Client-Server DEMO for DASH video streaming

MATTEO BIASETTON IACOPO MANDATELLI LUCA PIAZZON
University of Padua
January 9, 2017

Introduction

Dynamic Adaptive Streaming over HTTP (DASH) is an adaptive bitrate streaming technique over the Internet that relies on HTTP protocol.
The aim DASH is to serve a stable video stream that does not interrupt during playback. In order to achieve this, on server side, each video file is first compressed in different resolutions (identified by bitrate) and then divided into segments of fixed length (usually 2 to 4 seconds).
On the client side, while a segment is being reproduced by a DASH player at the application layer, the DASH algorithm selects (based on the current estimate of the link quality) the best bitrate for the next segment to be downloaded. Then the related request is sent to the server that will reply with the proper video segment.
The goal of the bitrate choice strategy is to select the highest possible bitrate that can be downloaded in time without causing interruptions in the playback of the video. Such an algorithm implemented by the client should adapt almost seamlessly to changing network conditions, ensuring to the user the best possible experience.
More advanced algorithms take into account principles such as fairness of the channel (equal share of the network resources among all the active players accessing the server), efficiency (highest feasible bitrate among those available) and stability (avoiding unnecessary bitrate switches to maximize user experience).
In the following chapters we are going to describe our demo setup, the different implemented algorithms and the experimental results.

1 Setup Description

The setup of our demonstration consist of three PCs: one configured as server and two configured as clients. The server runs Ubuntu Linux and execute an instance of Apache HTTP Server: the DASH Media Presentation Descriptor (MPD) file, the video segments and the corresponding init files (headers to be added to the segment file in order to be played by the Player) for each bitrate are stored in the "webapps" directory of Apache server.
Server and clients are directly connected to the router via Ethernet cables as shown in Figure 1.

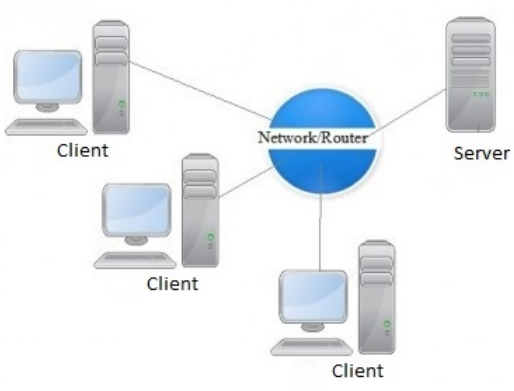

Figure 1: network topology

2 Software structure

The client we realized for this demonstration is written in Java: Figure 2 represents the UML diagram of the project.

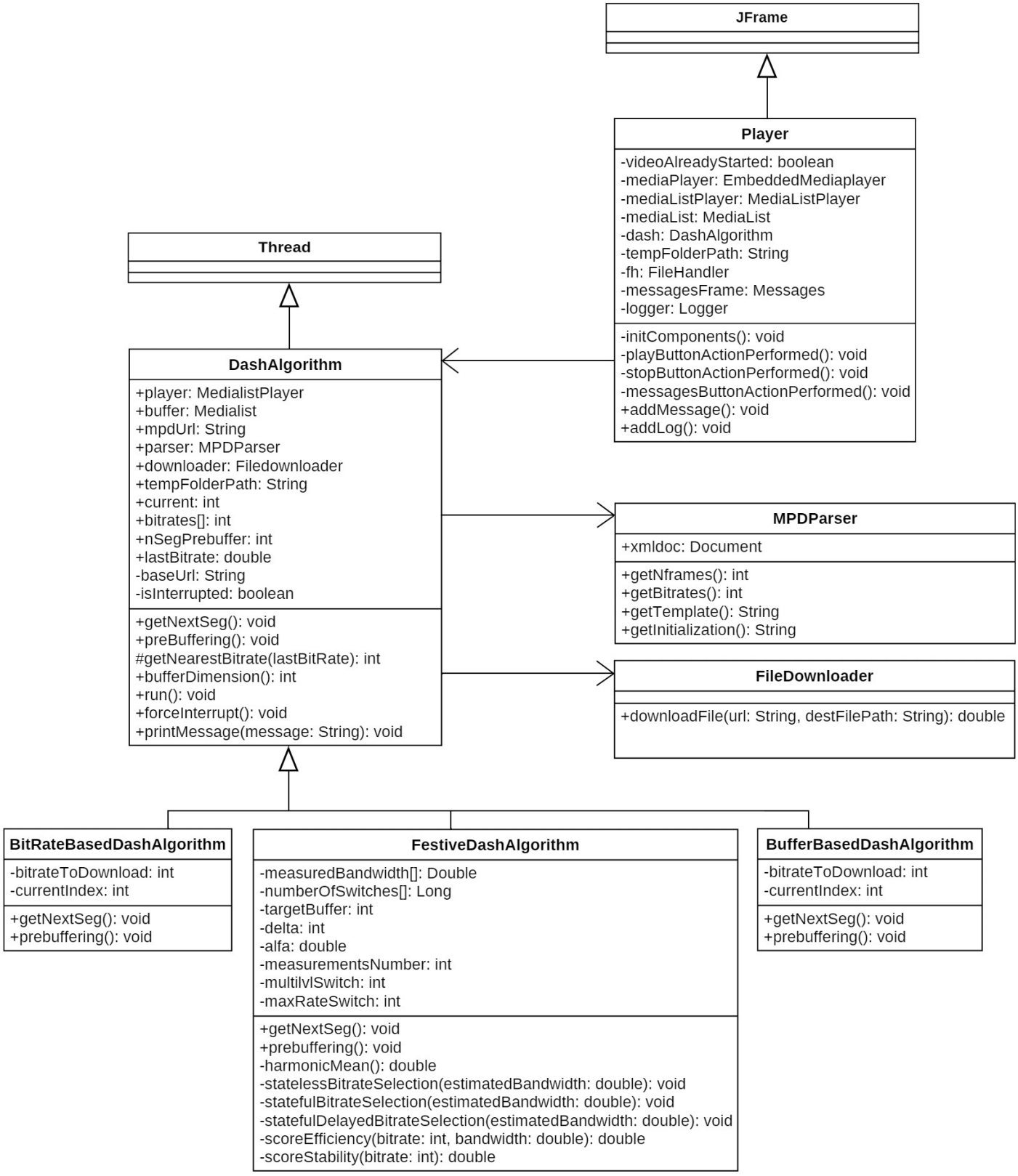

Figure 2: UML Diagram

The classes Player and DashAlgorithm are two different Java threads that run concurrently: since the downloaded video chunks are stored in a buffer, the player and the dash algorithm can concurrently consume and append the video segments to it.
The player is based on the vlcj library [1] which is an open source library that provides Java bindings and an application framework for the VLC media player from VideoLAN. This library requires a valid installation of VLC media player on the same PC (VLC architecture must be the same as the JRE, x86 or x64).
DashAlgorithm is an abstract class that defines the main features of a DASH algorithm. The different algorithms used in this demo are realizations of this class and will be discussed in the following sections.
MPDParser provides tools to interpret the MPD file of the video to be played. The MPD is an XML file containing information about media segments, their duration, encoding and other metadata that may be needed by clients. FileDownloader object is used in DashAlgorithm to download inits and segments from the server.

When the player starts, the user can select three different DASH algorithms from a drop-down menu. When the play button is pressed, first the downloader downloads all the .init files from the server and then the first three segments are downloaded (prebuffering) at low quality.
After the prebuffering, the selected algorithms starts and decides the bitrate of the segment to be downloaded: these information are passed to the downloader, which downloads the file from the server.
Once the file has been downloaded, it is concatenated on the fly with the respective .init, and the video chunk (now a .mp4 file) is added to the buffer, ready to be reproduced by the player. On the other hand, the player consumes the video segments stored in the buffer and reproduces the video; the player continues reproducing the video until the buffer is empty (e.g. due to network slowdown) or the video is finished.

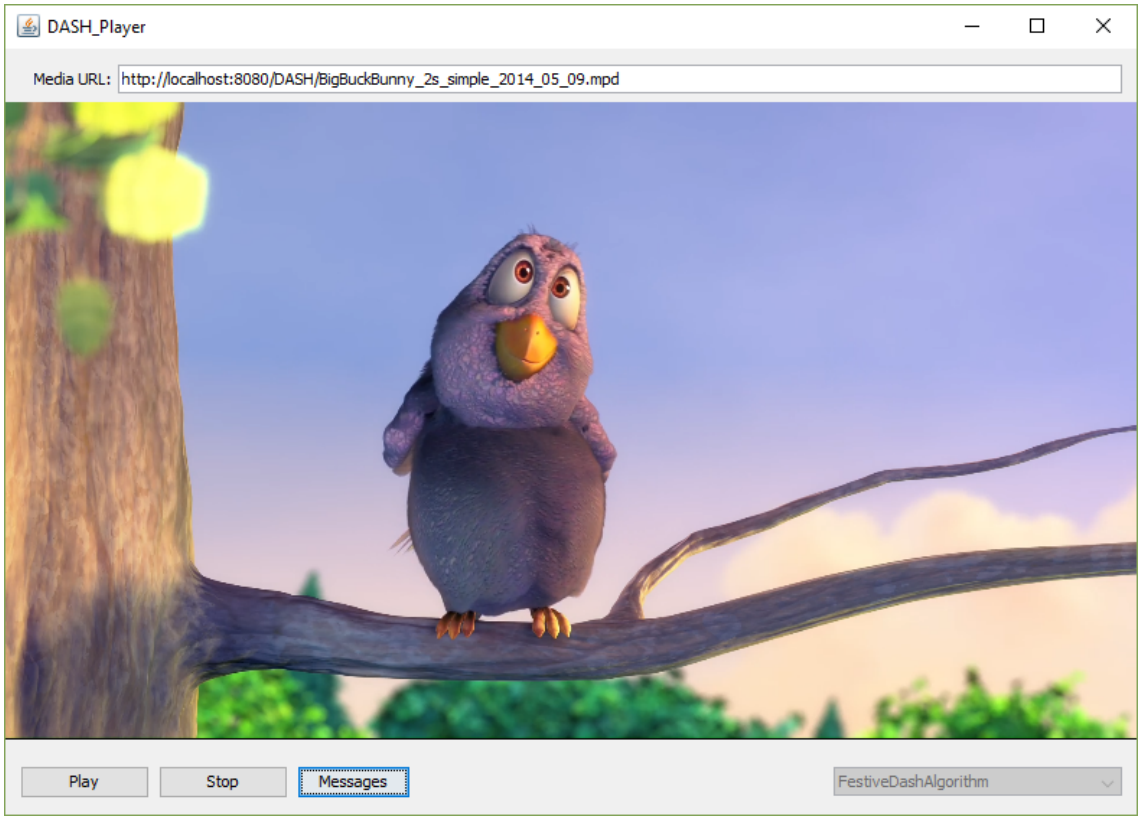

Figure 3: Client graphical interface running FESTIVE Algorithm

Although the dash algorithm tries different techniques to avoid emptying the buffer, sometimes it is inevitable (e.g. link failure, congestion on the network, etc.): in case of empty buffer, the dash algorithms we implemented do a prebuffering at low quality and then they resume their normal operation.
To simulate a real channel we made a script to limit the connection speed at the server side (the script runs Netem [2] commands). Changing connection speed during the execution of the various algorithms permit to see the differences between them and which is better in a given situation.

3 DASH Algorithms

In the following subsections are described the different dash algorithms we implemented, each one with it's own characteristic design idea.

3.1 BitrateBased algorithm

The first DASH algorithm we describe is the simplest one. In BitrateBased algorithm the choice of the next quality to download is based only on the connection speed calculated downloading the previous segment; e.g. if a segment has been downloaded with a bitrate of 100kb/s, then the quality of the next segment asked to the server will be the closest lower or equal to 100kb/s, among the available bitrates. This approach is very simple to implement but it performs good results only if the channel has a stable connection. In a common internet channel this is not generally true; small variation on the connection speed may lead to an interruption of video playback or to a bad playback experience due to frequent quality swaps.

3.2 BufferBased algorithm

The BufferBased algorithm, differently from the previous, does not look at the connection speed to choose the next segment quality to be downloaded, it only uses the buffer length to make all the decisions. The basic idea is that, fixed a desired max buffer length, the algorithm requests different quality according to the current buffer length: highest buffer length allows better quality.
In our implementation we chose to calculate the next quality to be downloaded with a simple proportion: let B_max be the maximum capacity of the buffer, Q_max the maximum available quality, B_cur the current number of elements in the buffer and let x be the quality of the next segment to be downloaded. This last value is calculated as the solution of the following proportion:

$$B\_max : Q\_max = B\_cur * x.$$

The next segment to be downloaded will have the closest quality lower or equal to x.

3.3 FESTIVE algorithm

The acronym FESTIVE stands for Fair, Efficient, Stable adapTIVE algorithm: these are in fact the characteristic features that its developers aimed for [3]. The algorithm is divided into three main components, with the following goals:

1. Estimate the network bandwidth.
2. Select a valid bitrate for the next segment.
3. Schedule when the next segment will be downloaded.

3.3.1 Bandwidth Estimation

The algorithm estimates the bandwidth of the network using a mean computed over the last 5 samples: the calculated value is more reliable than simply using the bandwidth observed for each chunk. To

make it robust to outliers we used the harmonic mean instead of the arithmetic mean.
The difference between our implementation and the description given in [3] is that, instead of calculating the mean over the last 20 samples, we do it with the last 5: by our experiments it's a more suitable compromise between robustness to sudden variations and reactivity to the occurrence of bandwidth drop.

3.3.2 Bitrate Selection

The policy for selecting the bitrate to download for the next segment follows the one given in [3] and it's done in multiple steps.
First there is a stateless selection of the bitrate: the algorithm chooses the highest available bitrate lower than the estimated bandwidth. To make this choice stateful, the algorithm allows multi-level bitrate switches and different-rate bitrate switches. With these two steps we ensured the bitrate convergence to a fair allocation among different players.
The third step takes into account the issue of stability: switching bitrate too frequently negatively affects the user's experience. The algorithm uses the previously calculated bitrate as a reference bitrate and, if this one is different from the last downloaded, decides whether to switch quality or keep the current quality, basing on a tradeoff between efficiency and stability.
In order to do this, we calculated an efficiency and stability score for both the reference bitrate, b_{ref}, and the last bitrate, b_{last}. Then the algorithm computes the combined weighted scores:

$$score_{stability}(b) + \alpha \times score_{efficiency}(b)$$

for both the bitrates, and picks the bitrate with the lower combined score. The α constant is a tunable knob which gives more weight to the stability or efficiency factor. We experimented a good value of α to be 0.2 with our harmonic mean of the last 5 measurements.
In addition to these rules for choosing the bitrate to download for the next segment, we added an empirical condition to desperately avoiding the stop of the video: when the actual buffer dimension fall below two segments, then the index of the bitrate to be downloaded is set to one less than the previous one. This condition gradually lowers the required bitrate until the buffer dimension grows above 2, or the minimum available quality is reached.

3.3.3 Chunk Scheduling

To avoid increasing the server's bandwidth costs due to the strategy of immediate download, and to avoid the bias induced by the initial conditions due to periodic requests of download, we opted for randomized scheduling.
Instead of using a constant playback buffer dimension, let's say *targetbuf*, we treat it as an expected value; specifically, for each chunk we choose randomly a target buffer size $randbuf_i$ from a uniform distribution [*targetbuf* - Δ, *targetbuf* + Δ].
At the steady state, the segments will be downloaded roughly periodically but with some jitter as the target buffer size is randomized. In our implementation we choose *targetbuf* = 10 and Δ = 1.

4 Experimental Results

To test the effectiveness of the algorithms we took different tests in different situations all using three clients concurrently:

- Same algorithm with stable channel, repeated for the three algorithms using different channel bitrate.
- Different algorithms with a stable channel
- Different algorithms with dynamic variation of the channel bitrate.

4.1 Stable channel using same algorithm

In this case all the algorithms perform very well. We tested three different situations: full bandwidth, 4Mbps upload limit and 1.6Mbps upload limit at server side. All these tests shown a good stability and a fair bandwidth share: the speed of the connection is equally divided among the three client and the quality is almost the same.

4.2 Stable channel using different algorithms

Also in this case stability and fair bandwidth share are verified. Some small variations are observable on the BitrateBased and the BufferBased algorithms, FESTIVE instead has a little better stability.

4.3 Variable channel using different algorithms

These tests are the most significant because they allow to observe advantages and disadvantages of each algorithm. In particular we report three tests we've done simulating some changes in the connection speed (the trend of the connection can be seen in the Bitrate graph because it tracks quite well the connection speed):

1. Connection with a low peak: when the connection has a quick variation from its target, BitrateBased follows the connection speed adding to the buffer a segment with very low quality. FESTIVE waits to lower the bandwidth, so it can perform smoother variations (Figure 4).

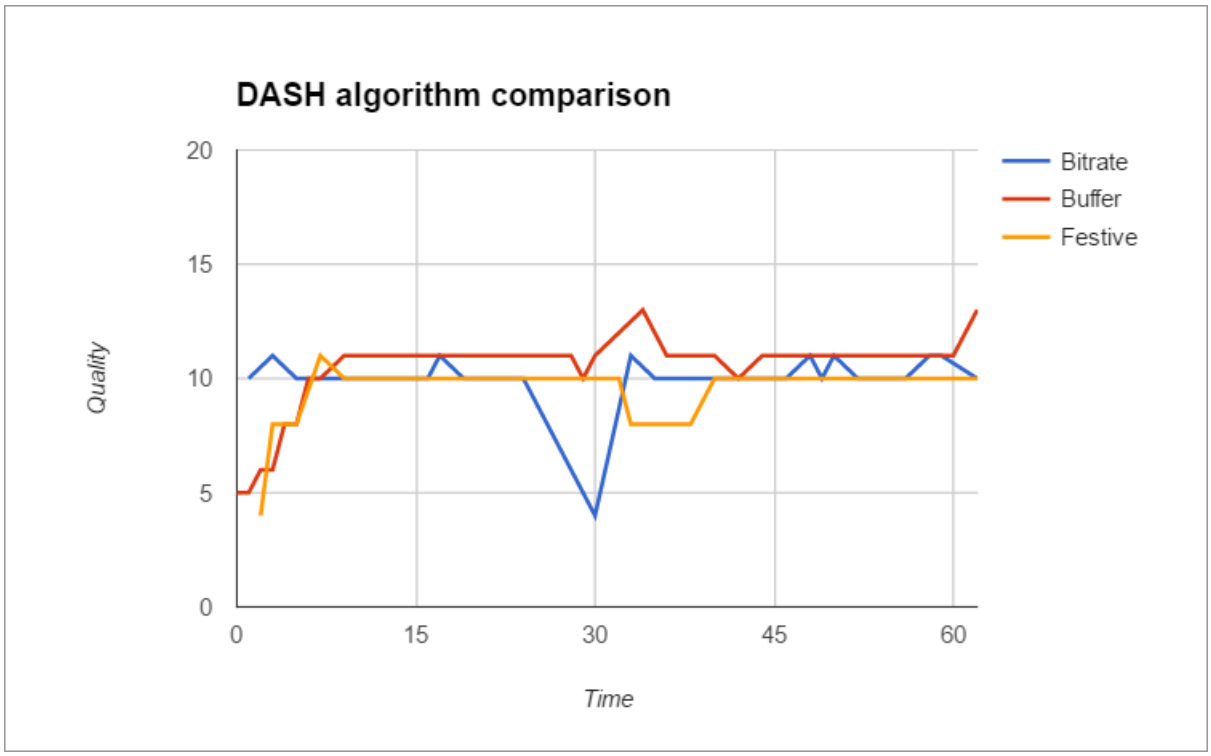

Figure 4: Low peak on connection during playback

2. Connection speed perform a low step during the video playback. As we can see in Figure 5 the three algorithms need different times to adapt to the connection speed variation: BitrateBased is the faster but also the most flickering one, BufferBased in our test is the one that ensures a little better average quality, FESTIVE is more cautious but more stable.

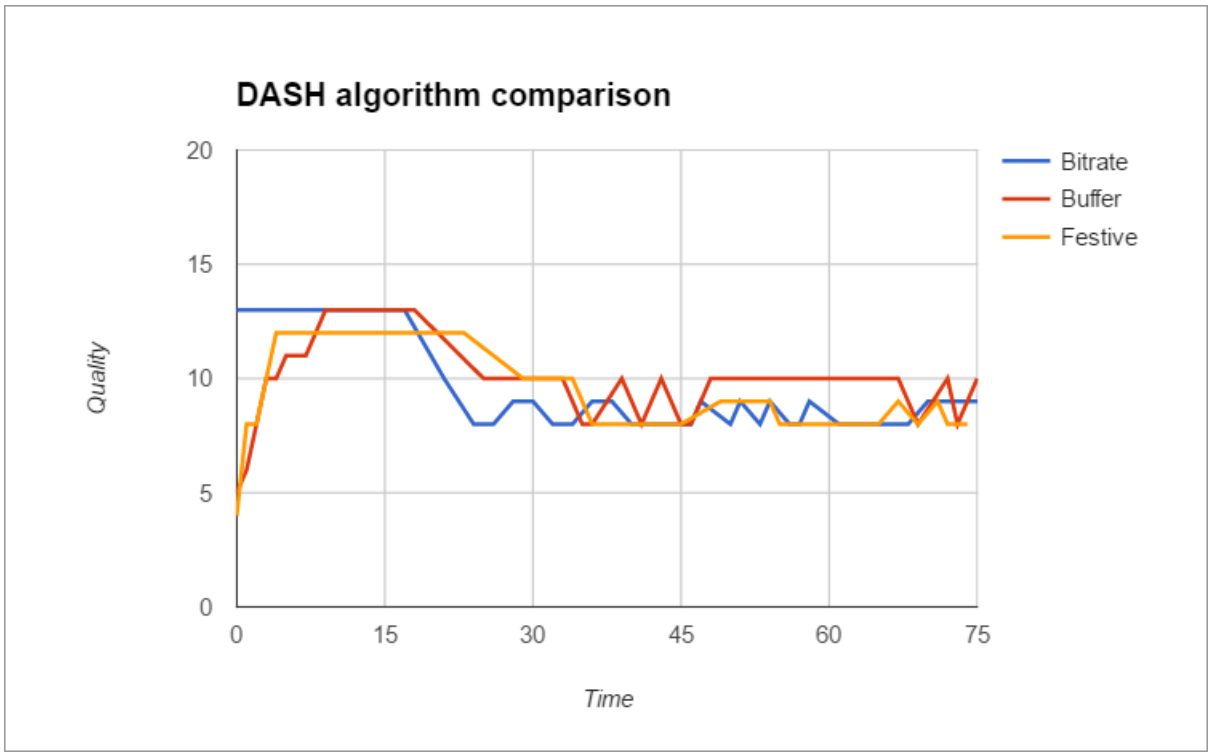

Figure 5: Low step on connection speed during playback

3. Upload limit of the server alternates between 4Mbps and 1.6Mpbs. As we can observe in Figure 6, in this case the three algorithms performs different choices. As soon as the connection speed changes, BitrateBased algorithm changes significantly the quality, at the opposite FESTIVE algorithm try to avoid sharp changes in quality to perform a better playback experience. The BufferBased algorithm is almost in the middle of this two approaches.

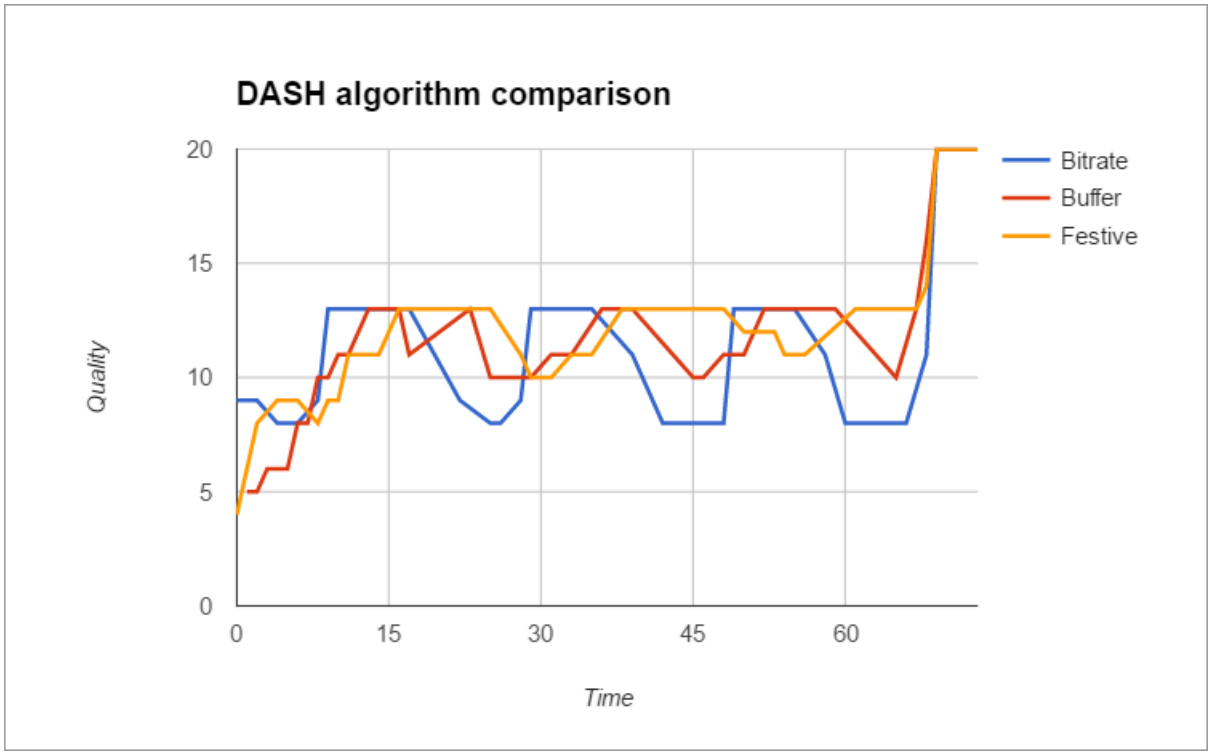

Figure 6: Connection speed alternates between two values and full speed at the end

5 Conclusions

In real internet connections an easy bitrate approach is not recommended because of the instability of the link, instead a buffer approach can be better in terms of user experience. However, if the connection is really unstable, BitrateBased is the one that can work without problems. A combination of BitrateBased and BufferBased might be also be better than using one or the other individually. FESTIVE algorithm, in practice, allows a good result in video quality stability, but performing the mean between a certain number of samples is not the better choice in extremely troubled networks.

References

[1] Vlcj library, used to develop the client. URL: http://capricasoftware.co.uk/#/projects/vlcj.

[2] Netem: used to limit the connection speed. URL: https://wiki.linuxfoundation.org/networking/netem.

[3] J. Jiang, V. Sekar, H. Zhang. Improving Fairness, Efficiency, and Stability in HTTP-based Adaptive Video Streaming with FESTIVE. Proc. CoNEXT '12, France 2012.