Buffer-based algorithm implementation to Rate Adaptation

Piergiorgio Ladisa
Nicola Sebastianelli

22/11/2018

Context

These days we are witnessing to the continuous growing demand of streamed content in network: anyone asks anywhere and in any-moment for contents to servers from their mobile devices. Hence, since the percentage of contents in the network is becoming everyday bigger, CDNs and ISPs are called to respond to this demand. The paradigm of the multimedia systems and for CDN is mainly the so-called Quality of Experience (QoE). This is very different from the paradigm on which Internet was designed, i.e. Quality of Service (QoS) and Best Effort, in fact the quality of experience is really subjective and very difficult to measure. Nevertheless, different protocols and strategies have been proposed in order to met the demand of users of streaming of contents. Firstly, different strategies of streaming exists:

- all-in-once, i.e. the content is entirely sent to the user. This approach is enforced by the server and the encoding rate is unintentional, since the client's buffer is filled up, in the same way in which the TCP buffer is filled up; since TCP performs the flow control, the client with this approach read at the encoding rate;
- throttling, i.e. the rate is adapted to the user with two possible approach:
  - on-off-S, in which the connection is maintained persistently;
  - on-off-M, in which the connection is not maintained persistently

  These two approaches are caused by the client application, that periodically stops reading from the TCP socket. In a general case of pseudo-streaming, the video is buffered until the buffer is enough full and, once reached the steady state, the controller starts doing an on-off step, downloading the block size of the video.

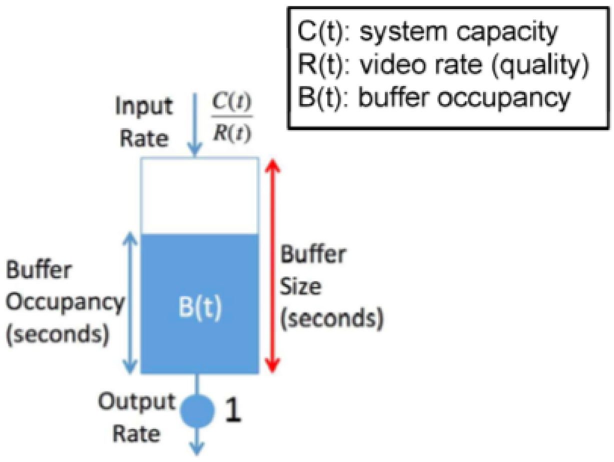
In this context, the mainly adopted protocol is HTTP Adaptive Streaming protocol (HAS). Here the idea is to make the requested bitrate of the video fit with the varying network resources, ensuring the best possible quality of experience for the user, based on the current situation of the network. This adaptation is usually done on client-side. Here the content is divided in chunks and available in different qualities. So HAS allows to encode each segment in different quality referring to the current bandwidth and so the stream is splitted into a sequence of segments, instead of downloading one entire large file (as in the all-in-once approach). The policies of HAS are:

- Rate-based, that select the highest possible video representation referring to the measured speed from the previous received chunk;
- Buffer-based, that uses different thresholds for the buffer in a way in which the more the buffer is filled, the more quality of the video is. Thus, with this approach, the quality can switch step-by-steo at each new request of a video segment: it is impossible to jump from the lowest to the highest quality just in one step;
- Buffer and Rate based. This is an hybrid approach, i.e. is based on the rate-based approach basically, but the previous chunk download speed is weighted using a factor that depends on the amount of the saturation of the buffer: if the buffer is depleting the previous chunk, the download speed will be considered to be less than the measured one and hence a lower quality is selected.

Chapter 1

A buffer-based approach to rate adaptation

When the client want to access to a video-content for streaming, it chooses which the video rate to stream by monitoring network conditions and estimating the available network capacity. This process is referred to as *adaptive bit rate selection* or *ABR*. ABR algorithms try to balance two opposite goals. The first goal is to maximize the video quality choosing as video rate the highest supportable by the network. The second goal is to minimize re-buffering events, which cause the video to halt if the client?s playback buffer goes empty. Data is requested in chunks and the buffer drains at 1 unit/second. So if the video-rate $R(t)$ is greater than the system capacity $C(t)$, new data is added at $C(T)/R(T) < 1$, depleting the buffer.



In order to maximize video quality, a service could just stream at the maximum video rate R_{max} all the time, but this would risk extensive re-buffering.

On the other hand, to minimize rebuffering, the service could just stream at the minimum video rate *Rmin* all the time but this extreme would lead to low video quality. Hence, the design goal of an ABR algorithm is to simultaneously obtain high performance on both metrics in order to give users a good quality of experience. Firstly, the client measures how fast chunks arrive to estimate capacity, let us say *C(t)*. The estimate enriched with knowledge of the buffer occupancy, which is represented with an adjustment factor *F(B(t))*, i.e. a function of the playback buffer occupancy. So, the selected video rate is *R(t)* = $F(B(t))C(t)$; different designs use different adjustment functions *F(.)*. When the buffer contains many chunks, $R(t)$ can safely deviate from $C(t)$ without triggering a rebuffer. The client can aggressively try to maximize the video quality by picking $R(t) = C(t)$. But when the buffer is low, the client should be more conservative.

We say that an ABR algorithm is *buffer-based* if it picks the video rate as a function of the current buffer occupancy $B(t)$. The region between:

- $[0, B_{max}]$ on the buffer-axis;
- $[R_{min}, R_{max}]$ on the rate-axis;

defines the feasible region. So, any curve $f(B)$ on the plane within the feasible region defines a *rate map*, i.e. a function that produces a video rate between R_{min} and R_{max} given the current buffer occupancy. Let us assume that:

- the chunk-size is infinitesimal;
- video rates within $[R_{min}, R_{max}]$ are continuous;
- videos are encoded at a constant bitrate;
- videos are infinitely long;

Thus, as long as $C(t) \geq R_{min}$ and $f(B)$ tends to R_{min} when B tends to 0, there will not be unnecessary rebuffering events. So, as long as $f(B)$ increases to R_{max}, the average video rate will match the average capacity when it is true that $R_{min} < C(t) < R_{max}$.

1.1 BBA0-Algorithm

The rate map implemented by the BBA0 algorithm is the following:

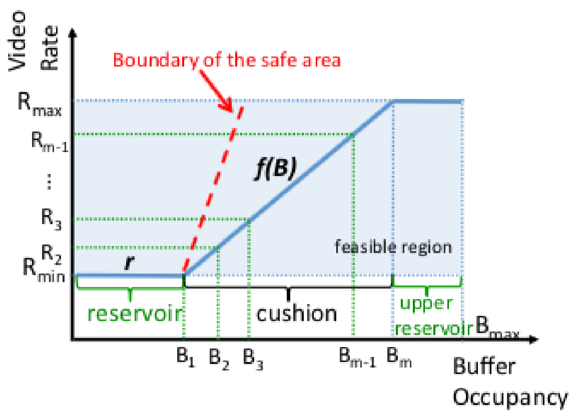


Figure 1.1: Rate map of the BBA0 algorithm

This algorithm add reservoir r to pad for finite chunk sizes and some $C(t)$ variation. So, while filling reservoir, it request R_{min} and must have r greater or equal to the minimum chunk size. In addition, an upper reservoir is added to reach R_{max} before the buffer B is full and above the "safe area" is where the buffer will not deplete into r if $C(t)$ suddenly drops, but does not reach values lower than R_{min}.

BBA0 Algorithm

The algorithm presented in [1], in pseudo-code, is the following:

Input:

- Rate$_{prev}$: previously used video rate;
- Buf$_{now}$: current buffer occupancy;
- r: size of reservoir;
- cu: size of cushion.

Output: Rate$_{next}$: next video rate.

```
if Rate_prev = R_max then
        Rate+ = R_max
else
        Rate+ = min{Ri : Ri > Rate_prev}

if Rate_prev = R_min then
        Rate- = R_min
else
        Rate- = max{Ri : Ri < Rate_prev}

if Buf_now <= r then
        Rate_next = R_min
```

```
else if Buf_now >= (r + cu) then
        Rate_next = R_max

else if f(Buf_now) >= Rate+ then
        Rate_next = max{Ri : Ri < f(Buf_now)};

else if f(Buf_now) <= Rate- then
        Rate_next = min{Ri : Ri > f(Buf_now)};

else
        Rate_next = Rate_prev;

return Rate_next;
```

This constitute a buffer-based algorithm to the rate adaptation in playing contents from the client.

Implementation of BBA0 Algorithm

The implementation that we have done of the BBA0 algorithm is written in Python and is presented below.

Since the BBA0 Algorithm regards the algorithm for the controller in the player, what we have implemented is a controller used by the player defined in the script *play.py*.

Firstly, we have defined the class BBA0Controller, in order to be imported by the player and executed through the specification of the use of this controller. So the definition of the class is the following:

```
class BBA0Controller(BaseController):

    def __init__(self):
        super(BBA0Controller, self).__init__()

    def __repr__(self):
        return '<BBA0Controller-%d>' % id(self)
```

The code below specifies the rate-map function $f(B)$, that is the line passing from the points $(reservoir, R_{min})$ and $(reservoir + cushion, R_{max})$. Hence the equation for the line is the following:

$$f(B) = B * (\frac{(R_{max} - R_{min})}{cushion} + (R_{min} - \frac{reservoir}{cushion}) * (R_{max} - R_{min})$$

so, the declaration of the function member of the class requires as input the object itself (i.e., *self*), the current value of the buffer (i.e., B_now), the value for the *reservoir*, the *cushion* and the maximum and minimum values of the rate (i.e., respectively R_max and R_min).

```
def f(self, B_now, reservoir, cushion, R_max, R_min):
    # The function f corresponds to the line equation between the
    # points (r,R_min) and (r+cu,R_max) when the value of Buf_now
    # is bounded by r and r+cu
    return B_now * ((R_max - R_min) / cushion) +
                    (R_min - ((reservoir / cushion) * (R_max - R_min)))
```

WRITE THE EXPLANATION OF FUNCTIONS MAXR AND MIN R

```
def maxR(self, constraint):
    Rates = self.feedback['rates']
    results = []
    for i in Rates:
        if (i < constraint):
            results.append(i)
    return max(results)

def minR(self, constraint):
    Rates = self.feedback['rates']
    results = []
    for i in Rates:
        if (i > constraint):
            results.append(i)
    return min(results)
```

The method member of the class defined below is the actual implementation of the BBA0 algorithm. In order to reproduce the behavior depicted in Figure 1.1, the percentages for the

- *reservoir* is setted at 15%;
- *cushion* is setted at 65%;
- *upper-reservoir* is setted at 20%;

```
def calcControlAction(self):

    # Retrive current iteration variables
    percentage_reservoir = 0.15
    percentage_cushion = 0.65
    percentage_upper_reservoir = 0.20
    reservoir = self.feedback['max_buffer_time'] * percentage_reservoir
    cushion = self.feedback['max_buffer_time'] * percentage_cushion
    R_max = self.feedback['max_rate']
    R_min = self.feedback['min_rate']
    R_curr = self.feedback['cur_rate']
    B_now = self.feedback['queued_time']

    # Compute upperbound
    if R_curr == R_max:
        R_plus = R_max
    else:
```

```
    R_plus = self.minR(R_curr)

# Compute lowerbound
if R_curr == R_min:
    R_minus = R_min
else:
    R_minus = self.maxR(R_curr)

# Compute new rate based in current buffer region

# Buffer in reservoir area
if B_now <= reservoir:
    Rate_next = R_min

# Buffer in upper reservoir area
elif B_now >= reservoir + cushion:
    Rate_next = R_max

# Buffer in cushion area
elif self.f(B_now, reservoir, cushion, R_max, R_min) >= R_plus:
    Rate_next = self.maxR(self.f(B_now, reservoir, cushion, R_max, R_min))
elif self.f(B_now, reservoir, cushion, R_max, R_min) <= R_minus:
    Rate_next = self.minR(self.f(B_now, reservoir, cushion, R_max, R_min))

else:
    Rate_next = R_curr

return Rate_next
```

Bibliography

[1] Te-Yuan Huang, Ramesh Johari, Nick McKeown, Matthew Trunnell, and Mark Watson. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. *SIGCOMM Comput. Commun. Rev.*, 44(4):187–198, August 2014.