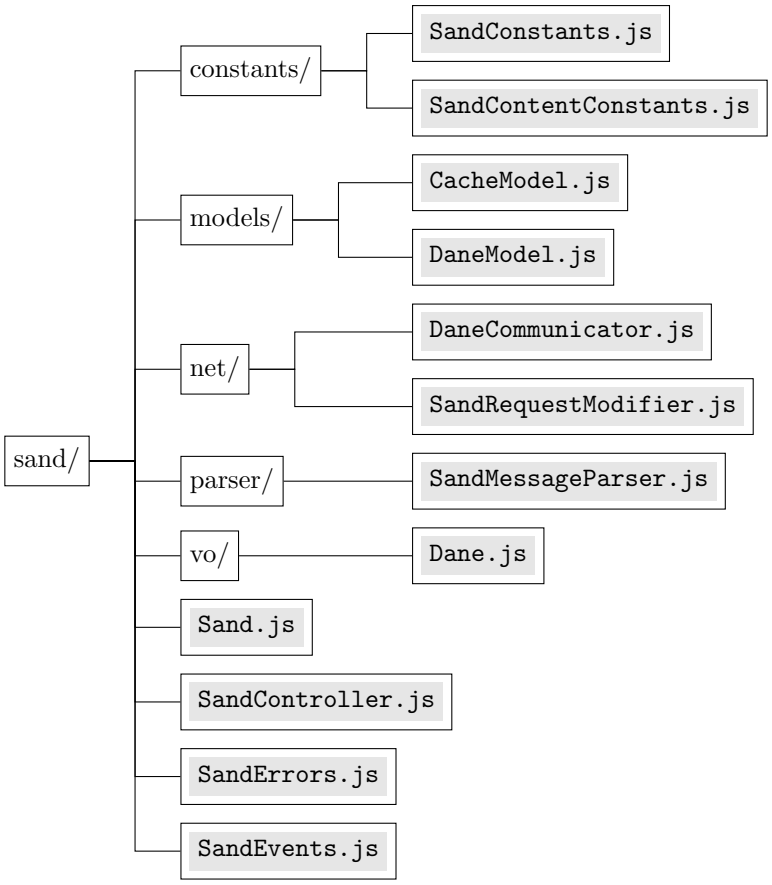

Figure 4.5: Sand system archictecture

File	Descriptions
`SandConstants.js`	Defines constants for internal workings pertaining to SAND.
`SandContentConstants.js`	Defines constants related to SAND messages. This file provides a class that maps the default SAND message set names to their message type. It provides the means to access both through `getMessageNameById()` and `getIdByMessageName()`.
`CacheModel.js`	Provides the necessary functionality about cached resources to internal systems. Whenever a system requires the knowledge about whether or not a certain segment is cached at the DANE side, calling `isSegmentCached(id)` will provide the answer in the form of a boolean.
`DaneModel.js`	Keeps track of an individual DANE known to the DASH client. `DaneModel` objects keep track of awaiting messages signaled via the `MPEG-DASH-SAND` header of the respective DANE and process these in a batched manner such that it does not stress the scheduling logic of the streaming system. In the event incoming messages get parsed correctly by the `SandMessageParser`, an internal event by the name `SAND_MESSAGE_LOADED` will be triggered with the parsed contents as payload.
`DaneCommunicator.js`	No communication logic is available in Dash.js due to the fact that MPEG-DASH only requires fetching (i.e., HTTP GET requests) from the origin server. The `DaneCommunicator` class provides an interface to send SAND messages (i.e., HTTP POST requests, see Section 3.3.1) to a DANE. If the DANE in question also returns the `MPEG-DASH-SAND` header, the corresponding `DaneModel` shall be notified through the `SAND_MESSAGE_AVAILABLE` event.
`SandRequestModifier.js`	The HTTPLoader subsystem from the streaming system provides methods for fetching different types of content (i.e., text based or byte based). It also provides an extension mechanism called the `RequestModifier` which is accessible as a plugin for modifying request headers and requests URLs (e.g., a Dash.js utilizer who wishes to make Dash.js compatible with their own system which requires a specific header). Instead of hooking into the plugin mechanism, we directly supply the HTTPLoader with our own request modifier which inserts the required `Sand-Client-ID` header as explained in Section 4.2.1. This class is used by `DaneModel` objects when processing awaiting messages.
`SandMessageParser.js`	A simplified parsing engine based on the parsers used by the streaming system. Incoming XML SAND messages are converted into JSON.
`Dane.js`	A VO that keeps track of DANE specific data such as its role (in-band-dane, support dane, ...), the supported capabilities the DANE reported during the handshake and whether or not we have encountered issues in the past contacting the DANE such that we can drop the DANE in the event it goes offline.

`Sand.js`	The glue component between Dash.js systems and the POC's sand system. The `Sand` class is directly inserted into the `dashjs` global variable, similar to how the core, DASH, MSS and streaming systems work. It provides the `addOutOfBandDane(url, role)` functionality to the MediaPlayer instance created when initiating Dash.js on a `<video>` element; Dash.js utilizers can signal their known support DANEs this way, e.g., `player.addOutOfBandDane("http://dane.com/", "supportDaneRole");`.
`SandController.js`	The `SandController` class provides functionality to process incoming SAND messages. It does this by hooking into the `SAND_MESSAGE_LOADED` event, triggered by a `DaneModel` object. It will in turn process the message and relay the processed SAND messages to their corresponding handler. Currently the SharedResourceAssignment SAND message will trigger a `SAND_BANDWIDTH_CAP_ISSUED` event with the corresponding bandwidth budget provided by the resource allocation entity. It will also trigger the `SAND_CACHED_SEGMENTS` event for when the DANE notifies the DASH client about cached segements.
`SandErrors.js`	Extension to the Dash.js error subsystem; provides error types and error messages which makes traceback easier.
`SandEvents.js`	Extension to the event bus subsystem; it contains all events that are triggered by the sand system for internal use.

Table 4.5: Overview of all sand system files used during our POC implementation in the Dash.js software architecture.

Hooks

With the acquired knowledge about the execution flow (see Section 4.3.2), we were able to pinpoint the areas of interest in the Dash.js subsystems in order for the above functionality to work in context of our POC.

For bandwidth guidance, we require to notify the DANE about our operation points at which we wish to operate (see example provided with the *SharedResourceAllocation* SAND message description in Table 3.3). The `TIME_SYNCHRONIZATION_COMPLETED` event signals that a fully fetched and parsed manifest is available for internal use, and that its information can be queried through the `DashAdapter` subsystem (see Section 4.3.1). We hook into this event to gather the required operation points. Listing 4.4 provides the pseudocode for the way the operation points are calculated in this POC; we combine all video bitrates together with the highest quality audio bitrate. After the calculations are finished, a SharedResourceAllocation SAND message is constructed which is sent to the support DANE using the `DaneCommunicator` object.

```
function onManifestAcquired() {
    videoRepresentations = dashAdapter.getRepresentationsForType("video");
    audioRepresentations = dashAdapter.getRepresentationsForType("audio");

    maxAudioBitrate = −1;
    for (every audio representation in audioRepresentations)
        maxAudioBitRate = max(maxAudioBitrate, audio.bitrate);

    operationPoints = [];
    for (every video in videoRepresentations)
        operationPoints.push(video.bitrate + maxAudioBitrate);

    return operationPoints;
}
```

Listing 4.4: Pseudocode explaining how operation points are calculated (actual calls to Dash.js subsystems are simplified).

Once the DANE notifies the DASH client of its bandwidth budget, the `SandController` will issue the internal `SAND_BANDWIDTH_CAP_ISSUED` event (as explained in Table 4.5). In order to apply the provided budget, we created a new secondary ABR rule called the `SandRule` which will listen for the `SAND_BANDWIDTH_CAP_ISSUED` event. When `SandRule` catches this event, it becomes active in the scheduling logic (see Section 4.3.1) and does two things:

- It advises the ABR logic not to go higher than the highest quality level possible under the provided bandwidth budget; this stops the DASH player from fetching content at bitrates that exceed its advised budget.
- It paces the segment fetching in a way that keeps the total throughput per second within the allocated bandwidth budget.

The total time to pace is calculated based on the selected bitrate, the provided bandwidth budget and the duration of the segments:

$$pace\ time = \frac{duration}{\frac{bandwidth\ budget}{selected\ bitrate}} * 0.9$$

We subtract a 10% margin from this calculation to compensate for JavaScript timer inaccuracies. A JavaScript timeout is designed to fire after a given delay and as soon as the main thread is free to handle its execution; in practise this results in timers sometimes being off by up to a second. The next segment can be fetched as early as:

$$next\ fetch\ time = current\ request\ time + pace\ time$$

In order to support smart caching, we expand the *SharedResourceAllocation* message being sent for bandwidth guidance with the `mpdUrl` attribute which contains the manifest URL from the origin server. From that point on, the DANE can decide to start caching the DASH content being consumed. In order to know if segments being requested by the streaming system are available at the support DANE, we first notify it of our anticipated segments via the *AnticipatedRequests* SAND message. The data required for this message is gathered in the `DashAdapter` subsystem which next to providing information to the streaming system (see Section 4.3.1) also keeps track of the segment index the playback is currently on. We thus expand the `DashAdapter` subsystem to also calculate the anticipated segments information for the upcoming 20 segments. This is done during scheduling when the streaming system queries the `DashAdapter` for a VO containing information for the next segment. We hook into this process to check if the next segment exceeds the anticipated message treshold, which is defined as follows:

$$anticipated\ request\ threshold = last\ anticipated\ request\ index - \frac{total\ anticipated\ segments}{2})$$

If $next\ request\ segment\ index > anticipated\ request\ threshold$, the `DashAdapter` will gather the anticipated request URLs and send them along with the `NEW_ANTICIPATED_MESSAGES` event and subsequently set a last anticipated request index which is $\frac{total\ anticipated\ segments}{2}$ higher than the previous one. The event is then handled by the `Sand` object which will send the data towards all support DANEs through the use of the `DaneCommunicator`. In case the DANE has cached segments available (see Section 4.2.4), it will notify the DASH client via the *ResourceStatus* or *DaneResourceStatus* SAND message. When this message arrives, it triggers the internal `SAND_CACHED_SEGMENTS` event which passes along the cached segments reported by the DANE. The `CacheModel` object will process this event and keep track of all cached segments available at the DANE. We also expand the scheduling logic to use the `CacheModel` during the two steps of scheduling. When the ABR logic is being processed, a query is sent to the DashAdapter to check if the upcoming segment request is cached (`DashAdapter` does this by querying our `CacheModel`). If this is the case, no bandwidth budget will be applied and the ABR logic will suggest to upgrade to the highest available quality. When the next step of scheduling requests the next segment VO from the `DashAdapter`, it will have replaced the origin server base URL with that of the DANE, thus resulting in the request being made to the DANE instead of to the origin server. It is important that the ABR logic advises the highest available quality, without it, the cache request would not work transparently towards the rest of the system. This could cause playback problems such as decoding errors when the wrong initialization segment is loaded (which is done whenever a quality level switch happens).

4.4 Python DANE implementation

For our POC DANE, we utilized the most recent version of Python (as explained in Section 4.1.2) which during the course of this thesis evolved from Python 3.5.6 to Python 3.7.3. The architecture we used for our design is described in Section 4.4.1 followed by the POC resource allocation and smart caching implementations explained in Sections 4.4.2 and 4.4.3.

4.4.1 Architecture

The design of the DANE was inspired by the inner workings of Dash.js and Flask[25]. The DANE itself - just like Flask - provides minimal bookkeeping and communication such that it can be flexibly imported in a Pythonic way to act as a library, we will call this the **core DANE** from now on. The actual functionality itself comes from the applications using the DANE library to implement behaviour. This results in a quick and easy way to deploy a DANE with custom behaviour which can easily scale to more complex scenarios. The way it allows to implement behaviour, is through an event based system, inspired by Dash.js. Listing 4.5 provides a small example of a DANE which prints out the capabilities of connecting DASH clients.

```
1  from dane import Dane
2  from dane.internals.eventBus import Events
3
4  supported_sand_message_ids = [12, 21]
5  dane_name = "example_dane"
6  app = Dane(supported_sand_message_ids, dane_name)
7
8  @app.event_bus.register_for(Events.SAND_MESSAGE_RECEIVED_STATUS_CLIENTCAPABILITIES)
9  def onClientCapabilities(event_name, client, message):
10     print(f"Client with the name '{client.client_id}' announced it has support for the following SAND messages: {
         message.supportedMessage}")
11     # Example: Client with the name 'client_a' announced it has support for the following SAND messages: [6, 7, 12, 13,
         14, 15, 21]
12
13 app.run()
```

Listing 4.5: Example DANE code showcasing the easy and quick setup procedure.

For the implementation of the core DANE, the following libraries and/or external projects were used:

- **Flask** micro-webframework for our HTTP entrypoints
- **Lxml**[26] XML toolkit for XML creation and parsing
- **SAND header parser**[27] from the DASH-IF conformance test vectors

The core DANE consists of four big internal systems (depicted in Figure 4.6):

- A webserver entrypoint enabled by Flask that allows POST and GET requests for communication with a DASH client (and possible future other DANEs)
- A SAND parsing and validation system
- An event bus that allows for behaviour implementation
- A basic bookkeeping system that keeps track of DASH client session data such as:
  - Session identification (i.e., the client ID set in the `Sand-Client-ID` header, see Section 4.2.2)
  - Client capabilities
  - Last connection time (updated each time the client polls)
  - Awaiting messages (retrieved by a client when polling the DANE)

[25] https://palletsprojects.com/p/flask/
[26] https://lxml.de/
[27] https://github.com/Dash-Industry-Forum/SAND-Test-Vectors

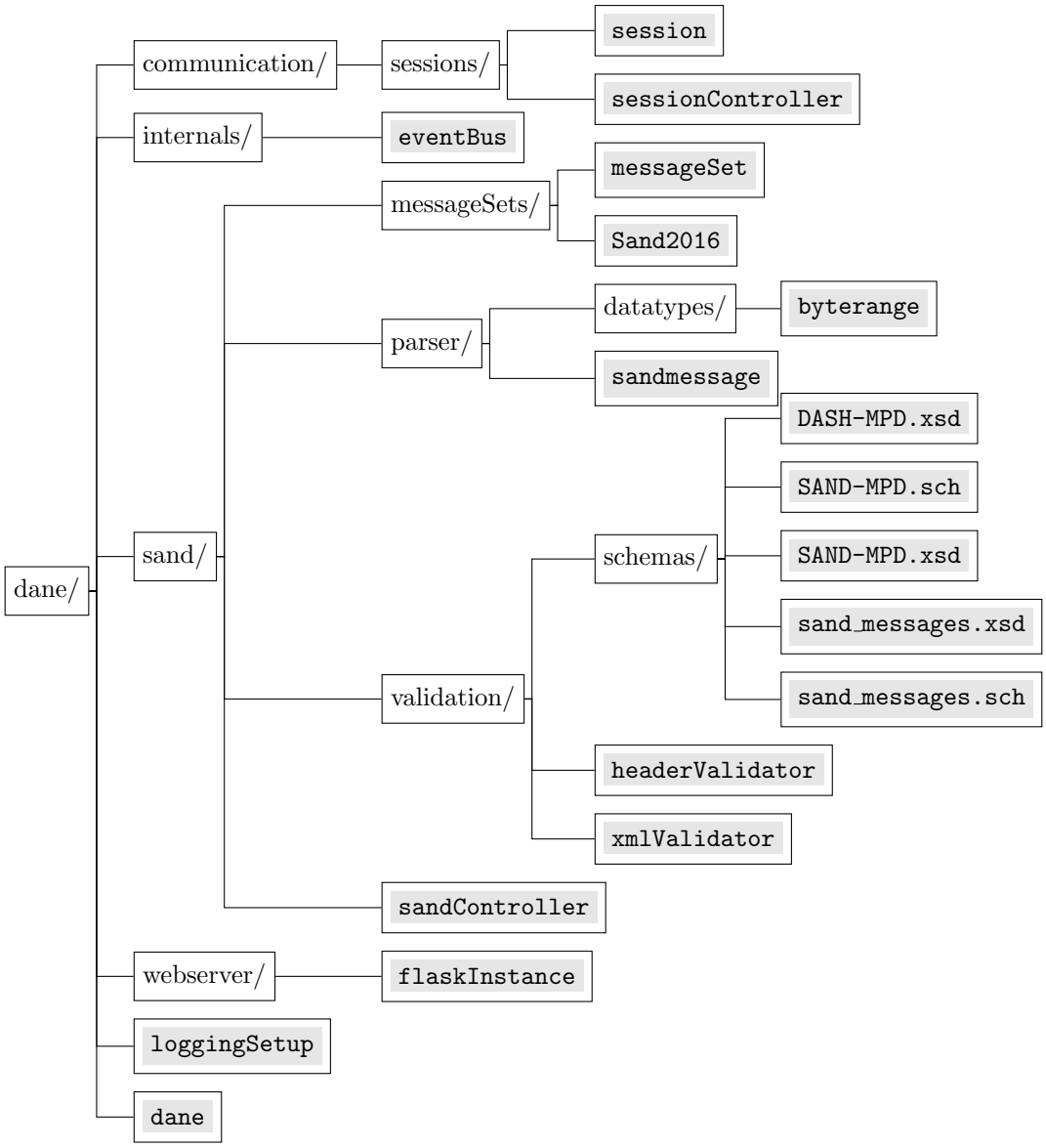

Figure 4.6: Core DANE architecture

The core DANE has two typical execution paths depending on which entrypoint the DASH client uses:

- HTTP POST for sending status messages to the DANE
- HTTP GET for fetching PER messages from the DANE (which can optionally include a SAND status message in the HTTP headers)

When DASH clients make a POST request to the DANE entrypoint (i.e., send a SAND status message), `flaskInstance` catches this and performs a precheck on the `Sand-Client-Id` header. If this header is not set, the execution pathway stops and the DASH client receives a HTTP 400 response indicating a malformed request. If it is set, an internal event by the name `SAND_CLIENT_CONNECTED` is triggered with the header value as payload. This event is then caught by the `sessionController` which compares it to its known clients: if the client already exists, its last connection time is updated; else a new session is created which also triggers the `SAND_CLIENT_ADDED` event with the newly generated `session` object as payload. When the precheck finishes, the Flask instance retrieves the request body and passes this along to the `sandController` instance which will try to validate and parse its contents. Our POC implements both the SAND HTTP and header channels (see Section 3.3.1), as such two data formats are possible. To simplify development, we created an abstraction layer called `sandmessage` which can handle both types of data formats and presents a unified API to the rest of system such that development should not worry about data formats. Every SAND message type defined in the SAND specification has its own

class which is derived from `sandmessage`. When the `sandController` finishes validation and parsing, each parsed SAND message will be accessible through its own respective `sandmessage` object. Once a list of `sandmessage` objects is gathered, the `sandController` will loop through them and trigger an event related to the SAND message type. The trigger contains the client `session` and `sandmessage` objects as payload. The following list enumerates all possible events based on the default SAND message set defined in the SAND specification:

- Metric Messages
  - SAND_MESSAGE_RECEIVED_METRIC_TCPLIST
  - SAND_MESSAGE_RECEIVED_METRIC_HTTPLIST
  - SAND_MESSAGE_RECEIVED_METRIC_REPSWICHLIST
  - SAND_MESSAGE_RECEIVED_METRIC_BUFFERLEVELIST
  - SAND_MESSAGE_RECEIVED_METRIC_PLAYLIST
- Status Messages
  - SAND_MESSAGE_RECEIVED_STATUS_ANTICIPATEDREQUESTS
  - SAND_MESSAGE_RECEIVED_STATUS_SHAREDRESOURCEALLOCATION
  - SAND_MESSAGE_RECEIVED_STATUS_ACCEPTEDALTERNATIVES
  - SAND_MESSAGE_RECEIVED_STATUS_ABSOLUTEDEADLINE
  - SAND_MESSAGE_RECEIVED_STATUS_MAXRTT
  - SAND_MESSAGE_RECEIVED_STATUS_NEXTALTERNATIVES
  - SAND_MESSAGE_RECEIVED_STATUS_CLIENTCAPABILITIES
- PER Messages
  - SAND_MESSAGE_RECEIVED_PER_RESOURCESTATUS
  - SAND_MESSAGE_RECEIVED_PER_DANERESOURCESTATUS
  - SAND_MESSAGE_RECEIVED_PER_SHAREDRESOURCEASSIGNMENT
  - SAND_MESSAGE_RECEIVED_PER_MPDVALIDITYENDTIME
  - SAND_MESSAGE_RECEIVED_PER_THROUGHPUT
  - SAND_MESSAGE_RECEIVED_PER_AVAILABILITYTIMEOFFSET
  - SAND_MESSAGE_RECEIVED_PER_QOSINFORMATION
  - SAND_MESSAGE_RECEIVED_PER_DELIVEREDALTERNATIVE
  - SAND_MESSAGE_RECEIVED_PER_DANECAPABILITIES

Implementations using the core DANE can hook into these events and define their own behaviour around it. If such implementations wish to send SAND messages to the DASH client, they have the opportunity to do so via the session object passed along with the event. The `session.send_message(sand_message_object)` allows for communicating back to the client. Once all hooks finish processing the events triggered by the `SandController`, the `FlaskInstance` will check if the detected DASH client session has awaiting messages. If this is the case, it will set the `MPEG-DASH-SAND` header with as value the URL (i.e., DANE root URL) from where the client can fetch the pending PER messages (i.e., by issuing a HTTP GET). If everything went alright, the DASH client shall receive an HTTP 200 response.

The second path of communication is via the HTTP GET entrypoint defined in `flaskInstance`. Similarly to the HTTP POST entrypoint, the GET entrypoint also checks for the `Sand-Client-Id` header presence. Because DASH clients can also send SAND messages via HTTP headers, this entrypoint will perform a presence check for such messages which will be subsequently passed to the `sandController` to be processed in the same way as explained for the HTTP POST entrypoint. If after all previous processing the DASH client has awaiting messages, the `flaskInstance` will retrieve these as XML formatted messages and pass them along with an HTTP 200 response. In the event that no messages are waiting for the DASH client, the `flaskInstance` will answer with a HTTP 204 response indicating that

no new content is available.

It is worth noting that the `sessionController` also performs a prune of inactive clients which have not connected with the DANE - either via the HTTP POST or HTTP GET entrypoints - within the last 30 seconds. If a prune happens, a `SAND_CLIENT_REMOVED` will be triggered with the respective DASH client `session` object. Implementations can use the `SAND_CLIENT_ADDED` and `SAND_CLIENT_REMOVED` events to keep track of added and removed clients such that they can update their own bookkeeping.

4.4.2 Resource allocation entity

The resource allocation entity is implemented in the `BandwidthController` class which hooks into the following core DANE events: `SAND_CLIENT_ADDED`, `SAND_CLIENT_REMOVED` and `SAND_MESSAGE_RECEIVED_STATUS_SHAREDRESOURCEALLOCATION` (see Section 4.2.3). Upon receiving the *SharedResourceAllocation* SAND message from a client, the `BandwidthController` will store the operation points included in the message and run the **basic** resource allocation strategy provided by the SAND specification (see Section 3.2.3). All clients will then be notified of a new bandwidth budget through a *SharedResourceAssignment* SAND message. It is important to note that our implementation subtracts a 10% margin from the total available bandwidth to compensate for packet header overhead (e.g., on a 10Mbit shared connection, the DANE would divide 9Mbit among its MPEG-DASH clients).

4.4.3 Smart caching entity

The smart caching entity is implemented in the `Cache` class which contains multiple subsystems:

- A cache queue algorithm named `CacheQueue`
- A caching algorithm named `MpdCache`
- A throughput calculation system named `Throughput`

The `Cache` itself hooks into the following core DANE events: `SAND_CLIENT_ADDED`, `SAND_CLIENT_REMOVED`, `SAND_MESSAGE_RECEIVED_STATUS_SHAREDRESOURCEALLOCATION` and `SAND_MESSAGE_RECEIVED_STATUS_ANTICIPATEDREQUESTS` (see Section 4.2.4).

Once at least two clients on the shared network connection are detected to be consuming the same DASH content (signalled via the *mpdUrl* attribute in the *SharedResourceAllocation* SAND message), a new `MpdCache` instance will be spawned for the MPD content in question. This system will start by fetching the manifest file and performing a minimalist parse to retrieve information about the highest quality video segments available (at this point in time, this POC only caches video segments). After gathering the necessary information about the segments, the `MpdCache` will try to allocate a budget for itself from the `BandwidthController` (see Section 4.4.2) such that fair guidance is still guaranteed to all clients. Once a budget is available for the `MpdCache`, it will start by creating a `CacheQueue` object that generates a dynamic priority queue filled with all the segments to be cached. One or multiple cache worker instances can then query this queue for the next segment which they will then fetch. The `Throughput` system is used to calculate the average throughput such that the cache workers pace themselves and do not exceed their allocated bandwidth budget (similar to to the pacing implemented in the Dash.js fetching logic, see Section 4.3.3).

The `CacheQueue` has the ability to work in two modi: **naive** or **furthest playback prioritized**. The **naive** approach will ignore DASH clients' position within the playback timeline and will always generate a queue in which segments are fetched in a backwards order. The reasoning behind this is that all clients progress their timeline from start to finish; if we cache from finish to start, all clients will reach a point during playback in which the DANE will be able to provide them cached segments at the highest quality for the rest of their playback duration. This mode also ensures that no cache misses occur from when a client switches to the DANE cache untill the end of playback. The **furthest playback prioritized** strategy utilizes the expected positions of DASH clients by looking at their anticipated requests. This strategy will find the client which has progressed the most on its playback timeline and prioritize the caching of all segments that follow from its highest anticipated segment index. The reason behind

this is to lower the bandwidth used in the shared connection by letting the DANE prefetch content which the client with the most playback progress will then fetch from the DANE instead of from the origin server. Since we are serving the client with the most progress, clients with an ealier position in the playback timeline, will naturally also reach a playback point for which cached segments are available. Figure 4.7 depicts the two strategies side by side and how they prioritize their queue based on incoming *AnticipatedRequests* SAND messages; the timeline is represented in the segment domain instead of the more conventional temporal domain. It is important to note that this depicted example does not remove segment IDs that are already cached for the sake of clearity; in our POC, cached segments disappear from the cache queue once they are fetched and available locally.

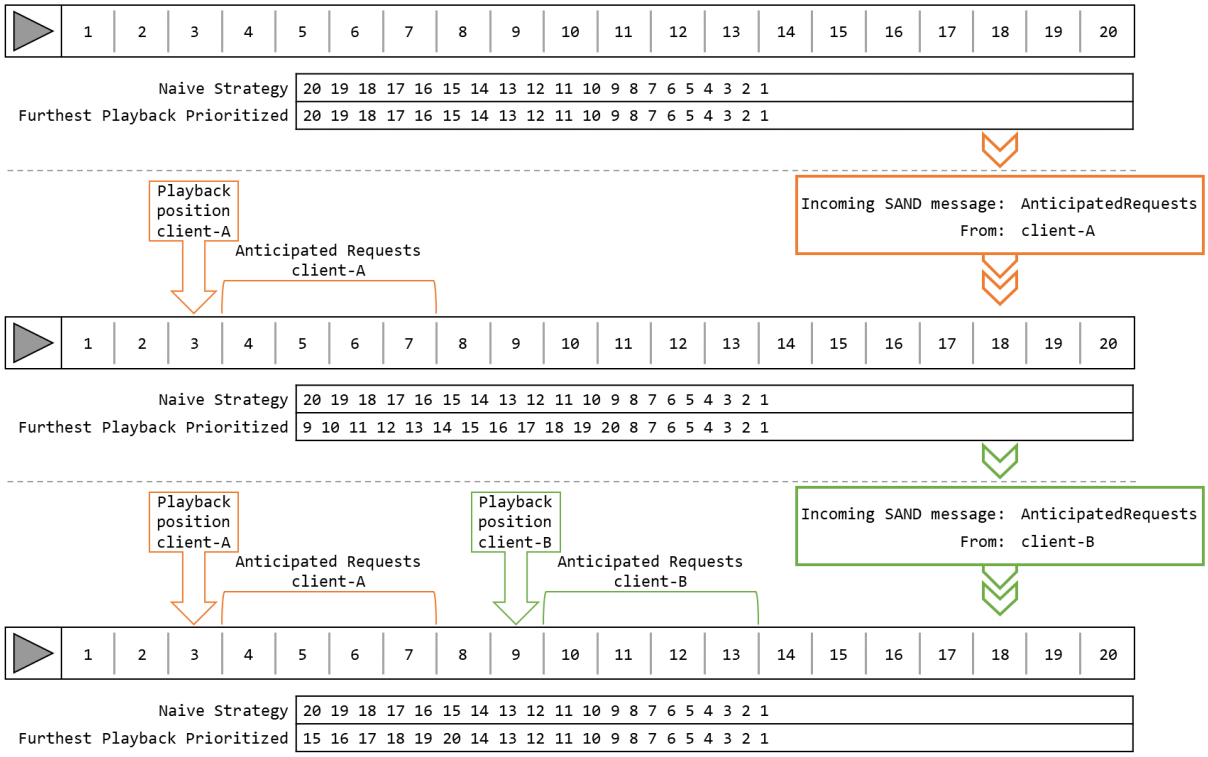

Figure 4.7: Visualisation of the naive versus furthest playback prioritized strategies implemented in the `CacheQueue` class. It depicts a timeline represented in the segment domain as clients send *AnticipatedRequests* SAND messages to the DANE which trigger the cache queue strategies to re-prioritize their cache sequences.

Chapter 5

POC Evaluation

This chapters describes the evaluation of this thesis' implementation. We evaluate our implementation through an experimental setup described in Section 5.1. Sections 5.2 and 5.3 describe the experiments themselves and their results, respectively.

5.1 Testbed

We evaluate our implementation in a locally wired network setup, as depicted in Figure 5.1. Our setup consists of a **client host** (which hosts four clients for each experiment), **DANE** and **origin server** all connected via a **1Gbit switch**. Table 5.1 describes the technical specifications of all devices involved. In order to simulate a realistic setup, the egress traffic of the origin server is limited to 10Mbit per second with the help of a Token Bucket Filter (TBF) [40] queuing discipline for the Linux traffic control **tc** [41] command: `tc qdisc add dev enp0s25 root tbf rate 10mbit burst 1.5kb latency 250ms`. We confirmed this command achieves the imposed 10Mbit limit by performing multiple throughput tests using the Iperf[1] toolkit; we used multiple intervals ranging from 0.5 to 10 seconds (-i flag) as well as multiple parallel connections (-P flag) ranging from 1 to 4. The Iperf tests proved a stable connection that averages around 9.7Mbit per second. The clients themselves are spawned via a small Python script that utilizes the built-in `webbrowser` module, see Listing 5.1. Our testbed did not have to worry about time synchronization since all clients were spawned on the same host machine.

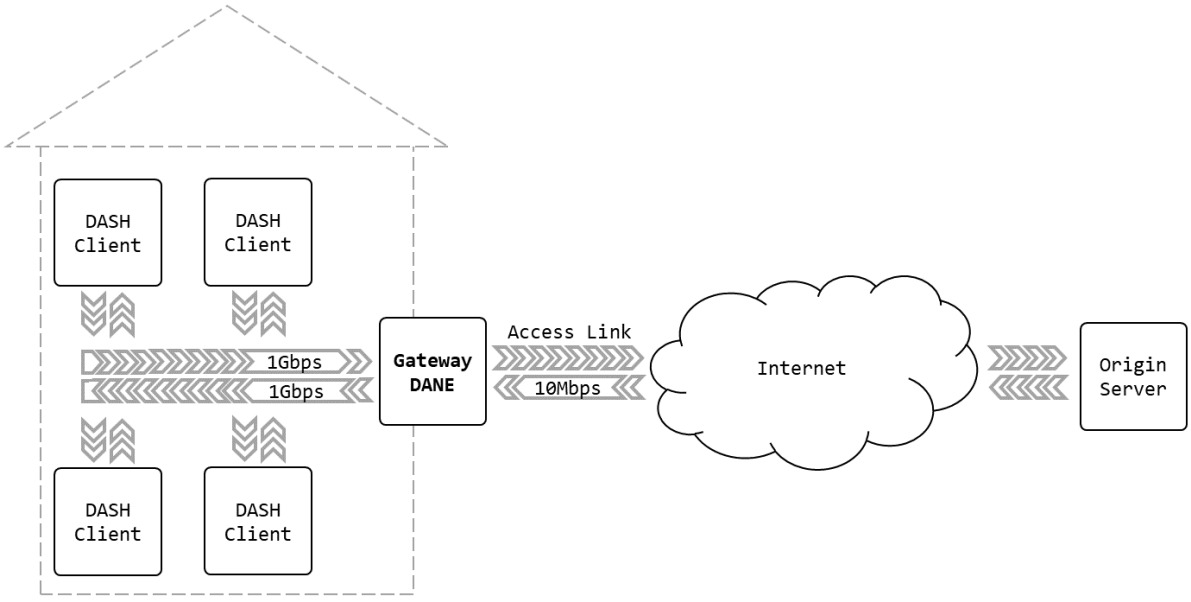

Figure 5.1: Network topolgy diagram of our testbed.

[1] https://iperf.fr/

Entity	Device
Client host	Dell XPS 13 9370 - Intel Core i7-8550U (4cores, 8 threads) @ 4.00GHz, 16GB RAM - Arch Linux - Linux kernel 5.2.0 - 4 parallel clients on Firefox 68.0 using our Dash.js implementation (see Section 4.3)
DANE	Custom PC - Intel Core i5-3570K (4 cores, overclocked) @ 4.20GHz, 8GB RAM - Dane implementation (see Section 4.4)
Origin Server	HP Probook 650 G1 laptop - Intel Core i5-4210M (2 cores, 4 threads) @ 2.60GHz, 4GB RAM - Ubuntu Server 18.04.3 LTS - Linux kernel 4.15.0 - NGINX[2] 1.14.0
Switch	TP-Link TL-SG1005D V5 (unmanaged)

Table 5.1: Overview of testbed device hardware.

```
import webbrowser
import time

interval_experiment = True
evaluation_script_index = 3

interval_values = []
if interval_experiment:
    interval_values = [0, 15, 15, 15]  # Clients start with a wait, specified as the list value, after the previous
else:
    interval_values = [0, 0, 0, 0]  # All clients start at the same time

for i in interval_values:
    time.sleep(i)
    webbrowser.open("http://127.0.0.1/evaluation_t{}.html".format(str(evaluation_script_index)))  # Opens a new tab in the OS default browser
```

Listing 5.1: Python client spawn script

5.2 Experiments

In order to verify our implementation described in Chapter 4, and subsequently tackle our research questions posed in Section 1.1, we will make use of three experimental setups. Every (sub-)experiment is run three times and uses the manifest file provided in Listing 5.2. The video utilized by our experiments comes from the Big Buck Bunny[3] project. The manifest provides a total of 20 representations with bitrates ranging from 45kb per second to 3.9Mbit per second, with a segment size of 4 seconds. Experiments involving SAND will respect the minimum buffer time of 10 seconds as specified by the manifest (see *minBufferTime* attribute on the *MPD* XML tag). The total playback time of the manifest is three minutes (see *mediaPresentationDuration* attribute on the *MPD* XML tag).

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- MPD file Generated with GPAC version 0.5.1-DEV-rev5379 on 2014-09-10T13:30:18Z-->
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT10S" type="static" mediaPresentationDuration="PT0H3M0S" profiles="urn:mpeg:dash:profile:isoff-live:2011">
  <ProgramInformation moreInformationURL="http://gpac.sourceforge.net">
    <Title>dashed/BigBuckBunny_4s_simple_2014_05_09.mpd generated by GPAC</Title>
  </ProgramInformation>
  <Period duration="PT0H3M0S">
    <AdaptationSet segmentAlignment="true" group="1" maxWidth="480" maxHeight="360" maxFrameRate="24" par="4:3">
      <SegmentTemplate timescale="96" media="bunny_$Bandwidth$bps/BigBuckBunny_4s$Number$.m4s" startNumber="1" duration="384" initialization="bunny_$Bandwidth$bps/BigBuckBunny_4s_init.mp4" />
      <Representation id="320x240 45.0kbps" mimeType="video/mp4" codecs="avc1.42c00d" width="320" height="240" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="45226" />
      <Representation id="320x240 89.0kbps" mimeType="video/mp4" codecs="avc1.42c00d" width="320" height="240" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="88783" />
      <Representation id="320x240 129.0kbps" mimeType="video/mp4" codecs="avc1.42c00d" width="320" height="240" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="128503" />
      <Representation id="480x360 177.0kbps" mimeType="video/mp4" codecs="avc1.42c015" width="480" height="360" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="177437" />
```

[3] https://peach.blender.org/

```
14      <Representation id="480x360 218.0kbps" mimeType="video/mp4" codecs="avc1.42c015" width="480" height="
        360" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="217761" />
15      <Representation id="480x360 256.0kbps" mimeType="video/mp4" codecs="avc1.42c015" width="480" height="
        360" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="255865" />
16      <Representation id="480x360 323.0kbps" mimeType="video/mp4" codecs="avc1.42c015" width="480" height="
        360" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="323047" />
17      <Representation id="480x360 378.0kbps" mimeType="video/mp4" codecs="avc1.42c015" width="480" height="
        360" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="378355" />
18      <Representation id="854x480 509.0kbps" mimeType="video/mp4" codecs="avc1.42c01e" width="854" height="
        480" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="509091" />
19      <Representation id="854x480 578.0kbps" mimeType="video/mp4" codecs="avc1.42c01e" width="854" height="
        480" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="577751" />
20      <Representation id="1280x720 783.0kbps" mimeType="video/mp4" codecs="avc1.42c01f" width="1280" height=
        "720" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="782553" />
21      <Representation id="1280x720 1.0Mbps" mimeType="video/mp4" codecs="avc1.42c01f" width="1280" height="
        720" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="1008699" />
22      <Representation id="1280x720 1.2Mbps" mimeType="video/mp4" codecs="avc1.42c01f" width="1280" height="
        720" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="1207152" />
23      <Representation id="1280x720 1.5Mbps" mimeType="video/mp4" codecs="avc1.42c01f" width="1280" height="
        720" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="1473801" />
24      <Representation id="1920x1080 2.1Mbps" mimeType="video/mp4" codecs="avc1.42c032" width="1920" height=
        "1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="2087347" />
25      <Representation id="1920x1080 2.4Mbps" mimeType="video/mp4" codecs="avc1.42c032" width="1920" height=
        "1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="2409742" />
26      <Representation id="1920x1080 2.9Mbps" mimeType="video/mp4" codecs="avc1.42c032" width="1920" height=
        "1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="2944291" />
27      <Representation id="1920x1080 3.3Mbps" mimeType="video/mp4" codecs="avc1.42c032" width="1920" height=
        "1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="3340509" />
28      <Representation id="1920x1080 3.6Mbps" mimeType="video/mp4" codecs="avc1.42c032" width="1920" height=
        "1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="3613836" />
29      <Representation id="1920x1080 3.9Mbps" mimeType="video/mp4" codecs="avc1.42c032" width="1920" height=
        "1080" frameRate="24" sar="1:1" startWithSAP="1" bandwidth="3936261" />
30    </AdaptationSet>
31  </Period>
32 </MPD>
```

Listing 5.2: Experiment manifest.

Experiment 1: No DANE involvement. For comparison to our own implementation, we will start four parallel clients at the same time over a 10Mbit per second shared network connection. This experiment will utilize vanilla Dash.js version 3.0 with all settings set to default and will serve as our baseline. This experiment will track the following metrics: buffer occupancy, selected bitrates, throughput, the amount of quality switches and stalls during playback for each of the four clients.

Experiment 2: Fair bandwidth guidance. This experiment will also start four clients at the same time over a 10Mbit per second shared network connection. The difference with experiment one, is that this setup incorporates our **resource allocation entity** described in Section 4.4.2. All clients will communicate with the DANE via SAND messages, thus allowing the DANE to steer all clients towards a fair bandwidth usage. This experiment will track the same metrics as experiment one..

Experiment 3: Smart caching. Experiment 3 will test our **smart caching entity** described in Section 4.4.3. The setup is similar to the one in experiment 2, with the exception of the DANE also prefetching content for all clients. We will perform tests for our two caching implementations; each caching implementation will in turn be tested with a different client startup rate: one where all four clients are started in parallel and one where clients start at 15 second intervals. This experiment will track the same metrics as experiment one plus cache hits[4] during playback.

5.3 Results

Following sections describe the results we obtained from the experiments described in Section 5.2. We ran each (sub-)experiment three times and we did not notice any deviating, worse or better behaviour in the results obtained from each run. As such, following sections will present the results of a singular run.

[4]We define a cache hit as a segment that the client is able to fetch from the DANE.

5.3.1 No DANE Involvement

Figures 5.2, 5.3, 5.4 and 5.5 depict the buffer occupancy, selected bitrates, individual throughputs and stacked throughputs respectively of all four clients involved in the experiment. As we hypothesized in Chapter 1, the limited throughput over the shared last mile - simulated by the 10Mbit per second egress traffic shaper on the origin server - causes competing behaviour between our four clients. By looking at the stacked area chart in Figure 5.5, we can see that the total throughput never exceeds the 10Mbit per second threshold. By looking at the buffer occupancy and selected bitrates, we can see client four thriving at the expense of clients one through three, mainly from the 100 second mark. The throughput graph shows inequality between all four clients which indicates an unfair bandwidth share. As a result, Dash.js' ABR logic - which is mainly throughput driven in non-stable buffer scenarios - will advice to switch to lower quality representations to compensate for this behaviour (this can be noticed by looking at the 120 second mark in Figure 5.3). In turn, this leads to client four receiving more breathing room and the capability to keep fetching its own content at the highest available quality. The behaviour we notice yields an overall bad QoE for the users, which we can confirm by the amount of quality switches and stalls experienced, represented in Table 5.2.

	Client 1	**Client 2**	**Client 3**	**Client 4**
Detected quality switches	32	37	32	12
Detected stalls	2	1	1	2

Table 5.2: No DANE involvement: Overview of the total number of stalls and quality switches experienced by all four clients.

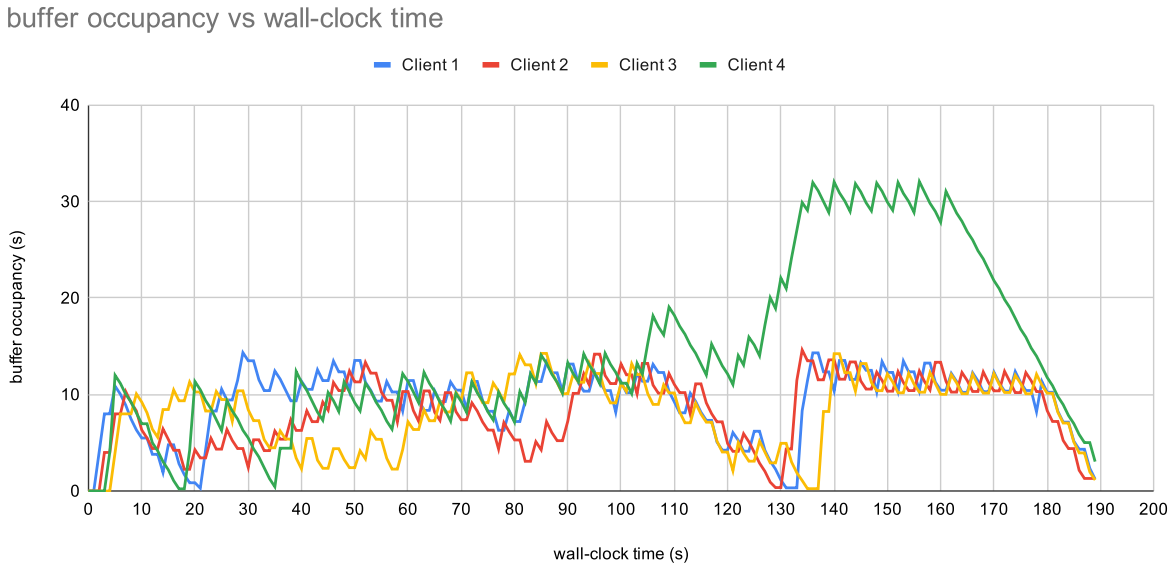

Figure 5.2: No DANE involvement: The buffer occupancy expressed in seconds for each client during playback.

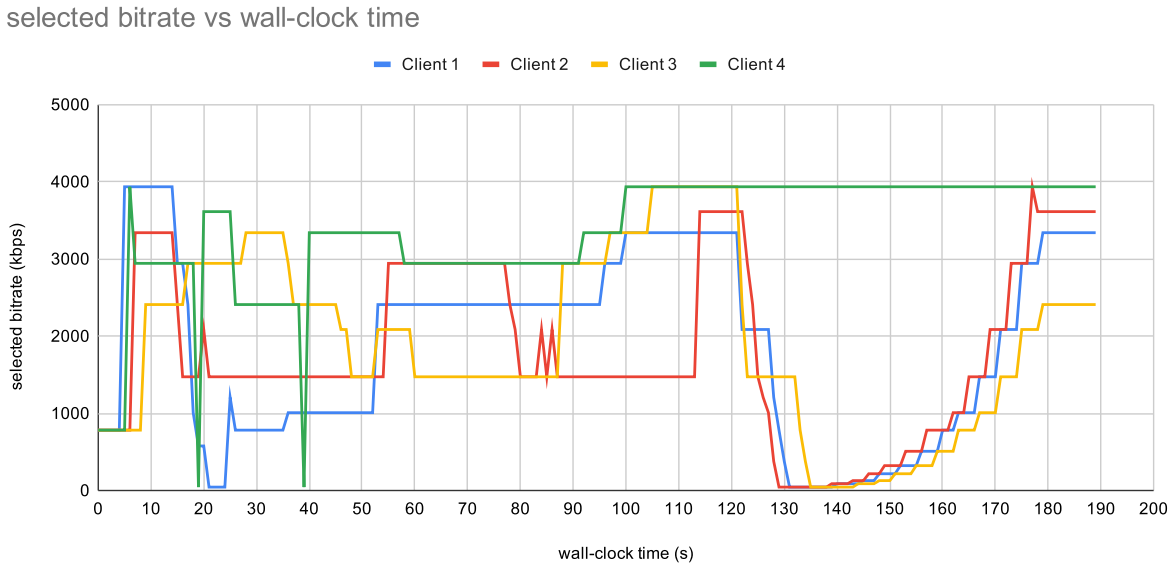

Figure 5.3: No DANE involvement: The selected bitrates expressed in kilobit per second for each client during playback.

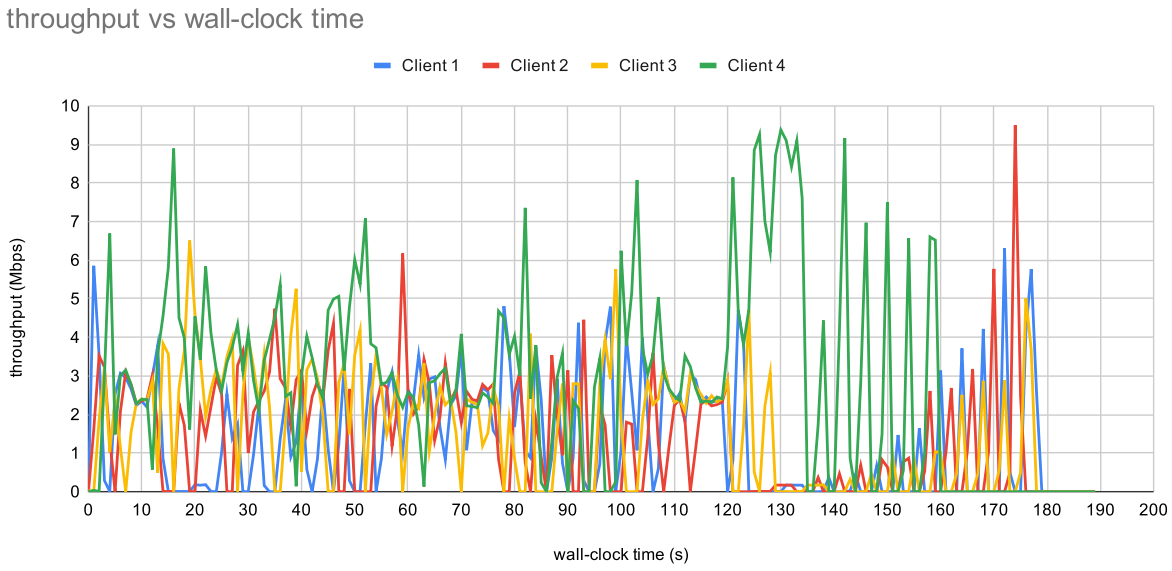

Figure 5.4: No DANE involvement: The throughput expressed in megabit per second for each client during playback, presented as a line chart.

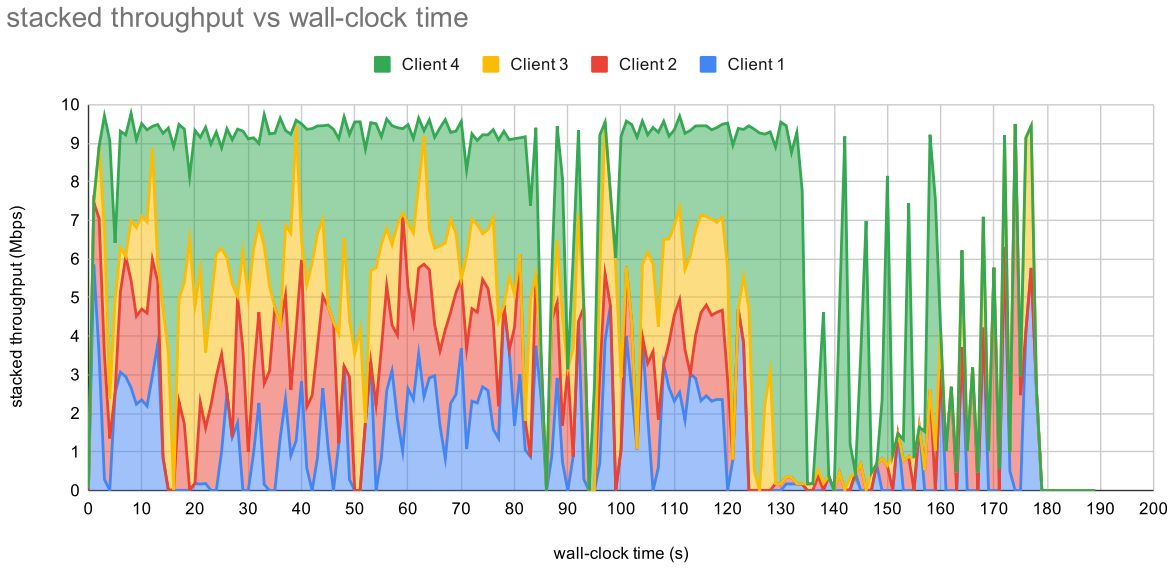

Figure 5.5: No DANE involvement: The throughput expressed in megabit per second for each client during playback, presented as a stacked area chart.

5.3.2 Fair Bandwidth Guidance

Figures 5.6, 5.7, 5.8 and 5.9 depict the buffer occupancy, selected bitrates, individual throughputs and stacked throughputs respectively of all four clients involved in the experiment. One can see that the **resource allocation** role assumed by our DANE introduces fairness in terms of bandwidth usage by looking at the bitrates selected by our clients (i.e., Figure **??**) and the stacked throughput graph. Clients never chose a bitrate higher than their given bandwidth budget (as communicated by the DANE), which together with the implemented pacing mechanism in the Dash.js segment scheduler results in a more evenly divided throughput as can be seen in the (stacked) throughput graph. This can also be seen by comparing the throughput graph of this experiment with the one of experiment 1, experiment 2 clearly shows all clients roughly staying around the 2-2.5Mb per second mark, whilst in experiment 1 this chaotically varied depending on the client. The results also show a drastic drop in the amount of quality switches and no detected stalls (see Table 5.3); this behaviour yields a better QoE for the user. The reason we see one quality switch for all clients stems from the buffer-fill phase during which Dash.js will fill its buffer in a lower quality until it reaches the steady buffer state, after which it will continue filling the buffer at highest possible quality (which is equal to the DANE's directive in this scenario). We can notice this behaviour in the buffer occupancy graph which starts showing a drop only after the steady buffer state is reached.

	Client 1	**Client 2**	**Client 3**	**Client 4**
Detected quality switches	1	1	1	1
Detected stalls	0	0	0	0

Table 5.3: Fair bandwidth guidance: Overview of the total number of stalls and quality switches experienced by all four clients.

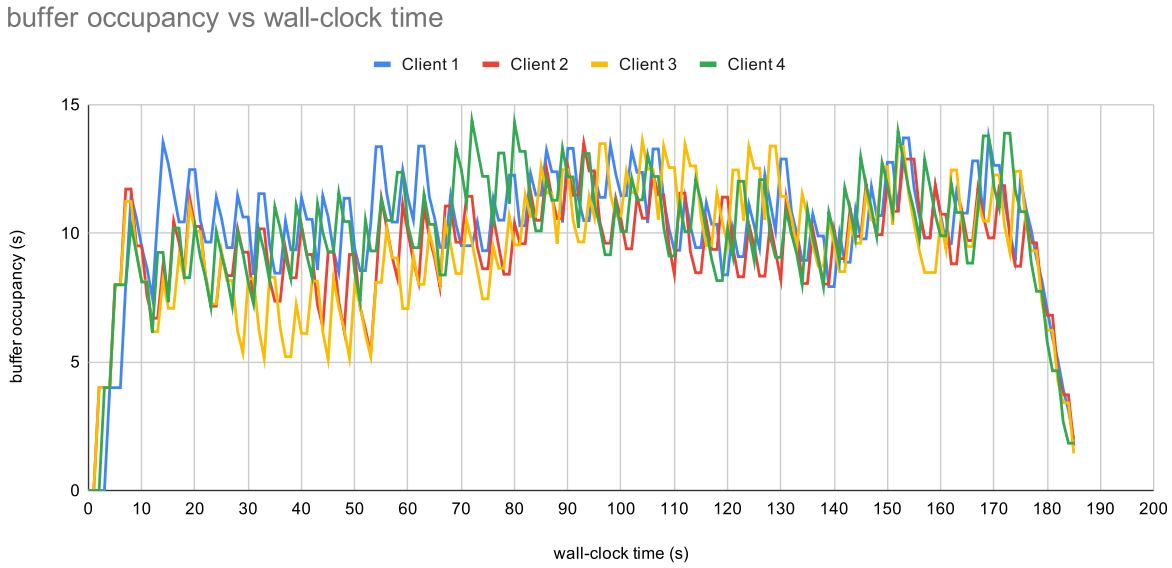

Figure 5.6: Fair bandwidth guidance: The buffer occupancy expressed in seconds for each client during playback.

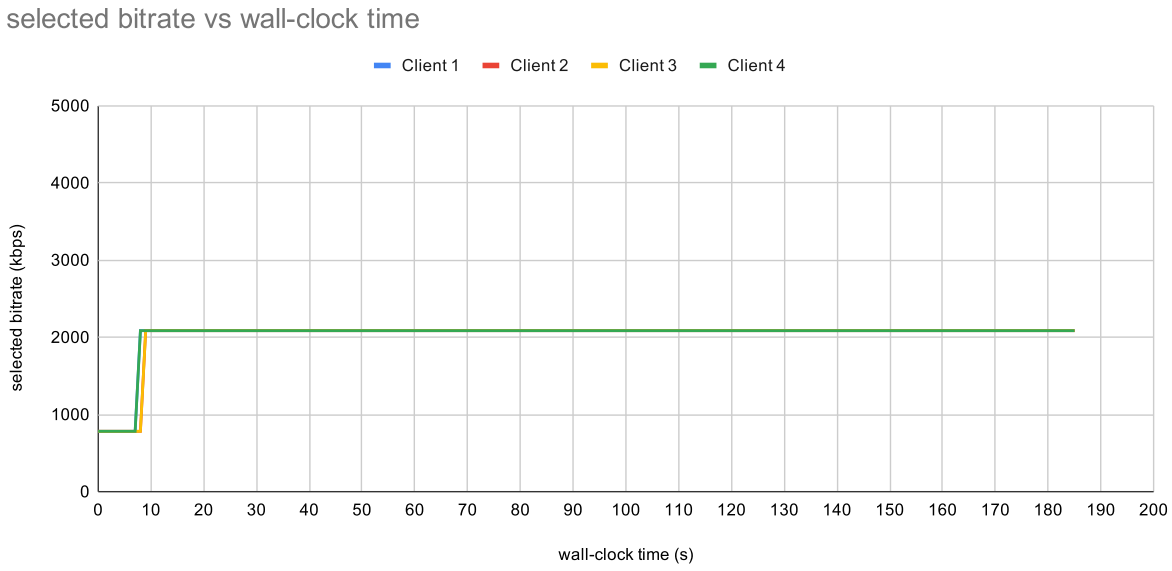

Figure 5.7: Fair bandwidth guidance: The selected bitrates expressed in kilobit per second for each client during playback. Even though clients one and two are not visible, in reality they follow the same bitrate selection trajectory as clients three and four.

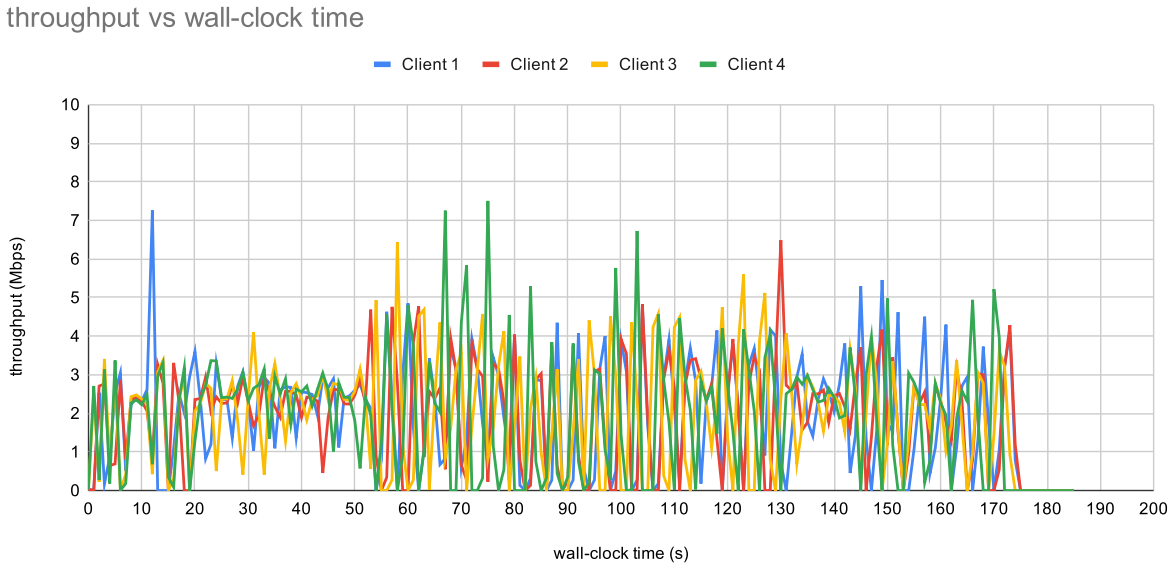

Figure 5.8: Fair bandwidth guidance: The throughput expressed in megabit per second for each client during playback, presented as a line chart.

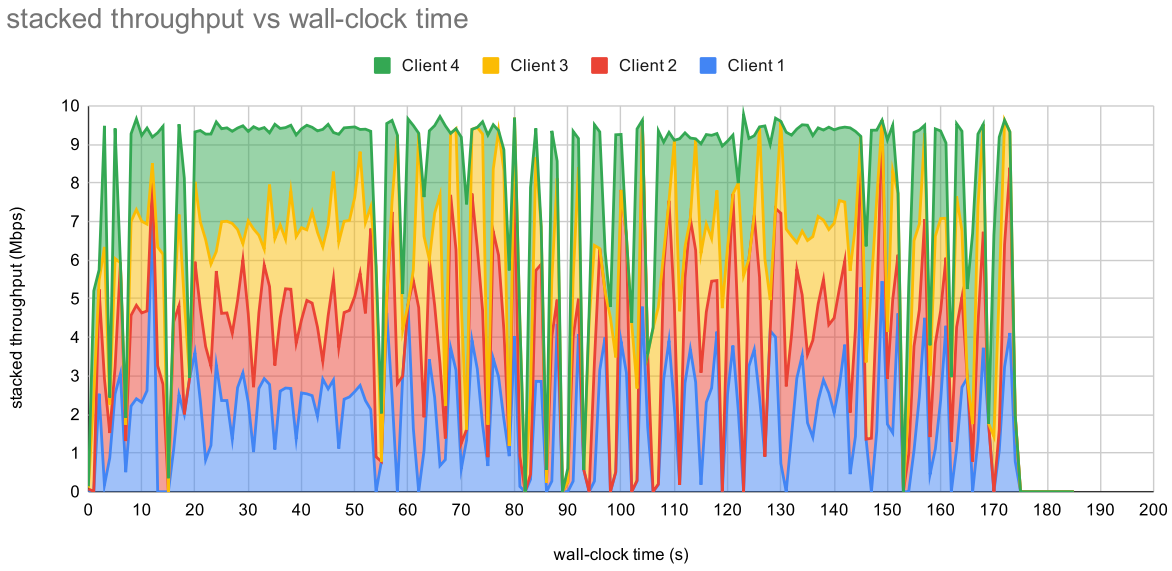

Figure 5.9: Fair bandwidth guidance: The throughput expressed in megabit per second for each client during playback, presented as a stacked area chart.

5.3.3 Smart Caching

For this experiment, we will only show the graphs for the **naive queue** implementation. The graphs for the **furthest player prioritized queue** implementation show similar findings in terms of buffer occupancy, selected bitrates and throughput. Figures 5.10, 5.11, 5.12 and 5.13 depict the buffer occupancy, selected bitrates, individual throughputs and stacked throughputs respectively of all four clients started in parallel. Figures 5.14, 5.15, 5.16 and 5.17 depict the same respective information but for the case where all clients are started at a 15 second interval. All graphs of both cases show high spiking behaviour from the moment a client is able to switch to cached segments (i.e., at the 85 and 100 second marks, respectively), this is because the client does not require pacing anymore and it can fetch the highest qual-

ity segments from the DANE over intranet speeds (i.e., 1Gb per second). The most interesting metric gathered during this experiment is the cache hit count for each individual client. Table 5.4 provides an overview of the amount of quality changes, stalls and cache hits each client encountered. As explained in Section 5.1, this experiment utilizes four second segments for a three minute long period. This results in **46 segments** being fetched, which is one more than anticipated; the segmentation process (see Section 2.4.1) during adaptive content preparation can sometimes introduce small deviations such as segments which are a couple of milliseconds shorter or longer than the intended duration.

At first sight, the two caching implementations seem similar in their achievements. A longer playback duration however would result in different results, the reason being that our POC implementation works with the next twenty anticipated requests (see Section 4.2.4), which in this setup is almost half of the segments. We also see that client one suffers from drawbacks if more clients join the stream at a later time. Because the DANE will start aggressively caching the highest quality available at two times the bitrate for that quality, all other clients are pushed towards a very low quality. Clients joining after client one do not notice this because they will be guided towards the right bitrate from the start; client one however has to significantly drop its quality. This process can take a while because client one is still fetching the highest available quality concurrently with the DANE; the DANE and client, in other words, compete for bandwidth for a small amount of time until client one finishes fetching the in-flight (high-quality) segment. We know that by looking at the bitrate selection graph and the throughput chart that the DANE paces itself correctly; if this were not the case, all clients would start experiencing a similar behaviour to the clients in experiment 1.

In this experimental setup, the caching algorithms are both able to push clients towards the highest available quality for over 50% of all segments. Our experiment is thus able to provide the highest quality of experience towards its clients whilst also reducing the overall traffic over the shared network connection. Table 5.5 provides an overview of the reductions compared to the total amount of traffic used during experiment two. With the provided testbed and experiments we are able to achieve reductions ranging from 29.49% to 42.68%.

		Client			
		1	**2**	**3**	**4**
	Naive cache queue				
Clients start at the same time	**Detected quality switches**	3	2	2	2
	Detected stalls	0	0	0	0
	Detected cache hits	$\frac{24}{46}$	$\frac{24}{46}$	$\frac{24}{46}$	$\frac{24}{46}$
Clients start 15 seconds apart from each other	**Detected quality switches**	8	4	3	2
	Detected stalls	1	0	0	0
	Detected cache hits	$\frac{24}{46}$	$\frac{24}{46}$	$\frac{25}{46}$	$\frac{32}{46}$
	Furthest player prioritized cache queue				
Clients start at the same time	**Detected quality switches**	3	3	2	2
	Detected stalls	0	0	0	0
	Detected cache hits	$\frac{24}{46}$	$\frac{24}{46}$	$\frac{24}{46}$	$\frac{24}{46}$
Clients start 15 seconds apart from each other	**Detected quality switches**	8	4	3	2
	Detected stalls	1	0	0	0
	Detected cache hits	$\frac{24}{46}$	$\frac{24}{46}$	$\frac{24}{46}$	$\frac{31}{46}$

Table 5.4: Smart caching: Overview of the total number of quality switches, stalls and cache hits experienced by all four clients. This setup was tested with two types of cache queue implementations as well as two different client arrival rates.

Experiment	Reduction in MB	%
Naive - Clients start at the same time	72.70	40.98
Naive - Clients start 15 seconds apart from each other	57.23	32.26
Furthest player prioritized - Clients start at the same time	75.73	42.68
Furthest player prioritized - Clients start 15 seconds apart from each other	52.32	29.49

Table 5.5: Smart caching: Overview of the total amounts of reductions in bandwidth consumption compared to experiment 2, expressed in both megabytes as well as percentage.

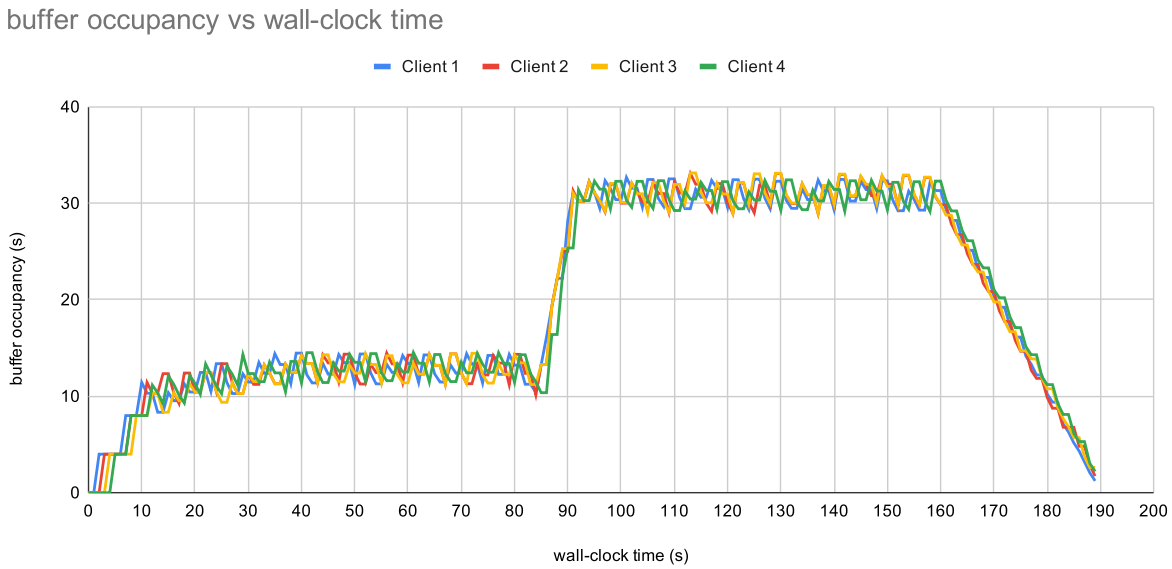

Figure 5.10: Smart caching: The buffer occupancy expressed in seconds for each client during playback. Clients were started at the same time and the DANE utilized the naive cache queue implementation.

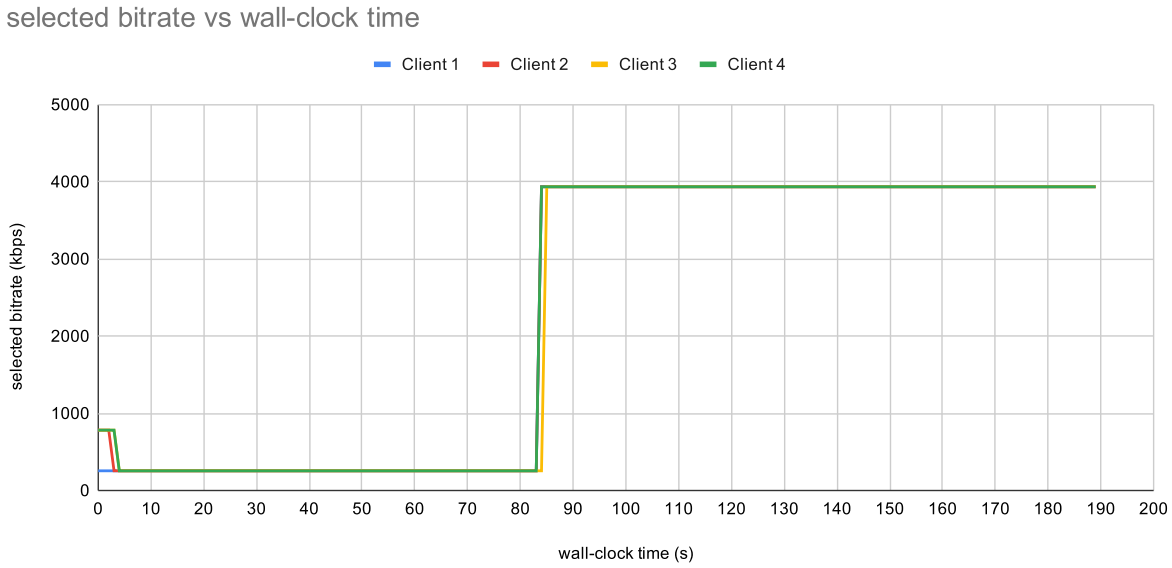

Figure 5.11: Smart caching: The selected bitrates expressed in kilobit per second for each client during playback. Clients were started at the same time and the DANE utilized the naive cache queue implementation.

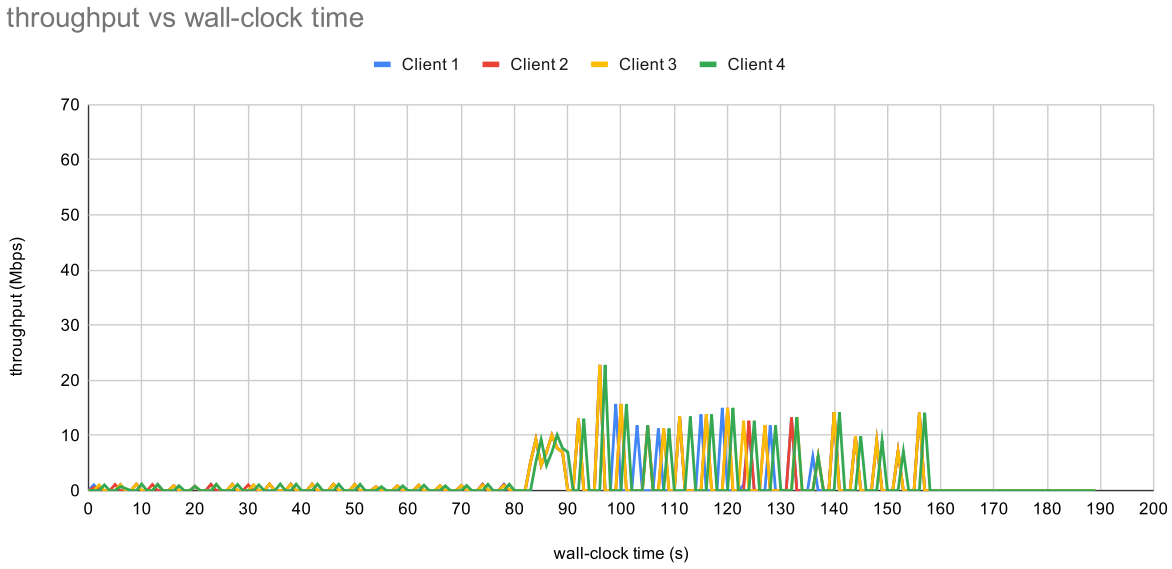

Figure 5.12: Smart caching: The throughput expressed in megabit per second for each client during playback, presented as a line chart. Clients were started at the same time and the DANE utilized the naive cache queue implementation.

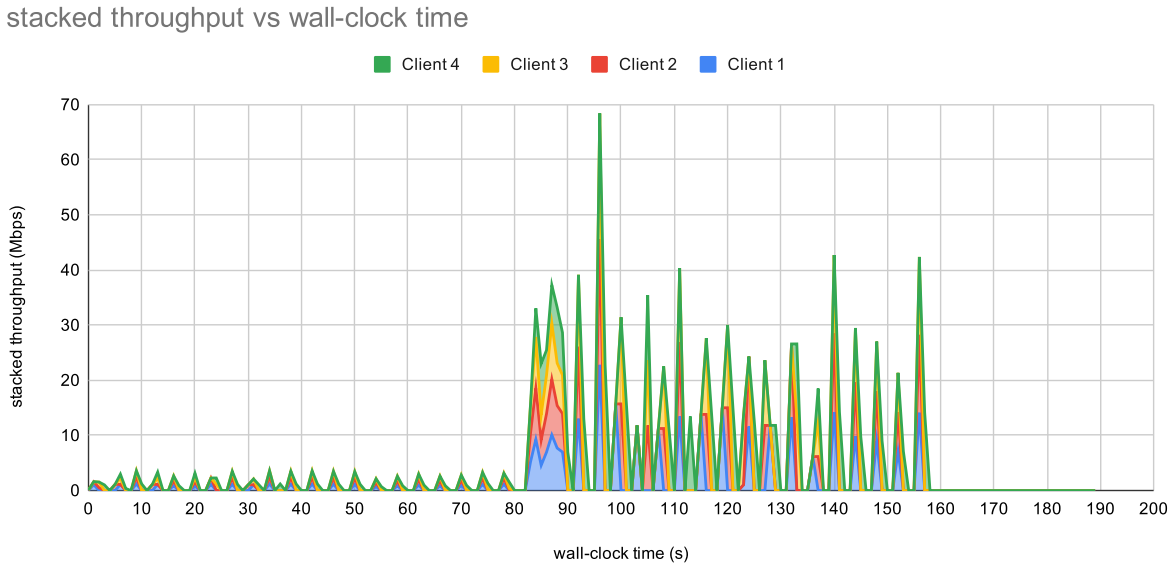

Figure 5.13: Smart caching: The throughput expressed in megabit per second for each client during playback, presented as a stacked area chart. The total throughput never exceeds the 10Mbit per second threshold when fetching content from the media origin server and is unrestricted when fetching cached content from the DANE. Clients were started at the same time and the DANE utilized the naive cache queue implementation.

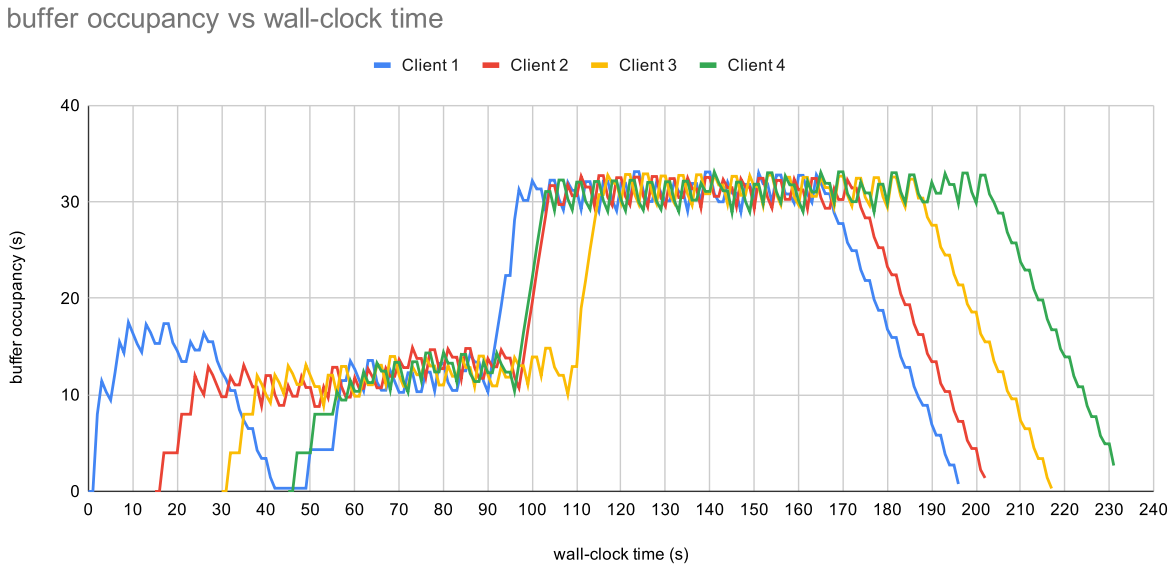

Figure 5.14: Smart caching: The buffer occupancy expressed in seconds for each client during playback. Clients were started using a 15 second arrival interval and the DANE utilized the naive cache queue implementation.

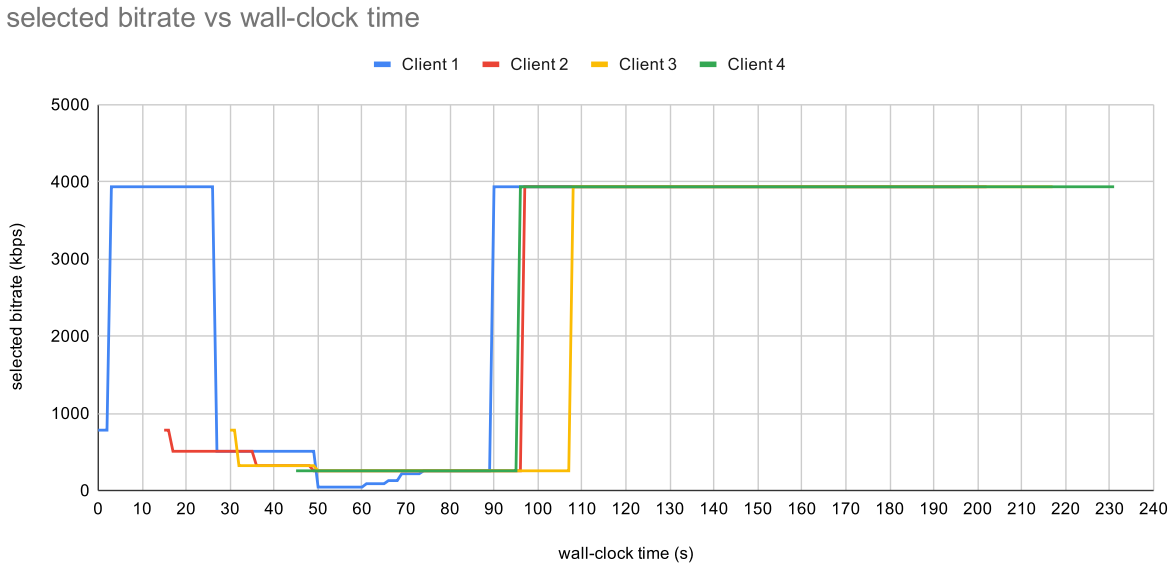

Figure 5.15: Smart caching: The selected bitrates expressed in kilobit per second for each client during playback. Clients were started using a 15 second arrival interval and the DANE utilized the naive cache queue implementation.

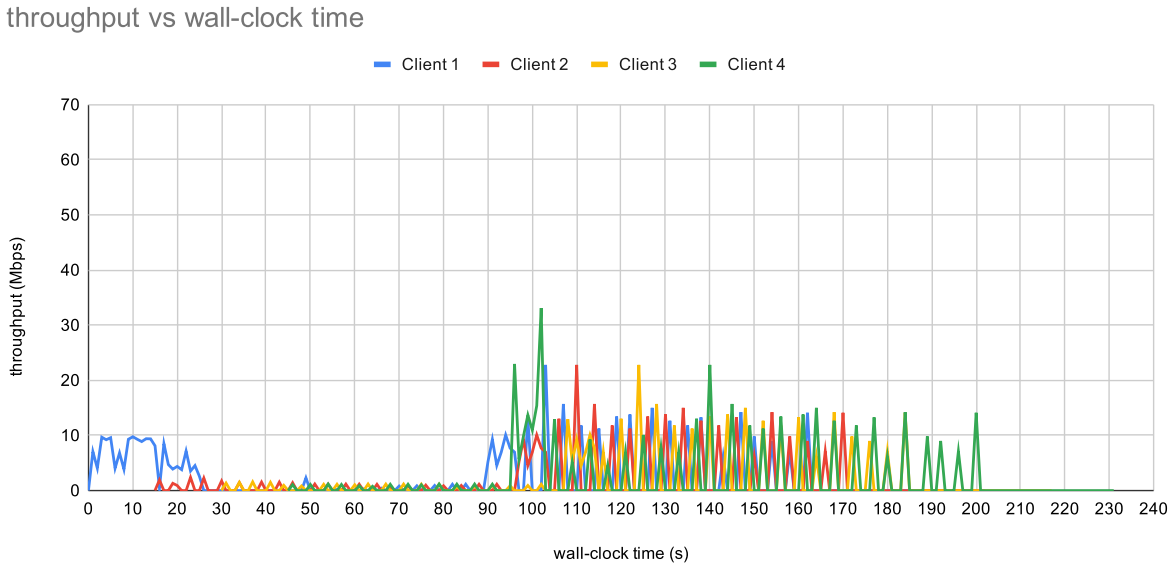

Figure 5.16: Smart caching: The throughput expressed in megabit per second for each client during playback, presented as a line chart. Clients were started using a 15 second arrival interval and the DANE utilized the naive cache queue implementation.

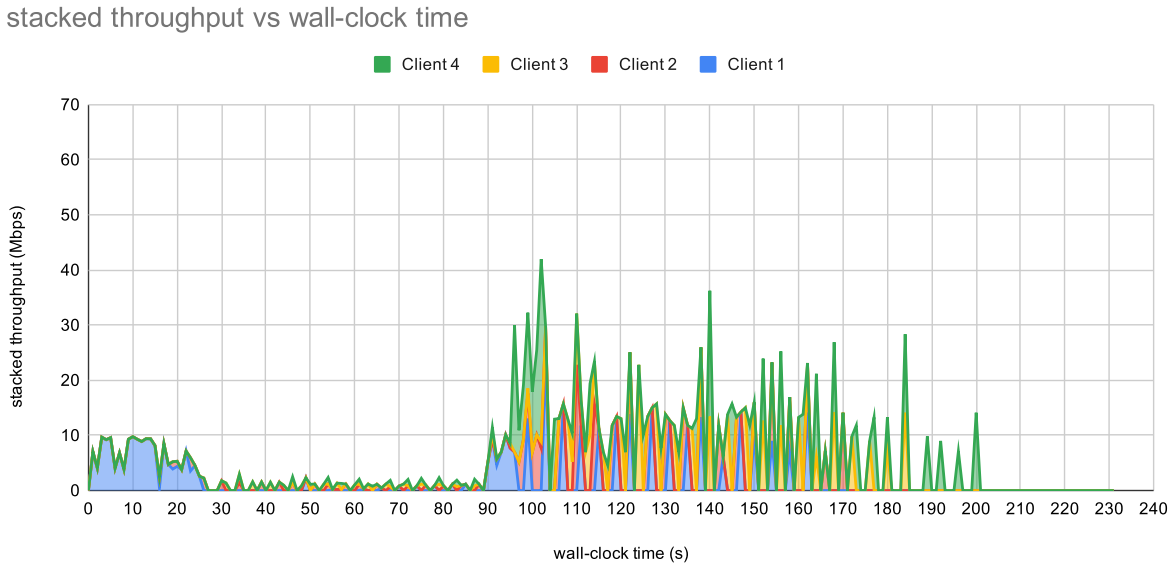

Figure 5.17: Smart caching: The throughput expressed in megabit per second for each client during playback, presented as a stacked area chart. The total throughput never exceeds the 10Mbit per second threshold when fetching content from the media origin server and is unrestricted when fetching cached content from the DANE. Clients were started using a 15 second arrival interval and the DANE utilized the naive cache queue implementation.

Chapter 6

Related Works

The following sections describe work related to server and network assistance in video streaming. Most works encountered in this field are related to **software-defined networking** (SDN), explained in Section 6.1. While realizing this thesis, works by the DASH-IF [42] and Pham et al. [43] were published which are closely related to our own work, these are described in Sections 6.2 and 6.3.

6.1 Software-Defined Networking for Improving DASH Streams

The work by Kleinrouweler et al. [44] focuses on performance problems when multiple DASH clients share the same network connection. Their work builds upon the findings by Huang et al. [4] and Akhshabi et al. [45] that show when multiple adaptive streaming clients compete for bandwidth, all clients start noticing a decrease in bitrate due to competing TCP flows. This inevitably leads to quality switches and stalling, both of these negatively impact the viewers' QoE.

Kleinrouweler et al. focus on large real network settings with up to 600 concurrent DASH clients. They introduce a DASH assisting network element (DANE), not to be confused with the DANE of SAND which got to see daylight at a later date. This DANE consists out of three elements:

1. A physical **Network Bridge** consisting of two physical network interface cards that are controlled in software via Linux tc [41] for packet forwarding and traffic control
2. The **Network Controller** is a piece of software that manages the network bridge traffic control settings
3. The **Service Manager** is the entity that talks to the network controller and all DASH clients

In their setup, DASH clients talk to the service manager about the stream they are consuming. The service manager equally divides the bandwidth among the DASH players and sends them directives about their fair share. The authors decided to implement the DASH players as headless entities; focus on the network bottleneck was their main priority, as such it was decided to ignore video decoding and displaying. Each headless DASH player comes equipped with three adaptation algorithms, of which the two first are based on the ABR logic from Dash.js (see Section 4.1.1):

1. **Throughput-based**
2. **BOLA**
3. **Assisted adaptation** (i.e., their DANE supported algorithm)

Even though their headless client incorporates some of the Dash.js ABR rules, they do not work in the same complex way. The first two rules work independently and are selected as the adaptation algorithm during tests. Their assisted adaptation however works in conjunction with the BOLA rule: if the DASH player buffer is higher than 10 seconds and the BOLA rule suggests a higher bitrate than the recommended bitrate from the service manager, the BOLA suggestions will be ignored; in cases where the buffer is lower than 10 seconds, DANE recommendations are also ignored.

Their testbed consisted of 30 Raspberry Pis each hosting a maximum of 20 headless DASH players. Two experiments were run on the above three adaptation algorithms:

1. **Parallel start-up** of multiple clients to simulate a popular stream (i.e. a new release of a popular series).

2. **Poisson process start-up** in which DASH players are started using a Poisson process. This makes the arrival rate of clients vary over time.

The results of their experiments corroborated earlier findings from Huang et al. [4] and Akhshabi et al. [45] when looking at throughput-based and BOLA-based adaptation algorithms. With their assisted adaptation, they were able to reduce stalling and quality switches when compared to standalone throughput or BOLA, Table 6.1 describes these reduction rates.

	Stalling	**Quality Switches**
Compared to Throughput	-95%	-94%
Compared to BOLA	-75%	-85%

Table 6.1: Reduction in stalling and quality switches for assisted adaptation versus througput and BOLA.

The work by Bhat et al. [46] focuses on network assistance in wide-area setups. They introduce a setup called **SABR** which stands for **SDN assisted Adaptive Bitrate Video streaming**. Their setup consists of three important parts:

1. An **OpenFlow Southbound API** (i.e., a controller which is used to send network directives to physical or virtual routers and switches within the network to change their behaviour) which is used to orchestrate SDN assisted CDNs for adaptive streaming.

2. An **OpenFlow Nortbound interface** which implements a **Representational State Transfer** (REST) API which can be queried by the clients to retrieve CDN information such as available caches, bottleneck bandwidth to those caches and the available DASH content within these caches.

3. A **SDN assisted adaptation algorithm** implemented in the open source Astream[1] Python based DASH client by Juluri et al. [47]

The SDN assisted adaptation algorithm is implemented in three classes of adaptation algorithms (similar to the approach of Dash.js, see Section 4.1.1):

1. **Rate-based algorithms**: SABR provides more accurate and realtime available bandwidth information; these outperform the estimations made on previously measured throughputs of already downloaded segments

2. **Buffer-based algorithms**: By letting SABR provide more accurate fetch times for segments and their corresponding quality, clients can anticipate when to fetch a new segment based on their buffer occupancy

3. **Hybrid algorithms**: Combines the strengths of previous two algorithms and their SABR improvements

Their testbed consists of a geographically distributed **CloudLab** [48] setup (i.e., a collection of software defined network switches co-hosted with storage and compute power for researchers to perform cloud-based experiments on) which tests a number of ABR algorithms in combination with caching algorithms (for a more detailed description, we redirect the reader to [46]). Their setup is built to test the QoE performance of clients with SABR and the caching performance with SABR in terms of cache hit rates on CDN nodes. Their findings indicate an improvement in the quantitative metrics of QoE, mainly because SABR provides accurate network characteristics which directly influence how well ABR algorithms perform. Their work also shows that the choice of caching algorithm at the CDN side can improve QoE as well as overall system performance.

Whilst our work focuses on similar topics, it takes a different route in achieving similar results. Our goal is to guide clients using MPEG-DASH SAND by only sending asynchronous network-to-client recommendations which are not enforced by network entities (e.g., software-defined networking) and only by the clients themselves.

[1] https://github.com/pari685/AStream

6.2 MPEG-DASH SAND Interoperability Guidelines

Whilst working on this thesis, DASH-IF released guidelines [42] which tackle our first research question (see Section 1.1) in an alternative way to our implementation. The DASH-IF guidelines describe several SAND modi for specific use cases together with a workflow. In our implementation, we opted for HTTP and header channels as our main way of exchanging SAND messages. The DASH-IF guidelines, however, indicate that next to the mandatory HTTP and header channel support, WebSocket is also mandatory and shall we used as the primary transport protocol. By using WebSocket, DASH-IF avoids having to deal with active versus non-active client identification (see Section 4.2.1) and thus does not require a polling mechanism. The following paragraphs describe the SAND modi. Clients and DANEs indicate support for the following modi in their *ClientCapabilities* and *DaneCapabilities* SAND message, where they set the `messageSetUri` parameter to one or more of the following URNs:

- Consistent QoE/QoS: `http://dashif.org/guidelines/sand/modes/qo`
- Proxy Caching: `http://dashif.org/guidelines/sand/modes/pc`
- Network Assistance: `http://dashif.org/guidelines/sand/modes/na`

Consistent QoE/QoS. This mode, also known as the **home gateway** use case, is used to provide a consistent QoE or QoS for DASH clients within a shared network. It resembles our **bandwidth guidance role** defined in Section 4.2.3. The following SAND messages are required to be implemented by DANE and DASH clients:

- ClientCapabilities
- DaneCapabilities
- SharedResourceAssignment
- SharedResourceAllocation
- QoSInformation

Proxy Caching. This mode, also known as **CDN edge**, is intended for enabling streaming enhancements via proxy caching. Similarities are present with our **smart caching role**; our implementation, however, is deployed in the same network as where the DASH clients reside. The following SAND messages are required to be implemented by DANE and DASH clients:

- ClientCapabilities
- DaneCapabalities
- AnticipatedRequests
- AcceptedAlternative
- DeliveredAlternative
- ResourceStatus
- MPDValidityEndTime
- NextAlternatives

Caches can indicate (partial) *representation* caching by sending the PER messages *ResourceStatus*, *DeliveredAlternative* and *MPDValidityEndTime*. To achieve next-segment caching, a client sends the status messages *AnticipatedRequests*, *AcceptedAlternatives* and *NextAlternatives*. No guidelines are provided on how to implement next-segment caching.

Network Assistance. This mode provides DASH clients with network information for their rate-based and buffer-based ABR algorithms in wireless scenarios. It provides two functions towards clients: bandwidth guidance per segment download and temporary delivery boost to avoid buffer under-runs. The second option is not mandatory for a DASH client to support. The following SAND messages are required to be implemented by DANE and DASH clients, with the final 6 message types being custom/non-standardized SAND extensions:

- ClientCapabilities
- DaneCapabilities
- SharedResourceAssignment
- SharedResourceAllocation
- NetworkAssistanceInitiationRequest
- NetworkAssistanceInitiationResponse
- NetworkAssistanceTerminination
- SegmentDuration
- DeliveryBoostRequest
- DeliveryBoostResponse

The DASH-IF guidelines also mention security considerations which match with our security considerations described in Section 3.4.1. It also introduces a new DANE discovery procedure with out-of-band DANEs using DNS. A DASH client can optionally implement this discovery protocol. The way it works is by querying the `dane` subdomain either through the Fully Qualified Domain Name (FQDN) or the Partially Qualified Domain Name (PQDN). The subdomain named `dane` is expected to group all DANEs that the network implements. A query shall result in all IP addresses of the respective DANEs. The specific modi supported by the network are identified as subdomains of the `dane` subdomain (i.e., `qo.dane`, `pc.dane` and `na.dane`). If a specific mode is queried, the DNS shall respond with only the IP of the DANE providing that mode. Our implementation does not handle out-of-band DANE discovery.

6.3 MPEG-DASH SAND for Improving DASH Streams

The work by Pham et al. [43], similar to our **bandwidth guidance role** (see Section 4.2.3), focuses on introducing fairness when bandwidth is shared by multiple clients. Just like our implementation, their setup makes use of Dash.js clients (although, the older version 2.5.0) with only the BOLA rule enabled (i.e., a buffer-based client). The way bandwidth guidance is applied to clients, however, differs much from our own implementation. Pham et al. utilize a traffic shaper on the client side to limit ingress traffic to the quality bitrate advised by their DANE. When the DANE advises a maximum quality, their solution informs Dash.js about the quality level limit and sets the maximum allowed ingress throughput to that level. Our solution on the other hand, deploys a per Dash.js client pacing algorithm which limits the bandwidth usage without explicitly throttling the client's network connection. Their testbed consists of 24 Dash.js clients connecting to the media origin server. With the DANE involved, their results match with ours in that the QoE is improved significantly as well as fairness is achieved. The advantage of our implementation is that it can be deployed in any browser without extra sofware required by the user; Pham et al. explicitly state that their implementation is a makeshift bandwidth limiting solution which works well for their setup (i.e., to prove their concept).

Chapter 7

Conclusion and Future Work

To conclude this thesis, we will recapitulate our findings and answer the questions we posed at the beginning of this thesis. After examining the current landscape together with future predictions, we are confident that a need for server and network assistance will rise soon. The Moving Pictures Expert Group predicted this by starting a Core Experiment in 2012 which led to the creation of SAND in 2017, which enables a standardized way of performing server and network assistance during MPEG-DASH streaming. We started this thesis with an interest in optimizing adaptive streaming QoE on shared network connections by means of server and network assistance.

The first question we asked ourselves, was what aspects are required to transition from the SAND specification to a deployable SAND implementation. To answer this question, we analyzed the SAND specification and looked into related works concerning client assistance during adaptive media streaming. We created a protocol for SAND message exchange between a client and an out-of-band DANE using the mandatory HTTP and header channels as mentioned in the SAND specification. The protocol supplies DANEs and clients with a mechanism to achieve out-of-band communication such that clients can share their operational characteristics and DANEs can provide guidance towards their clients.

The second question we asked ourselves, was if it is possible to guide multiple users on a shared network connection into a fair bandwidth usage using only SAND such that they do not experience the detrimental effects caused by bandwidth competition? We created a POC that bestows a resource allocation role on a DANE. By experimentally evaluating our POC, we were able to confirm that it is in fact possible to guide concurrent MPEG-DASH clients towards a fair bandwith share without having to restrict their ingress throughput by means of traffic shaping or software defined networking. The evaluation showed that the overall QoE (expressed in terms of the total amount of quality switches and playback stalls) improved when compared to a non-DANE scenario.

The last question we asked ourselves, was if we can support multiple clients consuming the same DASH stream within the same network in such a way that they enjoy the same or a better overall Quality of Experience and that the overall bandwidth consumption is lower than when the clients would individually compete for the same content at the quality provided by SAND? For this, we built upon the resource allocation role and added a smart caching role. This role enables a DANE to pre-fetch DASH media content and inform clients of its availability within the shared network. By performing an evaluation on this setup, we concluded that clients were able to view the content in a higher quality than should have been possible in a fair share scenario, with the penalty of temporarily having to share the network connection with an extra consumer, being the DANE itself. The results also show less total bytes spent over the shared network connection which results lower costs.

We can confidently say that the research questions we posed at the start of this work can all be answered in a positive way. This proves that server and network assistance optimizes our scenario and in all probability has to potential to do so too in other areas. With adaptive streaming becoming more popular and putting a significant requirement on shared network connections, we predict a rise in the popularity of SAND to enable more fair and better user experiences.

Future Work

The following paragraphs describe future work opportunities that were identified while working on this thesis.

Heuristic caching approach. The smart caching role implemented into our DANE currently exclusively pre-fetches DASH content at the highest quality available; this was done specifically in context of this thesis. The goal was to try and see if we could achieve a higher quality for all our clients streaming the same DASH content whilst saving on actual bandwidth consumption and providing a better QoE. In a more realistic scenario, however, only considering the highest quality DASH *representation* for DANE-side caching purposes would not suffice. We do not take into account what the actual client capabilities are (this could require a new type of SAND message) or what the actual throughput limit looks like. If, for example, the highest quality indicated by the manifest file were to be 10Mbps and the throughput is limited to 10Mbps, our setup would fail. A better approach would be to figure out what quality is best-fitting for the situation at hand and subsequently pre-fetch that quality for our clients. Another approach would be to group clients according to their capabilities and give them a certain priority, with these priorities then being incorporated in the decision process. There exist many ways one could improve this behaviour whilst keeping the fair use of the shared network more or less intact.

Heat-map cache queue implementation. During our implementation and evaluation, we focused on two types of cache queue implementations: naive and furthest playback prioritized. The first one will provide a better QoE but does not take client playback times into consideration whilst the second one fails in specific scenarios. Imagine a case where five clients are streaming the same DASH content, clients 1 through 4 are around the same playback timestamp at the beginning of the stream, but client 5 is near the end of the playback timeline. Our furthest playback prioritized cache queue would prioritize client 5, even though it would make more sense to prioritize clients 1 through 4 as this would result in less overall bandwidth usage. As such, we propose a heat-map cache queue which looks at where clients are situated on the timeline. If groups of players are clear, those should be prioritized.

DANE scalability. Our POC makes use of HTTP and HTTP headers for its message exchange. We implemented this in Python utilizing the Flask micro-webframework. All streaming related characteristics and client information are thus saved in memory on the spawned instance. In the event we want to spawn more instances (e.g., a load balancer such as NGINX[1] utilizes multiple instances to balance incoming traffic), this would result in a fragmented information base. As such, for scalability, we propose the use of a back-end database solution such as Redis[2]. This would decouple client related information, required for making decisions such as bandwidth guidance or smart caching, from the DANE instance itself, thus allowing multiple DANE instances to share client related information and providing clients with the right guidance.

[1] https://www.nginx.com/
[2] https://redis.io/

Appendices

Appendix A

MPEG-DASH Manifest

Figure A.1 represents an MPEG-DASH manifest hierarchy. Listing A.1 shows a real life example of such a manifest.

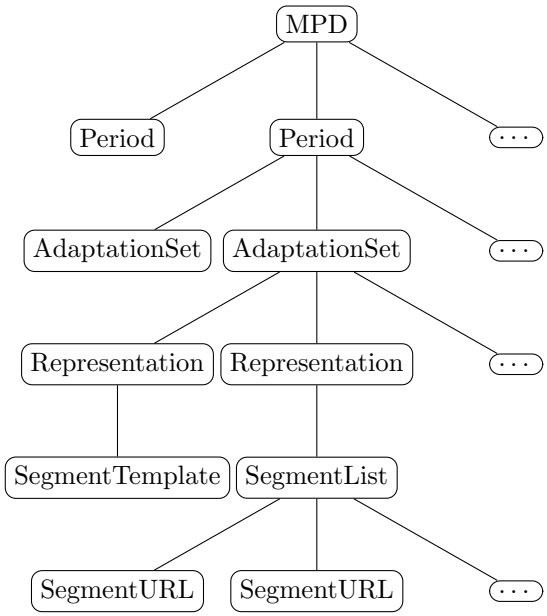

Figure A.1: DASH MPD file structure [25]

```
<?xml version="1.0" encoding="UTF-8"?>
<MPD xmlns="urn:mpeg:dash:schema:mpd:2011" minBufferTime="PT1.500S" type="static"
    mediaPresentationDuration="PT0H0M32.973S" maxSegmentDuration="PT0H0M3.000S" profiles="
    urn:mpeg:dash:profile:isoff-live:2011,http://dashif.org/guidelines/dash264">
  <Period duration="PT0H0M32.973S">
    <AdaptationSet segmentAlignment="true" maxWidth="1920" maxHeight="1080" maxFrameRate="24" par="16:9
    " lang="eng">
      <SegmentTemplate timescale="12288" media="$RepresentationID$/segment_$Number$.m4s" startNumber="1"
    duration="36864" initialization="$RepresentationID$/segment_.mp4" />
      <Representation id="1" mimeType="video/mp4" codecs="avc1.42C01F" width="640" height="360" frameRate=
    "24" sar="1:1" startWithSAP="1" bandwidth="485958" />
      <Representation id="2" mimeType="video/mp4" codecs="avc1.42C01F" width="640" height="360" frameRate=
    "24" sar="1:1" startWithSAP="1" bandwidth="779332" />
      <Representation id="3" mimeType="video/mp4" codecs="avc1.42C01F" width="1280" height="720" frameRate
    ="24" sar="1:1" startWithSAP="1" bandwidth="1465307" />
      <Representation id="4" mimeType="video/mp4" codecs="avc1.42C01F" width="1280" height="720" frameRate
    ="24" sar="1:1" startWithSAP="1" bandwidth="2374790" />
      <Representation id="5" mimeType="video/mp4" codecs="avc1.42C01F" width="1920" height="1080"
    frameRate="24" sar="1:1" startWithSAP="1" bandwidth="2956542" />
      <Representation id="6" mimeType="video/mp4" codecs="avc1.42C01F" width="1920" height="1080"
    frameRate="24" sar="1:1" startWithSAP="1" bandwidth="3945761" />
      <Representation id="7" mimeType="video/mp4" codecs="avc1.42C01F" width="1920" height="1080"
    frameRate="24" sar="1:1" startWithSAP="1" bandwidth="5898235" />
```

```
    </AdaptationSet>
    <AdaptationSet segmentAlignment="true" lang="eng">
      <SegmentTemplate timescale="48000" media="$RepresentationID$/segment_$Number$.m4s" startNumber="1"
     duration="144000" initialization="$RepresentationID$/segment_.mp4" />
      <Representation id="8" mimeType="audio/mp4" codecs="mp4a.40.2" audioSamplingRate="48000"
     startWithSAP="1" bandwidth="60067">
        <AudioChannelConfiguration schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011" value="2
     " />
      </Representation>
      <Representation id="9" mimeType="audio/mp4" codecs="mp4a.40.2" audioSamplingRate="48000"
     startWithSAP="1" bandwidth="39891">
        <AudioChannelConfiguration schemeIdUri="urn:mpeg:dash:23003:3:audio_channel_configuration:2011" value="2
     " />
      </Representation>
    </AdaptationSet>
    <AdaptationSet segmentAlignment="true" lang="eng">
      <SegmentTemplate timescale="1000" media="$RepresentationID$/segment_$Number$.m4s" startNumber="1"
     duration="3000" initialization="$RepresentationID$/segment_.mp4" />
      <Representation id="10" mimeType="application/mp4" codecs="wvtt" startWithSAP="1" bandwidth="1322" /
     >
    </AdaptationSet>
  </Period>
</MPD>
```

Listing A.1: An MPEG-DASH example manifest [25].

```
<!-- A representation with a SegmentList containing SegmengtURLs -->
<SegmentList>
  < Initialization  sourceURL="video/500kbit/init.mp4"/>
</SegmentList>
<Representation id="480p 500kbps" mimeType="video/mp4" frameRate="24" bandwidth="520929" codecs="avc1.4
     d4015" width="638" height="272">
  <SegmentList timescale="1000" duration="2000">
    <SegmentURL media="video/500kbit/segment_1.m4s"/>
    <SegmentURL media="video/500kbit/segment_2.m4s"/>
    <SegmentURL media="video/500kbit/segment_3.m4s"/>
    <SegmentURL media="video/500kbit/segment_4.m4s"/>
    <SegmentURL media="video/500kbit/segment_5.m4s"/>
    <SegmentURL media="video/500kbit/segment_6.m4s"/>
    <SegmentURL media="video/500kbit/segment_7.m4s"/>
    <SegmentURL media="video/500kbit/segment_8.m4s"/>
    <SegmentURL media="video/500kbit/segment_9.m4s"/>
    <SegmentURL media="video/500kbit/segment_10.m4s"/>
    <SegmentURL media="video/500kbit/segment_11.m4s"/>
    <SegmentURL media="video/500kbit/segment_12.m4s"/>
    <SegmentURL media="video/500kbit/segment_13.m4s"/>
    <SegmentURL media="video/500kbit/segment_14.m4s"/>
    <SegmentURL media="video/500kbit/segment_15.m4s"/>
    <SegmentURL media="video/500kbit/segment_16.m4s"/>
    <SegmentURL media="video/500kbit/segment_17.m4s"/>
    <SegmentURL media="video/500kbit/segment_18.m4s"/>
    <SegmentURL media="video/500kbit/segment_19.m4s"/>
    <SegmentURL media="video/500kbit/segment_20.m4s"/>
  </SegmentList>
</Representation>

<!-- A representation with a SegmentTemplate -->
<Representation id="480p 500kbps" mimeType="video/mp4" frameRate="24" bandwidth="520929" codecs="avc1.4
     d4015" width="638" height="272">
  <SegmentTemplate timescale="1000" duration="2000" media="video/500kbit/segment_$Number$.m4s" initialization
     ="video/500kbit/init.mp4" startNumber="1"/>
</Representation>
```

Listing A.2: An MPEG-DASH manifest extract showing the difference between a segment template and a segment list [25].

Appendix B

SAND Default Message Data Formats

B.1 SAND Message XSD Schema

Listing B.1 shows the default SAND message set. During the work on this thesis, an amendment [35] was published introducing changes to the XML Schema Definition (XSD) published in the original specification [29]; the XSD represented in Listing B.1 reflects these changes. Applications who wish to validate SAND messages can use the provided XSD to check whether or not the provided SAND messages are conform to the XSD schema.

```
<xs:schema
  targetNamespace="urn:mpeg:dash:schema:sandmessage:2016"
  attributeFormDefault="unqualified"
  elementFormDefault="qualified"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns="urn:mpeg:dash:schema:sandmessage:2016">

  <xs:annotation>
    <xs:appinfo>SAND Messages</xs:appinfo>
    <xs:documentation xml:lang="en">
      This Schema defines the Server And Network Assisted DASH (SAND) messages for MPEG-DASH.
    </xs:documentation>
  </xs:annotation>

  <!-- SAND message: main element -->
  <xs:element name="SANDMessage" type="SANDEnvelopeType"/>

  <!-- SAND common envelope Type -->
  <xs:complexType name="SANDEnvelopeType">
    <xs:choice maxOccurs="unbounded">
      <xs:element name="AnticipatedRequests" type="AnticipatedRequestsType"/>
      <xs:element name="SharedResourceAllocation" type="SharedResourceAllocationType"/>
      <xs:element name="AcceptedAlternatives" type="AcceptedAlternativesType"/>
      <!-- AbsoluteDeadline is not allowed in XML, only in HTTP headers -->
      <xs:element name="MaxRTT" type="MaxRTTType"/>
      <xs:element name="NextAlternatives" type="NextAlternativesType"/>
      <xs:element name="ResourceStatus" type="ResourceStatusType"/>
      <xs:element name="DaneResourceStatus" type="DaneResourceStatusType"/>
      <xs:element name="SharedResourceAssignment" type="SharedResourceAssignmentType"/>
      <xs:element name="MPDValidityEndTime" type="MPDValidityEndTimeType"/>
      <xs:element name="Throughput" type="ThroughputType"/>
      <xs:element name="AvailabilityTimeOffset" type="AvailabilityTimeOffsetType"/>
      <xs:element name="QoSInformation" type="QoSInformationType"/>
      <!-- DeliveredAlternative is not allowed in XML, only in HTTP headers -->
      <xs:element name="DaneCapabilities" type="DaneCapabilitiesType"/>
      <!-- ISO/ISC 23009-5 Annex D DASH metrics -->
      <xs:element name="TcpList" type="TcpListType"/>
      <xs:element name="HttpList" type="HttpListType"/>
      <xs:element name="RepSwitchList" type="RepSwitchListType"/>
      <xs:element name="BufferLevelList" type="BufferLevelListType"/>
      <xs:element name="PlayList" type="PlayListType"/>
      <xs:any namespace="##other" processContents="lax" minOccurs="0" maxOccurs="unbounded"/>
```

```
    </xs:choice>
    <xs:attribute name="senderId" type="xs:token"/>
    <xs:attribute name="generationTime" type="xs:dateTime"/>
    <xs:anyAttribute namespace="##other" processContents="lax"/>
  </xs:complexType>

  <!-- SAND message base Type -->
  <xs:complexType name="SANDMessageType">
    <xs:attribute name="messageId" type="xs:unsignedInt"/>
    <xs:attribute name="validityTime" type="xs:dateTime"/>
  </xs:complexType>

  <!-- AnticipatedRequests Type -->
  <xs:complexType name="AnticipatedRequestsType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:sequence>
          <xs:element name="Request" type="AnticipatedRequestType" minOccurs="1" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Request Type -->
  <xs:complexType name="AnticipatedRequestType">
    <xs:attribute name="sourceUrl" type="xs:anyURI" use="required"/>
    <xs:attribute name="range" type="ByteRangeSetType"/>
    <xs:attribute name="targetTime" type="xs:dateTime"/>
  </xs:complexType>

  <!-- SharedResourceAllocation Type -->
  <xs:complexType name="SharedResourceAllocationType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:sequence>
          <xs:element name="OperationPoint" type="OperationPointType" minOccurs="1" maxOccurs="unbounded"
    />
        </xs:sequence>
        <xs:attribute name="weight" type="xs:unsignedInt"/>
        <xs:attribute name="allocationStrategy" type="xs:anyURI"/>
        <xs:attribute name="mpdUrl" type="xs:anyURI"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- OperationPoint Type -->
  <xs:complexType name="OperationPointType">
    <xs:attribute name="bandwidth" type="xs:unsignedInt" use="required"/>
    <xs:attribute name="quality" type="xs:unsignedInt"/>
    <xs:attribute name="minBufferTime" type="xs:unsignedInt"/>
  </xs:complexType>

  <!-- AcceptedAlternatives Type -->
  <xs:complexType name="AcceptedAlternativesType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:sequence>
          <xs:element name="Alternative" minOccurs="1" maxOccurs="unbounded">
            <xs:complexType>
              <xs:attribute name="sourceUrl" type="xs:anyURI" use="required"/>
              <xs:attribute name="range" type="ByteRangeSetType"/>
              <xs:attribute name="bandwidth" type="xs:unsignedInt"/>
              <xs:attribute name="deliveryScope" type="xs:unsignedInt"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- MaxRTT Type -->
  <xs:complexType name="MaxRTTType">
    <xs:complexContent>
```

```
      <xs:extension base="SANDMessageType">
        <xs:attribute name="maxRTT" type="xs:unsignedInt" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- NextAlternatives Type -->
  <xs:complexType name="NextAlternativesType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:sequence>
          <xs:element name="Alternative" minOccurs="1" maxOccurs="unbounded">
            <xs:complexType>
              <xs:attribute name="sourceUrl" type="xs:anyURI" use="required"/>
              <xs:attribute name="range" type="ByteRangeSetType"/>
              <xs:attribute name="bandwidth" type="xs:unsignedInt"/>
              <xs:attribute name="deliveryScope" type="xs:unsignedInt"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- ClientCapabilities Type -->
  <xs:complexType name="ClientCapabilitiesType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:sequence>
          <xs:element name="SupportedMessage" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:attribute name="messageType" type="xs:unsignedInt" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="messageSetUri" type="xs:anyURI"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- ResourceStatus Type -->
  <xs:complexType name="ResourceStatusType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:choice minOccurs="1" maxOccurs="unbounded">
          <xs:element name="ResourceURLInfo" type="ResourceURLInfoType"/>
          <xs:element name="ResourceRepresentationInfo" type="ResourceRepresentationInfoType"/>
        </xs:choice>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- ResourceURLInfo Type -->
  <xs:complexType name="ResourceURLInfoType">
    <xs:attribute name="baseUrl" type="xs:anyURI"/>
    <xs:attribute name="status" type="ResourceStatusTypeStatusType" use="required"/>
    <xs:attribute name="reason" type="xs:string"/>
  </xs:complexType>

  <!-- ResourceRepresentationInfo Type -->
  <xs:complexType name="ResourceRepresentationInfoType">
    <xs:attribute name="repId" type="StringNoWhitespaceType"/>
    <xs:attribute name="status" type="ResourceStatusTypeStatusType" use="required"/>
    <xs:attribute name="reason" type="xs:string"/>
  </xs:complexType>

  <!-- ResourceStatus status enumeration -->
  <xs:simpleType name="ResourceStatusTypeStatusType">
    < xs:restriction  base="xs:string">
      <xs:enumeration value="available"/>
      <xs:enumeration value="cached"/>
      <xs:enumeration value="unavailable"/>
    </ xs:restriction >
```

```
188   </xs:simpleType>
189
190   <!-- DaneResourceStatus Type -->
191   <xs:complexType name="DaneResourceStatusType">
192     <xs:complexContent>
193       <xs:extension base="SANDMessageType">
194         <xs:sequence>
195           <xs:element name="resource" type="ResourceType" minOccurs="0" maxOccurs="unbounded"/>
196           <xs:element name="resourceGroup" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
197         </xs:sequence>
198         <xs:attribute name="status" type="DaneResourceStatusTypeStatusType" use="required"/>
199       </xs:extension>
200     </xs:complexContent>
201   </xs:complexType>
202
203   <!-- Resource Type -->
204   <xs:complexType name="ResourceType">
205     <xs:simpleContent>
206       <xs:extension base="xs:anyURI">
207         <xs:attribute name="bytes" type="xs:string"/>
208       </xs:extension>
209     </xs:simpleContent>
210   </xs:complexType>
211
212   <!-- DaneResourceStatus status enumeration -->
213   <xs:simpleType name="DaneResourceStatusTypeStatusType">
214     < xs:restriction  base="xs:string">
215       <xs:enumeration value="cached"/>
216       <xs:enumeration value="unavailable"/>
217       <xs:enumeration value="promised"/>
218     </ xs:restriction >
219   </xs:simpleType>
220
221   <!-- SharedResourceAssignment Type -->
222   <xs:complexType name="SharedResourceAssignmentType">
223     <xs:complexContent>
224       <xs:extension base="SANDMessageType">
225         <xs:sequence>
226           <xs:element name="ResourcePrice" type="xs:decimal" minOccurs="0" maxOccurs="unbounded"/>
227         </xs:sequence>
228         <xs:attribute name="clientId" type="xs:token" use="required"/>
229         <xs:attribute name="bandwidth" type="xs:unsignedInt"/>
230       </xs:extension>
231     </xs:complexContent>
232   </xs:complexType>
233
234   <!-- MPDValidityEndTime Type -->
235   <xs:complexType name="MPDValidityEndTimeType">
236     <xs:complexContent>
237       <xs:extension base="SANDMessageType">
238         <xs:sequence>
239           <xs:choice>
240             <xs:element name="MPDUrl" type="xs:anyURI"/>
241             <xs:element name="MPD" type="xs:base64Binary"/>
242           </xs:choice>
243         </xs:sequence>
244         <xs:attribute name="mpdId" type="xs:string"/>
245         <xs:attribute name="publishTime" type="xs:dateTime"/>
246         <xs:attribute name="validityEndTime" type="xs:dateTime" use="required"/>
247       </xs:extension>
248     </xs:complexContent>
249   </xs:complexType>
250
251   <!-- Throughput Type -->
252   <xs:complexType name="ThroughputType">
253     <xs:complexContent>
254       <xs:extension base="SANDMessageType">
255         <xs:attribute name="baseUrl" type="xs:anyURI"/>
256         <xs:attribute name="repId" type="StringNoWhitespaceType"/>
257         <xs:attribute name="guaranteedThroughput" type="xs:unsignedInt" use="required"/>
258         <xs:attribute name="percentage" type="PercentageType"/>
259       </xs:extension>
260     </xs:complexContent>
```

```
  </xs:complexType>

  <!-- Percentage Type -->
  <xs:simpleType name="PercentageType">
    <xs:restriction base="xs:unsignedInt">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- AvailabilityTimeOffset Type -->
  <xs:complexType name="AvailabilityTimeOffsetType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:attribute name="baseUrl" type="xs:anyURI"/>
        <xs:attribute name="repId" type="StringNoWhitespaceType"/>
        <xs:attribute name="offset" type="xs:unsignedInt" use="required"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- QoSInformation Type -->
  <xs:complexType name="QoSInformationType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:attribute name="gbr" type="xs:unsignedInt"/>
        <xs:attribute name="mbr" type="xs:unsignedInt"/>
        <xs:attribute name="delay" type="xs:unsignedInt"/>
        <xs:attribute name="pl" type="xs:unsignedInt"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- DaneCapabilities Type -->
  <xs:complexType name="DaneCapabilitiesType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:sequence>
          <xs:element name="SupportedMessage" type="xs:unsignedInt" minOccurs="0" maxOccurs="unbounded">
            <xs:complexType>
              <xs:attribute name="messageType" type="xs:unsignedInt" use="required"/>
            </xs:complexType>
          </xs:element>
        </xs:sequence>
        <xs:attribute name="MessageSetUri" type="xs:anyURI"/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- Metrics as defined in Annex D of ISO/IEC 23009-1 -->
  <!-- NOTE the naming convention complies with the keys defined in Annex D
       and with camelCase convention like the rest of the schema -->

  <!-- TcpList Type -->
  <xs:complexType name="TcpListType">
    <xs:complexContent>
      <xs:extension base="SANDMessageType">
        <xs:sequence>
          <xs:element name="TcpConnection" type="TcpConnectionType" minOccurs="1" maxOccurs="unbounded"/
    >
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>

  <!-- TcpConnection Type -->
  <xs:complexType name="TcpConnectionType">
    <xs:attribute name="tcpid" type="xs:unsignedInt" use="required"/>
    <xs:attribute name="dest" type="xs:string"/>
    <xs:attribute name="topen" type="xs:dateTime"/>
    <xs:attribute name="tclose" type="xs:dateTime"/>
    <xs:attribute name="tconnect" type="xs:unsignedInt"/>
  </xs:complexType>
```

```
<!-- HttpList Type -->
<xs:complexType name="HttpListType">
  <xs:complexContent>
    <xs:extension base="SANDMessageType">
      <xs:sequence>
        <xs:element name="HttpTransaction" type="HttpTransactionType" minOccurs="1" maxOccurs="
    unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- HttpTransaction Type -->
<xs:complexType name="HttpTransactionType">
  <xs:sequence>
    <xs:element name="Trace" type="TraceType" minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="tcpid" type="xs:unsignedInt" use="required"/>
  <xs:attribute name="type" type="HttpRequestTypeType"/>
  <xs:attribute name="url" type="xs:anyURI"/>
  <xs:attribute name="actualurl" type="xs:anyURI"/>
  <xs:attribute name="range" type="ByteRangeSetType"/>
  <xs:attribute name="trequest" type="xs:dateTime"/>
  <xs:attribute name="tresponse" type="xs:dateTime"/>
  <xs:attribute name="responsecode" type="xs:unsignedInt"/>
  <xs:attribute name="interval" type="xs:unsignedInt"/>
</xs:complexType>

<!-- HttpRequestType Type -->
<xs:simpleType name="HttpRequestTypeType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="MPD"/>
    <xs:enumeration value="XLink expansion"/>
    <xs:enumeration value="Initialization Segment"/>
    <xs:enumeration value="Index Segment"/>
    <xs:enumeration value="Media Segment"/>
    <xs:enumeration value="Bitstream Switching Segment"/>
    <xs:enumeration value="Other"/>
  </xs:restriction>
</xs:simpleType>

<!-- Trace Type -->
<xs:complexType name="TraceType">
  <xs:sequence>
    <xs:element name="b" type="xs:unsignedInt" minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:attribute name="s" type="xs:dateTime" use="required"/>
  <xs:attribute name="d" type="xs:unsignedInt" use="required"/>
</xs:complexType>

<!-- RepSwitchList Type -->
<xs:complexType name="RepSwitchListType">
  <xs:complexContent>
    <xs:extension base="SANDMessageType">
      <xs:sequence>
        <xs:element name="RepSwitch" type="RepSwitchType" minOccurs="1" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>

<!-- RepSwitch Type -->
<xs:complexType name="RepSwitchType">
  <xs:attribute name="t" type="xs:dateTime" use="required"/>
  <xs:attribute name="mt" type="xs:unsignedInt"/>
  <xs:attribute name="to" type="StringNoWhitespaceType"/>
  <xs:attribute name="lto" type="xs:unsignedInt"/>
</xs:complexType>

<!-- BufferLevelList Type -->
<xs:complexType name="BufferLevelListType">
  <xs:complexContent>
```

```
405       <xs:extension base="SANDMessageType">
406         <xs:sequence>
407           <xs:element name="BufferLevel" type="BufferLevelType" minOccurs="1" maxOccurs="unbounded"/>
408         </xs:sequence>
409       </xs:extension>
410     </xs:complexContent>
411   </xs:complexType>
412
413   <!-- BufferLevel Type -->
414   <xs:complexType name="BufferLevelType">
415     <xs:attribute name="t" type="xs:dateTime" use="required"/>
416     <xs:attribute name="level" type="xs:unsignedInt" use="required"/>
417   </xs:complexType>
418
419   <!-- PlayList Type -->
420   <xs:complexType name="PlayListType">
421     <xs:complexContent>
422       <xs:extension base="SANDMessageType">
423         <xs:sequence>
424           <xs:element name="Playback" type="PlaybackType" minOccurs="1" maxOccurs="unbounded"/>
425         </xs:sequence>
426       </xs:extension>
427     </xs:complexContent>
428   </xs:complexType>
429
430   <!-- Playback Type -->
431   <xs:complexType name="PlaybackType">
432     <xs:sequence>
433       <xs:element name="RenderingPeriod" type="RenderingPeriodType" minOccurs="1" maxOccurs="unbounded"/
        >
434     </xs:sequence>
435     <xs:attribute name="start" type="xs:dateTime"/>
436     <xs:attribute name="mstart" type="xs:duration"/>
437     <xs:attribute name="starttype" type="StartType"/>
438   </xs:complexType>
439
440   <!-- Start Type -->
441   <xs:simpleType name="StartType">
442     <xs:restriction base="xs:string">
443       <xs:enumeration value="New playout request"/>
444       <xs:enumeration value="Resume from pause"/>
445       <xs:enumeration value="Other user request"/>
446       <xs:enumeration value="Start of a metrics collection period"/>
447     </xs:restriction>
448   </xs:simpleType>
449
450   <!-- RenderingPeriod Type -->
451   <xs:complexType name="RenderingPeriodType">
452     <xs:attribute name="representationid" type="StringNoWhitespaceType" use="required"/>
453     <xs:attribute name="subreplevel" type="xs:unsignedInt"/>
454     <xs:attribute name="start" type="xs:dateTime"/>
455     <xs:attribute name="mstart" type="xs:duration"/>
456     <xs:attribute name="duration" type="xs:duration"/>
457     <xs:attribute name="playbackspeed" type="xs:decimal"/>
458     <xs:attribute name="stopreason" type="StopReasonType"/>
459   </xs:complexType>
460
461   <!-- StopReason Type -->
462   <xs:simpleType name="StopReasonType">
463     <xs:restriction base="xs:string">
464       <xs:enumeration value="Representation switch"/>
465       <xs:enumeration value="Rebuffering"/>
466       <xs:enumeration value="User request"/>
467       <xs:enumeration value="End of Period"/>
468       <xs:enumeration value="End of content"/>
469       <xs:enumeration value="End of a metrics collection period"/>
470       <xs:enumeration value="Failure"/>
471     </xs:restriction>
472   </xs:simpleType>
473
474   <!-- String without white spaces, same as MPD schema -->
475   <xs:simpleType name="StringNoWhitespaceType">
476     <xs:restriction base="xs:string">
```

```
      <xs:pattern value="[^\r\n\t \p{Z}]*"/>
    </xs:restriction>
  </xs:simpleType>

  <!-- 14.35.1 Byte Ranges of RFC 2616 -->
  <xs:simpleType name="ByteRangeSetType">
    <xs:restriction base="xs:string">
      <xs:pattern value="((\d+-\d*)|(\d*-\d+))(,((\d+-\d*)|(\d*-\d+)))*"/>
    </xs:restriction>
  </xs:simpleType>

</xs:schema>
```

Listing B.1: SAND urn:mpeg:dash:sand:messageset:all:2016 message set XSD schema

B.2 SAND Message header extensions ABNF

Listing B.2 represents the header extensions format in Augmented Backus–Naur Form (ABNF) format.

```
sand-message-value = sand-object
sand-object = sand-attr-or-list *( "," sand-attr-or-list )
sand-attr-or-list = sand-attribute / sand-list
sand-list = "[" sand-object *( ";" sand-object ) "]"
sand-attribute = sand-attribute-name "=" sand-value
sand-attribute-name = STRING
sand-value = QUOTEDSTRING / QUOTEDURI / TOKEN / INT /
             BYTERANGE / DATETIME / integer-list
integer-list = "[" INT *( "," INT ) "]"
```

Listing B.2: SAND urn:mpeg:dash:sand:messageset:all:2016 message header extensions ABNF

Bibliography

[1] Cisco Visual Networking Index. "Forecast and Trends, 2017–2022". In: *Cisco Systems* (2018), pp. 1–7. URL: https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.html (visited on 07/16/2019).

[2] S Shunmuga Krishnan and Ramesh K Sitaraman. "Video stream quality impacts viewer behavior: inferring causality using quasi-experimental designs". In: *IEEE/ACM Transactions on Networking* 21.6 (2013), pp. 2001–2014.

[3] Tobias Hoßfeld et al. "Quantification of YouTube QoE via crowdsourcing." In: *ISM*. 2011, pp. 494–499.

[4] Te-Yuan Huang et al. "Confused, timid, and unstable: picking a video streaming rate is hard". In: *Proceedings of the 2012 internet measurement conference*. ACM. 2012, pp. 225–238.

[5] *Speedtest Global Index*. URL: https://www.speedtest.net/global-index (visited on 07/22/2019).

[6] *Netflix Internet Connection Speed Recommendations*. URL: https://help.netflix.com/nl/node/306 (visited on 07/22/2019).

[7] Anthony T. S. Ho; Shujun Li. *Handbook of Digital Forensics of Multimedia Data and Devices*. John Wiley & Sons, 2015. ISBN: 9781118757079. URL: https://books.google.be/books?id=pDU0DAAAQBAJ&pg=PT146#v=onepage&q&f=false (visited on 07/31/2017).

[8] *What is Matroska?* URL: https://www.matroska.org/technical/whatis/index.html (visited on 07/27/2019).

[9] *Intel Quick Sync Video*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/quick-sync-video/quick-sync-video-general.html (visited on 07/27/2019).

[10] *ITU Recommendation H.323*. URL: https://www.itu.int/rec/T-REC-H.323-199611-S/en/ (visited on 07/22/2019).

[11] *RTP: A Transport Protocol for Real-Time Applications*. URL: https://tools.ietf.org/html/rfc3550 (visited on 07/22/2019).

[12] C. Perkins. *RTP: Audio and Video for the Internet*. Kaleidoscope Series. Addison-Wesley, 2003. ISBN: 9780672322495. URL: https://books.google.be/books?id=OM7YJAy9%5C_m8C.

[13] *SIP: Session Initiation Protocol*. URL: https://tools.ietf.org/html/rfc3261 (visited on 07/22/2019).

[14] *The Secure Real-time Transport Protocol (SRTP)*. URL: https://tools.ietf.org/html/rfc3711 (visited on 07/22/2019).

[15] *RTP Payload Format for High Efficiency Video Coding (HEVC)*. URL: https://tools.ietf.org/html/rfc7798 (visited on 07/22/2019).

[16] *Shooting Around the Corner: The Problem of Real-time Services*. URL: https://www.ietfjournal.org/shooting-around-the-corner-the-problem-of-real-time-services/ (visited on 07/22/2019).

[17] John D Day and Hubert Zimmermann. "The OSI reference model". In: *Proceedings of the IEEE* 71.12 (1983), pp. 1334–1340.

[18] Victor Paulsamy and Samir Chatterjee. "Network convergence and the NAT/Firewall problems". In: *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. IEEE. 2003, 10–pp.

[19] Jon Postel. "Transmission control protocol". In: (1981).

[20] Roy Fielding et al. "Hypertext transfer protocol–HTTP/1.1". In: (1997).

[21] Tim Berners-Lee, Roy Fielding, Larry Masinter, et al. *Uniform resource identifiers (URI): Generic syntax*. 1998.

[22] Michael Thornburgh. "Adobe's RTMFP Profile for Flash Communication". In: (2014).

[23] Jay Hoffmann. *Flash And Its History On The Web*. Aug. 7, 2017. URL: `https://thehistoryoftheweb.com/the-story-of-flash/` (visited on 08/01/2019).

[24] Stefan Lederer. *MPEG-DASH Content Generation with MP4Box and x264*. URL: `https://bitmovin.com/mp4box-dash-content-generation-x264/` (visited on 07/28/2019).

[25] Joris Herbots. "Bendwit: Platform for preparing content for adaptive streaming". 2017.

[26] *HTTP Live Streaming*. URL: `https://developer.apple.com/streaming/` (visited on 07/28/2019).

[27] *Smooth Streaming*. URL: `https://www.iis.net/downloads/microsoft/smooth-streaming` (visited on 07/28/2019).

[28] *Adobe HTTP Dynamic Streaming (HDS)*. URL: `https://www.adobe.com/devnet/hds.html` (visited on 07/28/2019).

[29] ISO ISO. *IEC 23009-5: 2017 Information technology–Dynamic adaptive streaming over HTTP (DASH)–Part 5: Server and network assisted DASH (SAND)*.

[30] ISO ISO. "ISO/IEC 23009-1: 2014: Information technology–Dynamic adaptive streaming over HTTP (DASH)–Part 1: Media presentation description and segment formats". In: *Geneva, Switzerland: International Organization for Standardization* (2014).

[31] Henry Thompson and Chris Lilley. "XML Media Types". In: (2014).

[32] Subodh Gangan. "A review of man-in-the-middle attacks". In: *arXiv preprint arXiv:1504.02115* (2015).

[33] Takeshi Imamura et al. "XML encryption syntax and processing version 1.1". In: *W3C, Recommendation* (2013).

[34] *Cross-Origin Resource Sharing (CORS)*. URL: `https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS`.

[35] ISO ISO. *IEC 23009-5: 2017 Information technology–Dynamic adaptive streaming over HTTP (DASH)–Part 5: Server and network assisted DASH (SAND) Amendment 1: Improvements on SAND messages*. 2019.

[36] *Media Source Extensions*. URL: `https://w3c.github.io/media-source/`.

[37] Kevin Spiteri, Rahul Urgaonkar, and Ramesh K Sitaraman. "BOLA: Near-optimal bitrate adaptation for online videos". In: *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE. 2016, pp. 1–9.

[38] P Leach, Michael Mealling, and Rich Salz. "RFC 4122: A universally unique identifier (UUID) URN namespace". In: *Proposed Standard, July* (2005).

[39] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[40] *tc-tbf - Linux man page*. URL: `https://linux.die.net/man/8/tc-tbf`.

[41] *tc - Linux man page*. URL: `https://linux.die.net/man/8/tc`.

[42] *Guidelines for Implementation: DASH-IF SAND Interoperability*. URL: `https://dashif.org/docs/DASH-IF-SAND-IOP-v1.0.pdf`.

[43] Stefan Pham et al. "Evaluation of shared resource allocation using SAND for ABR streaming". In: *Proceedings of the 10th ACM Multimedia Systems Conference*. ACM. 2019, pp. 165–174.

[44] Jan Willem Kleinrouweler, Britta Meixner, and Pablo Cesar. "Improving video quality in crowded networks using a DANE". In: *Proceedings of the 27th Workshop on Network and Operating Systems Support for Digital Audio and Video*. ACM. 2017, pp. 73–78.

[45] Saamer Akhshabi, Ali C Begen, and Constantine Dovrolis. "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP". In: *Proceedings of the second annual ACM conference on Multimedia systems*. ACM. 2011, pp. 157–168.

[46] Divyashri Bhat et al. “Network assisted content distribution for adaptive bitrate video streaming”. In: *Proceedings of the 8th ACM on Multimedia Systems Conference*. ACM. 2017, pp. 62–75.

[47] Parikshit Juluri, Venkatesh Tamarapalli, and Deep Medhi. “SARA: Segment aware rate adaptation algorithm for dynamic adaptive streaming over HTTP”. In: *2015 IEEE International Conference on Communication Workshop (ICCW)*. IEEE. 2015, pp. 1765–1770.

[48] Robert Ricci, Eric Eide, and CloudLab Team. “Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications”. In: *; login:: the magazine of USENIX & SAGE* 39.6 (2014), pp. 36–38.

[49] Emmanuel Thomas et al. “Applications and deployments of server and network assisted DASH (SAND)”. In: (2016).

[50] *Github discussion: Comparison between SHAKA player and dash.js player*. URL: `https://github.com/google/shaka-player/issues/1351`.

[51] Nick McKeown et al. “OpenFlow: enabling innovation in campus networks”. In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.