Dynamic QoE Optimisation for Streaming Content in Large-Scale Future Networks

Jeroen Famaey  Bart De Vleeschauwer  Tim Wauters  Filip De Turck  Bart Dhoedt  Piet Demeester

Department of Information Technology (INTEC), Ghent University – IBBT, Gaston Crommenlaan 8, bus 201, B-9050 Gent, Belgium
tel: +3293314938, fax: +3293314899
e-mail: jeroen.famaey@intec.ugent.be

Abstract—Multimedia content sharing services, such as YouTube, usually offer only low quality video streams. Additionally, it is expected the popularity of these services will increase even further in the future. Therefore, current streaming-content delivery architectures, with centralised searching and management components, might not be able to keep up with the ever-growing user bases.

In this paper, we propose a scalable and fully distributed end-to-end architecture for delivering streaming multimedia content. Additionally, we designed several algorithms that allow content servers to intelligently adapt content quality to the available access-link bandwidth and server resources. In this way, the service can improve Quality of Experience, by offering higher bit-rates at times of low load, while still being able to satisfy as many content requests as possible when the load increases.

In depth simulations were performed to validate and evaluate these algorithms. The results show that the proposed platform is capable of increasing delivered stream quality and accepted content requests in various large-scale scenarios.

I. INTRODUCTION

In the current Internet, many services exist for sharing and delivering video and other multimedia content. Examples include popular websites, such as YouTube and DailyMotion. Usually, such services offer videos only at low quality. Additionally, they employ centralised architectures for processing search results and serving content. It is expected, the popularity of content sharing services will increase even further in the future. In order to keep up with this growth, scalable architectures with fully distributed content searching and management components are needed.

In this paper, we propose a scalable and fully distributed architecture for sharing multimedia content. Additionally, the architecture supports mechanisms to improve the end-user perceived Quality of Experience (QoE). On one hand, overlay routing techniques are used to circumvent failures and bandwidth bottlenecks in the core Internet. On the other hand, content servers are capable of automatically increasing or decreasing stream bit-rates, based on the available bandwidth on the server and client access link. This allows the QoE to be increased at times of lower load, while still allowing the content servers to serve as many clients as possible when the amount of requested content increases. The proposed architecture is shown in Fig. 1.

The content servers are responsible for hosting content and streaming it to the clients. These servers may be backed by a data center for storing the actual content. As, for scalability reasons, every content server hosts only part of all available content, an additional mechanism is needed for content discovery. Therefore, all content servers take part in a distributed hash table (DHT) (e.g. Chord [1]). Whenever a content server comes online, it joins the DHT via a known bootstrap node and publishes its content by way of DHT insert messages. To discover the list of servers that contain a specific content item, one can simply send a DHT lookup message to an arbitrary content server.

Content servers are capable of adapting the bit-rate of content stream. This is done by way of a set of transcoding plugins. At the cost of additional computational resources, a plugin is able to transcode streams to a lower bit-rate, thus saving on bandwidth resources. In some deployments, clients might be allowed to share content themselves. These prosumers take up the role of both client and content server.

Jeroen Famaey is funded by the Institute for the Promotion of Innovation by Science and Technology Flanders (IWT).
Tim Wauters is funded by the Fund for Scientific Research Flanders (FWO).
Part of this research was funded by IBBT through the ISBO QoE project.
The client proxies should be positioned near a large set of clients (e.g. in the ISP’s access network). They act as a gateway for their clients. In this way, the actual client should only be capable of sending a content request to the nearest proxy, while the proxy takes care of resource monitoring, content discovery and server selection.

Fig. 1 also shows how content streams are routed from the content server, via the client proxy, to the client. When a fault occurs in the core Internet, the content server may route streams via another overlay server. This could be done via other content servers or via additional routing servers placed in the network.

The main contributions of this paper are a scalable protocol and algorithms for selecting a suitable content stream bitrate at the content servers. The scalable overlay routing mechanisms will be considered in future work. The rest of this paper is structured as follows. Section II summarises related work. Subsequently, the protocol and algorithms are described in Section III. Next, simulation results are discussed in Section IV. And finally, Section V summarises the important conclusions and future work.

II. RELATED WORK

Content Distribution Networks (CDNs) such as Akamai, have been widely used to efficiently deliver web-based content across the Internet [2], [3]. The protocol and algorithms proposed in this paper, are most closely related to CDN server selection protocols, which have been studied thoroughly in the past, e.g. [4], [5]. However, when considering streaming multimedia content, other factors, such as higher bandwidth usage and perceived QoE, have to be taken into account. Therefore, streaming CDNs have been proposed for delivering such content [6]. In [7], several server selection algorithms are proposed in the context of streaming CDNs.

However, most of these server selection approaches focus on minimising round trip time, while assuming enough resources (such as bandwidth) are available. On the other hand, our work focuses on maximising request acceptance rate and global QoE in face of limited bandwidth and computational resources.

The problem of adapting stream bit-rates in order to satisfy bandwidth constraints has already been studied in the context of setting up multicast trees for multimedia content delivery [8]. Nevertheless, their proposed algorithms are centralised, while we focus on distributed algorithms and scalability.

Also related are peer-to-peer (P2P) TV applications, such as PPLive and Zattoo. Little is known about the protocols and algorithms employed by these applications, as they are often proprietary. Nevertheless, some detailed measurement studies have been performed to study the used protocols [9], [10], [11]. Among other things, this showed that popular P2P-TV applications send video streams only in a single bit-rate and could benefit from our bit-rate adaptation techniques.

The architecture proposed in this paper, is based on earlier work [12], [13]. The focus of these works is an architecture for improving end-user perceived QoE by means of dynamically adapting stream bit-rates on the last mile. However, our current work focuses on scalable server selection and end-to-end bit-rate adaptation based on available server resources and bandwidth.

III. SERVER AND BIT-RATE SELECTION

In this section, the selection protocol and its algorithms are described in more detail. First, the selection protocol itself is explained. Subsequently, a static and dynamic version of the server and plugin selection algorithm is discussed. The selection protocol allows the client to select a content server, and the content server to select the stream’s bit-rate. The sequence diagram shown in Fig. 2 describes the message exchange of the protocol in more detail.

The protocol consists of three phases. The first phase is initiated when the client wants to request a multimedia stream. It sends the request to its client proxy, which then performs a lookup on the DHT to find out which content servers host the requested stream. This might either return all content servers that contain the content, or, for scalability reasons, only a subset of them. Subsequently, the client proxy starts the second phase by initiating the server selection algorithm to select the best server. Next, the client proxy asks the selected content server to send the requested stream. The request message contains information about the available incoming bandwidth of the client and client proxy. This content server then selects a plugin to transcode the stream, based on available computational resources and bandwidth (both outgoing server bandwidth and incoming client and client proxy bandwidth). Finally, the content server starts the third phase of the protocol. If it was successful at selecting a plugin (i.e. enough resources and bandwidth were available for at least 1 plugin), it will start sending the multimedia stream to the client (via the client proxy), and the protocol finishes (phase 3a). Otherwise, it will let the client proxy know it cannot send the stream (phase 3b), and the client proxy will repeat phase 2 with the next best server. This process is repeated until a content server is found that can send the stream, or all servers on the server list have been contacted. If no server is found, the client is
informed that it cannot receive the stream.

A. Notations and Assumptions

The notations used to describe the algorithms, are shortly summarised in this section. The set of content servers is given by \( S \). Every content server \( s \in S \) has a total amount of computational resources \( R_s \) and outgoing bandwidth \( B_s \). Its currently available and used resources and bandwidth are given by \( R^\text{avl}_s \), \( R^\text{usd}_s \), \( B^\text{avl}_s \) and \( B^\text{usd}_s \). Also given is a set of clients \( T \) and client proxies \( X \). Every \( c \in T \cup X \) has a total amount of incoming bandwidth \( B_c \). Again currently available and used bandwidth are given by \( B^\text{avl}_c \) and \( B^\text{usd}_c \). In the rest of this paper, it is assumed that the access network, and not the core network, contains the bandwidth bottleneck.

Therefore, a content server’s outgoing and client (proxy)’s incoming bandwidth equal the available bandwidth on their access link. The set of plugins available on every server is denoted as the set \( P \). Each plugin \( p \in P \) has a required amount of computational resources \( \rho_p \) and bandwidth \( \beta_p \). Additionally, a QoE-value \( \pi_c \) is associated with it. Finally, a set of content items \( C \) is also given. Every content server \( s \) hosts a subset \( C_s \) of this content. Furthermore, the content item \( c \in C \) has a duration of \( \delta \) time-units.

In reality, perceived QoE is a complex function of QoS parameters (e.g. jitter, bit-rate, delay), and how the end-user perceives them. However, the focus of this paper are video streaming services where the data is stored on content servers, and not streamed in real-time. Delay is thus less important. Additionally, the effects of jitter can be reduced by caching at the client-side. Therefore, we focus on bit-rate only and assume perceived QoE is directly proportional to the bit-rate of the stream. It is also assumed that the bit-rate of a plugin (and thus the QoE) is inversely proportional to its required resources.

B. Static Selection Algorithm

The selection algorithm consists of two parts. First, it allows the client proxy to select a content server. Second, it allows the content server to select a plugin for transcoding a multimedia stream. The algorithm allows the client proxies to sort a list of content servers from most to least favorite. It can then request the content with the content servers according to the order in the list. Additionally, the algorithm allows the content servers to sort their plugins from most to least favorite whenever a content item is requested. They can then iterate over the sorted list, until a plugin is found for which enough computational server resources, outgoing server bandwidth, and incoming client (proxy) bandwidth are available. The goal of the selection algorithm is two-fold. First, the content servers must be able to accept as many streams as possible. Second, the globally provided QoE must be maximised. In the static case, we assume that a stream can not be re-transcoded once the server starts sending it.

In the rest of this section, we discuss three metrics for the static selection algorithm. The first two are straight-forward and naive, and both work well in one specific scenario. Additionally, we propose a third, more intelligent, metric which combines the advantages of the two naive ones.

1) Minimum Resources Metric (MinRes): The minimum resources metric (MinRes) attempts to use as few computational resources as possible. As we assumed resource and bandwidth usage of a plugin to be inversely proportional, this metric attempts to send the stream in the highest possible bit-rate. Thus, it maximises the QoE on a stream per stream basis.

MinRes sorts content servers from most to least available resources and plugins from least to most used resources. It is clear that this metric performs well in a scenario where computational resources act as the bottleneck, and available bandwidth is very high.

2) Minimum Bandwidth Metric (MinBwd): The minimum bandwidth metric (MinBwd) is similar to MinRes, but instead of minimising resource usage it minimises bandwidth usage. As QoE and bit-rate are directly proportional, this minimises QoE on a stream per stream basis. Nonetheless, if bandwidth acts as the bottleneck this metric will be able to accept a lot more streams overall than MinRes, thus maximising the global QoE.

MinBwd sorts content servers from most to least available bandwidth and plugins from least to most bit-rate.

3) Weighted Metric (Wgt): As stated earlier, MinRes is expected to perform well when resources are the limiting factor and MinBwd when bandwidth is the bottleneck. However, neither would perform well in the opposite scenario. In this section, we propose a third metric, that is expected to perform well in both scenarios and in a third, more realistic, scenario where both resources and bandwidth are limited. The Wgt metric sorts content servers and plugins respectively according to the following functions:

\[
\begin{align*}
\min_{s \in S} \left( w_1 \times R^\text{usd}_s + w_2 \times B^\text{usd}_s \right) \\
\min_{p \in P} \left( w_1 \times \rho_p + w_2 \times \beta_p \right)
\end{align*}
\]  

with \( R^\text{usd}_s \), \( B^\text{usd}_s \), \( \rho_p \) and \( \beta_p \) normalised. Additionally, \( R^\text{usd}_s \) and \( B^\text{usd}_s \) are averaged over a time interval \( t \).

The parameters \( w_1 \) and \( w_2 \) represent the weight factor for resources and bandwidth respectively. In the rest of this section, suitable values for these weight parameters are derived, based on several observations. Let us first define \( \alpha \in [0, 1] \) and \( \beta \in [0, 1] \) as the current normalised resource and bandwidth usage. Client proxies calculate \( \alpha \) and \( \beta \) as follows. After discovering the content servers that host the requested content, the client proxy contacts them. The content servers reply with the normalised and averaged over time values of \( R^\text{usd}_s \) and \( B^\text{usd}_s \). The client proxy calculates the average of all received values, and uses them as \( \alpha \) and \( \beta \) respectively. The content servers use their own normalised values of \( R^\text{usd}_s \) and \( B^\text{usd}_s \) as \( \alpha \) and \( \beta \) when performing plugin selection. No assumptions are made on how content servers measure used computational resources and bandwidth. Though, many methods exist for this purpose (e.g. [14]).

If \( w_1 \) is highest, used resources are minimised, and if \( w_2 \) is highest, used bandwidth is minimised. The observations can
function \( p \in \mathcal{P} \) s.getPlugin()
1: sort(\( \mathcal{P} \)) /* from highest to lowest bit-rate */
2: for all \( p \in \mathcal{P} \) do
3: \( R^\text{av}_p \) \( \geq \) \( \rho_p \) and \( B^\text{av}_p \) \( \geq \) \( \beta_p \) then
4: return \( p \)
5: else if \( R^\text{av}_p < \rho_p \) then
6: return null
7: else
8: for all \( m \in \mathcal{P} \) do
9: transcode(m, \( p \))
10: if \( R^\text{av}_m \) \( \geq \) \( \rho_p \) and \( B^\text{av}_m \) \( \geq \) \( \beta_p \) then
11: return \( p \)
12: return null

procedure s.upscaleStreams()
1: sort(\( \mathcal{P} \)) /* from lowest to highest bit-rate */
2: for \( i = 0 \) to |\( \mathcal{P} \)| - 2 do
3: \( p \leftarrow \mathcal{P}_i; q \leftarrow \mathcal{P}_{i+1} \)
4: for all \( m \in \mathcal{P} \) do
5: if \( \beta_q - \beta_p < B^\text{av}_m \) then
6: return
7: transcode(m, \( q \))

Figure 3. Pseudo-code for the dynamic selection algorithm

then be summarised as follows

- If \( \alpha \) is low and \( \beta \) is high, then a low bit-rate plugin should be selected (thus \( w_2 \) should be highest).
- If \( \alpha \) is high and \( \beta \) is low, then a high bit-rate plugin should be selected (thus \( w_1 \) should be highest).
- If \( \alpha \) is high and \( \beta \) is high, then a medium bit-rate plugin should be selected (thus \( w_1 \) and \( w_2 \) should be equal).
- If \( \alpha \) is low and \( \beta \) is low, then a high bit-rate plugin should be selected (thus \( w_1 \) should be highest).

From this we can conclude that \( w_1 \) should be high if \( \alpha \) is high and if both \( \alpha \) and \( \beta \) are low, and \( w_2 \) should be high if \( \beta \) is high. Additionally, they should be about equal if \( \alpha \) and \( \beta \) are both high. From this we can derive values for the weight parameters: \( w_1 = 1 + \alpha - \beta \) and \( w_2 = \beta \).

C. Dynamic Selection Algorithm (DSA)

For the static version of the algorithm, it was assumed that the bit-rate of a stream can not be changed after the content server has started sending it. However, it is obvious that results could be improved if this were possible. Especially when the request count does not remain constant over time. Therefore, we propose an algorithm capable of dynamically changing the plugin associated with a stream. This however, does not make the static algorithms redundant, as dynamically adjusting bit-rates incurs additional overhead or might not be possible for some services or systems.

The server selection part of the algorithm is not changed, and we therefore assume a server is still selected using one of the metrics discussed above. Pseudo-code for the dynamic plugin selection algorithm is shown in Fig. 3. The variable \( B^\text{av}_s \) is the minimum of the available bandwidth of the server \( B^\text{av}_s \), client \( B^\text{av}_c \) and client proxy \( B^\text{av}_p \). For the requested stream, the values of \( B^\text{av}_s \) and \( B^\text{av}_c \) can be included in the request sent by the client proxy. However, this cannot be done for the streams that are already in the process of being sent. Therefore, a mechanism is needed for exchanging this information between the client proxy and content server. The client proxy could for example send recently measured values of \( B^\text{av}_s \) and \( B^\text{av}_c \) to all content servers it is using at regular intervals. Alternatively, the content server could request the information itself when necessary.

Whenever a new stream is requested, the getPlugin() function is called to find a suitable plugin for the stream. The method iterates over all plugins (from highest to lowest bit-rate) (lines 1–2). If there are enough resources and bandwidth for a plugin, it is returned (lines 3–4). As plugins are sorted from highest to lowest bit-rate (and thus from lowest to highest resource usage), no suitable plugin can be found if there are not enough resources for the current one, and null is thus returned (lines 5–6). Otherwise, the algorithm lowers the bit-rate of streams with a higher bit-rate than plugin \( p \) to that of \( p \), until there are enough resources, or no streams are left with a higher bit-rate (lines 7–11). In the first case, \( p \) is returned (line 11), otherwise the algorithm repeats for the next \( p \) or returns null if no more plugins remain (line 12). The getHigherBitRateStreams() function, returns all streams that are transcoded in a higher bit-rate than that of plugin \( p \). The transcode() procedure changes the plugin of stream \( m \) to \( p \).

Whenever a stream has finished, enough bandwidth might have become available to increase the bit-rate of another stream. Therefore, the upscaleStreams() procedure is called whenever a stream has finished. It attempts to increase the bit-rate of streams, starting off with the lowest bit-rate streams. The getBitRateStreams() function returns all streams that are transcoded with plugin \( p \).

Note that because resource usage and bit-rate are assumed to be inversely proportional, it is never necessary to change streams to a plugin that uses less computational resources in order to accommodate for new streams. This is because the algorithm always maximises bit-rate and thus minimises resource usage already.

IV. Evaluation

The architecture, protocol and algorithms were implemented in the PlanetSim peer-to-peer simulator [15]. In this section, we discuss the results of the simulations in order to compare and evaluate the static and dynamic algorithms. The static metrics were tested on three different static scenarios, where the client request count remained constant during the simulation run. The dynamic algorithm was evaluated on one static and two dynamic scenarios, where the request count changed during the simulation run. All results are averaged over 30 simulation runs. The time interval \( t \) over which bandwidth and computational resources are measured, was set to 1 minute for all scenarios. All results are evaluated in terms of two
different metrics, satisfied request count and obtained QoE (both expressed as percentage of the maximum value).

All simulated networks consisted of 50 client proxies and 20 content servers. To improve the performance of the simulator, the actual clients were not simulated. Instead, the client proxies acted as an abstraction of a set of many clients. Every content server contained three transcoding plugins, as shown in Table I. Computational resources are expressed as CPU cycles (in MHz). The bit-rate of $P_1$ and $P_2$ are comparable to respectively a low and high quality YouTube video stream, while $P_3$ provides a higher quality alternative. A lightweight transcoding technique was assumed, where the computational resources required to transcode to the lowest bit-rate are only ten times higher than for sending the stream in the highest bit-rate.

The content servers hosted a total of 100 different content items. Their duration was randomly generated following the distribution shown in Fig. 4, which is derived from the duration of over 2000 randomly sampled YouTube videos.

Every content item was hosted and requested by a percentage of content servers and client proxies equal to

$$1 - \frac{i - 1}{|C|}$$

with $i \in [1, |C|]$ the popularity-index of the item (lowest index being highest popularity). Which results in more popular content being hosted and requested by a larger number of servers and clients. Additionally, the total request count per item was chosen using a Zipf-like distribution [16]. The percentage of requests for the $i^{th}$ most popular content item was set to

$$\frac{i^{-\alpha}}{\sum_{n=1}^{|C|} n^{-\alpha}}$$

with $\alpha$ the Zipf parameter, set to 0.7, which has been shown to be a realistic value [17]. The arrival rate of the requests was chosen randomly with uniform distribution between the start and the end of the simulation run.

Statistical analysis was employed to interpret simulation results. A one-way ANOVA [18] was used to find significant differences between algorithms. All statistics were performed using a 5% significance level. The error bars in the graphs represent the interval in which the mean of the entire population is located with a 99% probability (the confidence interval).

### A. Static Scenarios

To evaluate and compare the three proposed static selection metrics, three different static scenarios were simulated. During a single simulation run, the client request count remained constant, and runs were performed with request counts from the set $\{1000, 2500, 5000, 10000, 20000, 30000\}$. In the first scenario, the available bandwidth was set to infinity, and the computational resources of each content server to 8000 MHz, making computational resources the limiting factor. In the second scenario, available computational resources were set to infinity and the outgoing bandwidth of each content server to 10 Mbps. Finally, in the third scenario, bandwidth of each server was set to 10 Mbps and computational resources to 8000 MHz per server. The results are shown in Fig. 5.

As expected, the results for limited computational resources (Fig. 5(a-b)) show that MinRes greatly outperforms MinBwd, both in terms of satisfied requests and obtained QoE. In contrast, MinBwd performs better than MinRes when bandwidth is the limiting factor (Fig. 5(c-d)). However, as MinBwd also minimises QoE, MinRes provides a better global QoE for request counts lower than 5000. At these low request counts, MinBwd uses $P_1$ for transcoding all streams, resulting in a maximum obtainable QoE of only 80%. Finally, both naive metrics perform poorly in the third scenario (Fig. 5(e-f)), where there is no significant difference in terms of satisfied request count.

The Wgt metric was created to combine the advantages of MinRes and MinBwd. As the results show, it does perform as desired in the first two scenarios, mimicking the behavior of respectively MinRes and MinBwd. It even performs significantly better than MinBwd in terms of obtained QoE for request counts between 2500 and 10000 in the second scenario, as, unlike MinBwd, it does not always use $P_1$ at these loads. Additionally, it performs significantly better than both naive metrics in the third scenario for request counts of 5000 and up.

### B. Dynamic Scenarios

To evaluate the dynamic selection algorithm (DSA), one static and two dynamic scenarios were simulated. In these scenarios, the performance of the dynamic algorithm is compared to that of the static algorithm with the Wgt metric. In all three scenarios, the bandwidth was set to 10 Mbps and the computational resources to 8000 MHz per content server. Fig. 6 shows the simulation results.

[Graph showing the duration (in minutes) of a random sample of over 2000 YouTube videos, measured over a 24 hour interval]

Figure 4. A histogram showing the duration (in minutes) of a random sample of over 2000 YouTube videos, measured over a 24 hour interval

Table I

<table>
<thead>
<tr>
<th>plugin</th>
<th>bit-rate (Kbps)</th>
<th>resources (MHz)</th>
<th>QoE (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>350</td>
<td>1000</td>
<td>80</td>
</tr>
<tr>
<td>$P_2$</td>
<td>700</td>
<td>500</td>
<td>90</td>
</tr>
<tr>
<td>$P_3$</td>
<td>1400</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

The Wgt metric was created to combine the advantages of MinRes and MinBwd. As the results show, it does perform as desired in the first two scenarios, mimicking the behavior of respectively MinRes and MinBwd. It even performs significantly better than MinBwd in terms of obtained QoE for request counts between 2500 and 10000 in the second scenario, as, unlike MinBwd, it does not always use $P_1$ at these loads. Additionally, it performs significantly better than both naive metrics in the third scenario for request counts of 5000 and up.

### B. Dynamic Scenarios

To evaluate the dynamic selection algorithm (DSA), one static and two dynamic scenarios were simulated. In these scenarios, the performance of the dynamic algorithm is compared to that of the static algorithm with the Wgt metric. In all three scenarios, the bandwidth was set to 10 Mbps and the computational resources to 8000 MHz per content server. Fig. 6 shows the simulation results. The first scenario (Fig. 6(a-b)) is identical to the third static scenario. In the second scenario, the client request count starts at 30 requests per minute and increases to 125 requests per minute from minutes 25 to 35, and then decreases to 30 again (Fig. 6(c-d)). In the third scenario, request count instantly increases from 8 to 25 per minute at minute 25 and instantly decreases again to 8
at minute 35 (Fig. 6(e-f)). Additionally, the duration of all video’s has been multiplied by 10 in the third scenario.

The results for the first scenario show that DSA performs only slightly better than Wgt when the request count remains constant (at most 2% better in terms of satisfied demand and obtained QoE). Statistically, performance of DSA is significantly better when the request count ≤ 5000.

However, in the second scenario DSA outperforms Wgt after the increase in request count (up to 5% better, both in terms of satisfied demand and obtained QoE). Statistically, DSA performs significantly better than Wgt between minutes 24 and 27. This shows that the dynamic algorithm is indeed better suited to handle scenarios with frequent and sudden changes in request count.

As the static algorithm is not capable of changing a stream’s bit-rate once transmission has started, it is believed that its performance will degenerate further if the average duration of videos increases. To prove this theory, another dynamic scenario was devised where the duration of all videos (cfr. Fig. 4) was multiplied by 10. The obtained durations are comparable to anything ranging from TV shows (20 - 30 minutes) to full length movies (90 - 110 minutes). The results of this simulation scenario are shown in Fig. 6(e-f). Even at first sight, it is clear that the difference between DSA and Wgt is a lot bigger in this scenario. Between minutes 30 and 40, results for the static algorithm decrease greatly, both in terms of satisfied demand (down to 62%) and obtained QoE (down to 54%). On the other hand, DSA is capable of handling at least 97% of all requests and even when the request count increases, it always obtains over 87% of the maximum QoE. After minute 40 the static algorithm recovers, but a difference of about 3 - 4% in terms of obtained QoE remains. ANOVA analysis showed that in terms of satisfied request count, the DSA algorithm performs significantly better than Wgt between minutes 31 and 37. In terms of obtained QoE, Wgt performs significantly better until minute 12, and DSA from minutes 15 to 51 and 54 to 60. These results show that the performance of DSA does indeed improve relative to that of the static algorithm as the average video duration increases.

V. CONCLUSIONS AND FUTURE WORK

The contributions of this paper are twofold. First, we proposed a scalable and fully distributed architecture for delivering streaming content in a future Internet context. Second, a distributed protocol and two algorithms were proposed for improving the end-user Quality of Experience (in terms of stream bit-rate), while maximising the amount of accepted content requests. These algorithms intelligently select stream bit-rates, based on the available access-link bandwidth, computational resources and expected future request load. The dynamic algorithm is additionally capable of changing the bit-rate of in-progress streams to better cope with sudden request rate changes.

Detailed simulations, performed with PlanetSim, showed that our static weighted bit-rate selection metric (Wgt) significantly outperforms more naïve selection metrics in several realistic scenarios. Additionally, it was shown that the dynamic algorithm indeed copes better with request rate surges and longer video durations (e.g. TV shows or movies).

In this paper, we focused on a YouTube like video sharing use-case. In future work, we plan to extend the evaluation of the bit-rate selection protocol to other streaming content...
delivery services such as Video-on-Demand (longer video durations) and Video Conferencing (real-time delivery). Additionally, the focus of this work was to cope with bandwidth limitations on the access links. In the future, we plan to further study and develop scalable overlay routing protocols for circumventing bandwidth bottlenecks and faults in the core Internet. Finally, methods exist in file-sharing and peer-to-peer TV services for receiving content from multiple sources at once. Methods for selecting multiple content servers, each sending part of the video stream, could be incorporated into the proposed architecture.

REFERENCES


