ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Informatica – Scienza e Ingegneria

TESI DI LAUREA
in
Reti di Calcolatori M

Evaluation of new HTTP Adaptive Streaming algorithms: the clients' and network's perspectives

Candidato:
Sergio Livi

Relatore:
Chiar.mo Prof. Ing. Antonio Corradi

Correlatori:
Lucile Sassatelli, Maître de conférences
Guillaume Urvoy-Keller, Professeur des universités
Université Nice Sophia Antipolis

Anno Accademico 2014/2015
Sessione III

Abstract. Lo *streaming* è una tecnica per trasferire contenuti multimediali sulla rete globale, utilizzato per esempio da servizi come YouTube e Netflix; dopo una breve attesa, durante la quale un buffer di sicurezza viene riempito, l'utente può usufruire del contenuto richiesto. Cisco e Sandvine, che con cadenza regolare pubblicano bollettini sullo stato di Internet, affermano che lo streaming video ha, e avrà sempre di più, un grande impatto sulla rete globale. Il buon design delle applicazioni di streaming riveste quindi un ruolo importante, sia per la soddisfazione degli utenti che per la stabilità dell'infrastruttura.
HTTP Adaptive Streaming indica una famiglia di implementazioni volta a offrire la migliore qualità video possibile (in termini di bit rate) in funzione della bontà della connessione Internet dell'utente finale: il riproduttore multimediale può cambiare in ogni momento il bit rate, scegliendolo in un insieme predefinito, adattandosi alle condizioni della rete. Per ricavare informazioni sullo stato della connettività, due famiglie di metodi sono possibili: misurare la velocità di scaricamento dei precedenti trasferimenti (approccio *rate-based*), oppure, come recentemente proposto da Netflix, utilizzare l'occupazione del buffer come dato principale *(buffer-based)*.
In questo lavoro analizziamo algoritmi di adattamento delle due famiglie, con l'obiettivo di confrontarli su metriche riguardanti la soddisfazione degli utenti, l'utilizzo della rete e la competizione su un collo di bottiglia. I risultati dei nostri test non definiscono un chiaro vincitore, riconoscendo comunque la bontà della nuova proposta, ma evidenziando al contrario che gli algoritmi buffer-based non sempre riescono ad allocare in modo imparziale le risorse di rete.

Contents

Chapter 1

Introduction

Like almost every piece of information, videos are seen as files in the digital world. In general we must download completely a remote file in order to open it. This is not true if we speak about video streaming: the playback starts potentially way before the end of the download; most people use it everyday, for example when using services like YouTube or Netflix.

Cisco [1] states that video streaming will be 79% of all consumer Internet traffic in 2018. Sandvine [2] declares that during peak hours in North America, Netflix and YouTube are together responsible for the 48.9% of the transfers towards the users. It is clear that video streaming is a topic with a rising impact on today's Internet. Badly designed applications may pose problems not only to end users, but also to streaming providers and ISPs (Internet Service Providers).

Services have some indicators to measure user satisfaction; these indicators are gathered in the so-called Quality of Experience (QoE): a collection of metrics, usually not so easy to measure (mostly because of their subjectiveness), that evaluates the user satisfaction with the service. QoE for video streaming is believed to heavily depend on two factors: rebuffering events (when the video stops because of insufficient network throughput) and bit rate obtained (the more the bit rate is high, the more the images are clear, the more the user is satisfied).

In order to maximize the QoE in video streaming applications, HTTP Adaptive Streaming (HAS) is widely used: it avoids rebuffering while at the same time getting a high bit rate. This is done by switching the bit rate on the fly while the playback is in progress, so as to absorb network fluctuations and obtain a high video quality.

Current HAS implementations measure the download throughput in order to chose the best bit rate sustainable, we'll refer to this approach as *rate-based*; as highlighted by [3, 4, 5, 6] and many more, this method could suffer from some issues related to the interactions with the underlying layers.

Video streaming applications do not show immediately the data received to the user; instead, they keep a *playout buffer* of a variable length which absorbs the unreliability of the network, as there aren't guarantees about the timing, even if the network is relatively stable (i.e. packets could arrive in bursts). An idea that comes from Netflix [7] suggests to choose the next bit rate using as input the utilization of this buffer, without trying to estimate the bandwidth at all, we'll call this technique *buffer-based*. The authors demonstrated that with their strategy, the Quality of Experience increased, as the users obtained similar bit rates averages with less rebuffering events.

The objective of this work is to dig on the two families of rate adaptation algorithms, and to compare some of the members of them, looking at metrics concerning QoE, network and competition between clients sharing a bottleneck; for the comparison we used VLC media player modified so to be capable of running different algorithms, on a testbed which permits network shaping, leveraging measurement tools to get the data from the clients, from the TCP stack and from the network equipment between the server and the clients.

In chapter 2 we provide an overview of the most important network technologies; the context of this work is provided in chapter 3, detailing the principles of HAS and how the different algorithms cope with the problem. The methodology and the details of the implementation are presented in chapter 4; a discussion of the results obtained is given in chapter 5.

Chapter 2

The global network and its protocols

In this chapter we introduce the structure of the Internet and one of its most important protocols: TCP, we will then talk about HTTP, an application protocol built on top of TCP, that is responsible of the most part of the traffic on the Internet.

2.1 The Internet

We can see the global network as an heterogeneous set of randomly-interconnected machines, where each of them can communicate with any other one, directly or by using one or more intermediate hops. An example, shown fig. 2.1.1, comes from the first stages of the Internet; we are in 1969, and this is the topology of ARPANET, the first computer network; each node in the picture is actually a couple of machines: the one represented by a circle is in charge of maintaining the *links*, while the other is the actual *host* to connect. This structure is still used in the current implementation: the so-called *routers* are responsible to manage the links and sort the traffic between end hosts.

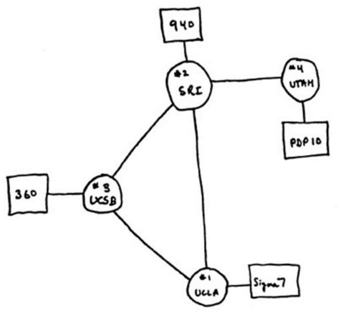


Figure 2.1.1: ARPANET in 1969: the first four nodes of the future Internet

ARPANET was at the beginning a military project, intended to keep under control the nuclear weapons in the USA territory even under a nuclear attack; under this conditions, it is easy to imagine how a part of the nodes could suddenly disappear. To raise the probability that every point remains accessible, it is necessary to add links in a web-like topology; having thus multiple possible paths between a given source and destination; for example in fig. 2.1.1, two of the three nodes on the left side could communicate between each other, even if the third one is down.

To exploit the multi-path topology, the information is split in *packets*, labeled with source and destination, as Donald Watts Davies and Paul Baran suggested in the sixties [8]. The packets reach independently the destination without any kind of reservation of the path, and without any confirmation of the delivery; this is the most basic and lightweight communication mean on the computer networks; the implementation of this protocol on the Internet is called Internet Protocol (IP), with two versions that currently live together, as the transition is in progress: IPv4 and IPv6.

An IP packet is made by an *header* and a *payload*; the header contains source/destination pair, among other things, while the payload carries the actual data to transfer. The machines are identified by their IP address, that is unique in the whole Internet.

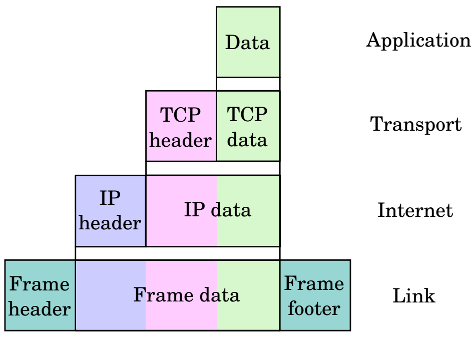


Figure 2.1.2: Encapsulation of data through the layers, as described in RFC1122[9]. Source: Wikipedia.

IP is simple. This is one of its points of strength that permitted its diffusion: it is fairly easy to support it over any physical mean of communication: copper, optical fiber, radio and so on. Fig. 2.1.2 shows the four layers of the Internet, as defined in RFC1122 [9]; IP sits in the middle of the “Internet hourglass”: many protocols carry IP packets as payload (*Ethernet*, *Wi-Fi*, *UMTS*...), and all standard protocols are built on top of IP.

On the *transport* layer, we find three different protocols that work over IP:

- ICMP (Internet Control Message Protocol): useful for debugging purposes, carries simple informations and error notifications.
- UDP (User Datagram Protocol): it adds the concept of *port*, giving thus the possibility to run simultaneously multiple network applications on the same host, routing the data to the correct one. There is no guarantee concerning the correct delivery nor the order of the packets received.
- TCP (Transmission Control Protocol): in addition to ports, it adds some guarantees, such as delivery confirmation and packet order. It aims to establish a connection between the end hosts, exchanging IP packets both for control messages and user data.

Routing

The Internet is an heterogeneous set of different interconnected IP networks, where each independent network is connected to a number of other networks; for each of them, the internal and external links are controlled by *routers*, machines configured for this purpose (represented by a circle in fig. 2.1.1). So the Internet could be seen as a graph, were the arcs are links and the nodes are routers. Our computers and servers are then connected to these routers, and all the data we exchange pass through the graph to reach the other peer. The links have in general different performances, due to physical reasons and depending on the quantity of data crossing them. Because of these differences, a router could see bursts of traffic coming from a high speed link and going to a lower speed one; this is why, for each output port, the routers maintain a *buffer* so to have a queue of outgoing packets that empties at the link's speed. When the amount of data flowing towards a link keeps having an unsustainable rate (higher than the capacity of the same link), this buffer could reach its limit; in this case, the router is forced to drop the subsequent packets; and the link is *congested*.

Without a specific mechanism to regulate traffic and avoid congestion, we could expect the most part of routers out-of-order and no communication possible across the Internet. For this reason, we need a mechanism to avoid congestion and ensure, as much as possible, a fair allocation of the available bandwidth; two approaches are possible: manage the traffic on the routers, or make the end hosts to send the data at a convenient rate. For practical reasons, the second approach was adopted at the first steps of the Internet and bundled inside TCP, under the name *congestion control*.

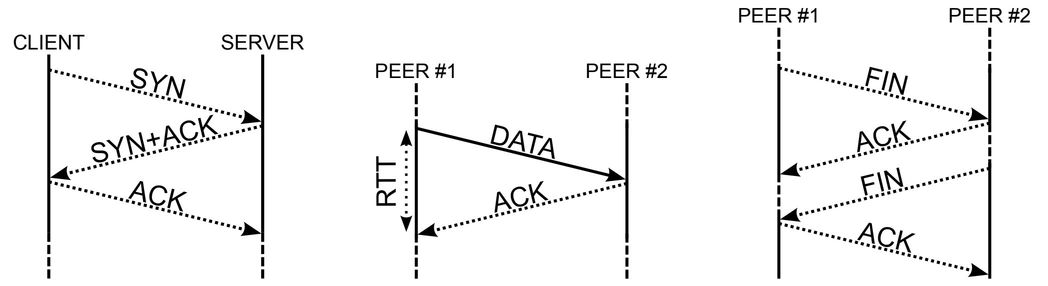
As the memory price decreases faster than how the link speed increase, the buffers tend to be oversized, especially on cheap, not-well-designed equipment; an excessive buffer size yields *bufferbloat*: the packets sits on the buffer, waiting to be routed for a time way longer than expected. This turns into an important issue in some fields, such as VoIP calls, on-line games or e-commerce web sites.

2.2 TCP

TCP was first defined in RFC793 [10], as *"a connection-oriented, end-to-end reliable protocol designed to fit into a layered hierarchy of protocols which support multi-network applications. The TCP provides for reliable inter-process communication between pairs of processes in host computers attached to distinct but interconnected computer communication networks"*.

TCP by sending IP packets, establishes a connection between two end hosts and exchange data in a reliable way. The two involved peers have different roles, as the model is service-user: one acts as a *server*, exposing an *open port* that accepts TCP connections, the other as a *client*, sending a special packet to request the communication.

The connection is initiated by the client that sends a "synchronization packet", SYN, as in fig. 2.2.1a; the server acknowledges the reception replying with a SYN+ACK packet; the client then ends the *handshake* sending a last acknowledgement, ACK. When the server receives this ACK, both sides consider the connection active, and identify it by the 4-tuple built with both



(a) *three-way-handshake*: the connection initialization

(b) *ACK* mechanism: reliability and order

(c) *four-way-handshake*: the end of the connection

Figure 2.2.1: TCP phases

IP addresses and both TCP ports. From this moment on, there is no more difference between the server and client roles.

A TCP connection, as seen from the upper layer, is a streams of flowing data for each of the two opposite directions. These two flows, to adhere to IP, must be split in packets, so to travel between end hosts over the network. Each data packet contains a *sequence number* that indicates its position on the stream, this ensures an ordered delivery to the upper layer even if the corresponding IP packets arrived out of order. For each chunk of data sent, the receiver generate an acknowledge, that contains the next sequence number expected, confirming at the same time the reception of all the data; the time passed between the sent of the packet and the reception of the ACK is called Round Trip Time (RTT), as shown in fig. 2.2.1b.

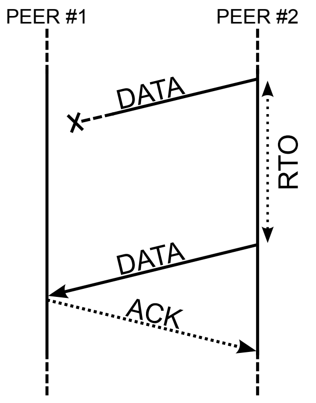


Figure 2.2.2: TCP, Retransmission Time-Out

The *reliability* of TCP is obtained in a quite simple way: the memory area corresponding to a data packet is erased only when the matching ACK is received; if no ACK arrives, the sender will wait a Retransmission Time-Out (RTO), and eventually resend the packet (fig. 2.2.2).

The connection is closed *asymmetrically*, by sending the control packets

FIN, that are confirmed by an ACK; the flag FIN announces to the peer that no more data will be sent in its direction; the connection is actually closed when both sides closed their side (fig. 2.2.1c). TCP includes also a mechanism to *abort* a connection, by sending a *reset* request (RST).

2.2.1 Flow control

The data is split into packets in order to travel through the network, and every packet is acked, so to ensure reliability. Wait for the previous ACK before sending the next packet, anyway, is not so effective.

An example scenario: the two end hosts connected through a 10Mbit/s Ethernet link. For Ethernet, the MTU (Maximum Transmission Unit, the maximum payload size allowed) is 1500 bytes, the Ethernet header is composed by 14 additional bytes, total: 1514 bytes; IP and TCP headers are 20 bytes each; so the best-value TCP packet on Ethernet carries 1440 bytes of user data. The time to transfer this packet will be 1.2 milliseconds, and the ACK could be delivered 50 microseconds after (excluding the computation time at the destination); but we have to take into account also extra delays, for instance caused by the speed of light: if our link is 1500km of wire, we have to add 5 milliseconds; the total time passed between the send of the packet and the reception of the ACK is 11.2ms, with an average throughput of about 1Mbit/s: ten times less the actual bandwidth of the underlying medium. There is a number of other reasons that could cause delays, including: the other peer isn't processing the packets because of high load, the data packets or the ACKs are stuck in queues on the intermediate routers.

To obtain better performances, TCP sends more than one IP packet at a time, allowing a certain number of unacked packets. This *receiver window* (RWND) is moved forward as ACKs arrive, so to maintain the number of *in flight* packets equal to its size.

As a first strategy, the window must be sized to not overflow the receiver's buffer, as there is no point in sending data that the other peer can't process; for this reason, the TCP header contains a field to specify the amount of free space on the receiving buffer. The sender will limit the transmission to this amount of bytes, and update the window size to the advertised value on every received packet (ACKs included).

2.2.2 Congestion control

The mechanism presented above is useful to limit traffic end-to-end, not considering the status of all routers relaying the traffic; the main goal of congestion control is to avoid overloading the intermediate network equipment and links, while letting the machines dynamically get their own fair share of the available bandwidth. The sender keeps a *congestion window* (CWND), in addition to the *receiver window* of flow control; the effective number of allowed unacked packets will be the $min(RWND, CWND)$.

This CWND starts from a predefined size (*initial window*). The sender will fill this window sending as fast as possible, it will then stop waiting for ACKs. When an ACK arrives, the window size will be increased, and one or more new packets will be sent. Conversely, the window will be decreased when a loss is sensed (suspect congestion), i.e. when one packet remains unacked while its successors are, or when the ACK does not arrive after a certain timeout.

The exact behavior and the precise amounts for increases and decreases depends on the chosen algorithm (for instance, in Ubuntu 14.10, 13 different algorithms are available to be plugged into the kernel); although, some guidelines are given in RFC5681 [11].

Slow start and congestion avoidance

MSS (Maximum Segment Size) is the maximum allowed size for the TCP payload, it depends on many factors, and its rationale is to avoid fragmentation at the lower layers; the transfer of a big file, for example, will be divided by MSS and produce a number of TCP packets.

RFC5681 [11] recommends to set the initial window to a low value (between 2×MSS and 4×MSS), and set a variable named *ssthresh* (slow start threshold) to an arbitrarily high value; when the current CWND value is less than the slow start threshold, the *slow start* algorithm is used, otherwise *congestion avoidance* is selected.

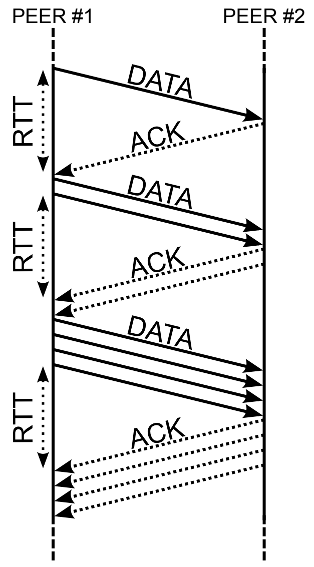


Figure 2.2.3: TCP slow start, how the CWND grows.

When in slow start, for every ACK received, the CWND will be increased by a MSS, so the resulting congestion window is doubled after an entire CWND sent and acked, or an RTT (see fig. 2.2.3); in this phase, the window grows exponentially to try to get to the maximum value quickly.

Congestion avoidance slows down the growth rate to linear: the increment for each ACK will be MSS×MSS/CWND, thus an MSS for each RTT; this slow increase is meant to adapt to the channel bandwidth in a graceful way, avoiding high congestion.

What happens upon loss

The first and most naive congestion control algorithm is named *Tahoe*; it uses a timeout to determine the existence of a loss: if the ACK is not received before the expiration of the RTO, it will be considered a loss. The ssthresh will be set to half of the current congestion window, and the congestion window will be reduced to 1 MSS.

Stepping back to a very low CWND will suddenly drop the speed of the transfer, and moreover there are some more hints that could drive to good decisions without waiting for the RTO, in particular *duplicate ACKs* (DUPACKs): if a packet is lost, for each subsequent packets delivered, the receiver should produce a duplicate ACK of the last in-order packet received (that is equivalent to ask for the missing packet).

TCP *Reno* exploit this information to improve the recovery performances: at the third DUPACK, it will resend the requested packet (without waiting for the timeout), halve the CWND and set the slow start threshold to the same value, thus avoiding the slow start phase and reacting as soon as possible, before the RTO. If, anyway, the RTO arrives, the behavior of Reno is the same as for Tahoe.

As said, there is a number of other possible behaviors, currently deployed on the Internet, that optimize the edge cases, or use other TCP options, such as selective ACKs (SACKs), a mechanism that aims avoiding the resend on already-received, out-of-order packets.

Congestion window behavior

In order to exploit the available bandwidth of a defined path, or to *fill the pipe*, we intuitively need a certain number of in-flight bytes; this number could be calculated by multiplying the bandwidth times the round trip time; this is called *bandwidth-delay product*, or BDP, and we expect the congestion window to tend spontaneously to this value. Conversely, we can affirm that the current value of CWND, specifying the number of in-flight bytes, indicates directly the instantaneous throughput of the corresponding flow.

Moreover, when in congestion avoidance, the CWND grows linearly and decreases exponentially; this approach is referred also as *additive increase multiplicative decrease* or AIMD. This strategy is vital in TCP, in particular when many active connections compete on the same bottleneck: it is demonstrated [12] that *eventually* each actor gets the fair share of the medium. It is important to note that this process takes time, depending on the characteristic of the links and on the congestion control algorithms in use.

2.3 HTTP

HTTP stands for Hypertext Transfer Protocol; it was used since 1990, and its first version is defined in RFC1945 [13]; created to transfer HTML hypertext files, it is now used with almost every kind of data.

HTML, Hypertext Markup Language, is a file format that permits to link different related documents; it introduces also a way to identify and locate this related content: URLs (Uniform Resource Locator), pieces of informations specifying the protocol that must be used and other protocol-specific data to obtain the document; HTML recommend but *not* enforce the use of HTTP. The applications that fetch and render HTML pages are called *web browsers*.

An HTTP URL is in the form `http://servername/path/to/file.html`, where `servername` indicates the IP address or the *hostname* (mnemonic name, from which the clients will find the IP address) of the server, and `path/to/file.html` is the name of the document; to retrieve the document, the browser will open a TCP connection to the server at port 80, and send:

```
GET path/to/file.html
[optional headers]
```

The server will then reply with some *headers*, specifying details about the file, such as the format, the modification date, etc., followed by the document requested. The connection is then closed.

HTTP is stateless, as the connection is established to download a single file, and no track is kept along the transfers; the *cookie* mechanism adds the concept of state in HTTP, by inserting a piece of information in the headers in both directions.

The currently most deployed version, HTTP/1.1 (defined in RFC2616 [14]), introduces a number of new functionalities. One of them, particularly useful for performance, is the `connection` header: when set to `keep-alive`, the TCP connection is not closed right after the transfer, and could be used for the next request; reusing the connection has two great advantages:

- reduces the latency of the order of one RTT: a new TCP three-way-handshake is avoided,

- reuses the old congestion windows, already tailored for the path between server and client.

Another functionality introduced in HTTP/1.1 is *pipelining*: the client could pack multiple requests without waiting the corresponding responses, which will be sent sequentially by the server in the same order; this approach speeds up the transfer of multiple resources hosted in the same machine.

A new version of HTTP is defined in RFC7540 [15], dated 2015; HTTP/2 adds some interesting capabilities, helpful to gain some milliseconds in page load:

- header compression: depending on the transferred contents, the HTTP headers could have a non-negligible share of the whole download;
- parallel transfers: concurrent HTTP requests and responses could be run over the same connection;
- *server push*: a mechanism that allows the server to send unsolicited data, predicting a future request, for example if an HTML document includes some images (even though an image could be embedded directly inside the HTML code, usually it is kept on the server as a separate file, with its own URL), the server could assume that the client will need them soon, and push them right after the document.

HTTP/2 is widely supported by the major web browsers, but not yet extensively used on web sites, as it requires an update of both the infrastructure and web development community.

Chapter 3

Streaming over the Internet

The context of this work is characterized in this chapter: it starts presenting how videos are represented in digital form, continuing with video streaming and HTTP Adaptive Streaming, explaining then the basics of rate-based algorithms and their drawbacks, followed by a description of the buffer-based algorithms.

3.1 Video compression

A video file contains separately the audio and the images, because they are digitalized using different strategies.

The audio is converted by taking the value of *amplitude* of the signal at regular intervals; this sampling has to be done in the order of thousands of times a second, so to respect the *Nyquist-Shannon sampling theorem*, that establish the minimum frequency at which an analog signal must be sampled so to be reconstructed from its digital form without distortion; in particular, if the original signal is limited below a frequency f_M, the sampling frequency must be $f_s > 2f_M$. Human hearing works in the range $20 - 20.000$Hz, so to reproduce accurately a sound, f_s must be greater than 40.000Hz.

The images are first split into a number of little squares, called *pixels*, in a grid with previously defined size, called *resolution*, for example 1280x720;

then, each of these pixels is saved extracting the value of three colors: red, green and blue; this procedure is repeated on a predefined frequency, called *frame rate*, for example 50 times per second.

Sound and images are compressed individually, using *codecs*. A codec is a piece of software (or hardware, in some cases) responsible for compressing or decompressing the media stream; a common parameter that a compressing codec accepts is the *bit rate*, representing the average number of bits that should be used to store a second of the content.

In a multiplexed video stream, the images would use much more space if compared to sounds; so from this moment on, for simplicity, we'll hypothesize that videos don't contain audio streams.

In particular, compression algorithms start from the assumption that a significant part of frames is similar to their predecessors, dividing thus the frames (or parts of frames) in two main categories: *key frames* and *delta frames*.

Key frames are stored independently, while delta frames contain only the differential part needed to transform the previous frame in the current one. As a consequence, the bit rate used depends on the motion of the video itself and changes along the content; codecs with this characteristic are called VBR (Variable Bit Rate).

3.2 Video Streaming

Video streaming is a technique to deliver video contents to the users through the network. When we say *streaming* we mean that the user starts enjoying the contents as soon as possible, before the transfer itself is completed. For example, watching a movie of 120 minutes on-line could mean downloading a 1.5GB file through a 10Mbit/s link, which means that the transfer will last about 20 minutes. But, given that the video data is spread sequentially on the file, there is no need to wait for the complete download before starting to display the movie to the user: the show could begin almost instantly.

As the network (the underlying medium) does not provide guarantees about bandwidth nor delay, it is not a good idea to show *immediately* the contents to the user: if for some reason the data is delayed, the playback would freeze until the bytes arrive. It is then crucial to store temporarily the contents in a *buffer*, so to absorb brief network fluctuations.

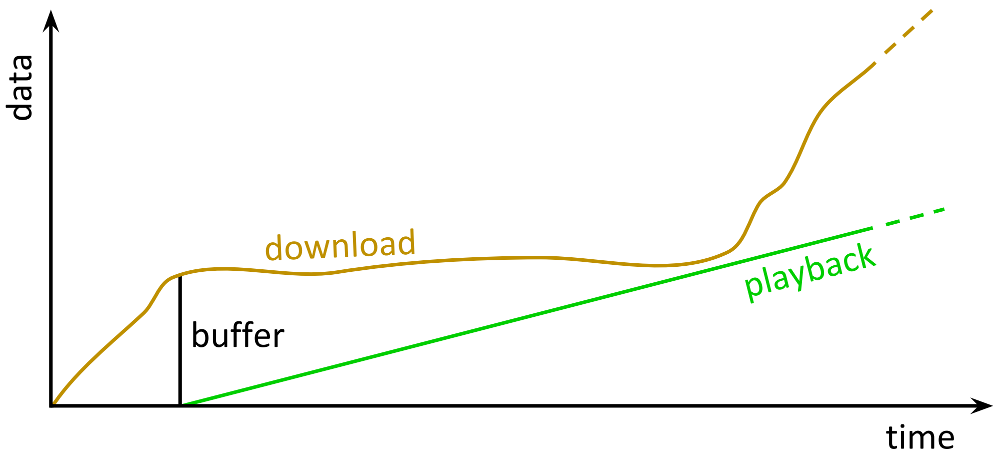


Figure 3.2.1: Video Streaming

This buffer will have a minimum and a maximum length. When its size is below the minimum, the player will wait for more data from the network (this is called *prefetch phase*); while when its size becomes more than the maximum, the player could ask the server to pause the transfer (in some implementations, this decision comes from the server), and resume it later, when the buffer depletes. This last behavior is called *ON-OFF*, and these bursty short transfers could break some assumptions in networking protocols, TCP in particular, and thus generate some problems to the underlying infrastructure.

Currently, the preferred protocol to transfer video files is HTTP, for convenience reasons: it is the standard for transferring web pages, so firewalls and network equipment are already well-shaped to work with it. In particular, video providers should be able to rely on Content Delivery Networks (CDN): groups of servers put in geographically strategic locations, in order to be as near as possible to end users; the use of HTTP imposes little or no changes to existing CDNs.

Quality of Experience

Given a service, the QoE is the quality level, as perceived by the human user. This is of course a subjective indicator, that can only be estimated. There is anyway a set of well defined objective metrics that are believed to have a direct impact on QoE.

Concerning video streaming, these are the main factors supposed to influence QoE, ordered by importance:

- *Rebuffering events*: when the network throughput is not high enough, the buffer runs out, the playback stops, and the user must wait for the new data before the playback could resume. This downtime is considered to be the most annoying issue in video streaming. Intuitively, these events happen when the throughput is lower than the bit rate, for a long period of time, depending on the buffer size and the ratio between throughput and bit rate.

- *Video quality*: as said, a video could be encoded at different *bit rates* (and with different *codecs*), the lower the bit rate, the lower the size of the resulting file, but also the lower the quality of the video; the lower the quality, the less the human eye will be able to see details on the images. Of course, also the user's device impact on this metric: smaller screen resolutions are not capable to display properly high video quality, so it is possible to obtain similar perceived quality with lower bit rates.

- *Startup delay*: as said, the playback does not start before the *prefetch phase*, while the user's desire is to start the show immediately. It should be noted that lowering artificially this delay (i.e. enforce a smaller buffer) could trigger rebuffering, especially if the bandwidth is not well higher than the bit rate.

3.3 The principle of HAS

The main goal of HTTP Adaptive Streaming is, clearly, maximize the Quality of Experience. In particular, the target is to get a high video quality, while avoid rebuffering events. This objective could be pursued by changing the bit rate of the video during the playback, on the fly, reacting to network fluctuations, adapting to the currently available bandwidth. It is worth to note that every change of the bit rate during the playback could disturb the user, thus impacting negatively on the QoE, especially if it is done often or if the bit rate jump is considerable (if the bit rates are relatively close the user could not even note the switch).

The main implementations currently used are:

- *MPEG-DASH*: is the international standard published by the MPEG working group in 2012, as ISO/IEC 23009-1 [16].
- *Adobe HTTP Dynamic Streaming*: the Adobe's version, supported in Flash Player and Flash Media Server, from version 10.1.
- *Apple HTTP Live Streaming*: Apple's implementation, part of QuickTime and iOS, supported since iPhone 3.0.
- *Microsoft Smooth Streaming*: on the server side, it is an IIS (Internet Information Services, the HTTP server developed by Microsoft) Media Services extension; on the client side, various software development kits are available, compatible with Windows, Apple iOS, Android, and Linux.

These implementations differ for the codecs used and the specific file formats, but the general structure is shared: the video is split in *segments* (or *chunks*) with a fixed duration (in general between 2 and 10 seconds); these segments are then encoded at different (2-8) bit rates. A *manifest file* will hold all these complementary informations (bit rates, exact durations, URLs...). The client could decide to change the selected bit rate between the downloads.

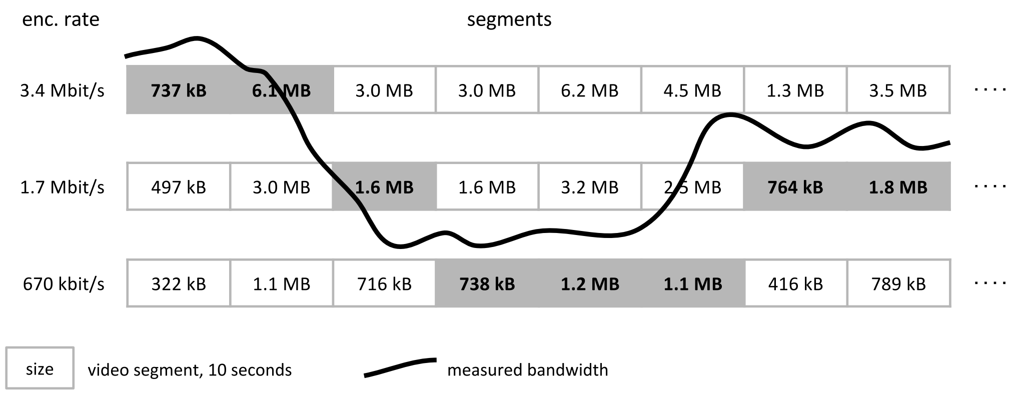


Figure 3.3.1: HTTP Adaptive Streaming

It is worth to remember that the *nominal* encoding bit rate is not dutifully respected by the codec, when the compression is VBR: the *instantaneous* rate depends directly on the motion present on the video contents; this is clear in fig. 3.3.1, where the segments have all equal duration but the resulting file sizes differ.

3.4 Rate-based algorithms (RBAs)

The easiest and most intuitive way to adapt the bit rate to the network is to estimate the current bandwidth to base the choice. Proprietary adaptation algorithms are not identical between each other, but in general, the prediction is heavily based on the measured throughput during the previous downloads. As an example, in [4, 5] we can find a simplified algorithm believed to mimic Microsoft Smooth Streaming client:

The player keeps two metrics related to the bandwidth: A, the throughput of the latest downloaded chunk, and $\hat{A}$, the running average of A. The value of $\hat{A}$, after the download of the segment i, is:

$$\hat{A}(i) = \begin{cases} \delta\hat{A}(i-1) + (1-\delta)A(i) & i > 0 \\ 0 & i = 0 \end{cases}$$

with $\delta = 0.8$.

We assign to each available bit rate an index, from the lowest to the highest. ϕ_{cur} denotes the currently selected bit rate index. The player downloads the first segment with $\phi_{cur} = 0$, the lowest bit rate.

The next candidate profile ϕ is obtained by:

$$\phi = \max \left\{ i : b_i < c \times \hat{A} \right\}$$

where $c = 0.8$ is used to absorb encoding and bandwidth fluctuations. ϕ_{cur} is updated following this algorithm:

```
if $\phi > \phi_{cur}$ then
    increase $\phi_{cur}$ by one
else if $\phi < \phi_{cur}$ then
    decrease $\phi_{cur}$ by one
else
    no action
```

Moreover, the player have two different states: *buffering* and *steady state*. While in *buffering* phase, the player continuously downloads segments, until the buffer reaches a predefined size (30 seconds). It then stops downloading and switches to *steady state*, in which the buffer size is maintained almost constant: a new segment is downloaded after the complete playback of the current one. This generates the well-known *ON-OFF behavior*: the player downloads a segment then it stops for some time, then it downloads a new segment, then it stops, and so on. The duration of ON and OFF periods depends on the ratio between the selected bit rate and the throughput: the former indicates how fast the data is consumed, while the latter denotes the speed of new arrivals. For example, if the throughput is twice the bit rate, the two periods are likely to be equal (fig. 3.4.1); con-

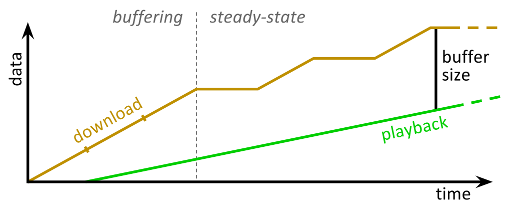


Figure 3.4.1: HAS states

versely if the throughput and the bit rate are similar, we don't expect to see OFF periods.

The importance of ON-OFF behavior becomes clear if we think that:

- It is not given that user's device has enough memory to keep the entire file. Think about an entire movie stored in RAM: this is not sustainable, as is would consume for example, half of the available memory.
- More importantly, network resources are expensive, so it is essential to avoid wastes as much as possible; the user could abandon the playback in any moment, so there is no point in downloading a big amount of data.

Real world clients would have more elaborate approaches. For example, it is reported that some of them integrate current buffer size in the behavior: they became more conservative when the buffer is low, and bet more when it is in a healthy state.

3.4.1 Interplay with TCP and client competitions of RBAs

The algorithm explained above does its best to estimate the available bandwidth, but maybe that is *not* the answer to the question *which bit rate should I chose?* [3]. In practice, the *stream selection* we introduced is a control loop (fig. 3.4.2), which has as output the next bit rate, as input the measured throughput, and internally a mechanism that ignores completely what is happening on the lower layers: in fact, by choosing the bit rate, it will implicitly choose the size for the next file to download, and this could have an

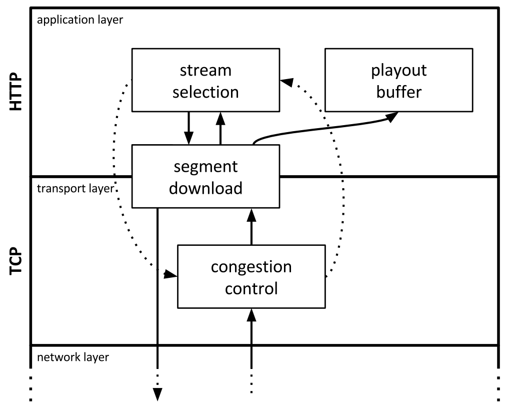


Figure 3.4.2: Rate based algorithms: interaction between control loops

impact on the next measured throughput value, re-impacting then on the next decisions. The reason is the existence of another control loop, running on the transport layer: TCP congestion control.

Interactions between loops

Each time the player asks for a new segment using HTTP, a new transfer over TCP will be made from the server to the client; as said, the CWND directly controls the throughput of TCP transfer, and the rate-based algorithms depend directly on the throughput, so the two control loops (stream selection and congestion control) behave as a “nested double feedback loop” [4], something notably hard to predict.

The bit rate adaptation mechanism does not (and it is not supposed to) have access to the CWND value, also because they are located on the opposite sides of the communication: respectively the client and the server. So to measure the throughput, the client simply calculates the average throughput of the current segment, dividing the segment size by the time spent to download it. TCP, in the meantime, continuously changes the transmission rate trying to dynamically adapt to the fair share of the medium; a strategy that actually works well for long transfers, but does not guarantee optimal results for short transfers: upon a loss (or even without losses), the CWND could drop, and because of the length of the transfer, there wouldn’t be the time to grow back to a good value. This could unnecessarily force the stream selection to a lower profile. The impact of the losses on throughput is well explained in [6], where The the impact of a single loss on a segment transfer is analyzed, depending on the position of the lost packet relative to the begin and end of the download.

The lower the bit rate, the smaller are the sizes of the segments, the harder becomes make the CWND grow, as reported in [3] while studying the effect of a long flow competing with an HAS session. The authors detailed the bad cooperation between the side TCP flow and the HAS *ON-OFF* behavior, and tried to find some viable solutions.

Competition between RBAs clients

Moreover, it is worth to note that bandwidth estimation utilizes data taken only when a download is in progress. Table 3.1 [5] spots some peculiar cases of possible errors when two competing clients in *steady state* share the same bottleneck link with capacity **C**. When is *steady state*, as said, the player shows an *ON-OFF* behavior to keep the playout buffer as constant as possible. This could generate strange scenarios in which the intermittent measurements yield insidious results.

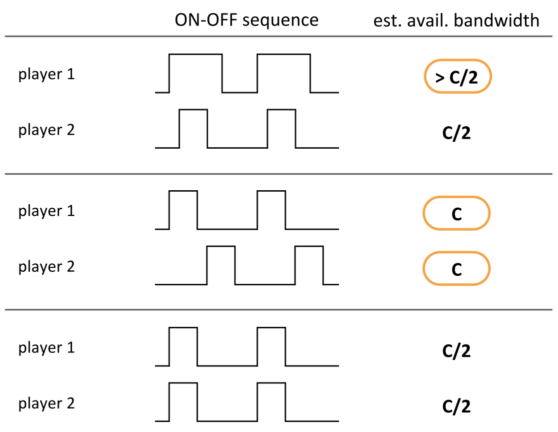


Table 3.1: Estimation inaccuracy for competing HAS players [5]

- In the first example, player 1 downloads segments that are a little bit bigger than the ones downloaded by player 2. In this particular configuration, player 2 will experience correctly half of the capacity of the link, while player 1, because of the fraction of time it was downloading alone, will measure more. This overestimation could drive player 1 to wrong bit rate decisions.

- The second case is an example where the players are downloading in a mutually exclusive fashion. In this case, they will both wrongly measure the full link capacity, and so they are likely to be pushed towards a higher bit rate, in which the segments will be bigger; this is obviously unsustainable, and the players will eventually step back to a lower bit rate.

- The last example shows the best case: both players are getting half of the capacity, their estimation should be correct.

It is noteworthy that overestimation is not always an issue: if the error is negligible compared to the distance between the available bit rates, no switch would be triggered; algorithms usually have a safety margin that prevents this, among other inconveniences.

3.4.2 Huang et al. algorithm

In [3], Huang et al. propose some modifications to a rate-based algorithm so to get better rates when a competing long flow shares the bottleneck link. The authors first reproduced a commercial HAS video player, then introduced these variations:

- Less conservative: increased the aggressiveness from the initial $c = 0.6$ value of their baseline algorithm to $c = 0.9$; TCP guarantees in any case that the client don't get more than the fair share.
- Better filtering: instead of calculating the moving average, the authors used medians and quantiles; considering the 80th percentile the vulnerability to outliers is greatly reduced.
- Bigger segments: by requesting five segments at once, it's possible to let TCP reach the optimum CWND size, improving the throughput and the decision taken by the algorithm.

3.4.3 Sabre

In [17], Mansy et al. analyze the bufferbloat effects caused by HAS, stating that it could easily add a delay of one second or more in residential connections. The proposition aims to minimize the impact on the buffers by limiting the amount of in-flight bytes; this could be done on the client side shaping the TCP receiver window, an effect that could be obtained by:

- HTTP pipelining: not waiting the end of the previous download before asking the next segment; if the receiver buffer is empty, the RWND will be at its maximum, and the next transfer will start with a burst of packets, causing bufferbloat; keeping the HTTP pipeline non-empy make the RWND controllable.

- Reading the receiver buffer at a specified rate: in order to smooth the buffer fluctuations, the buffer must be emptied at a *target_rate* similar to the corresponding video bit rate; the throughput will follow, avoiding thus bursts of data.

As the throughput will be then measured by the client to choose the next bit rate, it is important to not limit it too much. Because of that, the algorithm works in two states, depending on the current buffer level:

- Refill:
  if the buffer drops below *refill_thresh*,
  target_rate = $\lambda \times R_h$, with $\lambda > 1$, where R_h is the maximum bit rate;

- Backoff:
  if the buffer exceeds *backoff_thresh*,
  target_rate = $\delta \times R$, with $0 < \delta < 1$, where R is the current bit rate.

The player will start in refill mode, download smoothly but anyway having the possibility to get the R_h. When the buffer reaches an high occupancy value, the algorithm won't stop downloading, but it'll limit the throughput slightly lower than the current bit rate; the buffer will then decrease, until the *refill_thresh* is met, and so on.

3.4.4 Festive

Jiang et al, in [18], focus on efficiency, fairness, and stability on competing players. The proposed approach has these properties:

- Randomized scheduling: in order to avoid synchronized downloads which bias the measurements, as table 3.1 second case, the requests are anticipated or delayed, by randomizing the maximum buffer capacity.

- Stateful bit rate selection: as in table 3.1 first case, players selecting high bit rates will tend to see higher throughputs; the proposition is to use the current bit rate as a status for the selection, making the selection more aggressive if the current value is low and more conservative if it is already high; this could be easily done by letting the rate switches being frequent for low bit rates and sporadic for high bit rates.

- Delayed bit rate update: the previous point introduces instability; in order to limit the impact, the result of the stateful selection is taken as an advice, and a concrete choice is taken after calculating and comparing the costs in term of stability and efficiency for both proposed and current bit rates.

- Harmonic mean: the bandwidth is estimated by calculating the harmonic mean of the last 20 transfers' throughput; this mean is more robust to large outliers, if compared to the running average; this is particularly important as the randomized scheduler increases the possibility of encountering throughput outliers: the number of competitors could vary greatly between segment downloads.

3.5 Buffer-based algorithms (BBAs)

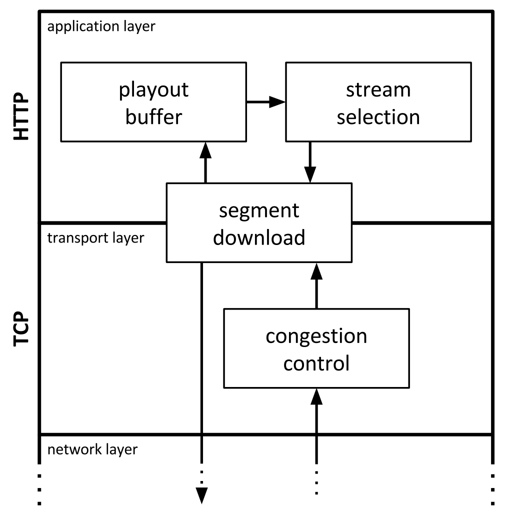


Figure 3.5.1: Buffer based algorithms: what we want to control is part of the loops

As [3] suggested, maybe it is not a good idea to try to measure the bandwidth when the first goal is to avoid the rebuffering events, while maximizing in the meanwhile the video quality delivered to the user. People from Netflix [7] agree: their data indicates that the throughput the single users get is far from being constant, but conversely it is quickly variable, between 500kb/s and 17Mb/s, and so it is almost impossible to try to predict what is going to happen on the wire looking directly at the past throughput.

The idea proposed in [7] is: base the decision as much as possible on the playout buffer occupancy, as that is the main state variable we want to control. This is not always possible, for example during the start up phase the buffer does not contain yet enough data to drive to a good decision, so at the beginning it is in some way necessary to measure the available bandwidth.

There are four different flavors in this suggested buffer-based family:

- *BBA-0:* an initial version that draws the idea.
- *BBA-1:* variability of segment size is taken into account. The resulting choice is more tailored on the next segment to download.
- *BBA-2:* at the beginning of the playback, the buffer is empty, but this does not mean that the bandwidth is small. This version adds some simple throughput estimation to ramp up the buffer quickly in the start-up phase.

- *BBA-Others:* looking ahead to future segments, it tries to smooth the bit rate changes, believed to penalize the QoE.

3.5.1 BBA-0

The first algorithm of the family is an initial version to draw and validate the idea. In fig. 3.5.2 is presented the *rate map*, a function that ties the rate selection to the amount of data contained in the buffer; on the x-axis there is the playout buffer occupancy, measured as time, on the y-axis the video bit rates.

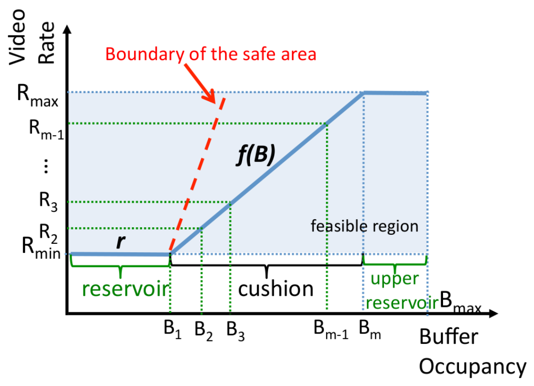


Figure 3.5.2: *BBA-0 rate map*. Source: [7]

The buffer is split into three zones: *reservoir*, *cushion* and *upper reservoir*.

- The *reservoir* is meant to protect the buffer from draining; to understand its usefulness, it is important to add a couple of hypotheses: it is not possible to abort a download, and a different bit rate could be chosen as soon as the current transfer is completed. Thanks to the *reservoir*, the player can safely finish downloading a big segment even if the bandwidth drops (this is the meaning of *safe area*). When the buffer occupancy is between 0 and the *reservoir* size, the algorithm will recommend the lowest bit rate.

- The *cushion* is where the algorithm linearly choose a bit rate depending on the current buffer occupancy. There is not direct link between the decision and the experienced throughput.

- The *upper reservoir* allows the player to get the maximum bit rate; without it, the client would choose the highest rate only when the buffer is *exactly* full, while the whole area is actually safe enough.

In the paper's implementation, segments are 4 seconds long, the buffer size is 240 seconds, the *reservoir* is set to 90 seconds, the *upper reservoir* to 24 seconds (10% of the total).

As said, when the buffer occupancy corresponds to the *reservoirs*, the player will get the respective bit rate. On the *cushion*, the algorithm first calculates the value of the $f(B)$ function, which transforms the current number of seconds of video in the buffer in a continue value of bit rate (so normally, a bit rate that does not exist in the *manifest file*), then it follows these rules:

```
if $Rate_{cur+1}$ exists and $f(B) \geq Rate_{cur+1}$ then
    increase $cur$ by one
else if$Rate_{cur-1}$ exists and $f(B) \leq Rate_{cur-1}$then
    decrease $cur$ by one
else
    no action
```

Where the bit rates available are ordered from the smallest to the greatest, cur denotes the index of the currently selected bit rate (zero-based), and $Rate_{index}$ indicates the specific bit rate value.

It is worth noting that in the *BBA* family, the *ON-OFF behavior* is mostly avoided: segments are downloaded continuously, and if the buffer keeps growing, the algorithm would barely select a higher bit rate. There is only one case in which the algorithms could stop downloading: if the buffer is full (but in this case the player is downloading segments from the highest bit rate).

3.5.2 BBA-1: VBR encoding

As shown in fig. 3.5.3 the segment size is highly variable; as said before, the VBR compression algorithms use an amount of bytes that depends on the motion of the specific scene. A direct effect of this characteristic is that the download of two different segments (encoded at the same nominal rate) could take a hugely different amount of time, even if the network is relatively stable.

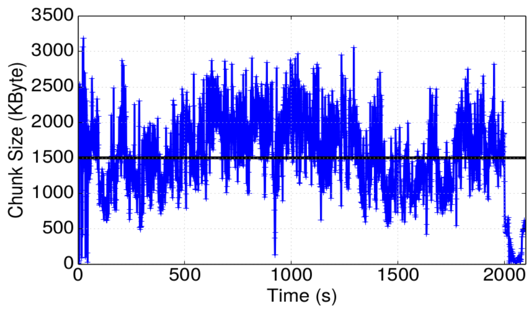


Figure 3.5.3: The size of 4-second chunks of a video encoded at an average rate of 3Mb/s. Note the average chunk size is 1.5MB (4s times 3Mb/s). Source: Te-Yuan Huang et al. [7]

The main implementations of HAS do not carry the segment size in the manifest file; it is anyway not problematic to add these informations, this is way *BBA-1* exploit these clues and use them instead of the average bit rates, as they are much more precise than the averages. Specifically, *BBA-1* utilizes the segment size to build a *dynamic reservoir* and to choose the next bit rate being aware of the real bit rate of the corresponding segment.

In order to calculate *reservoir* size, we assume to have a client with an available bandwidth equal to $Rate_0$, and we expect this player to stream the lowest quality without rebuffering events. This could be actually true if the instantaneous bit rate for all the segments is equal to the nominal average; as in that case the time to download any segment is equal to the segment duration; but the story changes if the segment has a higher or lower instantaneous bit rate, and it is especially problematic if the segment size is above the average, as in this case the buffer will shrink. We need then to dimension the *reservoir* to absorb these bit rate variations.

The player defined above will have a playback free from rebuffering if its *reservoir* size is equal the sum of the amount of seconds that the client will play minus the amount of data that it will download, during the next X seconds. In the paper, X is set to the double of the buffer size: 480 seconds. The *reservoir* is dynamically and continuously calculated, with extra safety bounds: between 8 and 140 seconds (respectively 2 segments and 35 segments).

While the *reservoir* is now dynamic, the *cushion* doesn't change its size; on the other hand, the *rate map* is substituted with a *chunk map*: with respect to the buffer occupancy, while in the *cushion*, the algorithm will choose a segment size, and not a rate; in this way the selection is more tailored to the specific instantaneous bit rate.

The *upper reservoir* follows the variations of the *reservoir* to keep intact the total maximum size of the buffer.

3.5.3 BBA-2: the start-up phase

At the beginning of the streaming session (or after a seek on the video), the buffer is empty; *BBA-0* and *BBA-1* fail interpreting this phenomenon as low-capacity network, and applying a conservative behavior; this is not always a good choice, as with some probability the network could safely give more. In these conditions, where the buffer does not hold informations, it is necessary to estimate the available throughput; doing so, it is possible to make the buffer grow faster in an initial phase.

In practice to ramp up in bit rates, the ratio between download time and playback time for the last downloaded segment should be bigger than a certain factor. At the beginning, this factor is equal to *eight*, decreasing then linearly as the buffer grows up through the *cushion*. *BBA-2* keeps staying in this start-up phase until the *chunk map* (*BBA-1*) gives a higher bit rate or the buffer starts decreasing.

3.5.4 BBA-Others: instability and outage protection

There is still some issues running on that could cause problems to end users: *Temporary network outages* can show up for example on residential ADSLs or because of Wi-Fi interferences, and the *instability* is introduced by the choice to rely on segment size in *BBA-1*. In this context, *instability* indicates how often bit rate is changed, and it is a direct effect of the variable segment size, as reported in figure 3.5.4. *Instability* has a bad impact on QoE, as frequent bit rate switches could displease the user.

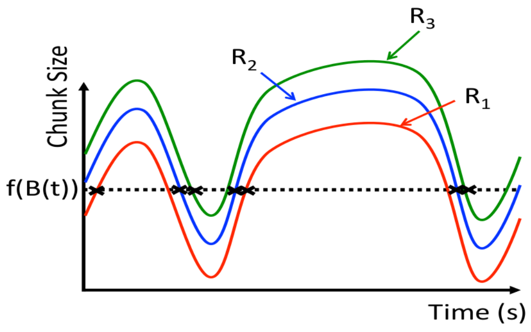


Figure 3.5.4: *Chunk map* increases *instability*. Even if the buffer level and the mapping function remains constant, the variability of segment sizes can make *BBA-1* often switch between rates. The lines in the figure represent the chunk size over time from three video rates, R_1, R_2, and R_3. The crosses represent the points where the mapping function suggests a rate change. Source: [7]

BBA-Others aims to both reduce *instability* and protect from *network outages*, with two strategies:

- The *reservoir* is bound only to grow, but not to shrink; this helps keeping an extra amount of *reservoir*, useful in case of network issues, and stabilizes the bit rate selection, as the *chunk map* will move less frequently.
- Before choosing to a higher bit rate, the algorithm will look ahead to next segments, avoiding the switch if with the current conditions it would take the opposite choice soon. We could expect that the bigger the amount of look ahead, the more it will smooth the rate; in [7], the authors propose to look ahead to the same number of segments currently held in the buffer. Note that the rate change is *not* avoided if the *chunk map* suggested to decrease the bit rate, so to maintain a good resiliency to rebuffering events.

Chapter 4

Investigating BBAs performances: the empirical approach

This chapter proposes a methodology to compare the families of algorithms: the testbed and the metrics.

Buffer based algorithms have been tested directly in the wild, through the Netflix service, for single client metrics, in [7], and it is demonstrated that they succeeded to decrease the impact of rebuffering events while obtaining almost the same bit rates in average as the rate based algorithm used at Netflix at the time. This work aims to compare the algorithms in a controlled environment, focusing on other metrics, representing the impact on the network and the competition between the clients.

4.1 Testbed

The testbed run entirely on top of VirtualBox, a hypervisor, orchestrated with Vagrant. The configuration files hold all the informations concerning the virtual machines (VM), in this way the set of VMs could be regenerated and automatically configured with a single command.

Each box in fig. 4.1.1 represents a virtual machine in the VirtualBox environment. There are different (virtual) networks between the VMs, so that the

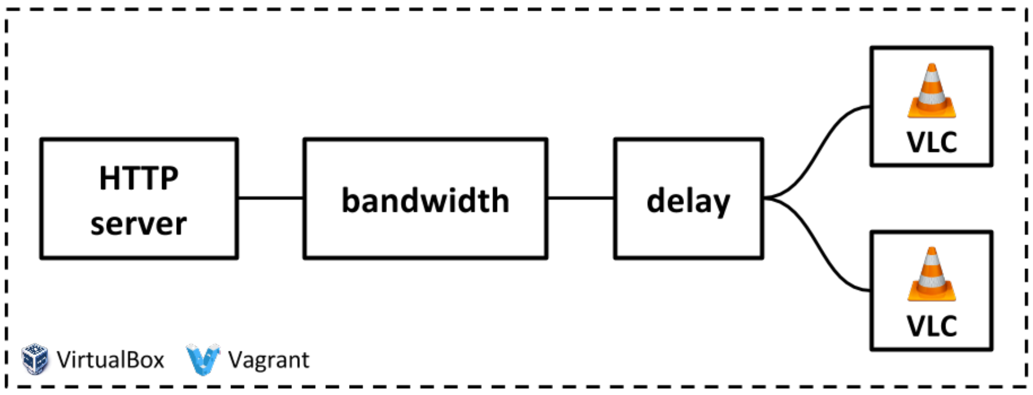


Figure 4.1.1: The testbed

packets are forced to flow through the bottleneck emulation (bandwidth+delay).

HTTP Server

HAS protocols do not require particular server configurations for the Video On Demand part (this is not true for Live video streaming, that anyway is not our case), so the HTTP server implementation is a simple *node.js* instance that serves static files.

Bandwidth and delay

These two machines are meant to simulate the bottleneck. They both use *tc-netem* Linux tool, from the *traffic control* suite to shape the traffic so to emulate a link with given bandwidth and delay, respectively.

Clients

For simplicity, we preferred the Apple HTTP Live Streaming (HLS) protocol. The clients run a modified version of VLC player[1], so to behave like the simplified player in [4, 5]; the *libcurl* library was linked to allow the

[1]http://www.videolan.org/vlc

reuse of the TCP connection between the downloads, and all the BBA family presented in [7] was implemented. The resulting application is capable to run different configurations by passing specific parameters as environment variables.

4.2 Metrics

The tests run on the testbed are classified by looking at specific metrics, concerning the Quality of Experience, the network utilization and the competition between the clients.

4.2.1 QoE metrics

As cited before, the important metrics regarding Quality of Experience, are: *rebuffering events*, *bit rate* and *instability*.

Rebuffering events

This is the most important metric impacting QoE. A rebuffering event happens when the player suddenly stops the playback, because the buffer got empty. Note that if a segment is not completely downloaded, it is not available to the player.

To characterize the impact of rebuffering, we could:

- Count the number of *rebuffering events* in a session. This could be a good idea, but it is implicitly tied to other variables: video duration and segment duration. The first is pretty clear: the shortest the video, the less data to download, the less opportunities to have problems. For the second, if the playback continuously stops after each segment waiting for the next one to be downloaded, it is clear the the longer the segments are, the less the rebuffering events will be. It should be noted, also, that the cardinality of rebuffering events could be artificially lowered, forcing longer waiting times.

- Calculate the *rebuffering ratio*: the idea here is to measure the fraction of time spent waiting for new data, dividing the sum of the duration of all the rebuffering events by the total duration of the session. In other words, it is the fraction of time while the player was in rebuffering state.

Bit rate

To evaluate the bit rate obtained by the clients, three metrics could be interesting:

- *average bit rate*: simply obtained by summing the nominal bit rate for each segment downloaded and dividing by the number of segments.
- *average relative bit rate*: the average bit rate divided by the bottleneck capacity, so that measures coming from different tests (i.e. with different bottleneck capacities) could be compared.
- *average quality level*: the bit rates are not spread uniformly, but the higher the bit rate, the bigger is the distance with its neighbors. This metric takes the average of the indexes of the bit rates of the downloaded segment and scales it as a percentage.

Instability

The metric for instability, as in [5], is obtained by dividing the number of bit rate switches by the total number of segments.

4.2.2 Network metrics

Congestion window

The congestion window (CWND) is the parameter that auto-adapt the TCP flows to the current network conditions. It is an important metric to understand how the data is exchanged on the network. To capture the value, we used the kernel module *tcp_probe*.

Router buffer

The simulated bottleneck is preceded by a buffer, that is emptied at the rate imposed; this buffer has a maximum length and it grows and decreases as time passes, depending on the network activity; it is possible to poll *traffic control* to get its current status.

As the maximum buffer capacity is not constant between the tests, we chose to take also the average value relative to the maximum.

Round Trip Time

We impose a static delay to the packets, but they could experience an additional delay, caused by the combination of the said router buffer and the bottleneck capacity. We recorded the average RTT perceived, relative to the base we enforced.

Link utilization

It is interesting to see how much of the bottleneck capacity is actually exploited by the HAS flows. For this purpose we took a copy of the headers for each packet arrived and departed from the *bandwidth* VM, using *tcpdump*; sampling the time in hundredths of second, we then counted the number of packets, and bytes. We used these samples to obtain the network rate entering and leaving the bottleneck link.

4.2.3 Competition metrics

The metrics about competition denote various aspects that arise when the bottleneck is shared between HAS clients.

- *bit rate unfairness*: the absolute value of the difference between the two average bit rates.
- *average bit rate unfairness*: the difference between the bit rates divided by the bottleneck capacity.
- *quality level unfairness*: the absolute value of the difference between the two average quality levels.

Chapter 5

Assessment results of BBAs: the client and network perspectives

This chapter presents a description of both the algorithms and the videos involved in the tests; a part of the results obtained follows, concerning single-client, two-clients and three-clients sessions, under different points of view: QoE, network and competition metrics.

Videos

Tests were conducted over two videos with peculiar characteristics: the first presents a constant segment size and four bit rates, the second is a real movie with high variability and eight quality levels.

Apple's BipBop

For testing purposes, and to get the first results, it is useful to have a video without segment size variability (i.e. *Constant Bit Rate*, CBR). Some initial tests were run against a video with this characteristic, before switching to a real movie.

This video is the basic example for the HLS protocol, as provided by Apple[1]; it contains constant motion, so as a result, there is almost no variability in

[1]https://developer.apple.com/streaming/examples/basic-stream.html

segment size: even if the codec used is VBR. The entire run time is 30 minutes, with 181 ten-seconds segments (the last is 4 seconds long).

Available bit rates are:

- 232 kbit/s at 400×300 pixels,
- 650 kbit/s at 640×480 pixels,
- 1 Mbit/s at 640×480 pixels,
- 2 Mbit/s at 960×720 pixels.

Big Buck Bunny

It is an open source movie, freely downloadable from the project web site[2], re-encoded using *ffmpeg*[3] to prepare HLS segments and *manifest* files. The video duration is about 10 minutes, 299 two-seconds segments. For the lowest bit rate, the biggest segment is 2.2 times the nominal average.

Available bit rates are:

- 350 kbit/s at 320×176 pixels,
- 470 kbit/s at 368×208 pixels,
- 630 kbit/s at 448×256 pixels,
- 845 kbit/s at 576×320 pixels,
- 1130 kbit/s at 704×400 pixels,
- 1520 kbit/s at 848×480 pixels,
- 2040 kbit/s at 1056×592 pixels,
- 2750 kbit/s at 1280×720 pixels.

[2]https://peach.blender.org/
[3]http://ffmpeg.org

Algorithms

These are the bit rate decision algorithms tested:

classic: inspired from the simplified player in [4, 5]. The available bit rates are compared with the running average of the obtained throughput, multiplied to 0.8 as safety margin. This throughput is obtained by dividing the segment size by the time span between the start of the HTTP request and the end of the response.

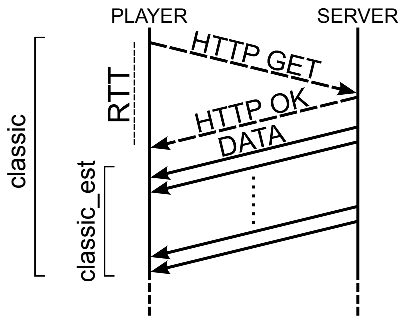


Figure 5.0.1: Time-line of data exchanged between the peers

classic_est: mostly as *classic*, but measuring the throughput as the segment size divided by the time span between the first and the last received bytes. The difference between the two is shown in fig. 5.0.1: *classic* starts measuring the time at the beginning of HTTP GET, while *classic_est* at the beginning of the data stream.

Both *classic* and *classic_est* were tested with two different playout buffer sizes: 30 seconds, as in [4, 5], and 240 seconds, as for the BBA family.

BBA-0, BBA-1, BBA-2, BBA-Others: as defined in [7]. *BBA-Others* was further modified halving the look-ahead window, to frame the impact of this mechanism on the metrics.

Tests performed

Various tests were run against both of the videos cited above, with different network configuration and different number of clients, on all the algorithms presented. We ran experiments with both constant and variable bottleneck capacity, with different router buffer limits and dropping policies, and with different delay values. Each experiment ran twice if one client was involved, four times otherwise, to ensure a good confidence interval on the results.

5.1 Quality of Experience metrics for a single client

As a first glance, we analyze single-client sessions, to study the behavior for Quality of Experience metrics, on the simplest environment imaginable: all network settings are fixed in each experiment, and no competition is imposed.

5.1.1 Constant bottleneck capacity

BipBop

A batch of 20 different experiments, with fixed delay (200ms) and fixed bottleneck capacity (ranging from 300kbit/s to 2.2Mbit/s), was run twice.

Concerning the rebuffering ratio, there is not a particular difference between the algorithms, but we clearly noted a divergence on the bit rate obtained: the average bit rate weighted on the link capacity got by BBAs is around 90%, while for RBAs it is less than 55%; two factors could explain this gap:

- The highest bit rate is wrongly reported as 2Mbit/s in the manifest file (taken directly from Apple), while instead it is 1.5Mbit/s; this information, in buffer based family, is utilized only by *BBA0* to build the rate map, but it does not have the same impact on the choice as for the RBAs.
- The rate-based algorithms stick to the highest bit rate below the supposed link capacity, while the buffer-based oscillate to between two bit rates as the current buffer occupancy suggests, taking sometimes an unsustainable bit rate and stepping back when needed, without impact on rebuffering. This second point drives the analysis to the instability: RBAs are around 1%, BBAs get obviously an higher value, 4%; this is not problematic, as 4% means 8 rate switches, in this 30-minutes video.

Big Buck Bunny

To check dependency on network shaping, tests with various conditions were run on this video. We made vary not only the bottleneck capacity, but also the delay and the router buffer size, imposing a fixed value and a Bandwidth-Delay Product (BDP) fraction. Values used are: for capacity 400kbit/s to 3Mbit/s, in 100kbit/s steps; for delay 100ms, 200ms and 400ms; for router buffer 200 packets, 100%, 50%, 25%, 10% of BDP.

The rebuffering ratio grows, in most cases, with the delay and the inverse of the router buffer size: high RTT and small buffers make difficult to obtain high throughputs, and this could lead to rebuffering, if the capacity is already small; comparing to BipBop video, the rebuffering ratio is at least ten times more, probably because of the spikes in instantaneous bit rate (VBR: the segments have different file sizes).

A curious case is *classic_est*, as it shows high values for 400ms delay, due probably to the way it measures the bandwidth: it does not take into account the HTTP handshake, that is at least an RTT, needed to have the video data, as shown in fig. 5.0.1. This is not a problem in most cases; but when the delay gets big enough to be comparable to the duration of the subsequent transfer, it starts to impact on the download itself and should be taken into account. Note that the RTT value is problematic only at the time of the HTTP request, so in this case the experienced RTT is the base RTT: 400ms, as the router buffer has just been emptied. The delay impact less on the other algorithms as it is implicitly considered, on the throughput for classic or on the download duration for BBAs. The introduction of *HTTP/2* [15], and in particular the ability that allows the server to proactively send contents to the client, called *server push*, could mitigate this issue; but the algorithms should be completely redesigned so to take advantage, and moreover, the server might have to be part of the decisional process.

The difference of the bit rate obtained by the two families blurs, if compared to *BipBop*: BBAs lose 15 to 30 percentage points, while RBAs gain about 10; on the other hand, gaps and similarities between the algorithms within the families become clear:

classic_est sees an higher bandwidth, and so as expected takes an higher bit rate if compared to *classic*, of 5 to 10 percentage points. The instability is around 4% for both of them. As said, the RBAs were tested with both 30s and 240s playout buffer, and no sensible differences were observed between the two options.

BBA-1 and *BBA-2* take a bit rate a little bit higher than *classic_est*, but with the downside of an high instability: they switch more than 60 times in this 10-minutes video. These algorithms work on the chunk map, meaning that they switch between bit rates also depending on the segment sizes. Moreover, the reservoir size is dynamic, depending on the size of a number of future segments; this moves continuously the chunk map, increasing the instability.

BBA-Others loses about 10 percentage points in bit rate, compared to *BBA-1* and *BBA-2*. Two new elements were added in this version: rate smoothing and outage protection; the first was done introducing look-ahead, that avoid switching to an higher bit rate if there is the risk to step back soon, and both of them by letting the reservoir only to grow. The rate smoothing succeeded indeed, cutting the instability by two thirds, to about 7% (that is 20 changes).

In this particular case, the rate smoothing had a little impact: halving the look-ahead window does not have an influence on bit rate, nor instability; this could be a video-dependent observation, anyway, as this video present a peak in the segment size at the beginning, as shown in fig. 5.1.1; because of this, the resulting reservoir in *BBA-Others* is quite big and, in practice, constant. This interplay could be an explanation for the low instability and low bit rate obtained by the algorithm: it is pushed to an extreme of its behavior.

Differences in link utilization depend on RTT, router buffer size and playout buffer size. Taking the subset of algorithms with the playout buffer of 240s (excluding the RBAs with 30s buffer), for a given couple of (RTT, router buffer size), the link utilization differs at maximum 3.8 percentage points.

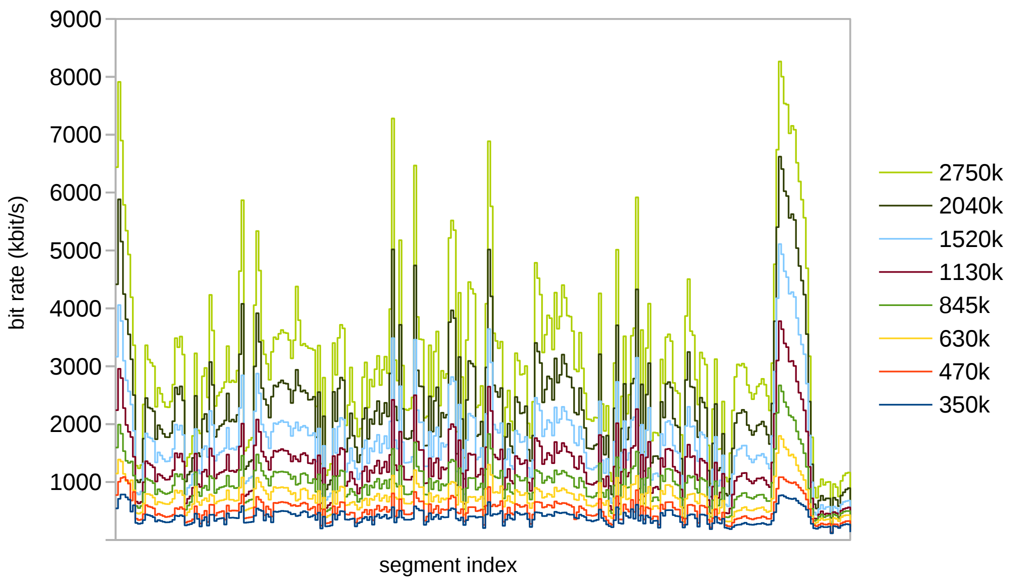


Figure 5.1.1: Instantaneous bit rates for single segments. Big Buck Bunny

RBAs with smaller playout buffer tend to show lower link utilization. To explain this phenomenon, we split the typical RBA streaming session in three parts: start-up phase, ON-OFF phase, ending phase. In the start-up phase the client downloads continuously segments until it reaches the upper bound of the playout buffer; in the ON-OFF phase it lazily downloads segments to maintain the buffer level; in the ending phase it had already downloaded all the video data and it empties the buffer without touching the network. The link utilization metric is taken only on the first two parts. Independently of the playout buffer maximum size, we expect the player to download the entire video, at similar bit rates; a client with a smaller buffer however, will spend less time on the start-up phase and especially on the ending phase. In this way, the same download is spread in a bigger amount of time, giving thus a lower link utilization value.

5.1.2 Variable bottleneck capacity

Fig. 5.1.2 shows a session of *classic_est* with variable bottleneck bandwidth, streaming Big Buck Bunny. In this case, it is clear that the algorithm is not

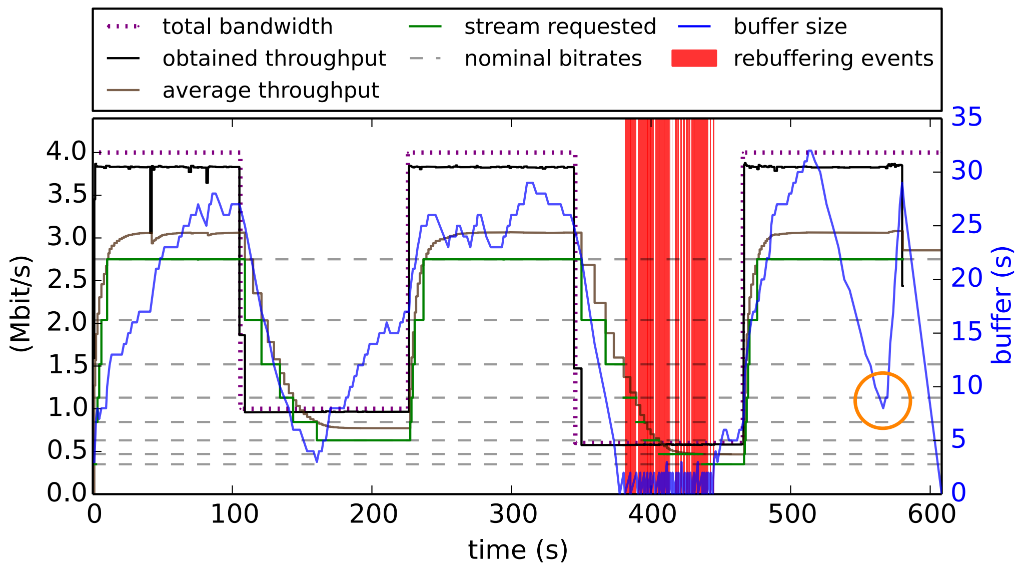


Figure 5.1.2: *classic_est* with variable bottleneck capacity

able to detect the bandwidth drop quickly enough, because the running average as defined in [4, 5] does not converge quickly enough: the rebuffering ratio is 8.2%. The average bit rate obtained is 1936kbit/s, with 10% instability. Conversely *BBA-2* succeeds in avoiding completely the rebuffering events, as shown in fig. 5.1.3; the average bit rate is lower: 1592kbit/s, and the instability has more than doubled, as expected by the design of the algorithm: 23%.

It is necessary to note that average bit rate and rebuffering ratio metrics are at the opposite sides: for example we could code an algorithm that always chooses the highest bit rate, but obviously it will keep rebuffering after each segment; or we could avoid rebuffering events by choosing always the lowest bit rate; clearly both of them are wrong approaches, resulting in long waiting times or low quality, degrading in both cases the Quality of Experience.

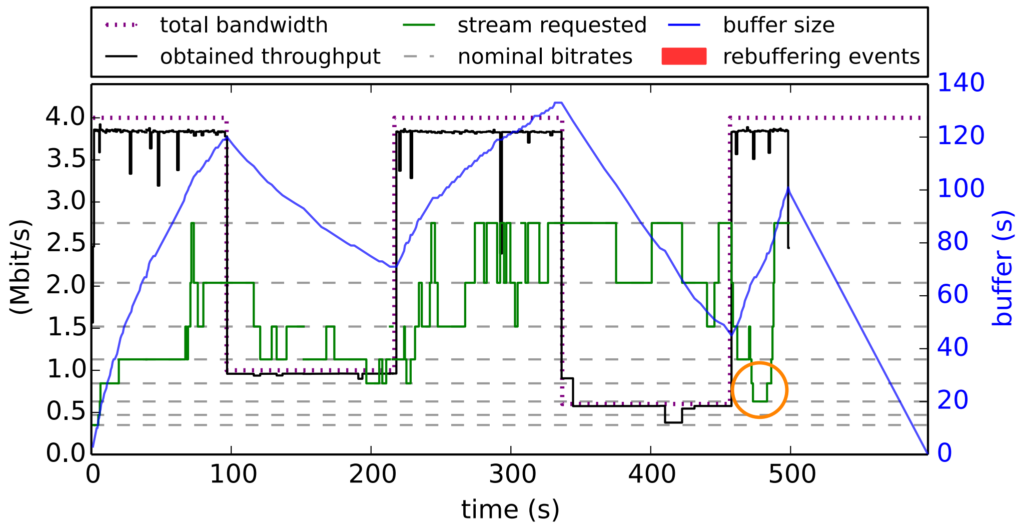


Figure 5.1.3: *BBA-2* with variable bottleneck capacity

In fig. 5.1.4 we see *BBA-Others*: the instability is around 8%, even less than with *classic_est*, and the average bit rate is 1368kbit/s. Figures 5.1.2, 5.1.3 and 5.1.4 exhibit the differences between the three algorithms:

- *classic_est* closely follows the running average of the throughput (brown line).
- *BBA-2* is harder to understand because it bases the choice on the play-out buffer (blue line), but is also influenced by the reservoir size, which depends on the size of the following segments; the segment size is heavily variable, as shown in fig. 5.1.1, and so is the reservoir.
- *BBA-Others*, by limiting the variations of the reservoir, sticks more on the current buffer occupancy.

Figures 5.1.2 and 5.1.3 also show implicitly the effects of variable segment size: in fig. 5.1.2 there is a drop on the buffer occupancy at about 580s (circled in orange); we can find the same drop, but in the value of the bit rate selected, in fig. 5.1.3, at 480s (circled in orange): these drops correspond to the download of the same group of segments. These segments are bigger than the average, as shown by fig. 5.1.1.

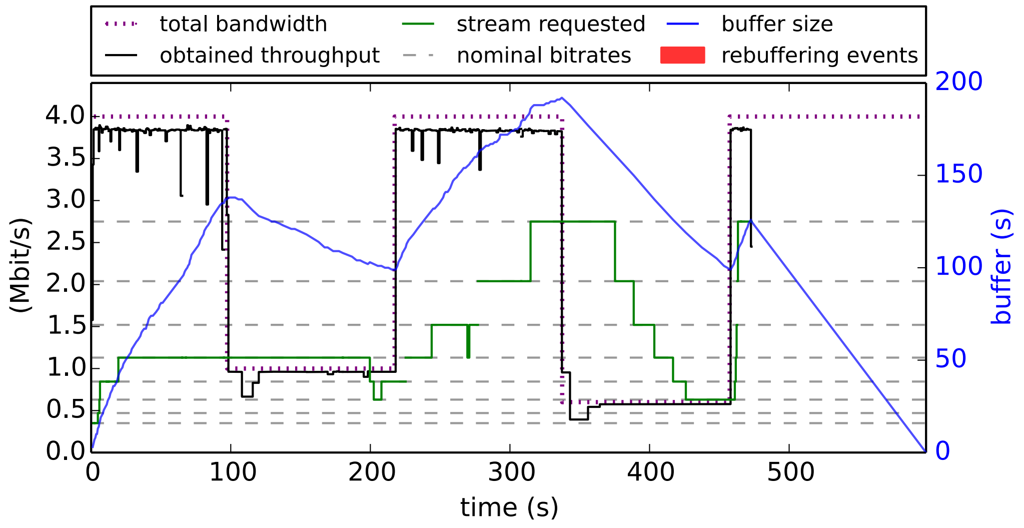


Figure 5.1.4: *BBA-Others* with variable bottleneck capacity

5.2 TCP metrics for a single client

Zooming in to the network activity of the single downloads, it is possible to see the impact on TCP behavior, and of TCP on the streaming algorithms.

5.2.1 How CWND grows

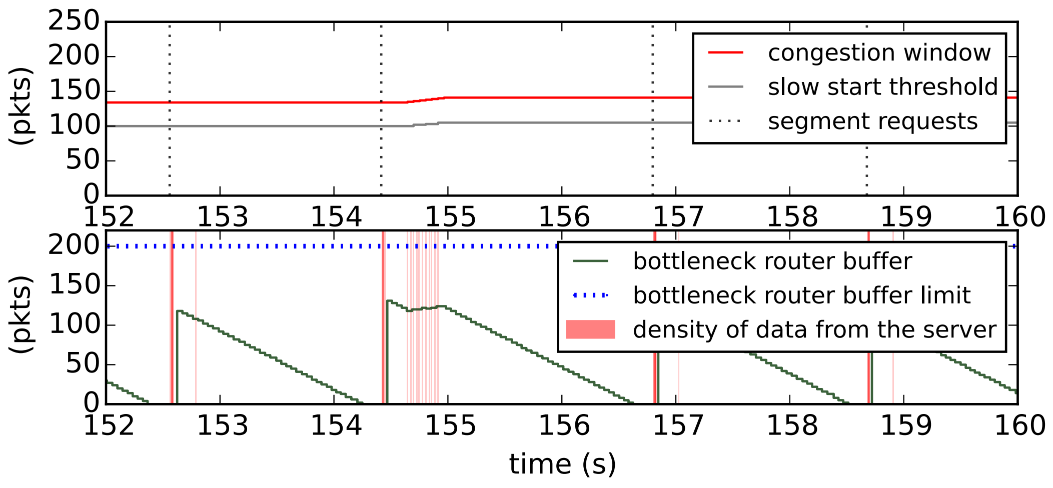


Figure 5.2.1: *BBA-Others*, Big Buck Bunny, capacity: 900kbit/s, RTT: 200ms, router buffer: 200 packets. Detail.

Fig. 5.2.1 present how the congestion window and the router buffer interact during segment transfers. We can see two interesting examples of transfer:

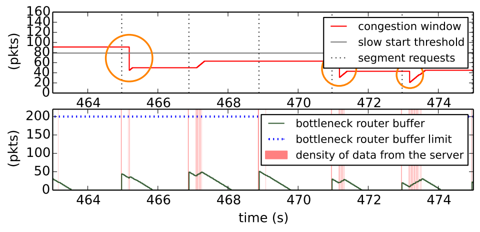
- The majority of the downloads are like the first one, on the left (152.5s). The server sends *almost* all the segment's data in an unique burst at the beginning (the red vertical line). The router buffer grows to the CWND value, and the packets are slowly released, as the limitation imposes.
- The segment sent during second transfer (154.5s) is bigger. In this case, the congestion window is not large enough to contain all the packets, so the server waits for the first ACK to make it grow and send more packets, entering in the so-called *ACK-clocking* phase of TCP: send new packets upon ack reception; the light red vertical lines in the plot represent these time-spaced transmissions. Obviously the router buffer increases in size, too.

Even if, like in fig. 5.2.1, the segments are downloaded one next to the other, the router buffer occupancy oscillates continuously, going to zero at the end of each transfer; this burstiness is a characteristic of HAS and does not happen for instance during the transmission of long files, where the buffer hardly goes to zero.

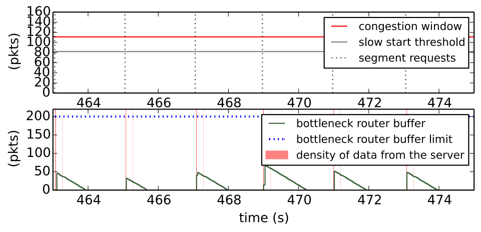
5.2.2 CWND idle reset

During OFF phases (mainly for rate-based algorithms, but also for buffer-based, when the buffer occupancy is high), there is the risk that the congestion window is reset to a lower value: *initial_window_size*, even if the connection is not closed. This could happen after relatively long periods of inactivity, in the order of the Retransmission Time-Out (RTO) [11].

Some examples are circled in figure 5.2.2a, where the congestion window drops by one half; note that the window size was decreased some time before the drop is visible on the figure; indeed if we look at the corresponding



(a) *slow_start_after_idle* enabled



(b) *slow_start_after_idle* disabled

Figure 5.2.2: *classic* algorithm, Big Buck Bunny, capacity: 900kbit/s, RTT: 200ms, router buffer: 200 packets. Detail.

transfer we can see that the initial burst is about 50 packets (from the router buffer occupancy), but a non negligible amount of data is sent after the first ACK, and this would not happen if the CWND was ~90pkts. The delay on the plot is probably a glitch of *tcp_probe*, the tool used to inspect the TCP internals: it looks like it updates its status only upon ACK reception.

It is possible to disable this "window timeout" behavior, so to prove that the CWND drops are due to it, we run the same test with the feature disabled (fig. 5.2.2b). To prevent the behavior it is enough to run as root on the server machine:
echo 0 > /proc/sys/net/ipv4/tcp_slow_start_after_idle
As you can see in fig. 5.2.2b, the congestion window for the very same downloads, does not drop; in this way we expect to have better performances, as throughput is directly proportional to CWND. It is interesting to see that in this case there isn't any impact on obtained average bit rate (354kbit/s in both cases) nor on throughput (564kbit/s in both cases).

To understand better this behavior, we run a similar test using BipBop video (fig. 5.2.3); this video has longer segments (10 seconds instead of 2 seconds), so the OFF phases will be longer; letting us expect a bigger impact on the results.

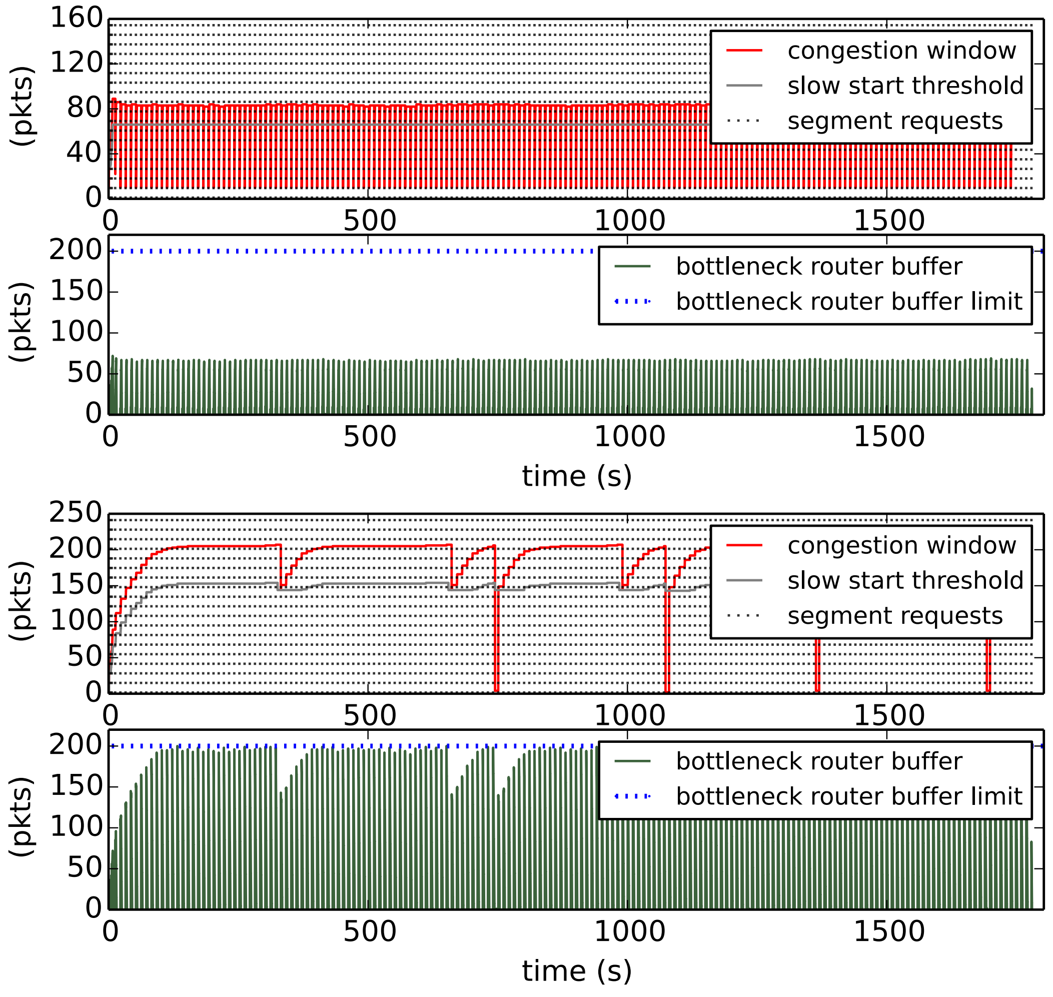


Figure 5.2.3: *classic* algorithm, BipBop, capacity: 900kbit/s, RTT: 200ms, router buffer: 200 packets, entire run.
above: *slow_start_after_idle* enabled, below: disabled

The measured throughput has only a little gain when *slow_start_after_idle* is disabled: from 775kbit/s to 781kbit/s. This is probably because when the window is reset, slow start[4] is triggered, so the previous CWND is quickly restored (note that slow start threshold is left untouched).

A bigger impact could be observed when the available bandwidth is much higher than the selected bit rate: in this case, the segments are quite small

[4]Slow start is a mechanism in TCP to make the window quickly grow at the beginning of the connection: for each ACK, the CWND is increased by one and two packets are sent. This works until the CWND value reaches the slow start threshold

and there isn't enough time for the congestion window to grow up to the optimal value. As an example: BipBop video, *classic* algorithm, bottleneck at 10Mbit/s. With idle detection, the measured throughput is about 5.8Mbit/s, while without it we measure 8.4Mbit/s.

5.3 Two competing clients

What happens when two clients share the same bottleneck? Previous work [5] underline some issues with the rate based algorithms. Does it still hold with buffer based family?

5.3.1 Constant bottleneck capacity

BipBop

As for the single-client scenario, experiments were run with 200ms delay and 200 packets router buffer; we doubled the value of bottleneck capacities, ranging from 600kbit/s to 4.4Mbit/s; all experiments were run four times.

These are the main differences comparing to single-client tests:

- All the clients get in general an higher bit rate. Approximately the BBAs get +3 percentage points, while the RBAs +20 p.p. The families are nearer, while keeping a solid distance of 25 percentage points. The big grow of RBAs is not unexpected: for each test we doubled the number of clients but also the bottleneck capacity, and there is always the possibility for a single player to take the entire channel during the OFF phase of the other client. Conversely, BBAs have *almost never* OFF phases[5], so this possibility is very rare for the family.
- The trend for instability is growing for rate-based algorithms and decreasing for buffer-based, even if the differences are not so sharp (1-2 p.p., meaning 2-4 switches).

[5]Any buffer-based algorithm switches to ON-OFF behavior when the playout buffer is full; in fact, it could happen only when the throughput keeps higher than the highest bit rate

- Unfairness: the RBA family shows values around 20%, while the BBAs around 10%. This indicates a better fair allocation of the resources by buffer-based algorithms. Note that unfairness is calculated as difference of the averages of the bit rate: any good value of this metric could hide unfair clients; for example the clients could keep oscillating between the perfect fair allocation, fail to get it and keep changing; for this reason, unfairness should be compared to instability.

Big Buck Bunny

The same tests as for the single-client case were run, but with the bottleneck capacity doubled: so with different delays (100ms, 200ms and 400ms), router buffer limits (200 packets, 100%, 50%, 25%, 10% of BDP) and capacities 800kbit/s to 6Mbit/s, in 200kbit/s steps.

Looking at rebuffering ratio, we note that:

- *classic_est*, on big router buffer, shows high values also for smaller RTTs (from 15% to 43%). The roots are probably the same as for single-client: the algorithm does not take into account the delay introduced by the HTTP handshake before the actual download (fig. 5.0.1); with two clients, the effect is more explicit as at the time a client issues an HTTP request, the other client probably has some packets on the router buffer, inflating the delay.
- For router buffer depending on the BDP, the value of rebuffering ratio tends to be half if compared to single-client sessions; it should be noted that the value is biased by the lowest bottleneck capacity imposed, that does not permit to stream the lowest bit rate (because of VBR and TCP/HTTP overhead) for single-client sessions, but is doubled for two-clients sessions; in this case the clients could alternatively exceed the fair share, obtaining, in this way, lower rebuffering.

Comparing to single-client tests, all the players take higher bit rates. The increase is particularly significant for RBAs, being around +10 percentage points; note that for RBAs also the instability grows, while BBA's instability is almost untouched.

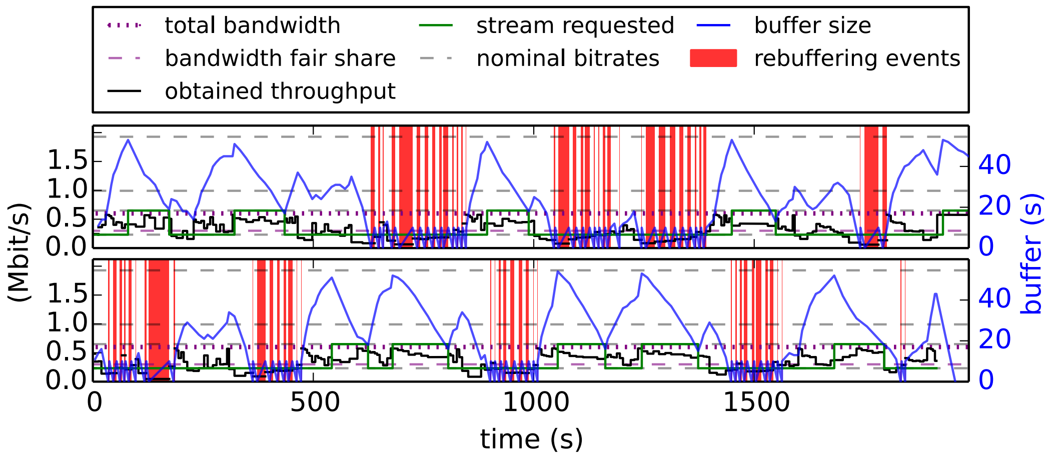


Figure 5.3.1: *BBA-1*, BipBop, capacity: 600kbit/s, RTT: 200ms, router buffer: 200 packets. The two plots correspond to the two clients.

We see that in average the unfairness is bigger for bigger router buffers. This is true in particular for *BBA1-2* and *classic_est*; other metrics suggest that these algorithms are the most aggressive. A possible scenario that explains the issue is:

- Both clients are downloading at a good rate (CWND have similar values).
- It could happen that the router buffer is near to the maximum when a new transfer starts.
- The bottleneck router drops packets when the buffer reaches its maximum; so in this situation, most of the packet of the initial burst are dropped.
- The sender believes that the channel is heavily congested, and it shrinks the congestion window to a very low value; the corresponding client starts experiencing low throughput, and its buffer starts decreasing; eventually, it could experience rebuffering.

At this point, the two cwnd are different, and we expect tcp to take them back to similar values, but this does not happen because:

- The "unlucky" client asks low size segments, that are smaller and "require" a smaller CWND.
- The "lucky" one gets higher throughput as the leftover is bigger, the buffer will increase and a higher bit rate will be selected.

Until the opposite happens, and the clients switch between each other.

5.3.2 Variable bottleneck capacity

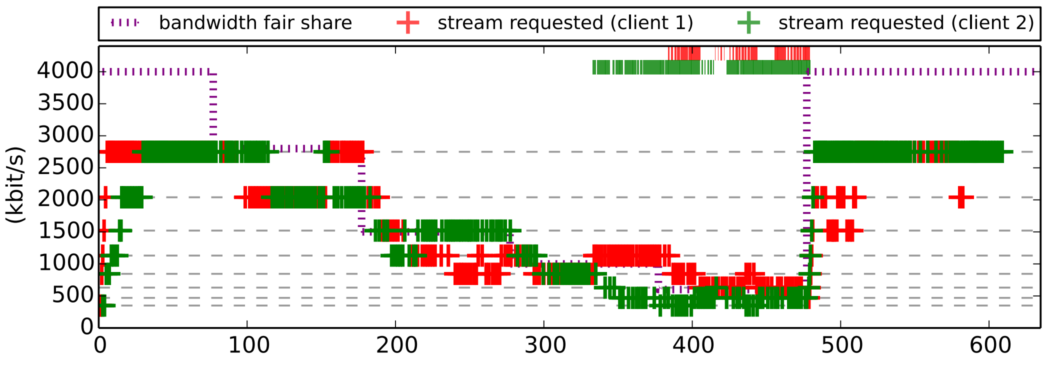


Figure 5.3.2: *classic_est* with variable bottleneck capacity

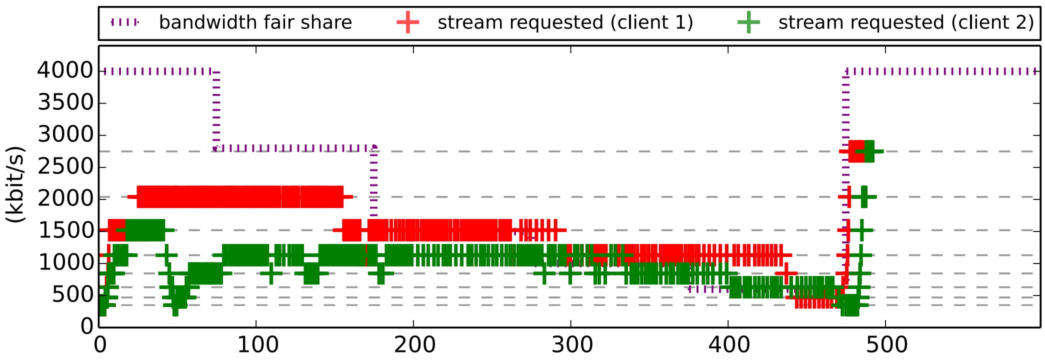


Figure 5.3.3: *BBA-Others* with variable bottleneck capacity

Figures 5.3.2 and 5.3.3 show two examples of the behavior of competing clients on a bottleneck with variable capacity. The colored stripes on top of the images represents the rebuffering events; we note that there is no rebuffering events for *BBA-Others*, while the rebuffering ratio for *classic_est* clients is 16.2%±3.3 (averaging on four different runs of this same experiment). Each cross represent a segment download. The purple dotted line is the *fair share*, half of the bottleneck capacity. The average bit rate obtained by *classic_est* is higher than *BBA-Others*: respectively 1816±34kbit/s and 1305±236kbit/s; but as said before, an high value is not so significant when also the rebuffering ratio is high. On the fairness side, *classic_est* seems behave better compared to *BBA-Others* in this test case; as they respectively get 78±5kbit/s and 514±196kbit/s; *BBA-2* gives a better result: 338±138kbit/s, with high instability: 26%, that is comparable to the single-client average instability, but it could still indicate an oscillating allocation of the resources.

5.4 Three competing clients

Some tests were run with three competing clients over constant bottlenecks. In this case the packet were delayed by 200ms and the router buffer limit was set at the BDP; the bottleneck capacities 1.2Mbit/s to 9Mbit/s, in 300kbit/s steps. The algorithms show similar results as in two-clients sessions.

5.5 Final remarks

In order to clarify the performances concerning QoE, network and competition, we ran some tests using both algorithmic families; our tests shed light on some of the strong and weak points of the strategies, not claiming, however, a winner: the algorithms we tested behave in general within acceptable boundaries. In particular, rate-based algorithms are more stable, while rate-based tend to be more resilient to highly variable available bandwidth.

Netflix [7] compared its proprietary rate-based algorithm with the buffer-based family presented here; the tests were conducted in a high variable scenario, the real life, where the available bandwidth it is not stable at all and could differ by an order of magnitude between the subsequent transfers. *BBA-Others* got similar bit rates in average, but succeeded reducing rebuffering events by 20-30%. Even if the test conditions were quite different, our results don't contradict Netflix outcome.

Chapter 6

Conclusions

Video streaming is a technology that delivers multimedia contents across the Internet, enabling a nearly instant access to the content; as an example, YouTube and Netflix are two services that use this mechanism: after a short wait, during which a safety buffer (playout buffer) is filled, the player can play the video. Cisco and Sandvine [1, 2], observing Internet traffic, agree that video streaming has a significant influence on today's and tomorrow's global computer network: the design of streaming applications plays an important role for user satisfaction and infrastructure stability.

While the network status is objective and measurable, user satisfaction is subjective; depending on the service type, some indicators could give an idea of the Quality of Experience (QoE). For video streaming, important (and opposite) indicators are rebuffering and bit rate; the former denoting video pauses for insufficient bandwidth, the latter being direct responsible for image and sound quality. User satisfaction has a fundamental influence especially for commercial services, as dissatisfied users are likely to quit the service and eventually switch to a competitor.

Designing protocols and applications meant to widely run over the Internet is a critical task, as the scenario is complex and the interactions with other net citizens difficult to predict; in order to have a strong implementation it is necessary to match the theoretical model with empiric tests.

Buffer-based HAS strategy is an interesting proposition, already validated by Netflix in the global case, with good outcome; it still needs, anyway, experiments in specific cases to understand better the client behavior.

Our tests confirm the value of the proposition, revealing on the other hand some cases in which the algorithm does not yield a fair allocation of the network resources; worth to note that our research is far from being complete and exhaustive: *future work* section gives some pointers on the possible extensions of the test cases.

Future work

At a first step, additional video resources should be added to the testbed: the results we obtained could have being biased by specific traits of the videos we used; thus, more videos should produce more trustworthy outcome.

It would be interesting also to see more refined rate-based strategies, closer to the market counterparts, for example the proposition of Huang et al. [3], Sabre [17] and Festive [18]; these proposals indicate some changes to the plain rate-based strategy presented here, improving performances and competition.

Looking at the tests framework, it should be broaden to support experiments closer to facts that could normally happen in residential Internet connections, like:

- competition with stable flows, as in [3];
- concurrent streaming sessions from different services, that are likely to use adaptation algorithms from different families, so to see the level of compatibility between the opposed strategies;
- interactions between HAS streaming and interactive traffic, such as VoIP calls.

Looking at the network structure in the results presented, the router buffer queues were managed in the most naive way: drop the arriving packets when the buffer is full (*drop tail*). There are however, other interesting policies (AQMs: Active Queue Management) that could indirectly change the behavior of the clients, for instance:

- Adaptive Random Early Detection (*A-RED*): packets are dropped with a probability that depends on the current queue size, avoiding the congestion by signaling, in fact, the status of the link;
- Constant Delay (*CoDel*): a parameterless policy that aims to limit bufferbloat, by recognizing *bad queues* and dropping probabilistically packets so to maintain an acceptable delay.

These policies, meant to help the peers to fairly use the network, would have an impact on the allocation of the resources on streaming applications.

As a last thought, we could investigate on more complex bottleneck configurations, so to simulate, for instance, a Wi-Fi client competing with an Ethernet client on the same ADSL line.

Bibliography

[1] Cisco. Cisco Visual Networking Index: Forecast and Methodology, 2013-2018, 10 Jun 2014.

[2] Sandvine. Global Internet Phenomena Report, Second Half 2014.

[3] Te-Yuan Huang, Nikhil Handigol, Brandon Heller, Nick McKeown, and Ramesh Johari. Confused, timid, and unstable: picking a video streaming rate is hard. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 225–238. ACM, 2012.

[4] Saamer Akhshabi, Ali C Begen, and Constantine Dovrolis. An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In *Proceedings of the second annual ACM conference on Multimedia systems*, pages 157–168. ACM, 2011.

[5] Saamer Akhshabi, Lakshmi Anantakrishnan, Ali C Begen, and Constantine Dovrolis. What happens when HTTP adaptive streaming players compete for bandwidth? In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, pages 9–14. ACM, 2012.

[6] Jairo Esteban, Steven A Benno, Andre Beck, Yang Guo, Volker Hilt, and Ivica Rimac. Interactions between HTTP adaptive streaming and TCP. In *Proceedings of the 22nd international workshop on Network and Operating System Support for Digital Audio and Video*, pages 21–26. ACM, 2012.

[7] 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. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 187–198. ACM, 2014.

[8] History of the Internet – Packet Switching. `http://www.securenet.net/members/shartley/history/packet.htm`.

[9] RFC1122, Requirements for Internet Hosts – Communication Layers. `http://www.rfc-editor.org/info/rfc1122`, October 1989.

[10] RFC793, Transmission Control Protocol. `http://www.rfc-editor.org/info/rfc793`, September 1981.

[11] M. Allman, V. Paxson, and E. Blanton. RFC5681, TCP Congestion Control. `http://www.rfc-editor.org/info/rfc5681`, September 2009.

[12] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[13] T. Berners-Lee, R. Fielding, and H. Frystyk. RFC1945, Hypertext Transfer Protocol – HTTP/1.0. `http://www.rfc-editor.org/info/rfc1945`, May 1996.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC2616, Hypertext Transfer Protocol – HTTP/1.1. `http://www.rfc-editor.org/info/rfc2616`, June 1999.

[15] M. Belshe, R. Peon, and M. Thomson. RFC7540, Hypertext Transfer Protocol Version 2 (HTTP/2). `http://www.rfc-editor.org/info/rfc7540`, May 2015.

[16] ISO/IEC 23009-1: ”Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats”.

[17] Ahmed Mansy, Bill Ver Steeg, and Mostafa Ammar. Sabre: A client based technique for mitigating the buffer bloat effect of adaptive video flows. In *Proceedings of the 4th ACM Multimedia Systems Conference*, pages 214–225. ACM, 2013.

[18] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with FESTIVE. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2012.