Simulation Framework for HTTP-Based Adaptive Streaming Applications

Harald Ott
Technische Universität Berlin
Einsteinufer 25
10587 Berlin, Germany
haraldott@win.tu-berlin.de

Konstantin Miller
Technische Universität Berlin
Einsteinufer 25
10587 Berlin, Germany
konstantin.miller@tu-berlin.de

Adam Wolisz
Technische Universität Berlin
Einsteinufer 25
10587 Berlin, Germany
adam.wolisz@tu-berlin.de

ABSTRACT

The popularity of Internet-based video services has significantly increased over the past years. The de facto standard technology for Internet-based Video on Demand is HTTP-Based Adaptive Streaming (HAS), which is also increasingly used for live services. A core component of a HAS client is the adaptation algorithm, which dynamically adjusts the video representation to the network conditions. Meanwhile, there exists a large body of work on adaptation algorithms. Unfortunately, many experimental studies lack a thorough performance evaluation. Often, the reason is the use of an unrealistic network environment, or incomparability of results with other studies, or a too narrow subset of evaluated parameter configurations. We argue that a simulative approach can help resolving these issues by requiring less efforts to set up a realistic network environment, by assisting to reproduce an experiment, and by allowing to parallelize simulations, and potentially run them faster than in real time. The contribution of the present work is a design and implementation of a simulation model for a HAS-based application, including both the client and a server side. It has a clean modularized structure allowing for an easy integration of different adaptation algorithms. The client behavior is defined by a Finite-State Machine that can easily be extended to include additional functionality. Moreover, the model provides extensive logging functionality for monitoring the Quality of Experience (QoE). We integrate three state-of-the-art algorithms into the model: FESTIVE, PANDA, and TOBASCO2. We demonstrate the usefulness of the model by running a set of experiments using a simulated indoor Wi-Fi environment.

CCS CONCEPTS

•Information systems →Multimedia streaming; •Networks →Network simulations;

KEYWORDS

Adaptive Streaming, ns-3, MPEG-DASH, Simulation Model

ACM Reference format:
Harald Ott, Konstantin Miller, and Adam Wolisz. 2017. Simulation Framework for HTTP-Based Adaptive Streaming Applications. In *Proceedings of the 2017 workshop on ns-3, Porto, Portugal, June 2017 (WNS3 2017),* 8 pages.

1 INTRODUCTION

Humans are predominantly visually oriented creatures. We often rely on visual input as our main source of information. Therefore, it is by no means surprising that with the rise of mobile devices with high-resolution screens, these devices are increasingly used to display video content. However, watching locally stored content is just one limited use case. The broad availability of high-speed wireless access networks has led to a massive proliferation of mobile applications involving video transmission. Very diverse forms of video-supported entertainment, information dissemination, education, monitoring, or communication are becoming increasingly popular. As a consequence, by 2020, video traffic is predicted to account for over 80% of the total consumer traffic on the Internet [6].

All Internet-based video services face the challenge to serve an extremely heterogeneous set of clients under extremely heterogeneous network conditions. Both the user on an otherwise idle high-speed fixed-line connection, and the fast-moving user on the edge of a busy mobile cell, should be able to receive the video representation which is most appropriate for their conditions. This has led to a wide adoption of adaptive streaming technologies, such as HTTP-Based Adaptive Streaming (HAS), also known as Dynamic Adaptive Streaming over HTTP (DASH) [21], that is capable of providing a smooth viewing experience in dynamic environments. Prominent examples include MPEG-DASH [1] and Apple HTTP Live Streaming (HLS) [19].

Due to their indisputable commercial success, Hypertext Transfer Protocol (HTTP)-based streaming technologies have attracted a considerable amount of attention from the research community. A large number of novel adaptation approaches have been proposed. Unfortunately, many of the studies have one weak aspect, which is the performance evaluation. Three most common problems are: (i) use of an unrealistic network environment, (ii) lack of comparability with other studies, and (iii) focus on a narrow subset of possible parameter configurations.

An example for the first problem is an overly simplified network environment consisting of a dedicated link with a piecewise-constant throughput function, with few throughput transitions per minute or in total. Such a setting is often emulated using the Linux traffic control tool `tc`[1], or DummyNet [4]. An example for

[1]http://www.lartc.org/manpages/tc.txt

the second problem is the use of a test bed environment that is sophisticated enough to confront a streaming client with realistic network conditions, but that can hardly be reproduced by a different research team, due to the lack of a detailed description, or due to a disproportionate amount of required effort. An example for the third problem is a performance evaluation, in which the baseline approaches, or the proposed approach itself, are evaluated using only a single parameter configuration.

We argue that a simulative approach can help resolve these issues. First, it permits to create a realistic network environment with significantly less effort. In advanced network simulators, such as ns-3[2], there exist models for state-of-the-art network technologies, including the Internet protocol stack, Wi-Fi and Long-Term Evolution (LTE) networks, physical environments, such as the buildings model, and various mobility models. Second, network environments can be reproduced precisely. And finally, it allows creation of parallelized simulations, and sometimes, run them faster than in real time.

In this work, we present our design and implementation of a simulation model for a HAS application. The model consists of a server module and a client module. Due to the receiver-driven nature of HAS, most of the functionality is contained in the client. It has a clean modularized structure that allows for an easy integration of adaptation algorithms. At the core of the client is a Finite-State Machine (FSM) that determines its behavior, and that can, if required, be extended to include additional functionality. The model provides extensive logging functionality, necessary to analyze the behavior of adaptation algorithms and evaluate their performance.

We integrated three adaptation algorithms into the developed model: FESTIVE [12], PANDA [13], and TOBASCO2, which is an extension of TOBASCO [18]. To demonstrate the developed functionality, we run a series of experiments using these three algorithm in an indoor Wi-Fi environment, using the Wi-Fi model[3], and the buildings model[4] of ns-3.

The paper is structured as follows. Section 2 outlines the related work. Section 3 provides a brief overview of the core principles of HAS. The components of the developed model are presented in Section 4. Section 5 describes the setting and the results of the performed experiments. Finally, Section 6 concludes the paper.

2 RELATED WORK

The last years have seen a vivid activity in developing adaptation algorithms for HAS clients. Various approaches have been proposed that are based on control theory [24], Markov decision processes [3], machine learning [7], and heuristics that are based on the client's playback buffer level and throughput estimations [12, 18], to name just a few. While most of them target Video on Demand (VoD), some of them address live streaming [16]. The demand to evaluate these algorithms against each other, allowing to select the best approach for a given use case, is greater than ever. Consequently, several studies focus on the development of experimentation platforms for HAS.

De Cicco et al. [8] present A Tool for rApid Prototyping of Adaptive Streaming Algorithms (TAPAS), which is a streaming client written in Python that allows to design and carry out experimental performance evaluations of adaptation algorithms, without having to implement a complete client. TAPAS supports both DASH and HLS. It has been designed to minimize Central Processing Unit (CPU) and memory usage so that multiple experiments can be performed in parallel on a single machine. In contrast, we focus on a simulation platform, which allows to more efficiently perform a large number of experiments, in a reproducible way, using realistic network environments.

Fouda et al. [9] and Quinlan et al. [20] present emulation platforms that allow to evaluate streaming solutions over LTE networks. However, since emulations run in real time, they do not offer the scalability of simulations. Moreover, the models are not released as open source and no documentation or description of the design is available.

Vergados et al. [23] have developed an open source simulation model for HAS. Unfortunately, the authors do not present the design of the developed software or any documentation, which would allow to assess its flexibility and extensibility.

In order to provide comparability among different adaptation approaches, other authors resort to direct comparison by testing multiple algorithms in the same test bed or emulated environment. Unfortunately, such studies are inevitably limited in the amount of considered approaches, network environments, and parameter configuration. Timmerer et al. [22] performed one of the most broad comparisons by evaluating 10 adaptation algorithms in an emulated environment. The authors used one throughput trajectory, and one set of configuration parameters per algorithm. They used two segment sizes: 2 seconds and 10 seconds. Another comparison was performed by Akhshabi et al. [2], who evaluated 3 clients.

Yet another approach to achieve comparability is to evaluate streaming clients against optimal performance [11, 17, 26]. Such approaches typically suffer from high computational complexity, which restricts their applicability to realistic scenarios. Moreover, optimal performance is often achieved using additional information such as the knowledge of the future throughput, which also limits the applicability of results.

3 HTTP-BASED ADAPTIVE STREAMING

Before we introduce our developed simulation model, we describe the main principles of HAS. HAS is currently one of the most successful streaming technologies for VoD. Among the first works proposing HAS-based approaches are [5, 14]. However, it took almost a decade until HAS has seen wide deployment and commercial success. HAS flavors such as HLS [19] or Microsoft Smooth Streaming [25] has been used by various streaming services, and have been supported by most operating systems and browser environments. In 2011, the open standard MPEG-DASH [1, 21] was created to facilitate the interoperability.

In a HAS system, the video content is encoded in several representations that may differ w.r.t. various characteristics such as the resolution, frame rate, compression technology, compression rate, video format, etc. Each representation is split into segments, typically containing several seconds of video data. Each segment

[2]http://www.nsnam.org
[3]http://www.nsnam.org/docs/models/html/wifi.html
[4]http://www.nsnam.org/docs/models/html/buildings.html

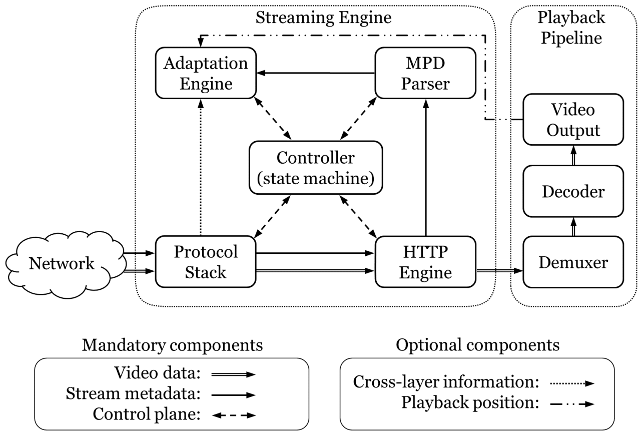

Figure 1: Main functional blocks of a HAS client

starts with a random access point of the stream such that segments from different representations can be concatenated to obtain a valid video file. Consequently, switching the representation is feasible on each segment boundary.

The client issues a series of HTTP GET or GET RANGE requests to download the segments, selecting a representation for each of them from the set of available representations. After a segment is downloaded, it is stored in the playback buffer until the playback of the previous segment has been completed. If the download is not completed in time, the playback is halted. To avoid playback interruptions, a client strives to keep the playback buffer at a level that is high enough to mitigate the impact of throughput drops and link outages.

At the core of a HAS client is an adaptation algorithm that selects the representation for each of the downloaded video segment. The adaptation algorithm is not part of the MPEG-DASH standard, and thus it has been subject to intensive research efforts over the last years. To meet its adaptation decisions, an adaptation algorithm is using the information it can acquire from its environment. Most often this information includes the dynamics of the past throughput, with a level of details depending on the underlying platform, and the dynamics of the level of the playback buffer. More sophisticated approaches leverage cross-layer information from lower layers of the protocol stack or other network entities, such as a local network controller. They may also leverage context information, such as the location, mobility pattern, or available sensor data.

The adaptation algorithm used by a streaming client has a strong impact on the Quality of Experience (QoE). If, e.g., the adaptation decisions follow the short-term changes of the network throughput, the resulting average video quality may be quite high but the overall QoE will decrease due to a high rate of quality fluctuations. On the other hand, a conservative approach might reduce the risk of playback interruptions due to video segments missing their playback deadline, but at the same time decrease the overall video quality. Thus, a good adaptation strategy needs to maintain a balance between several trade offs.

A functional diagram illustrating the main components of a HAS client is provided in Figure 1.

4 SIMULATION MODEL

In this Section, we describe in details the components of the developed simulation model.

4.1 Overview

Our HAS simulation model has been designed with two main objectives in mind. First, it had to have a clean modularized structure with logically separated functional blocks. Second, we aimed at keeping the complexity of the model as low as possible, implying clean and simple Application Programming Interfaces (APIs). Moreover, users should not have to modify the code in order to integrate their own adaptation algorithms. An overview of the model, summarizing the functionality of the modules, is given in Figure 2.

The model consists of three main functional blocks: client, adaptation algorithm, and server. The **client module** contains most of the functionality of the model. It collects and maintains the information required by the adaptation algorithm, such as the buffer level, throughput information, and meta information about the streamed video. The client module is derived from the `Application` class of ns-3. The **adaptation algorithm module** contains the actual adaptation logic. It inherits from the base class `AdaptationAlgorithm` and must implement the method `GetNextRep()` returning the selected video quality, and, potentially, an inter-request delay. Finally, since HAS is client-driven, the **server module** has very little functionality. It listens for incoming Transmission Control Protocol (TCP) connections, and then serves the requests by returning the requested amount of bytes. The server module is also derived from the `Application` class of ns-3.

The latest version of the code and the documentation can be downloaded at https://github.com/haraldott/DASH-NS3.

4.2 Client Module

The main tasks of the client module include:

- Processing input arguments, such as stream meta information.
- Instantiating of an adaptation algorithm object.
- Running the streaming session by establishing a TCP connection to the server, and requesting segments from the server.
- Simulating the playback.
- Monitoring the buffer level and the throughput, providing this information to the adaptation algorithm.
- Logging.

The behavior of the client is defined by a FSM that we proposed in our previous work [15]. It is depicted in Figure 3, and the corresponding states, events, and actions are defined in Tables 1, 2, and 3, respectively. This state machine is sufficient to reflect most types of VoD clients. A state machine for live streaming is provided in our previous work [15]. In the following, we will briefly describe the individual states, and the possible transitions—including events and actions—among them.

4.2.1 States. The states are presented in Table 1.

In the initial state S_0, the client creates an adaptation algorithm object of the required type, reads the stream meta information from

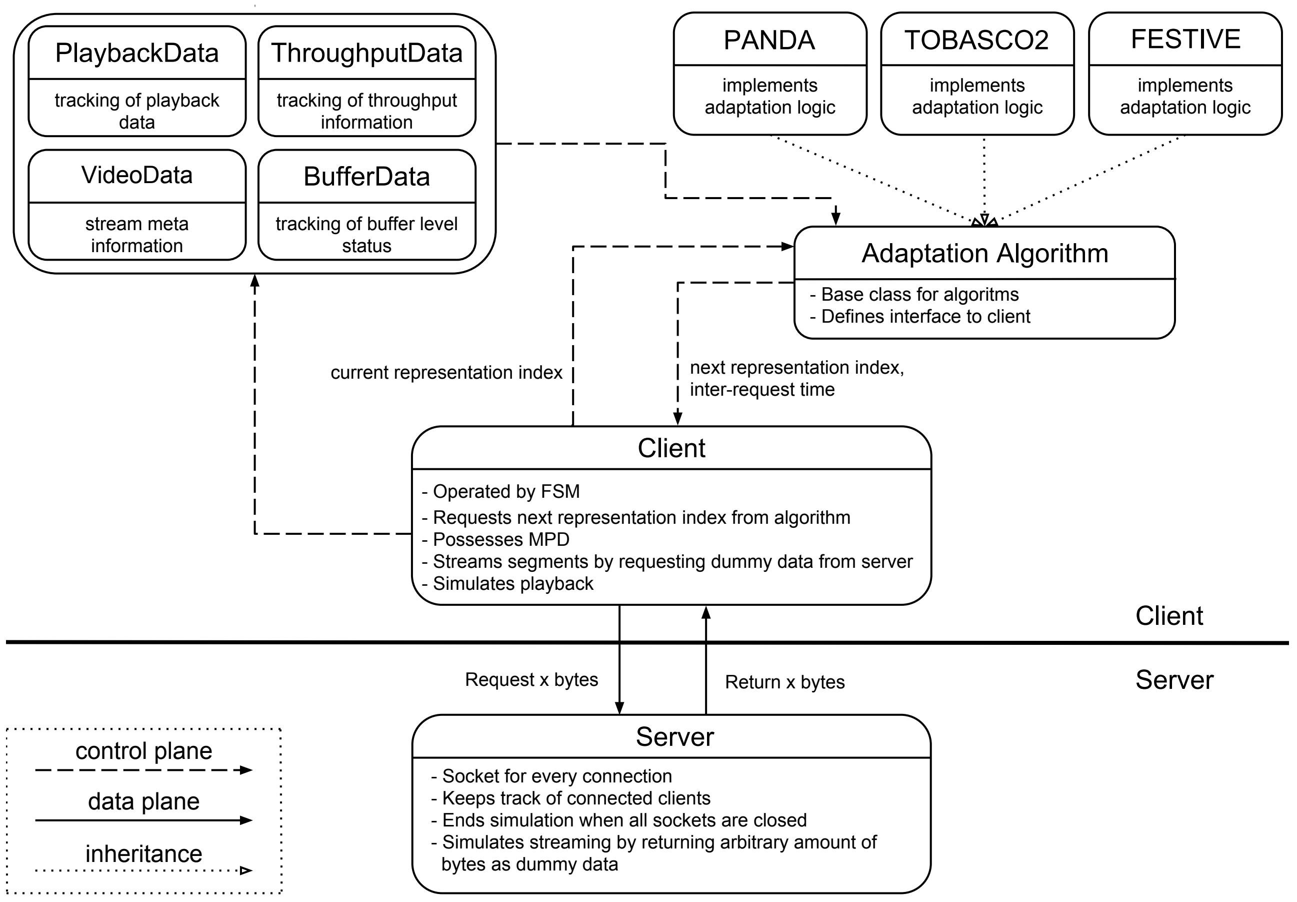
PlaybackData
tracking of playback data
ThroughputData
tracking of throughput information
VideoData
stream meta information
BufferData
tracking of buffer level status
PANDA
implements adaptation logic
TOBASCO2
implements adaptation logic
FESTIVE
implements adaptation logic
Adaptation Algorithm
- Base class for algoritms
- Defines interface to client
current representation index
next representation index, inter-request time
Client
- Operated by FSM
- Requests next representation index from algorithm
- Possesses MPD
- Streams segments by requesting dummy data from server
- Simulates playback
Client
Server
Request x bytes
Return x bytes
Server
- Socket for every connection
- Keeps track of connected clients
- Ends simulation when all sockets are closed
- Simulates streaming by returning arbitrary amount of bytes as dummy data
control plane
data plane
inheritance

Figure 2: Functional diagram of the simulation model

a file, and initializes log files. It also calls the method `GetNextRep()` which selects the representation for the first segment to download. The method `GetNextRep()` also returns the duration of a potential inter-request delay, however, prior to requesting the first segment of the streaming session, we expect that any reasonable adaptation algorithm returns the value 0.

At the end of the simulation, the client makes the transition to the terminal state S_T, in which the connection to the server and the log files are closed. Most of the time the client spends circulating between the states S_D, S_P, and S_{DP}.

In state S_D, the client is downloading, while no playback takes place. This state occurs at the very beginning of the streaming session, when the client is downloading the first segment(s) and has nothing to play out. This is often called pre-buffering, initial delay, or start-up delay. State S_D also occurs when the playback buffer runs empty because of a mismatch of the video bit rate and available throughput. This is often called buffer underrun or a stalling event.

In state S_P, the client is only playing but not downloading. This state occurs during an inter-request delay, which may be introduced by the client to prevent the buffer from exceeding a given threshold. Such a threshold may result, on the one hand, from memory limitations (note that 10 minutes of video at 10 Mbps require approximately 750 MB of storage space.) On the other hand, however, such a threshold may be set by the service provider in order to avoid wasting resources in case the user quits the streaming session prematurely.

Finally, in state S_{DP} the client is both downloading and playing, which is the state in which the client typically spends most of the time.

4.2.2 *State Transitions.*

The events that drive the state transitions are presented in Table 2, and the accompanying actions in Table 3. In the following, we briefly describe the transitions, mentioning some of the actions performed during these transitions. All transitions are accompanied by logging actions that are described in Section 4.6, and thus are not explicitly mentioned here.

The only feasible event in S_0 is e_0, which is a manually created event that signals that the model is initialized, including establishing a TCP connection to the server, and selecting the representation for the first segment. It results in the transition $S_0 \xrightarrow{e_0} S_D$. The associated action is issuing the request to the server.

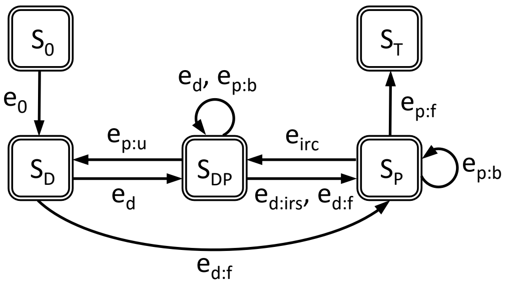

Figure 3: State diagram of the streaming client

Table 1: Table of states

S_0	initial state
S_D	downloading only
S_{DP}	downloading and playback
S_P	playback only
S_T	terminal state

In the state S_D, once a segment is completely downloaded, and if it is not the last segment in the stream, the client determines the representation for the next segment by calling the method `GetNextRep()` of the adaptation algorithm module. As in S_0, in S_D the adaptation algorithm is not expected to generate any inter-request delay (since S_D corresponds to pre-buffering or re-buffering). This corresponds to event e_d, which leads to the transition $S_D \xrightarrow{e_d} S_{DP}$. This transition is associated with the action to (re-)start the playback. Playback management is performed by the `PlaybackHandle()` method of the client module. When the playback is started, it decrements the level of the playback buffer by one segment. It then sets the timer to the next call of `PlaybackHandle()` to the duration of the video contained in one segment. Another action associated with the transition $S_D \xrightarrow{e_d} S_{DP}$ is the start of the new segment download.

The only other possible transition from S_D is $S_D \xrightarrow{e_{d:f}} S_P$. Event $e_{d:f}$ represents the end of the download of the last segment in the stream. Since no more segments can be downloaded, the client proceeds to the state S_P. Similar to the transition $S_D \xrightarrow{e_d} S_{DP}$, this transition (re-)starts the playback by calling the `PlaybackHandle()` method.

In the state S_{DP}, there are 5 feasible events: e_d, $e_{d:f}$, $e_{d:irs}$, $e_{p:u}$, and $e_{p:b}$. The event e_d, which corresponds to the end of the download of a segment, which is not the last segment in the stream, induces the self-transition $S_{DP} \xrightarrow{e_d} S_{DP}$. Note that e_d is generated if the adaptation algorithm (during the call to `GetNextRep()` which is performed immediately after the download is completed) decides that no inter-request delay shall be introduced. The generated action is the start of the new segment download. The `PlaybackHandle()` method need not be called since the playback is already running. The event $e_{d:f}$ corresponds to the end of the download of the last segment in the stream. It induces the transition $S_{DP} \xrightarrow{e_{d:f}} S_P$. During this transition, no further actions have to be performed. The event $e_{d:irs}$ occurs if the adaptation algorithm introduces an inter-request delay before the download of the subsequent segment. This event induces the transition $S_{DP} \xrightarrow{e_{d:irs}} S_P$. During this transition, a timer is set to the duration of the inter-request delay.

The two new events that can occur in S_{DP} are $e_{p:b}$, and $e_{p:u}$. $e_{p:b}$ occurs when the timer set by the `PlaybackHandle()` methods expires, representing the end of playback of a segment, and if the download of the next segment to be played has been completed (the b in $e_{p:b}$ stands for "buffered"). This event induces the self-transition $S_{DP} \xrightarrow{e_{p:b}} S_{DP}$, which starts the playback of the new segment, and re-sets the playback timer. The corresponding event for the case that the next segment has not yet been completely downloaded is $e_{p:u}$ (where u stands for "unbuffered"). In this case, the client proceeds to the state S_D (re-buffering).

Finally, in the state S_P, there are three feasible events: $e_{p:b}$, $e_{p:f}$, and e_{irc}. $e_{p:b}$ represents the end of playback of a segment, given that the subsequent segment has already been completely downloaded. It induces the self-transition $S_P \xrightarrow{e_{p:b}} S_P$ (recall that S_P represents an inter-request delay, or the playout of the final segments, when all segments of the video have been completely downloaded). $e_{p:f}$ represents the end of playback of the final segment, and thus of the video, and induces the transition to the terminating state $S_P \xrightarrow{e_{p:f}} S_T$, closing the connection to the server. Finally, e_{irc} occurs when the timer set for the inter-request delay expires ("irc" stands for "inter-request completed"), inducing the transition $S_P \xrightarrow{e_{irc}} S_{DP}$, during which the playback is re-started.

Note that for the sake of completeness, we have included selecting the video quality into the set of actions presented in Table 3. This is not entirely correct, since video quality selection is performed *prior* to a state transition. Moreover, the outcome of the video quality selection effects the performed transition. For example, upon completing a segment download in state S_{DP}, either event e_d or $e_{d:irs}$ may be generated, depending if the adaptation algorithm introduces an inter-request delay or not.

4.3 Adaptation Algorithm Module

The adaptation algorithm module consists of the abstract class `AdaptationAlgorithm`. All adaptation algorithms must inherit from it, and thus implement the method `GetNextRep()`, which defines the interface towards the client module.

The constructor of the `AdaptationAlgorithm` receives from the client module references to objects that hold information that may be used by the implemented adaptation algorithm: on throughput, on the buffer level, on the playback, and on stream meta data. Further extensions may be performed, for example, for algorithms that require cross-layer information.

The `GetNextRep()` method is called immediately when the download of a segment is completed. Based on the (dynamically updated) information contained in the corresponding objects, it determines the representation for the next segment to download, and the duration of the inter-request delay (which can be 0). Both values are aggregated in the `AlgorithmReply` structure returned to the client.

Table 2: Table of events

e_0	start of the streaming session
e_d	download of a segment completed, next download starts immediately
$e_{d:irs}$	download of a segment completed, next download is delayed
$e_{d:f}$	download of the final segment completed
$e_{p:b}$	playback of a segment completed, next segment is buffered
$e_{p:u}$	playback of a segment completed, next segment is not buffered
$e_{p:f}$	playback of the final segment completed
e_{irc}	inter-request delay ends

Table 3: Table of actions

$S_0 \xrightarrow{e_0} S_D$	select video quality, request first segment, logging
$S_D \xrightarrow{e_d} S_{DP}$	select video quality, request next segment, (re-)start playback, logging
$S_{DP} \xrightarrow{e_d} S_{DP}$	select video quality, request next segment, logging
$S_{DP} \xrightarrow{e_{p:b}} S_{DP}$	start playback of next segment
$S_{DP} \xrightarrow{e_{p:u}} S_D$	logging
$S_{DP} \xrightarrow{e_{d:irs}} S_P$	select video quality, schedule next request, logging
$S_{DP} \xrightarrow{e_{d:f}} S_P$	logging
$S_P \xrightarrow{e_{p:b}} S_P$	start playback of next segment, logging
$S_P \xrightarrow{e_{irc}} S_{DP}$	request next segment, logging
$S_P \xrightarrow{e_{p:f}} S_T$	disconnect from server, logging

4.4 Server Module

The two main tasks of the server module are to manage the connected clients, and to serve the clients' requests.

To simplify the information flow, we keep the information about the segment sizes, which needs to be provided to the simulation model as part of the stream meta information, at the client module. The servers receives the number of bytes it has to send as part of each incoming request. Thus, it does not have to know anything about the video.

Initially, the ns-3 framework calls the `StartApplication()` method, which sets the callbacks for the `Accept`, `Close`, and `Receive` socket calls.

After a client is done requesting segments from the server, it closes the socket. This causes the client to be removed from the list of connected clients. When the last client closes the socket and the list of connected clients is therefore empty, the simulation is terminated by the server.

4.5 Input Parameters

The input of the model includes stream meta information, such as the segment size matrix, and the segment duration. It also includes the type of the adaptation algorithm to be used. In addition, a simulation ID is used to identify the produced output files, and to initialize the random number generator. The duration of the simulation is provided implicitly by the segment sizes matrix—the simulation terminates when all clients have played out the complete video.

4.6 Logging

All logging activity is performed by the client module. It can be roughly grouped into four categories: download, buffer level, playback, and adaptation.

For each downloaded segment, the client logs its index, the time when the request is sent, the time when the first byte is received, the time when the download is completed, and the number of downloaded bytes. In addition, the clients logs the download progress on the packet level, recording the time and size of each received TCP packet. After each segment download, the clients logs the buffer level.

For each played segment, the client logs its index, the start of the playback, and the quality level (representation index).

Finally, for each performed adaptation decision, the client logs the segment index, the selected representation, the time of the decision, and the reason for the decision. The latter is a numerical identifier corresponding to the specific branch of the decision process, performed by the algorithm.

The logged values allow to estimate the QoE, which is mainly influenced by the duration of playback interruptions (total re-buffering time), average video quality, number of quality transitions, and the initial (or start-up) delay.

5 EXPERIMENTS

To demonstrate the functionality of the developed simulation model, we perform experiments using three state-of-the-art adaptation algorithms that we have integrated with the developed model. Note that the focus of the present work is on the simulation model, rather than on streaming client evaluation. Consequently, the presented experiments lack an exploration of parameter space of the considered clients. They also lack the diversity of network environments that should be considered by a proper evaluation campaign.

5.1 Test Video

Since no playback is performed during the simulation, no actual video data is required. Still, to receive valid results, it is necessary to use realistic segment sizes, ideally from a real DASH-encoded video. The video used in our work is a 442 second long excerpt from the movie Sintel[5]. It contains 221 segments (2 seconds per segment), and is encoded in eight representations, whose average bit rates range from 0.088 Mbps to 20.5 Mbps. The bit rates for the representations are selected so as to obtain a roughly linear

[5]https://durian.blender.org/

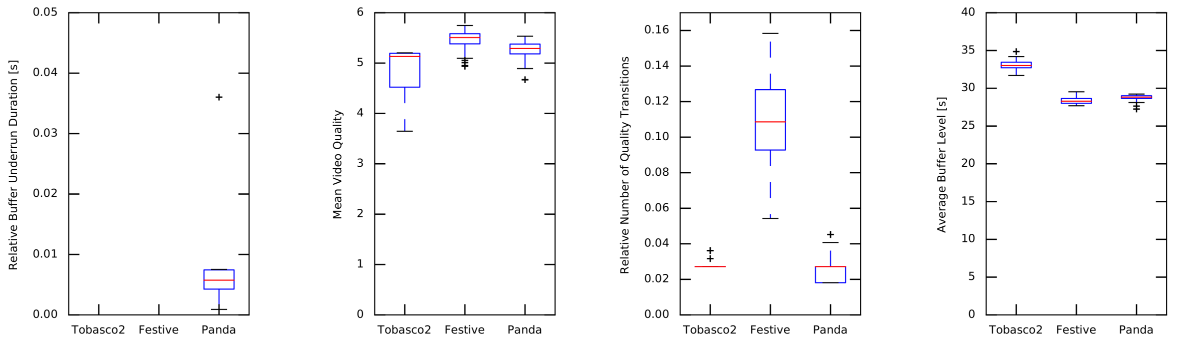

Figure 4: Results from measurements with 5 clients

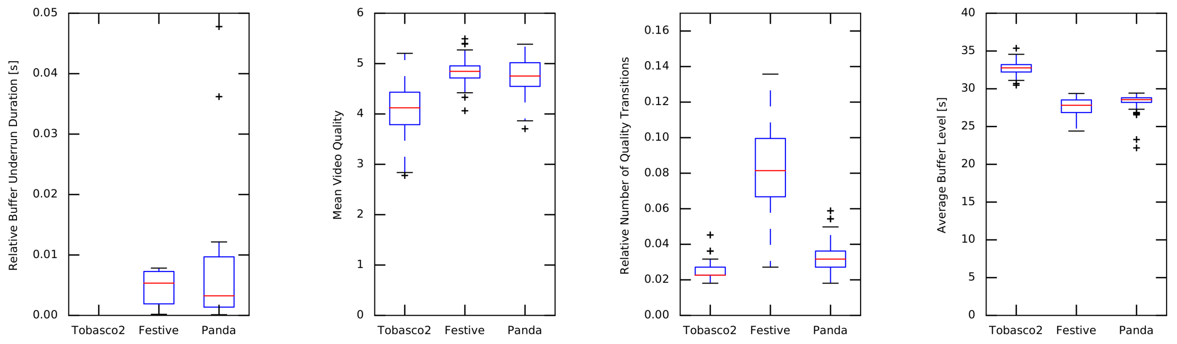

Figure 5: Results from measurements with 10 clients

increase of the video quality, in terms of the Peak Signal-to-Noise Ratio (PSNR), which allows us to use the representation index as an indicator of the video quality. We used Variable Bit Rate (VBR) encoding.

5.2 Network Environment

Our goal was to run the measurements in a realistic environment. To achieve this, we used the Buildings Model provided with ns-3 to create a building of 30 x 18 x 18 meters, with 6 floors, and 8 x 3 rooms on each floor. The Access Point (AP) position was fixed at (1, 1, 1), which corresponds to the left lower room on the first floor. Clients were placed randomly within the building, using the random position allocator provided by the ns-3 framework. We used the IEEE 802.11n Wi-Fi mode, with configuration values taken from [10]. Finally, we increased the TCP send and receive buffer sizes to 524288 bytes, and the Maximum Transmission Unit (MTU) to 1446 bytes.

5.3 Results

All experiments were repeated 15 times with randomly selected client positions. The number of clients per experiment was varied from 1 to 10. In the following, we exemplarily present results for 5 and 10 clients.

One of the most important factors influencing the QoE is the amount of playback interruptions. The left-most subfigure in Figures 4 and 5 depicts the distribution of total durations of playback interruptions per client, over clients and over runs, relative to the duration of the video. For example, given a video duration of 442 seconds, a value of 0.01 corresponds to 4.42 seconds spent in re-buffering during the streaming session. We observe that TOBASCO2 successfully avoids interruptions, partially due to its aggressive reduction of the video quality if the buffer level falls below a critical threshold. The amount of interruptions for the other two algorithms increases with the number of clients, as expected, due to the increased scarcity of resources.

Another factor which has a high influence on the QoE is the video quality. The second column in Figures 4 and 5 shows the distribution of average video quality achieved by a client, over clients and over runs. To estimate the average video quality, we use the average representation index over all segments in a streaming session. Note that, as described in Section 5.1, we encoded the video such that the representation index is proportional to the PSNR. We observe that TOBASCO2 offers a slightly lower video quality, than the other two algorithms. The rationale behind this behavior is to slightly sacrifice quality in order to avoid playback interruptions and excessive quality transition.

Multiple studies have shown that quality transitions may annoy the user and decrease the overall QoE. Consequently, we included the number of quality transition into the set of studied performance metrics. It is depicted in the third column of Figures 4 and 5. More precisely, the figures show the distribution of the relative number of quality transitions per client, over clients and over runs. Thus, a value of 0.01 corresponds to one segment in 100 played in a different quality than its predecessor. Given a segment duration of 2 seconds, this implies one quality transition per 200 seconds on average. We observe that FESTIVE is particularly prone to excessive quality transitions, while TOBASCO2 and PANDA offer similar performance.

Finally, we include the average buffer level into the set of performance metrics, in order to verify if the comparison of the selected algorithms is a fair one or if any of them is using a significantly higher average buffer level to achieve a higher performance. It turns out that all three algorithms exhibit similar values, with TOBASCO2 showing a slightly higher value. It is worth noting that a low average buffer level may as well be an indicator of low performance if an algorithm systematically fails to adjust the video bit rate to the throughput, resulting in low buffer levels, and even buffer underruns.

6 CONCLUSION

In the present work, we addressed the problem of developing a simulation model for HAS-based streaming clients. We adopted a modular and flexible design that allows to easily integrate adaptation algorithms, and extend the model by adding new functionality. At the core of the model is a FSM that reflects the main operation principles of an adaptive streaming client.

To demonstrate the model, we integrated three state-of-the-art adaptation algorithms. We evaluated them using the Wi-Fi model of ns-3, and the building model, which facilitate simulating wireless indoor channels.

ACKNOWLEDGMENTS

This work was supported by the ZIM programme of the German Federal Ministry for Economic Affairs and Energy (BMWi) in the context of the research project ANOPAS.

REFERENCES

[1] I. 23009. 2012. *Dynamic Adaptive Streaming over HTTP (DASH), First Edition.* International Standard.

[2] S. Akhshabi, A. C. Begen, and C. Dovrolis. 2011. An Experimental Evaluation of Rate-Adaptation Algorithms in Adaptive Streaming over HTTP. In *Proc. of ACM Multimedia Systems Conference (MMSys).*

[3] A. Bokani, M. Hassan, and S. Kanhere. 2013. HTTP-Based Adaptive Streaming for Mobile Clients using Markov Decision Process. In *Proc. of International Packet Video Workshop (PV).* San Jose, CA.

[4] M. Carbone and L. Rizzo. 2010. Dummynet Revisited. *ACM SIGCOMM Computer Communication Review* 40, 2 (2010), 12.

[5] S. Carmel, T. Daboosh, E. Reifman, N. Shani, Z. Eliraz, D. Ginsberg, E. Ayal, and K. Saba. *Network Media Streaming.* Patent No. 6,389,473. Filed 24 Mar 1999, Issued 14 May 2002.

[6] Cisco Systems, Inc. 2016. *Cisco Visual Networking Index: Forecast and Methodology, 2015 - 2020.* White Paper.

[7] M. Claeys, S. Latré, J. Famaey, and F. De Turck. 2014. Design and Evaluation of a Self-Learning HTTP Adaptive Video Streaming Client. *IEEE Communications Letters* 18, 4 (2014), 716–719.

[8] L. De Cicco, V. Caldaralo, V. Palmisano, and S. Mascolo. 2014. TAPAS: a Tool for rApid Prototyping of Adaptive Streaming algorithms. In *Proc. of the 2014 Workshop on Design, Quality and Deployment of Adaptive Video Streaming (Video Next).*

[9] A. Fouda, A. N. Ragab, A. Esswie, M. Marzban, A. Naser, M. Rehan, and A. S. Ibrahim. 2014. Real-Time Video Streaming over ns3-based Emulated LTE Networks. *International Journal of Electronics Communication and Computer Technology* 4, 3 (2014), 659–663.

[10] P. Gawowicz, S. Zehl, A. Zubow, and A. Wolisz. 2016. *NxWLAN: Neighborhood eXtensible WLAN.* Technical Report TKN-16-002. Technische Universität Berlin.

[11] D. Jarnikov and T. Özçelebi. 2011. Client Intelligence for Adaptive Streaming Solutions. *Signal Processing: Image Communication* 26, 7 (2011), 378–389.

[12] J. Jiang, V. Sekar, and H. Zhang. 2014. Improving Fairness, Efficiency, and Stability in HTTP-Based Adaptive Video Streaming With FESTIVE. *IEEE Transactions on Networking* 22, 1 (2014), 326–340.

[13] Z. Li, X. Zhu, J. Gahm, R. Pan, H. Hu, A. C. Begen, and D. Oran. 2014. Probe and Adapt: Rate Adaptation for HTTP Video Streaming at Scale. *IEEE Journal on Selected Areas in Communications* 32, 4 (2014), 719–733.

[14] A. F. Lippman. 1999. Video Coding for Multiple Target Audiences. In *Proc. of Visual Communications and Image Processing.*

[15] K. Miller. 2016. *Adaptation Algorithms for HTTP-Based Video Streaming.* Ph.D. Dissertation. Technische Universität Berlin.

[16] K. Miller, A.-K. Al-Tamimi, and A. Wolisz. 2016. QoE-Based Low-Delay Live Streaming Using Throughput Predictions. *ACM Transactions on Multimedia Computing, Communications, and Applications* 13, 1 (2016).

[17] K. Miller, S. Argyropoulos, N. Corda, A. Raake, and A. Wolisz. 2013. Optimal Adaptation Trajectories for Block-Request Adaptive Video Streaming. In *Proc. of the Packet Video Workshop.*

[18] K. Miller, E. Quacchio, G. Gennari, and A. Wolisz. 2012. Adaptation Algorithm for Adaptive Streaming over HTTP. In *Proc. of 19th International Packet Video Workshop (PV).*

[19] R. Pantos and W. May. 2016. HTTP Live Streaming, version 20. *IETF Informational Internet-Draft* (2016).

[20] J. J. Quinlan, D. Raca, A. H. Zahran, A. Khalid, K. K. Ramakrishnan, and C. J. Sreenan. 2016. D-LiTE: A Platform for Evaluating DASH Performance Over a Simulated LTE Network. In *Proc. of IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN).*

[21] I. Sodagar. 2011. The MPEG-DASH Standard for Multimedia Streaming Over the Internet. *IEEE Multimedia* 18, 4 (2011), 62–67.

[22] C. Timmerer, M. Maiero, and B. Rainer. 2016. Which Adaptation Logic? An Objective and Subjective Performance Evaluation of HTTP-based Adaptive Media Streaming Systems. *arXiv preprint 1606.00341* (2016).

[23] D. J. Vergados, A. Michalas, A. Sgora, D. D. Vergados, and P. Chatzimisios. 2016. FDASH: A Fuzzy-Based MPEG/DASH Adaptation Algorithm. *IEEE Systems Journal* 10, 2 (2016), 859–868.

[24] X. Yin, V. Sekar, and B. Sinopoli. 2014. Toward a Principled Framework to Design Dynamic Adaptive Streaming Algorithms over HTTP. In *Proc. of 13th ACM Workshop on Hot Topics in Networks (HotNets).* Los Angeles, CA.

[25] A. Zambelli. 2009. *IIS Smooth Streaming Technical Overview.* Technical Report. Microsoft Corporation.

[26] X. K. Zou, J. Erman, V. Gopalakrishnan, E. Halepovic, R. Jana, X. Jin, J. Rexford, and R. K. Sinha. 2015. Can Accurate Predictions Improve Video Streaming in Cellular Networks?. In *Proc. of the International Workshop on Mobile Computing Systems and Applications (HotMobile).*