DASH Policy Algorithm

Luca Crema

12/06/2020

1 Introduction

The following report will explore a custom policy for the **Dynamic adaptive streaming over HTTP** or **DASH**–like client protocol to reproduce a video to maximize the overall perceived video quality.

2 Environment simulation

To work on the client policy it has been set-up an environment that could simulate the communication delays, the consumption of the downloaded video segments, and the eventual freezes. The simulation has been realized using `C++14` because it's a convenient object-oriented language and the one I know the most among the given options.

2.1 Segments

As defined in the DASH protocol the video is split into segments: for this simulation all of the segments have the same length, and each one has five levels of encoding available that differ in **bit size and quality**.

2.2 Client-Server

The client sequentially asks the policy for two things: the index of a segment and the encoding level to build a request to the server. The server will receive the request from the client and a bitrate value from a list then returns a response containing the time taken to transfer the chosen segment. The client proceeds to store the downloaded segment in its buffer and updates the media playtime accordingly to how much time passed to get the response and how much buffered time it has, then it starts over. To avoid the overwriting of already–played segments, the client has implemented protection that discards the downloaded segment if the current one in the buffer has already been played.

3 Quality metrics observations

We can notice that the penalty that weights the most in quality is long delays, this means that having long waiting times before the playout of the first segment is not profitable in the short term and shorter delays are preferable ($\exp(A+B+C) \gg \exp(A)+\exp(B)+\exp(C)$ for A,B,C big enough).

4 Custom policy

The policy starts by downloading the first segment at the highest possible encoding level to minimize initial waiting time. In every other case, it calculates the average bitrate of the previous $min(N_{avg}, L)$ elements (where L is the number of segments in the buffer and N_{avg} is a policy constant for the number of elements to calculate the average of) and underestimates it by multiplying it by $^2/_3$.

$$R_k^* = \frac{2}{3} \times \frac{1}{N_{avg}} \sum_{i=L-N_{avg}+1}^{L} R_i$$

There are now two possibilities:

1. There are still segments to download ($L < N_{tot}$).
2. The buffer is full and we can improve the segments already in the buffer.

4.1 First case - buffer not full

The algorithm chooses to request the segment following the last one ($k = L + 1$) in the buffer and calculates the encoding c_k that gives the maximum U_k. If $B(t_{k-1}) - t_k^{'} = 0$, where $t_k^{'}$ is the amount of playout time consumed when downloading the k's element, it means that the video has frozen so the policy chooses the highest encoding level right away. For this simulation the minimum encoding has been capped to 2 as it's been experimentally proven to be more efficient; I couldn't come up with a certain explanation but I suppose it's because lower encoding's bit sizes increase the probability for long waiting times when low–bitrate spikes occur. The estimated delay time φ_k^* is given by

$$\varphi_k^* = \frac{M_k(c)}{R_k^*} - \frac{B(t_{k-1}) - t_k^{'}}{\lambda}$$

where λ is a policy constant that defines how many segments should be ready in the buffer at any given time. The quality is then calculated using the given quality function:

$$U_k = q_k(c_k) - \beta|q_{k-1}(c_{k-1}) - q_k(c_k)| - \exp(\gamma\varphi_k^*/T_s)$$

4.2 Second case - buffer full

The algorithm looks for a segment to improve starting from the one after what's currently playing, for each encoding available it calculates whether there's a valuable improvement and if it can be downloaded in time before that very segment has to be played, when it finds that segment with such encoding it chooses it to be downloaded and overwritten.

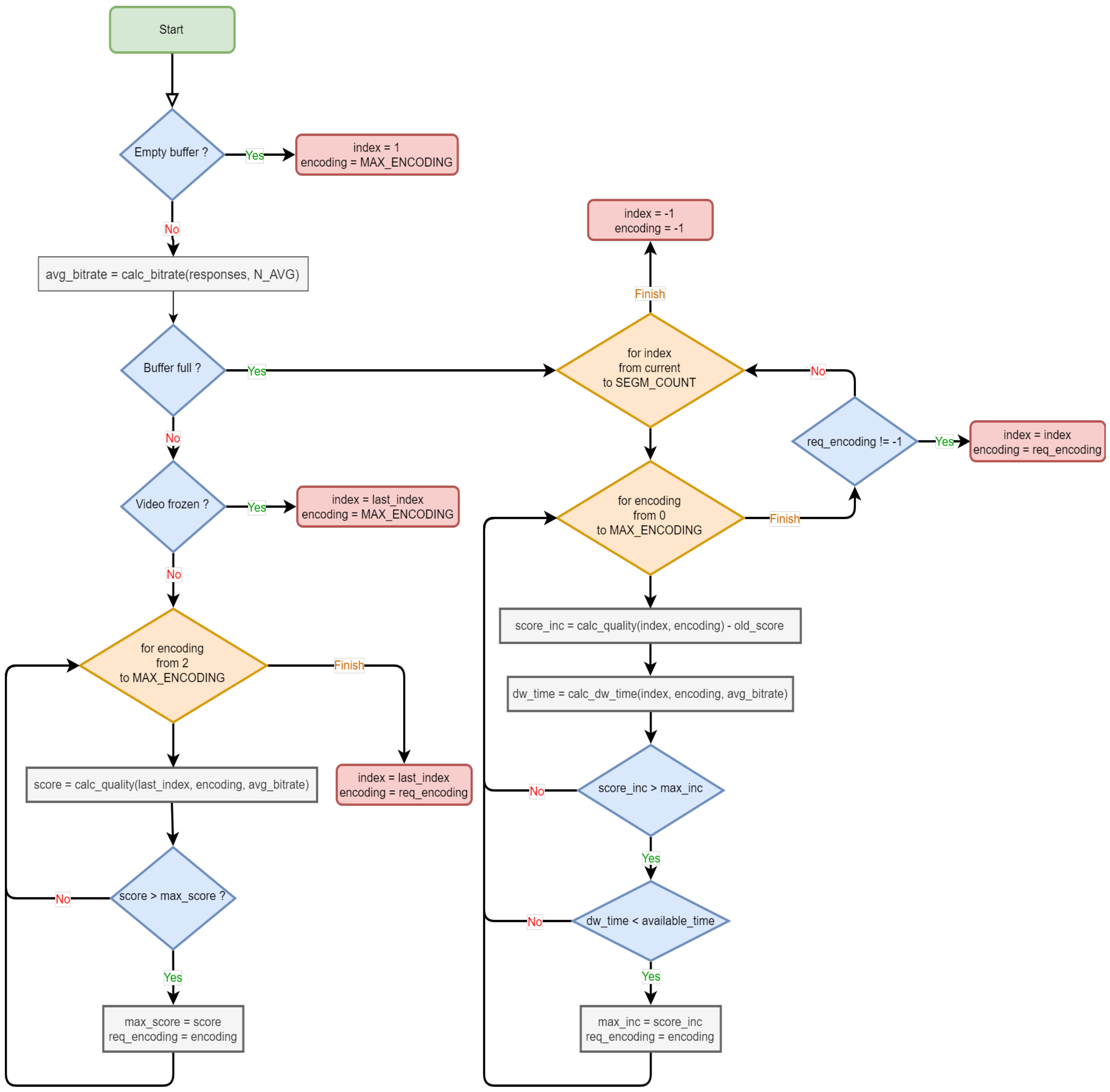

Figure 1: Flow chart describing the algorithm.

5 Performance

By testing with the given `MPD.txt` containing a Media Presentation Descriptor–like structure and `channel.txt` containing a list of bitrates the final perceived quality of the user has been somewhere around the values of **34200** and **34500** (depending on some variables such as N_{avg} and λ) which, by checking with other colleagues, is the best performance any of us managed to reach. To put it in perspective here are some other policy's performances:

All encoding level 4	29000
All encoding level 3	33000
Professor's suggested policy	30000

Although it seems that the performance is similar to the trivial policies, it's exponentially harder to improve the algorithm and gain quality after a certain value around 33500.