Multimed Tools Appl (2018) 77:6471–6502
DOI 10.1007/s11042-017-4556-6

Interactive live-streaming technologies and approaches for web-based applications

Luis Rodriguez-Gil[1,2] · Pablo Orduña[1,2] · Javier García-Zubia[1,2] · Diego López-de-Ipiña[1,2]

Received: 25 August 2016 / Revised: 9 January 2017 / Accepted: 27 February 2017 /
Published online: 11 March 2017

Abstract Interactive live streaming is a key feature of applications and platforms in which the actions of the viewers affect the content of the stream. In those, a minimal capture-display delay is critical. Though recent technological advances have certainly made it possible to provide web-based interactive live-streaming, little research is available that compares the real-world performance of the different web-based schemes. In this paper we use educational remote laboratories as a case study. We analyze the restrictions that web-based interactive live-streaming applications have, such as a low delay. We also consider additional characteristics that are often sought in production systems, such as universality and deployability behind institutional firewalls. The paper describes and experimentally compares the most relevant approaches for the study. With the provided descriptions and real-world experimental results, researchers, designers and developers can: a) select among the interactive live-streaming approaches which are available for their real-world systems, b) decide which one is most appropriate for their purpose, and c) know what performance and results they can expect.

Keywords Webcam · Live streaming · Remote laboratories · Online learning tools · Rich interactive applications

✉ Luis Rodriguez-Gil
luis.rodriguezgil@deusto.es

Pablo Orduña
pablo.orduna@deusto.es

Javier García-Zubia
zubia@deusto.es

Diego López-de-Ipiña
dipina@deusto.es

[1] Faculty of Engineering, University of Deusto, Avda. Universidades, 24, 48007, Bilbao, Spain

[2] DeustoTech - Deusto Foundation, Avda. Universidades, 24, 48007, Bilbao, Spain

1 Introduction

The latest social trends and technological advances have led to the emergence of various popular web-based live streaming platforms, such as YouTube Live,[1] TwitchTV,[2] Instagram Livestream[3] and Facebook Live.[4] These platforms are designed to maximize scalability and, though they are indeed live, they still allow a relatively high delay (several seconds or more). This enables them to use a larger buffer, heavier compression and more effective transcoding techniques than they otherwise could. The work in [48] provides further detail on these issues and outlines the TwitchTV architecture, which is a good example. Specifically, the measured broadcast delay of that platform varies between 12 to 21 seconds. The negative impact on user experience is not too high, because for non-interactive live-streaming applications —such as live sports—, such a delay is acceptable.

However, there are also many applications of live streaming which need to be interactive. In interactive live streaming systems, the viewers affect the content of the stream. A common example is a videoconference application, in which viewers interact with each other. Other example are remote laboratories, which will be used in this work as the main case study. These labs allow remote students to view specific hardware through a webcam and interact with it remotely in close to real time. Figure 1 characterizes the different types of streaming and some of its applications.

Interactive live-streaming systems share some challenges with standard live-streaming platforms. One of those is the importance of being web-based. Throughout the last years there has been a powerful trend towards shifting applications to the Web. However, certain features, such as multimedia, have traditionally had more limited support [31, 41]. Applications that depended on them had to find workarounds: many chose to rely on non-standard plugins [9], such as Java Applets[5] or Adobe Flash.[6] Others accepted a significant decrease in their quality or performance, or could not be migrated at all. Today, with HTML5 [16] and with other related Web standards such as WebGL [44], this is starting to change. One of the features for which applications have traditionally had to rely on external plug-ins was video streaming. Now, as an example, large websites such as Youtube or Netflix[7] rely by default on HTML5 [47].

Applications that require interactive live-streaming, however, have additional requirements, expectations, and limitations. VOD (Video-On-Demand) streaming applications, such as Youtube or Netflix, are the most common platforms. Because videos exist far in advance before the users view them, they can be preprocessed at will. They can use heavy compression and prepare the video for different qualities and transmission rates. Also, they can be streamed through adaptive streaming with relative ease. Also, they rely on buffering to provide a greater quality despite network issues, and to be able to use a larger compression frame. Live applications, however, have limitations at those respects. As previously mentioned, those that are not interactive (e.g., broadcasting a live sports event), can withstand

[1] https://www.youtube.com/live

[2] https://twitch.tv

[3] https://instagram.com/livestream

[4] https://live.fb.com

[5] http://java.com

[6] http://www.adobe.com/es/products/flashplayer.html

[7] https://www.netflix.com

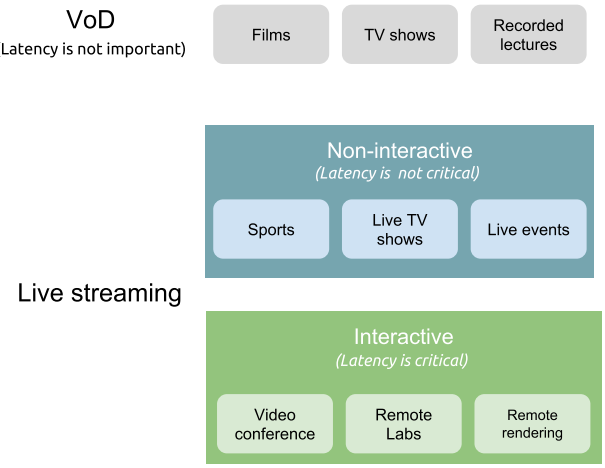

Fig. 1 Characterization of the different types of streaming and some of its applications

several seconds delay without issues. For those that are interactive (e.g., remote laboratories, collaborative tools, video conferencing applications) more than a second delay is already high: according to some HCI analysis, beyond a 0.1 seconds delay the user can notice a system is not reacting instantaneously, and beyond 1 second the user's flow of thought is interrupted [27].

In this context, researchers and system designers and developers that want to implement interactive live-streaming systems face certain difficulties. Major live-streaming platforms are closed and proprietary. It is difficult to use them for learning and research purposes [48], and they are not suitable for interactive live-streaming or as middleware for other applications. Moreover, the schemes that are available for implementing interactive live-streaming are complex. For a real-world usage, the adequacy of a scheme may depend on the video format, on the communication scheme, on the compatibility of different browsers, on the resources and bandwidth available, etc. Most of those aspects, individually, are examined in the literature. However, the real-world performance and limitations of the different real-world schemes cannot be readily predicted from it. There is little real-world experimental data that researchers and developers may use to take a truly informed decision on the approaches they choose. The main goal of this work is to provide them with that information.

In this paper, in Section 2, we describe the goals and contributions of this work and the particular requirements of web-based interactive applications that rely on live-streaming that we will consider. To illustrate the case practically, we put special focus on remote laboratories and educational applications. We propose some criteria through which the effectiveness of each approach can be compared. Then, in Section 3, we examine and describe several approaches that Web-based interactive applications may use for providing live-streaming capabilities. The five of them that seem potentially more relevant according to the previously defined criteria are described in more detail, and selected for further experimental comparison. In Section 4 we describe the experiments that have been conducted to measure the effectiveness and real-world performance of those five approaches. In Section 5, we compare the results of the different experiments. In Section 6 we examine the results and comparison and we offer an interpretation and some guidelines for potential application.

Finally, in Section 7 we draw a number of conclusions and we outline some possible future lines of work.

2 Motivation

2.1 Challenge and purpose

In a live streaming system the content is typically produced while it is being broadcast. That is, essentially, what differentiates it from non-live systems. However, there is still a very significant delay between the moment a frame is captured and the moment it is displayed in the target device [1, 15]. This delay is not only the result of network or hardware latency. It is built into the design to achieve a higher scalability [43]. In non-interactive live streams, a delay of seconds does not typically harm the QoE (Quality of Experience), and it makes it possible to leverage techniques such as buffering, video segmentation and high-compression motion codecs. An example of this is the case of the Twitch[8] platform. It relies on different techniques depending on the target device, but it tends to have a higher than 10 seconds delay [48]. Other example is the YouTube[9] live streaming platform, which lets users choose between better quality or lower latency. Even in the lower latency mode a capture-display delay higher than 20-30 seconds is, reportedly, not unexpected.[10] This is appropriate for several types of applications, such as broadcasting a sports event. However, for certain interactive applications, delays higher than a second, as previously established, can already be considered high.

An *interactive* live streaming application differs from a non-interactive one. Users are not simply passive spectators to the content. Instead, they are able to interact with it or through it, affecting the stream [48]. This imposes a strong constraint on the maximum acceptable capture-display delay that is not present in other types of live streaming. It could thus also be considered as *near-real-time* streaming. Some of the potential applications for interactive live streaming (see also Fig. 1) are the following:

- **Videoconferencing software**, such as Skype,[11] Google Hangouts,[12] or Apple Facetime,[13] in which users view, listen and react to each other in *near-real-time*.
- **Surveillance systems**, in which the viewer should be able to see what is happening in almost real-time.
- **Remote rendering systems**, in which the server handles the rendering and sends the video to the client in real time. An example is cloud-based gaming [36]: rendering a videogame in the server-side and forwarding the input from the client. Other example is free-viewpoint rendering [40]: in such a system, with many video inputs, the server has a huge amount of video data. To reduce bandwidth requirements, only the relevant portions are served to the client in real time.

[8] http://www.twitch.tv

[9] http://www.youtube.com

[10] Though no official figures are provided by YouTube, several observations and informal tests are available, such as those found at http://blog.ptzoptics.com/youtube-live/low-latency-streaming/ or at the Google product forum (https://productforums.google.com/forum/).

[11] https://www.skype.com

[12] https://hangouts.google.com

[13] https://apple.com/facetime

– **Interactive remote laboratories**, in which users interact with real physical equipment located somewhere else with a webcam stream as their main input.

The contributions of this work are intended to be useful for any web-based interactive live streaming application. However, due to the experience and background of the authors, the examples of this work will mainly relate to this fourth type of application: remote laboratories. Nowadays, remote laboratories often rely on relatively old technologies and approaches to provide interactive live streaming. Examples of such an approach is refreshing an image from JavaScript, or relying on the M-JPEG codification scheme. It is currently not clear, however, which of these relatively old approaches are more effective. Also, it is not clear whether newer approaches are not being used due to:

– Inertia and developer preference.
– More advanced technologies (such as adaptive streaming, video segmentation, or high-compression codecs) not being effective for near-real-time streaming.
– Newer approaches having significant real-world issues, such as portability issues, low reliability or difficulties to deploy behind institutional proxies.
– No literature available on the approaches available, their effectiveness, and the expected real-world outcome.

This work thus aims to shed more light in that area. The goal is that the remote laboratory community in particular and other interactive live streaming applications in general have the information to make better decisions on which streaming approaches to implement. And, moreover, so that they can know what effectiveness and performance they can expect by doing so. It also aims, specifically, to describe the currently used approaches and their architecture, and to propose some novel ones.

2.2 Contributions

The contributions of this work are thus the following:

– A brief analysis of which characteristics are important for interactive live streaming applications.
– Description and architecture of the most common interactive live streaming approaches that are currently used by remote laboratories (JavaScript-based image refreshing, and native M-JPEG).
– Description and architecture of some more advanced approaches, which, to our knowledge, have not been used in real-world remote laboratories but which could be superior. (JavaScript-based M-JPEG, JavaScript-based MPEG-1 and JavaScript-based H.264/AVC, all three relying on Web Sockets as a transport).
– Experimental analysis of the support for these approaches across all major desktop and mobile browsers.
– Experimental performance comparison of those described approaches that are most relevant.
– Scientific knowledge for existing developers of systems that rely on interactive live-streaming, that enables them to make educated decisions on the feasibility and convenience of incorporating alternative technical approaches into their implementations.
– Conclusions, based on the results of the experiments, on which approaches would be more appropriate depending on the type of remote laboratory required.

Of all of those contributions, the main one is the experimental performance comparison among the most relevant web-based approaches.

2.3 Remote laboratories

A remote laboratory is a software and hardware tool. It allows students to remotely access real equipment located somewhere else [9, 13, 24]. They can thus learn to use that equipment and experiment with it without having it physically available. Research suggests that learning through a remote laboratory, if it is properly designed and implemented, can be as effective as learning through a hands-on session [5]. Additionally, they can offer advantages such as reducing costs [26] and promoting sharing of equipment among different organizations [29]. Many remote laboratories feature one or several webcams. Through them, users can view the remote equipment. Simultaneously, they can interact with the physical devices using means such as virtual controls that are physically mapped to them. (e.g., [14, 20, 42, 46]). Some remote laboratories are even designed to allow access from mobile devices [8]. An example of remote laboratory is depicted in Fig. 2. In this particular case,[14] the students experiment with the Archimedes' principle. They can interact with 6 different instances of equipment, for which 6 simultaneous webcam streams are needed.

2.4 Technical goals and criteria

We propose a set of technical goals and criteria to compare and evaluate the different interactive live streaming approaches that will be examined.

The key technical goals that will be considered are the following:

- **Near-real-time**: The delay between the actual real-life event and the time the user perceives it —the latency— should be minimum for the interaction to be smooth.
- **Universality**: The applications should be deployable under as many platforms, systems and networks and as easily as possible.
- **Security**: The applications should be secure.

Though less critical, the following traits significantly affect the Quality of Experience and will be taken into account when evaluating the different possible approaches:

- **Frame rate**: The higher the better.
- **Quality**: The higher the better.
- **Network bandwidth usage**: The lower the better.
- **Client-side resources**: CPU and RAM usage. The lower the better.

Server-side processing is also an important consideration, especially for production systems. Though it will be considered and discussed evaluating it quantitatively is beyond the scope of this work — which focuses mainly on the client-side. Therefore, the experiments themselves include no server-side measurements.

A last consideration is the **implementation complexity** of each approach. Beyond the previously mentioned criteria, in practise, the knowledge, cost and effort required for implementing a specific interactive live-streaming approach is also, in many cases, a determining factor. Evaluating this complexity quantitatively is beyond the scope of this work.

In the following subsections we briefly describe a simplified streaming platform model. Additionally, we provide further detail and rationale about the aforementioned technical goals and criteria.

[14] The Archimedes' principle remote laboratory is usually publicly available at: https://weblab.deusto.es/weblab/labs/Aquatic%20experiments/archimedes/

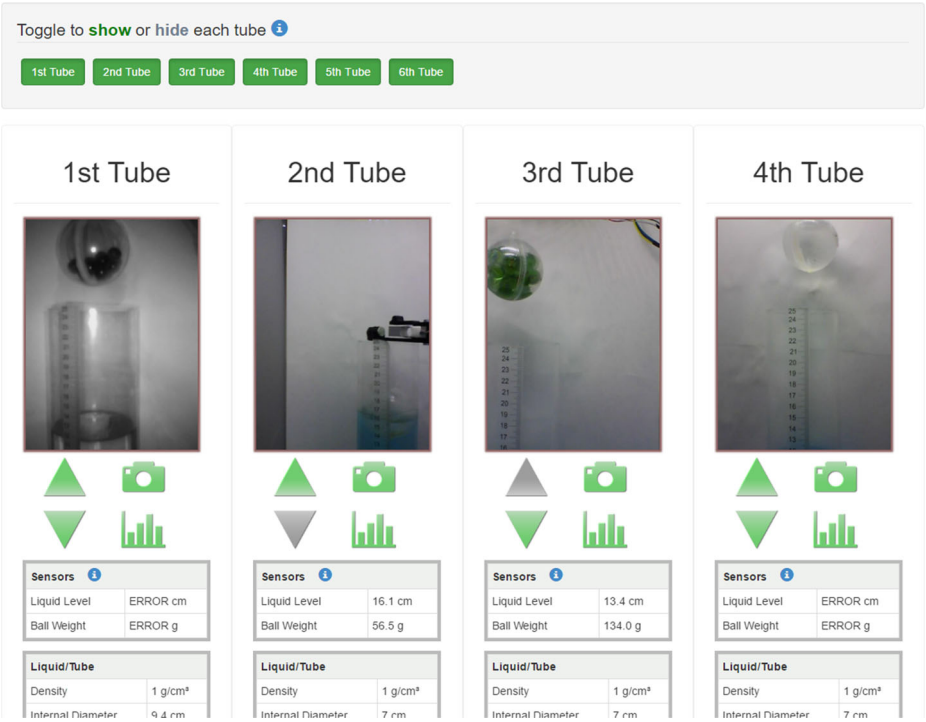

Fig. 2 Archimedes principle remote laboratory at the University of Deusto

2.4.1 Simplified live-streaming platform model

Different live-streaming platform models may exist. A simplified one is shown in Fig. 3. It is also the general model that is considered in this work. A set of IP cameras provide their input to the streaming platform through a camera output format. The particular format will vary, because different camera models support different formats. Common ones are, for instance, JPG snapshots, M-JPEG streams, and, in newer models, the H.264 format. The streaming platform receives the input and transcodes it into the target format. Often, the platform will also briefly act as a cache server for the input, so that it can scale for

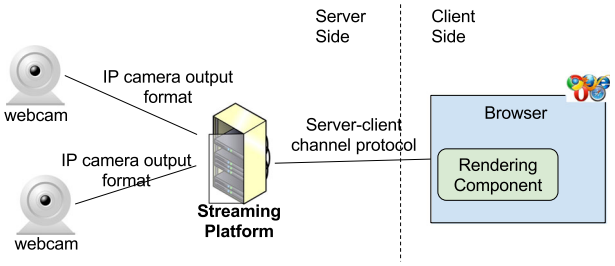

Fig. 3 Simplified live-streaming platform model

an arbitrary number of users without increasing the load on the webcams. The transcoded output is served through the server-client channel protocol (e.g., standard HTTP, AJAX, Web Sockets) to the client's browser. Depending on the approach, the browser will render it natively or through other means.

2.4.2 Near-real-time

In a live-streaming context end-to-end latency (sometimes also known just as *latency*), is generally considered to be the time that elapses between the instant a frame is captured by the source device and the time it is displayed on the target device. For most types of live streaming applications a relatively high (some seconds) latency can be tolerated without significantly harming the user's experience [6, 32]. Latency is introduced in each stage of the process. Noteworthy delays are the latency introduced by the camera, the latency introduced by the server-side encoding, the latency introduced by the network and the latency introduced by the client (decoding and displaying). These sources of latency are analyzed and discussed in detail in the white paper by Axis Communications [22]. Tolerating a relatively high delay is a significant advantage. Especially in a bandwidth-constrained network, codecs that provide large compression but which require heavy pre-processing can be used. Issues such as *jitter* can be solved with a longer buffer. Most HTTP streaming methods rely on buffering to provide adaptation for bandwidth fluctuation, and often separate the stream into multiple media segments. This adds an unavoidable capture-display delay [23].

Interactive live-streaming applications are much less tolerant to latency. The actions of the users depend directly on what they are currently seeing on the stream. A few seconds delay is enough to severely harm their Quality of Experience. Exactly how much latency can be tolerated and how much it affects user experience varies depending on the application. For example, some works report that in conversational applications (e.g., IP telephony, videoconferencing) 150 ms is a good latency, while 300-400 ms is not acceptable [32]. For cloud-based games some studies suggest that approximately for each additional 100 ms latency there is a 25% decrease in player performance [3]. For many other types of common interactive live streaming applications, such as remote laboratories, there is, to our knowledge, little specific research available on how much increased latency affects user experience. However, the interaction style and pace of many of them, such as remote laboratories themselves, is generally similar to that of a standard application or interactive website. Thus, it is reasonable to assume that generalist interaction conclusions are appropriate. In this line, according to works such as [27], beyond a 0.1 seconds delay the user can notice that a system is not reacting instantaneously, and beyond 1 second the user's flow of thought is interrupted.

Due to all this, supporting near-real-time (which for the purpose of this work, we will consider as being able to provide a relatively low end-to-end latency) is a particularly important requirement for an effective interactive live streaming approach, and the set of techniques that can be applied are significantly different than those that are applied for standard live-streaming or for VoD (Video on Demand). Modern techniques which are very popular and effective for standard streaming are sometimes not an option anymore, or are severely limited:

- **Buffering**: Would add a delay of at least the buffer length, so it can't be used or needs to have a minimal length.
- **Segmented streams**: Would add a delay of at least the segment length.
- **Pre-transcoding**: Not really an option if a small delay is required.

2.4.3 Universality

The meaning and usage of *universality* varies between contexts, but in this paper we will use it to refer to the degree to which an application is technically available to those who may want to use it. Aspects which increase universality are, among others, the following:

- Being cross-platform
- Being web-based
- Being available across many types of devices (PCs, mobile phones, tablets)
- Having less technical requirements to run properly
- Requiring less user privileges to run
- Being deployable behind more strict institutional firewalls and proxies

Universality is generally positive, but it is important to note that, in practise, it often implies important trade-offs. Depending on the particular context, needs and requirements of an application, the actual importance of universality will vary. In the case of remote laboratories, research suggests that it is one of the most important characteristics [9], but in other cases this might differ. It is noteworthy that this work aims to contribute to web-based interactive applications, which, for being web-based, already tend to provide relatively high universality.

2.4.4 Security

Being secure can be considered a goal of any application. However, the importance will vary depending on the context. Some technologies tend to provide greater security than others. For example, remote laboratories and other educational applications are often hosted by universities. Their IT teams are often hesitant to offer intrusive technologies to students to avoid exposing them to security risks, for which the university could be liable [9]. All things equal, *non-intrusive* technologies are thus preferred.

2.4.5 Frame rate

The frame rate is measured through the frames-per-second (FPS) metric. In some contexts, 50-60 FPS is considered to be a satisfactory visual quality at which increases can hardly be noticed. However, in practise, in many cases, significantly lower frame rates are used [32]. This is often in fact the case for many interactive live streaming applications.

2.4.6 Quality

Quality is hard to measure because it is actually a qualitative perception that is affected by many (qualitative and quantitative) variables. Sometimes (e.g., in Youtube) it is used as a synonym for *resolution* or *pixel density*. For simplification, in the comparison of the different approaches, we will rely the most on the *resolution*. The particular video codec that is used also has great influence in the final quality of the stream.

2.4.7 Network bandwidth usage

Live-streaming applications consume significant amounts of network bandwidth. This is because video content tends to consume significant bandwidth itself, and because often it has to be provided to many users [38]. Bandwidth usage can thus be a significant cost

and limitation, and all things equal approaches that preserve network bandwidth are preferred. Unfortunately, there tends to be an inverse correlation between network bandwidth usage and required server-side and client-side processing. That is, the codecs that require the less bandwidth tend to also be the ones that require the more processing power to code (server-side) and to decode (client-side). Sometimes specialized hardware is relied upon to provide more efficient decoding. Adding to the difficulty, some network setups, particularly mobile ones, are inherently unstable and their bandwidth capacity cannot be predicted reliably [23, 39].

2.4.8 Client-side resources

Different approaches and implementations require different amounts of CPU power and RAM. The codecs used, particularly, have a very significant influence at that respect. Client-side processing effort tends to be higher for the codecs that require the less bandwidth. To compensate for this, however, many devices also provide hardware-level support for particular codecs. Relying on hardware-level support is most of the time significantly more efficient, in terms of processing and energy usage. At the same time, because support tends to vary between different devices, it can sometimes make portability harder. In this work, the client-side processing effort will be measured in terms of CPU and RAM usage, though additional variables could be taken into account, such as energy cost, I/O usage or discrete graphic card usage. It is noteworthy that some applications have different client-side processing restrictions than others. All things equal, lower resources usage is better: A Video on Demand (VoD) application, for instance, could admit a relatively high usage in exchange of low bandwidth and high quality. There is a single active stream and the user is not expected to be multi-tasking. However, a remote laboratory or an IP surveillance application which requires being able to observe many cameras at once would often have stricter limits. See for instance the remote laboratory in Fig. 2. The students have access to 6 different simultaneous streams. Through them, they must be able to interact with the equipment in real-time. Thus, the resource usage of an individual stream must be significantly conservative.

2.4.9 Server-side processing

Server-side processing can be very high due to the pre-processing, compression and encoding that is sometimes used. Large media servers and systems, and especially those that aim to scale to many concurrent users per stream, such as Wowza,[15] YouTube and Netflix, encode a given source video into many separate formats and qualities. Thus, they can dynamically adapt the stream to the bandwidth and technical restrictions of each user. A higher or lower quality stream can be served depending on the bandwidth that the user has available. Also, one format or another can be served depending on whether the user's device or browser supports that format or not, and depending even on whether the user's device supports hardware acceleration for that format. For interactive applications the possible choice of codecs and formats is more limited, because the latency cannot exceed certain values. Also, it is noteworthy that for applications which do not aim to scale to many concurrent users per stream, but which instead aim to serve a relatively high number of different streams (such as many remote laboratories) it is sometimes convenient to accept a higher bandwidth usage in exchange of a lower processing effort.

[15] http://www.wowza.com

Though server-side processing could thus be an important consideration, this paper focuses on the client-side and therefore, though server-side considerations will be briefly described, experiments will focus on the client-side.

2.4.10 Implementation complexity

In practise, in production systems, the main factor for choosing an interactive live-streaming approach will often not be the technical characteristics or performance, but its implementation complexity. Technically superior approaches may be overlooked in favour of approaches that require less knowledge and effort and have a lower cost to implement. The quantitative evaluation of the implementation complexities of each of the different approaches is beyond the scope of this work. It would be hard to do and very difficult to reach meaningful results, due to its often developer-specific and subjective nature. Nonetheless, the architecture used for each experiment will be described and thus the implementation complexity may be partially inferred from it.

3 Interactive live-streaming approaches

In this section we will describe and analyze some of the different approaches that are available for web-based interactive live streaming. The first of them (image-refreshing and native M-JPEG based approaches) are often used by the architectures, use-cases and implementations of live streaming applications that can be found in the literature. Additionally, we include some novel approaches which are more rarely used, some of which have only recently become available due to their reliance on new or under-development Web standards.

The diagram in Fig 4 shows the generalized (and simplified) process that the interactive live streaming platforms follow from the time a frame is captured to the time it is displayed. The flow starts when the IP webcam captures a frame. Many different IP webcam models exist, and different models support different output formats. Some of the most common are JPG (discrete images), M-JPEG and H.264 streams. Through those formats, the output

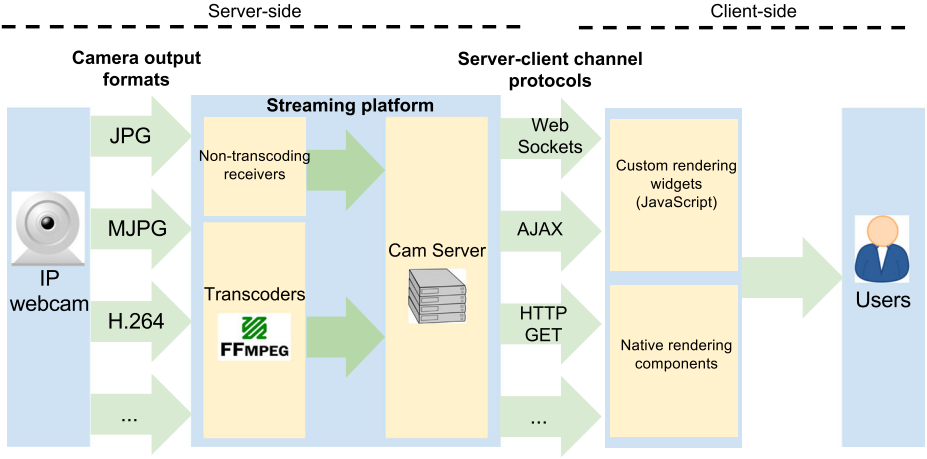

Fig. 4 Interactive live-streaming platform process

from the cameras is forwarded to the streaming platform. Depending on the source and target formats, it might or might not be necessary to transcode it into a different format. Transcoding can take a significant amount of processing power and adds some latency. The cameras server will briefly store these images, and will be responsible of serving them to the browser. Different server-client channel protocols are available. Once delivered, the browser will render the provided data to the user. Depending on the particular scheme, it will rely on a native component (such as for native M-JPEG videos or for image refreshing) or it will process, decode and render the data through JavaScript.

In the following subsections we will describe in more detail the different schemes. We will describe more thoroughly those schemes which best seem to meet the criteria described in Section 2, and which we will select for the experimental comparison. For most schemes, we will describe separately the server-side and client-side.

3.1 JavaScript-based image refreshing

This is a mature technique which is technically very simplistic, both in the client and the server-side. The server simply provides individual access to the current frame and the client repeatedly asks for a new frame using JavaScript. Despite its technical shortcomings it is used by many applications in the literature, sometimes as a fallback. Particularly, it is used by many remote laboratories for which a high FPS isn't absolutely necessary, including most of the remote laboratories of WebLab-Deusto,[16] LabsLand[17] and RemLabNet[18] [34].

3.1.1 Client-side

In the client-side only a browser with JavaScript support is required. The HTML contains an `<img>` tag which points at the webcam image. Then, from JavaScript, the image tag's `src` attribute is constantly modified. Under all modern browsers —and most legacy ones— when the `src` tag is changed the new image is loaded. To prevent some technical issues, generally some additional low-level considerations are taken:

- To modify the webcam image's URL in some way (such as by adding a random number to the *GET query* parameters) so that no cache issues occur.
- To query for a new image only after the `load` event has fired, sometimes after a small delay, so that requests don't accumulate on slow networks.

3.1.2 Server-side

Server-side this particular approach is also relatively simple. Most IP webcams provide an image URL that serves one frame, so that URL just needs to be made available either directly or through a standard web server. Some applications require several concurrent users and the hardware of webcams is sometimes not particularly powerful, so a cache server can be used. In that case, a cache server in the same network would continuously request frames to the camera. Then, whenever the web server asks for a frame, the last cached frame is served.

[16] https://weblab.deusto.es

[17] http://labsland.com

[18] http://remlabnet.eu

3.2 Motion JPEG

Motion JPEG, often known as just M-JPEG, is a video format with intraframe-only compression: each frame is essentially a separately compressed JPEG image. Because of this, its compression rate tends to be significantly lower than that of most modern interframe formats [4] (e.g., H.264/MPEG-4 AVC [19], H.265 [18], VP8 [2], VP9 [11]). However, it does have some significant advantages [21]: it is simple to implement, it requires little memory and processing power, it responds better than other formats to packet loss, fast image changes [4], and network jitter [28]. Many IP webcams provide native support for M-JPEG streaming, and M-JPEG is supported by most modern browsers, including the desktop and mobile versions of Google Chrome, Mozilla Firefox, Apple Safari and Microsoft Edge. It is not natively supported, however, by Microsoft Internet Explorer, and there have been significant issues with the implementations of most of these browsers. Examples of interactive live streaming applications that currently rely on M-JPEG are the RexLab.[19] Other example are laboratories based on the iSES[20] SDK [33], which support both browser-native M-JPEG and a JavaScript-based M-JPEG decoder which receives the frame from the server through WebSockets.

Natively, browsers and HTTP servers implement M-JPEG by sending the videos through special mimetypes such as `multipart/x-mixed-replace` and using the Chunked Based Coding feature of HTTP 1.1 (RFC2616 [7]), which is a data transfer mechanism that allows the transmission of dynamically generated content. When the request is started the total length does not need to be known. Instead, an undetermined number of *chunks* can be sent, and the request can be completed any time by a final zero-length chunk. In the case of a M-JPEG stream, servers keep a long-running request, sending the separate JPEG images that compose the video as separate chunks.

3.2.1 Limitations of browser-native M-JPEG implementations

Although the M-JPEG video format itself is relatively effective for live streaming [10, 21] and it is indeed a popular format for applications such as webcams, digital cameras or remote laboratories, there are in practise some technical and reliability issues with the native M-JPEG implementations of current browers.

Though most major browsers nominally support M-JPEG natively, it fails to work under certain versions of Google Chrome, Mobile Safari and Firefox. Also, when the stream is interrupted or fails, browsers do not recover. Under some circumstances, such as receiving at a higher FPS than the bandwidth allows, Chrome and Firefox often stop displaying the image but keep internally receiving them.

Motion-JPEG has no particular bandwidth-control or timing provisions. Browsers tend to render the frames just as they are received. This works as expected when the bandwidth is high enough for the FPS. The latency is even particularly low when compared to modern formats such as H.264 because there is no interframe compression or buffering delay. However, when the server is sending frames at a higher rate than the client can receive and display, the frames tend to accumulate and thus a significant and growing latency is

[19] Brazillian consortium headed by the Federal University Santa Catarina (UFSC) remote laboratories. On 21th April 2016 at least 8 different remote laboratories are available at http://relle.ufsc.br, all of which rely on browser-native M-JPEG.

[20] http://www.ises.info

introduced. Avoiding this is non-trivial and is not always possible, and the implementations observed in this work do not make any particular provision at this respect.

3.2.2 Client-side

The client-side is, at least in principle, straightforward for those browsers that natively support M-JPEG, because including an `<img>` tag is all that is needed. In practise, some implementations will provide a fallback mechanism (e.g., in [25]) for browsers that do not support it or that encounter issues, which is not uncommon.

As described in the previous section, reliability and bandwidth control is often an issue. Browsers provide no particular Motion-JPEG-related functionality or semantics through JavaScript, so detecting and recovering from failures is non-trivial. Under the tested browsers (Google Chrome, Mozzilla Firefox) no *load* or *error* events are raised on the `<img>` element that hosts the M-JPEG image when the stream is interrupted. In principle, the binary data of the image could be repeatedly accessed in JavaScript through the HTML5 APIs and explicitly compared to verify whether it has changed or not. In practise, however, this is very costly in terms of performance and is by default disallowed for externally hosted images due to CORS restrictions. Furthermore, browsers seem to have issues handling high FPS, with Chrome, for instance, soon using 100% CPU on that tab and showing a blank image.

Most observed implementations simply choose to stream at a fixed and relatively low FPS to partially avoid these issues, or let the user or administrator configure the FPS.

An alternative is to avoid relying on the native M-JPEG capability of the browsers (which tends to be flawed) and to receive and render the stream through JavaScript instead.

3.2.3 Server-side

Many IP webcam models support M-JPEG natively, so some implementations simply redirect that stream to the end-users. However, if the system should support several concurrent users, a caching server is required to achieve an acceptable Quality of Service. The server can try to adapt to available bandwidth by avoiding to queue frames and instead sending always the latest captured frame. Thus, theoretically, if the client received and displayed the latest frame just when it is received, the capture-display delay would be minimized and the server and client would be mostly synchronized. In practise, however, intermediate systems (including the browsers themselves) often buffer the TCP stream, so this scheme does not always work as well as expected.

To avoid relying on the (often flawed) native M-JPEG browser support, and to rely on JavaScript instead, the server will have to send the stream through an alternative means, such as Web Sockets, WebRTC or AJAX. This way, the previously mentioned glitches can be bypassed, and it becomes feasible to adapt to the available bandwidth. This comes at the cost of a higher client-side processing.

3.3 High-compression formats

Image-refreshing and M-JPEG, as discussed, provide relatively poor compression, but they are nonetheless often used for certain types of real-time interactive applications. That seems to be, mostly, because:

– They require little client-side and server-side processing power.

- They are simple to implement and maintain.
- Encoding and decoding takes very little time so very little capture-display latency is added.
- They can be used in almost any platform and browser.

In other contexts, however, such as non-interactive live streaming, or such as VoD, those formats are generally not used and would often be regarded as suboptimal formats. Instead, those contexts often favour formats and approaches that take higher processing time, require a more complex architecture, and add some latency; but that require lower bandwidth and provide higher quality. Example formats are, for instance: MPEG-1 [17], H.264/MPEG-4 AVC [19], H.265 [18], VP8 [2], or VP9 [11]. Today, some of those are natively supported by some browsers and systems —though not necessarily for live-streaming—, and some formats can be decoded, at a potentially high processing cost, through JavaScript. Therefore, in the current state of things, approaches that rely on this kind of codecs could today be an effective alternative for near-real-time live video streaming. Especially, since they offer a particularly high compression-rate and a relatively high quality [12].

3.3.1 Client-side

Client-side there are, again, two possibilities. First, in the most ideal case, the browser and hardware will support the format natively through the HTML `<video>` tag, and the support will also provide enough facilities for near-real-time live streaming of that format. Unfortunately, this is not always a valid approach due to the following:

- The HTML5 standard provides the `<video>` tag but it does not include live-streaming support. Though that is likely to change in the future through the MediaExtensions API.
- Each browser supports a different set of codecs, so the server needs to generate different streams to be truly cross-platform.

As an alternative, or as a fallback, it is possible to use JavaScript-based decoders for some of the formats. Such a system receives the stream data through Web Sockets, AJAX, or similar; decodes it through JavaScript, which depending on the format can require significant resources; and then renders it into an HTML5 canvas. Though more costly in terms of processing power, it is truly cross-platform and compatible with any modern browser.

3.3.2 Server-side

Server-side this approach tends to be significantly more costly in terms of resources, especially if several streams need to be provided. If a single stream is provided, it is also more costly than in the case of image-refreshing or M-JPEG due to the compression and intra-frame nature of these formats. A significant difference is also that for an user to be able to join an ongoing stream, initialization steps are required. That sometimes involves sending initialization packets through a secondary data channel, and/or waiting for specific periodic frames before being able to join.

3.4 Non-standard plugins

Traditionally, multimedia features in the browser have been limited. This has led many media-dependant applications to rely on non-standard plugins such as Adobe Flash or Java

Applets. The remote laboratory described in [37], for instance, displays a webcam through the YawCam[21] Java Applet.

Adobe Flash, Java Applets and similar plug-ins have access to native TCP and UDP sockets, which makes it possible to use non-web streaming protocols such as RTSP [35]. Although this makes this approach particularly powerful, it also implies that it is less universal. The plugins themselves need to be previously installed, which requires administrator privileges that are not always available. Also, Java Applets, for instance, are not supported in mobile devices, and Flash support is very limited. Support in desktop browsers is better, but still, Chrome has dropped support for Java Applets, and Firefox is expected to drop it soon. Other browsers are likely to follow similar paths. These issues are also discussed in [30]. Furthermore, the plugins themselves and the usage of non-HTTP protocols have security implications. As a result, applications that rely on those would be unable to be deployed under many institutional firewalls and proxies without significant configuration and policy changes [9].

3.5 WebRTC

WebRTC (Web Real-Time Communication) is an API standard that is currently under development by the W3C. It is oriented for peer-to-peer multimedia communication. This technology can be very useful for certain applications, such as videoconferencing ones, which can benefit from being decentralized and peer-to-peer. However, it has certain constraints:

- It still requires a server to handle the connection process and signalling; and to route all data for those cases where the NATs or firewalls of the clients prevent them from directly connecting to each other.
- It is oriented towards peer-to-peer connections, so its usefulness is limited when the source of the content is a traditional centralized server.
- It is not yet an accepted standard, and it is not yet supported by every major browser.

For these reasons, WebRTC-based approaches will not be considered for the purposes of this work, though it may be useful to also compare them in the future.

3.6 HLS and MPEG-DASH

HLS (HTTP Live Streaming) is a protocol created by Apple which is intended to answer some of the challenges described in this work. However, it is, at least for now, not natively supported by all browsers, and it is not standard, so *universality* is limited.

MPEG-DASH (Dynamic Adaptive Streaming over HTTP) is an adaptive bitrate streaming standard that is currently in draft form. Support is growing and in the future it is likely to be a very effective alternative.

3.7 Limitations

Though several approaches were described in the previous section, there are many more potential ones which could be feasible, and more are likely to appear in the future. Thus, it is noteworthy that the previous list or this comparison is not meant to be exhaustive.

[21] http://www.yawcam.com/

4 Experimental work

In this section, we experimentally evaluate the performance of the five most relevant schemes (described in the previous section). First, we detail the experimental setup and methodology. Next, for each scheme, we:

- Implement the scheme.
- Conduct qualitative experiments to verify whether the scheme does indeed run under different systems and devices.
- Conduct quantitative experiments to evaluate its performance.

4.1 Experimental setup

At this stage we have selected 5 approaches to be compared quantitatively (which we will refer to as image-refreshing, native M-JPEG, JS M-JPEG, MPEG-1, H.264/AVC). We are interested in measuring the performance under real-world conditions. Thus, beyond the specific scheme being analyzed, there are several other variables which may affect the measurements. Some of these are the following:

- **FPS:** Target Frames Per Second.
- **Client device:** Computer or mobile device to render and take measurements in.
- **Network:** Latency, available bandwidth, etc.
- **Browser:** Browser and specific version.

Conducting experiments for every combination would be impractical, and not particularly meaningful. Certain restrictions have been applied for each of these variables, and will be described next.

4.1.1 FPS

Though some systems rely on a variable FPS, in this case we will set a target FPS. This makes it possible to obtain comparable results for RAM, CPU and bandwidth, and is consistent with real-world usage. When applicable,[22] we will conduct the experiments against three different FPS values: 5, 10 and 25. For some schemes, we will also measure the maximum average FPS they can achieve.

4.1.2 Client device

All the quantitative experiments have been conducted under *Device A*. In addition to the quantitative experiments, several qualitative ones were conducted to verify whether the specific schemes are indeed cross-platform. For those, two additional devices were used. The specification of the three devices are the following:

- **Device A** Mac Book Pro 13' Mid 2014: 2.6 GHz Intel Core i5, 8 GB RAM, 256 GB SSD, Intel Iris 1536 MB Graphics Card. Running OS X 10.11.5).
- **Device B** Desktop PC. Intel Core i7, 8 GB RAM. Running Windows 10.
- **Device C** Samsung Galaxy S7 Edge (SM-G935F).

[22] As described in more detail in later subsections, MPEG-1 will only be measured with 25 FPS, because its standard does not allow 5 or 10 FPS.

4.1.3 Network

There are mainly two network parameters which could be considered: bandwidth and latency. The experiments were conducted in a local network, with around 100 Mbps of bandwidth and around 20 ms of latency. Though both parameters are important, preliminary experiments suggest that they do not significantly affect the comparison.

In those preliminary experiments, the bandwidth did not affect the measurements, except when the measured bandwidth usage started getting close to the maximum network bandwidth. In these cases, either the target FPS could not be met (image-refreshing scheme) or the capture-render delay started growing beyond what could be reasonable for an interactive live-streaming system (the other schemes). Differences in latency seemingly have no effect in RAM, CPU or bandwidth. As expected, however, an increased network latency results in an increased capture-render delay. The relationship seems to be, as expected, mostly linear. So, though the measurements would vary on a slower network, the comparison and the conclusions should not.

4.2 Methodology and measurements

The implementations have been deployed in a server in the local network. The video feed is obtained from a local IP webcam. The metrics that will be measured are the RAM usage, CPU usage, downstream bandwidth and capture-render latency. They all will be measured in relation to a target FPS. We have kept them as separate metrics because, depending on the specific application, the most significant ones might vary. For instance, in certain mobile networks minimizing the bandwidth usage could be the most important criteria [39]. In other cases, such as in networks with more bandwidth, minimizing RAM or CPU usage could be more appropriate.

Bandwidth measurements are incoming-only, and have been obtained either through the Chrome Task Manager or the Mac OS activity monitor (because web socket bandwidth usage is not shown in Chrome). RAM and CPU were measured through the Chrome Task Manager. 5 measurements were taken in each case. The highest and lowest measurement were discarded. The 3 remaining measurements were used to compute the average and the standard deviation, which are listed in tables for each scheme.

For measuring the latency, an IP webcam was pointed towards a desktop computer displaying a clock on the screen. Then, the test laptop (Device A) was placed next to it, rendering the webcam image through the experimental streaming system, using the particular settings of each experiment. The latency is thus equal to the difference between the live clock (in the desktop) and its image rendered in the test laptop. For each experimental combination (FPS and streaming approach) 5 pictures were taken of both screens, and the difference measured. For these 5 measurements, the average and standard deviation was computed. Figure 5 shows a picture of this setup. The screen (in the middle) shows a clock. An IP webcam (in the lower left) is pointing to it and forwarding the stream to our interactive live-streaming platform. Then, the stream is being rendered in the laptop (to the right), using one of the schemes and configurations. When a picture is taken of both screens, the capture-render delay, at that moment, will thus be the difference between the clock and its render.

4.3 JavaScript-based image refreshing

The setup for this set of experiments can be observed in Fig. 6. The live images are retrieved from the IP camera and stored into a Redis cache server. A Python-Flask-based server serves

Fig. 5 Picture measuring the capture-render delay

those images to the browser, which simply applies the image refreshing technique from JavaScript to obtain the images through standard HTTP and then displays them at a given frame rate.

The implementation was run on device A (with Chrome, Firefox, Safari), device B (with Chrome, Firefox, Internet Explorer and Edge) and device C (with mobile Chrome and mobile Firefox). Works as expected and without noticeable issues in all of them.

The performance of the experimental implementation (conducted under device A and Chrome) is summarized in Table 1. It is noteworthy that RAM usage is particularly high. It increases steadily since the first image is loaded, and, after a while, it stabilises at around 600 MB. It is hypothesised that this is because of the Chrome caching and memory optimization schemes, which retains copies of the previously loaded images up until a certain point. The latency (capture-display delay) is relatively low. For all the tested frame rates, it is within the 223ms to 316ms range. The latency is lowest for 25 FPS. Although not covered in the table, it is noteworthy that at 10 FPS but with a constrained, simulated *Good 2G* connection, the delay is stable at around 1726 ms.

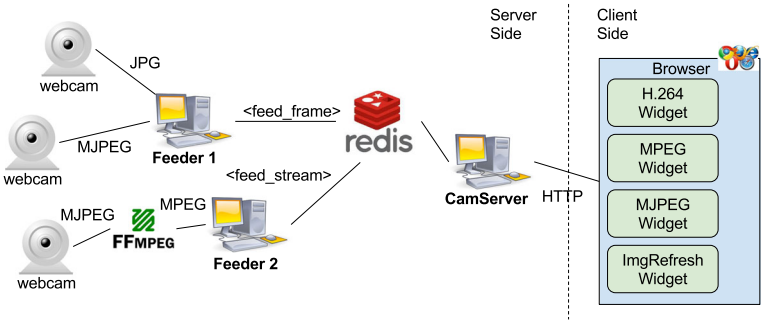

Fig. 6 Architecture and deployment setup for the different experiments

Table 1 JavaScript-based image refreshing performance

		Mean	S.D.
5 FPS	RAM	618.0 MB	2.646
	CPU	16.57%	0.115
	Bandwidth	233.0 KB/s	4.359
	Latency	316 ms	104.083
10 FPS	RAM	634.7 MB	0.577
	CPU	31.8%	0.100
	Bandwidth	461.3 KB/s	0.577
	Latency	246 ms	104.083
25 FPS	RAM	649.67 MB	17.098
	CPU	85.5%	0.709
	Bandwidth	1130.3 KB/s	1.528
	Latency	223 ms	107.480
Max	Achieved FPS	41 FPS	
	RAM	671 MB	
	CPU	116.9%	
	Bandwidth	2100 KB/s	

4.4 Native M-JPEG

The setup for these experiments is similar to the previous ones, and also depicted in Fig. 6. The live images are retrieved from the IP camera and stored into a Redis cache server. A Python-Flask-based server provides a M-JPEG stream from them to the client at a fixed target frame rate. The client displays the M-JPEG stream natively by simply using an ¡img¿ element to the stream. No particular JavaScript is needed. Thus, no particular 'Native M-JPEG' widget is present in the figure.

The implementation was tried on device A (with Chrome, Firefox, Safari), device B (with Chrome, Firefox, Internet Explorer and Edge) and device C (with mobile Chrome and mobile Firefox). Runs in all browsers except on Internet Explorer.

The performance of the experiment (conducted under device A and Chrome) is summarized in Table 2. Similarly to the previous experiment, the RAM raises steadily since the first image is rendered, and increases steadily until it reaches around 600 MBs. When not bandwidth-constrained, the capture-display delay is in the 218-515 ms range, being lowest for 25 FPS. When bandwidth-constrained, the delay steadily increases and can be higher than 60 seconds. No data for a maximum FPS is provided, because at higher FPS the stream fails sooner. This is likely due to the native M-JPEG limitations described in Subsection 3.2.1. Particularly, it seems that once a single image fails, the browser stops updating the image. And because there is no JavaScript API and no JavaScript error event raised, recovery is non-trivial.

4.5 JavaScript-based M-JPEG

The setup for this set of experiments is also depicted in Fig. 6. The live images are retrieved from the IP camera and stored into a Redis cache server. A Python-Flask-based server provides a M-JPEG stream from them to the client at a fixed target frame rate through

Table 2 Native M-JPEG

		Mean	S.D.
5 FPS	RAM	605.3 MB	1.155
	CPU	12.40%	0.265
	Bandwidth	235.3 KB/s	4.726
	Latency	515 ms	149.921
10 FPS	RAM	617.3 MB	6.658
	CPU	24.5%	0.866
	Bandwidth	445.7 KB/s	32.624
	Latency	356 ms	51.394
25 FPS	RAM	625.3 MB	1.155
	CPU	60.13%	0.503
	Bandwidth	1126.7 KB/s	12.014
	Latency	218 ms	55.426

Web Sockets and the *socket.io* library. The client renders each frame to an HTML5 Canvas through JavaScript.

The implementation was tried on device A (with Chrome, Firefox, Safari), device B (with Chrome, Firefox, Internet Explorer and Edge) and device C (with mobile Chrome and mobile Firefox). Runs in all browsers. No particular issues were observed in any of them.

The performance of the experiment (conducted under device A and Chrome) is summarized in Table 3. The RAM usage seems to increase slowly but it lowers periodically, and does not increase proportionally to the FPS. When not bandwidth-constrained the latency is

Table 3 JavaScript-based M-JPEG

		Mean	S.D.
5 FPS	RAM	166.0 MB	11.136
	CPU	6.867%	0.058
	Bandwidth	315.7 KB/s	3.215
	Latency	289 ms	71.924
10 FPS	RAM	430.0 MB	18.330
	CPU	12.5%	0.100
	Bandwidth	528.0 KB/s	34.511
	Latency	216 ms	57.735
25 FPS	RAM	306.3 MB	47.480
	CPU	28.433%	0.231
	Bandwidth	1494.3 KB/s	92.376
	Latency	284 ms	54.262
Max	Achieved FPS	115-127 FPS	
	RAM	263 MB	
	CPU	20.8%	
	Bandwidth	8434.3 KB/s	

relatively low and quite stable. It is within the 284-289 range, and it is lowest for 10 FPS, though given the small difference, it might be due to random fluctuations.

Higher than 115 FPS could be reached, and, strangely, in that case the CPU usage was actually lower than at 25 FPS.

4.6 JavaScript-based MPEG-1

MPEG-1 [17] is a very mature format and in many traditional streaming contexts it could be considered *legacy*. More modern formats such as H.264 —which will also be tested in later sections,— provide better quality and compression [45]. However, potentially, its simplicity also implies a lower processing cost and a lower latency. Considering this and that currently many applications (such as most remote laboratories) still rely on apparently less efficient approaches such as image-refreshing or M-JPEG; MPEG-1 could still be an effective option in some current contexts.

The setup for this set of experiments is also depicted in Fig. 6. It is slightly more complex than the previous ones because a transcoding component (from M-JPEG to MPEG) is necessary. Thus, in the server, a *ffmpeg* instance retrieves the live stream from the IP webcam (through M-JPEG), transcodes it into MPEG-1 and sends it to a *feeder* server. The *feeder* forwards the stream to the web server through Redis channels. The client receives the stream from that server through Web Sockets and *socket.io*. The stream is decoded using a JavaScript decoder and rendered into an HTML5 canvas. The MPEG-1 standard supports a limited number of FPS options, so in this case, the experiment was conducted only with 25 FPS (5 FPS and 10 FPS are not really allowed by the standard). The codec was configured to use an 800 Kbps constant bitrate.

Specifically, the system relies on the following ffmpeg command to transcode the stream:

```
ffmpeg -r 30 -f mjpeg -i <webcam_mjpeg_url> -f mpeg1video
-b 800k -r 25 pipe:1
```

Client-side, a pure JavaScript decoder has been used. The decoder is Open Source and was originally created by Phoboslab.[23] It has been modified to add support for the *socket.io* library, which is a wrapper around Web Sockets but falls back to a long-polling system if the specific deployment does not support the former. The modified decoder is available at Github.[24]

The setup was tried on device A (with Chrome, Firefox, Safari), device B (with Chrome, Firefox, Internet Explorer and Edge) and device C (with mobile Chrome and mobile Firefox). Runs in all browsers. No particular issues were observed in any of them. It is noteworthy, though, that MPEG-1 quality was lower than the previous approches (which was to be expected due to the higher compression).

The performance of the experiment (conducted under device A and Chrome) is summarized in Table 4. The RAM usage is very steady. The latency is on average 610 ms when not bandwidth-constrained. Though not included in the table summary, it is noteworthy that when simulating a *Good 2G* connection under Chrome, the latency is similar, probably because due to its low bandwidth requirements, the restricted bandwidth is not really constrained either.

[23] http://phoboslab.org

[24] https://github.com/zstars/jsmpeg

Table 4 JavaScript-based MPEG-1

		Mean	S.D.
25 FPS	RAM	104.0 MB	0.000
	CPU	6.9%	0.351
	Bandwidth	39.57 KB/s	3.066
	Latency	610 ms	325.087

4.7 JavaScript-based H.264/MPEG-4 AVC

H.264/MPEG-4 AVC [19] is currently a popular high-compression format, which is particularly appropriate for streaming, and is commonly supported as an output format for many modern IP webcams. The performance of a system relying on H.264 will vary significantly depending on the specific codec implementation, on the specific parameters for the codec, and on other factors. Thus, these experiments are not intended to evaluate the performance of the H.264 format itself, but, instead, the potential performance of a real-life interactive live streaming system that relies on it.

The setup for these experiments is depicted in Fig. 6. Its architecture is similar to the one that was used for the MPEG-1 experiment, and it also relies on a transcoding component (from M-JPEG to H.264). Thus, in the server, a *ffmpeg* instance retrieves the live stream from the IP webcam (through M-JPEG), transcodes it into H.264 and sends it to a *feeder* server. The *feeder* forwards the stream to the web server through Redis channels. The client receives the stream from that server through Web Sockets and *socket.io*. The stream is decoded using a JavaScript decoder and rendered into an HTML5 canvas.

The *ffmpeg* instance was configured to use the *libx264* codec, with the baseline profile, constant bitrate mode (set to 1500Kbps) and with several low latency flags enabled. Specifically, the *ffmpeg* commandline that was used is:

```
ffmpeg -r 30 -f mjpeg -i <webcam_mjpeg_url> -flags +low_delay
-probesize 32 -c:v libx264 -tune zerolatency -preset:v ultrafast
-r <target_fps> -f h264 1500k pipe:1
```

The client renders the stream through a modified *Broadway.js* decoder. *Broadway.js* is a heavily optimized, Open Source, H.264 JavaScript decoder, which has been compiled through Emscripten and is futher optimized to use WebGL. To it, we have added *socket.io* support, so that it can gracefully fall back to AJAX under systems and deployments where Web Sockets are not functional. The modified decoder is Open Source and is hosted at GitHub.[25]

The setup was tried on device A (with Chrome, Firefox, Safari), device B (with Chrome, Firefox, Internet Explorer and Edge) and device C (with mobile Chrome and mobile Firefox). Runs in all browsers. No particular issues were observed in any of them.

The performance of the experiment (conducted under device A and Chrome) is summarized in Table 5. RAM usage is steady despite the FPS. CPU usage increases with the FPS. Latency is highest at 5 FPS (under which it averages 1083 ms) and lowest at 25 FPS (under which it averages 417 ms). This difference is quite significant, and, in part, is probably due to internal buffers in the codec being filled faster at higher framerates.

[25] https://github.com/zstars/h264-live-player

Table 5 JavaScript-based H.264/AVC

		Mean	S.D.
5 FPS	RAM	177.3 MB	1.528
	CPU	11.67%	1.155
	Bandwidth	90.9 KB/s	22.848
	Latency	1083 ms	29.445
10 FPS	RAM	178.3 MB	5.859
	CPU	15.67%	1.155
	Bandwidth	96.2 KB/s	1.258
	Latency	734 ms	27.731
25 FPS	RAM	176.0 MB	1.000
	CPU	27.0%	2.646
	Bandwidth	100.3 KB/s	8.314
	Latency	417 ms	160.728

5 Comparison

Table 6 summarizes the browser support for each chosen approach and implementation. For the most part, it is very wide. This is to be expected because they were chosen, precisely, for providing relatively high *universality*.

Figure 7 compares the client-side RAM usage observed during the experiments. In the case of image refreshing and native M-JPEG it is, apparently, very high. However, it is noteworthy that when the page loads it starts small, and progressively grows until it stabilizes at the figures that are displayed. That is likely due to Chrome's caching and RAM management scheme. In the case of JavaScript-rendered M-JPEG RAM usage is significantly smaller, and it is rather volatile —which probably explains why for 25 FPS the measured RAM usage is smaller. The MPEG-1 and H.264 implementations consume, by far, the least RAM.

Figure 8 compares CPU usage. As one would normally expect, it seems to increase almost linearly with the FPS. Interestingly, however, native M-JPEG and image-refreshing consume the most CPU, while JavaScript-based M-JPEG and the high-compression-based methods —MPEG-1 and H.264— consume the least. This could be considered counter-intuitive, because MPEG-1 and H.264 have a significantly more powerful and complex, interframe compression. Although explaining these results would require further analysis

Table 6 Browser support for each approach and implementation

	Chrome	Safari	IE	Edge	Firefox	Chrome Mobile	Samsung Mobile	Firefox Mobile
Image Ref.	✓	✓	✓	✓	✓	✓	✓	✓
Native M-JPEG	✓	✓	✗	✓	✓	✓	✓	✓
JS M-JPEG	✓	✓	✓	✓	✓	✓	✓	✓
JS MPEG-2	✓	✓	✓	✓	✓	✓	✓	✓
JS H.264	✓	✓	✓	✓	✓	✓	✓	✓

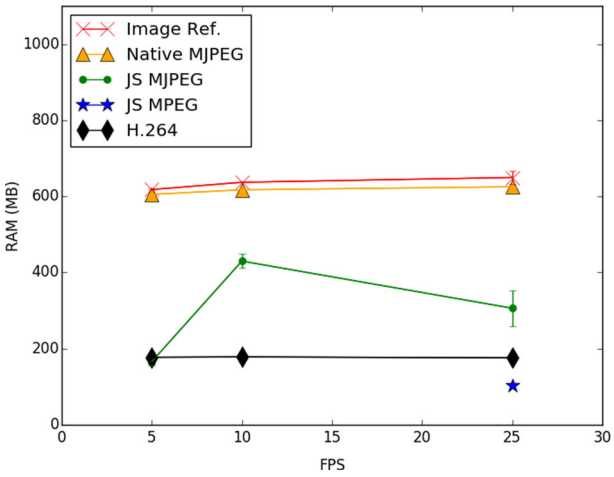

Fig. 7 Client-side RAM usage comparison

and is not within the scope or target of this work, some of the factors involved could potentially be:

– Higher bandwidth-requirements of the lower-compression formats increase the CPU usage as compared to the higher-compression but lower-bandwidth ones, especially in a browser environment where probably the buffers involved are copied several times.
– Image-refreshing is not what the browsers' image component and system was designed for, and modifying an image repeatedly is not necessarily meant to be efficient: it involves cache and DOM operations which are appropriate for single images but not so much for videos.

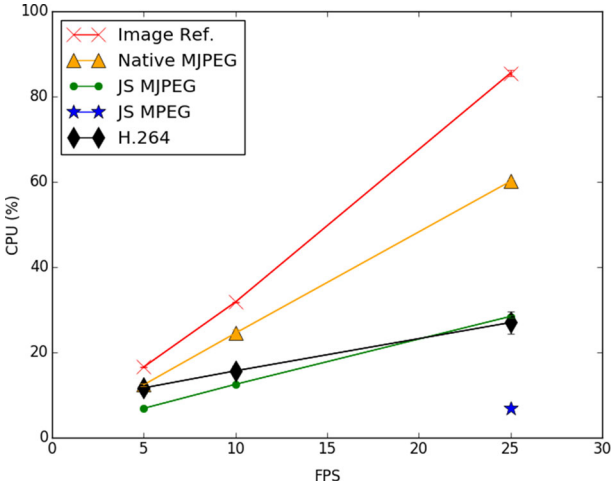

Fig. 8 Client-side CPU usage comparison

– The native M-JPEG implementation of Chrome might be significantly suboptimal, while the JavaScript engine is very heavily optimized, which would explain the difference between the JavaScript-based M-JPEG and the native M-JPEG experiments.
– The H.264 decoder has the advantage of being heavily optimized (compiled into JavaScript through Emscripten, and even using WebGL for some specific tasks such as color conversion) while the MPEG-1 decoder has the advantage of being relatively simple.

Figure 9 compares downstream bandwidth usage. Comparatively, MPEG-1 and H.264 consume a very small amount of bandwidth. Image-refreshing, native M-JPEG and JavaScript-based M-JPEG, which do not rely on interframe compression, consume several times more bandwidth and it increases proportionally to the FPS. JavaScript-based M-JPEG seems to consume slightly more than native, specially at higher framerates, possibly due to *socket.io* or WebSocket overheads.

Figure 10 compares the latency of the different approaches. For the low framerates image-refreshing and JavaScript-based M-JPEG are the fastest. H.264 is the slowest. However, for the higher framerates, the difference among the difference approaches decreases. The change is most apparent for H.264, which goes from an around 1100 ms delay at 5 FPS to a quite low around 500 ms delay at 25 FPS. This non-linear change is probably due to the codec's implementation details. Particularly, internal buffers are probably being filled faster at the higher framerates.

6 Discussion

We have examined and compared some techniques and implementations for web-based near-real-time interactive live streaming. Of cross-platform web-based techniques, image refreshing and native M-JPEG are currently among the most commonly used ones for applications such as remote laboratories. We have also examined and compared some approaches (Java-Script based M-JPEG rendering, JavaScript-based MPEG-1 rendering,

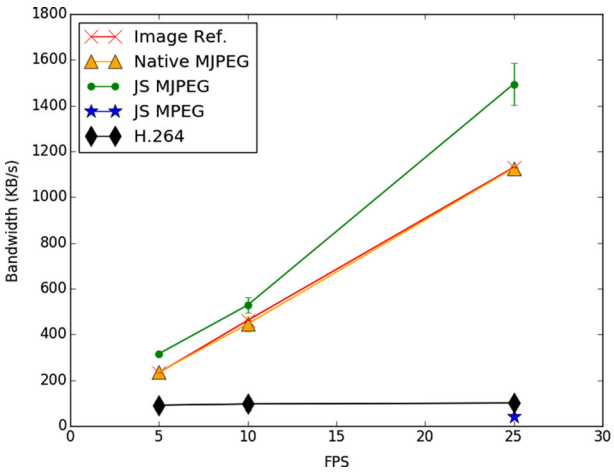

Fig. 9 Client-side downstream bandwidth usage comparison

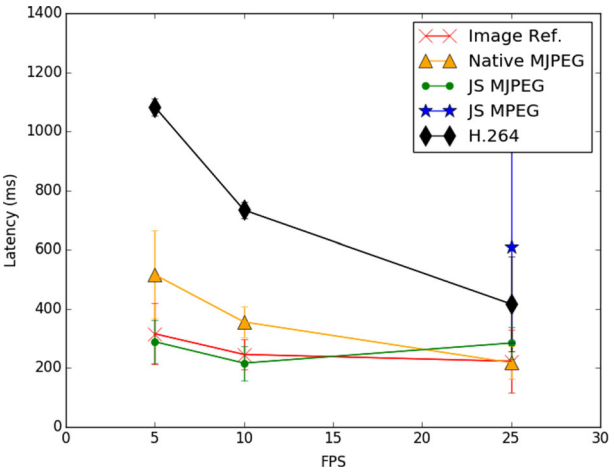

Fig. 10 Latency comparison

and JavaScript-based H.264/AVC rendering) which as far as we know, have not been used for remote laboratories (and possibly not for similar purposes such as IP camera servers).

The formats that some of those approaches rely on (particularly, M-JPEG and MPEG-1) are far from new. However, M-JPEG is still very popular for interactive live streaming because of its simplicity and because of the low capture-display latency that it provides. MPEG-1, similarly, is very mature. Before the appearance of newer formats, it used to be popular for web streaming. Some of those newer formats are H.264 AVC [19], H.265 [18], VP8 [2] and VP9 [11]. They provide a significantly better quality and compression rate, but they are more complex. They generally have a higher processing cost. Of those newer formats, we have considered an approach relying on JavaScript and H.264.

The results of the experiments show that no single approach is necessarily the best for all cases. They show, however, that some approaches can probably be significantly more advantageous than others for certain purposes.

One of the first conclusions that we learn from the experiments is that relying on the native M-JPEG scheme does not seem to be convenient in any case. This is remarkable because, currently, many remote laboratories and other applications, such as IP camera servers, rely on it. However, the only advantage seems to be a slightly lower bandwidth usage than JavaScript-based M-JPEG. In exchange:

- A major browser lacks support (Internet Explorer)
- Other browsers have several reported bugs and a relatively poor track-record at supporting it
- It consumes significantly more RAM
- It consumes significantly more CPU power

Additionally, other significant observation is that the performance of the most simplistic approach —image refreshing— is as good as that of native M-JPEG in RAM and bandwidth usage, and is not far in CPU power. Considering the aforementioned issues that come with native M-JPEG, and the significant advantages of image refreshing such as simplicity and trivial error-recovery and bandwidth-adaptation capabilities, we conclude again that in most cases, there would be no advantage in relying on native M-JPEG.

Other significant observation is that JavaScript-based MPEG-1 and JavaScript-based H.264 seem to be significantly advantageous in many of the evaluated variables. This is remarkable because, as far as we know, they have not been used in the context of remote laboratories. They require much less RAM, extremely lower amounts of bandwidth than the other approaches, and also less CPU power —especially in the case of MPEG-1—. The lower bandwidth requirement is expected due to their interframe compression, and the fact that a fixed bitrate was used. However, it is still noteworthy that it can be achieved while maintaining a (subjectively) good image quality. The lower CPU requirement is less obvious (some possible explanations are suggested in Section 5). These results suggest that, for some remote laboratories, particularly those that require a high FPS and which are particularly bandwidth-constrained (for example, because they are intended to be used on mobile devices, or because they feature many simultaneous webcams) relying on MPEG-1 and H.264 could be a very effective choice.

These formats, however, do have some disadvantages. First, the image quality —with the constant-bitrate, low-latency configuration that was chosen— is not bad, but is worse than for the other approaches. Second, they do indeed raise the latency. Nonetheless, by choosing the appropriate low-latency configuration, most measurements remain lower than 1 second. This is still acceptable for many interactive live streaming applications. However, it is certainly higher than with other more *traditional* approaches such as image refreshing or M-JPEG. Thus, despite the mostly superior results, they are not necessarily the best choice for all applications. It is also noteworthy that they have some additional disadvantages:

- The implementation and deployment is more complex. It requires a transcoding server, and a specific JavaScript-based player in the client.
- Deployment is more complex due to the transcoding server that is required. Server-side, more processing resources are required.
- Error recovery and bandwidth adaptation is harder than with other approaches such as image-refreshing (for which it is trivial).

7 Conclusions and future work

In this work we have described two of the most common approaches that are nowadays used for web-based near-real-time interactive live streaming (image refreshing and native-M-JPEG). Additionally, we have proposed and compared three additional approaches (JavaScript-based M-JPEG, JavaScript-based MPEG-1 and JavaScript-based H.264). Those last, to our knowledge, are not currently used in this context.

The results suggest that, in many cases, avoiding the native M-JPEG scheme in favour of one of the other four would be advisable. It seems to provide very little benefit over the alternatives. They also suggest that the three JavaScript-based alternative approaches (JavaScript-based M-JPEG, JavaScript-based MPEG-1 and JavaScript-based H.264/AVC) provide comparatively good performance.

The schemes based on MPEG-1 and H.264/AVC could be the most appropriate for certain types of interactive applications. Particularly, for those, such as certain remote laboratories, which can withstand a relatively high latency (around 1 second) but which require a low bandwidth usage at a high FPS. Nonetheless, the image-refreshing scheme, despite its simplicity, could still be the most appropriate scheme for those interactive applications which require a very low latency, especially at lower FPS rates.

In the future, it would be interesting to compare new approaches. MPEG-DASH is a promising HTML5-related standard which will likely provide near-real-time live-streaming. Once the support for it is wider, it would be useful to evaluate whether a low capture-display delay can be achieved, and how its performance compares against the alternatives.

Additionally, it would be interesting to evaluate more modern interframe compression formats, and with more configurations. Although the most modern formats (such as VP-9) are expected to require a much higher amount of resources, especially if decoding through JavaScript, a format such as VP-8 could still give interesting results. Especially, if the implementation was heavily optimized and relied on *asm.js* or a similar scheme.

Acknowledgments This work has received financial support by the Department of Education, Language policy and Culture of the Basque Government through a Predoctoral Scholarship granted to Luis Rodriguez-Gil.

References

1. Akhshabi S, Begen AC, Dovrolis C (2011) An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP. In: Proceedings of the second annual ACM conference on multimedia systems. ACM, pp 157–168
2. Bankoski J, Wilkins P, Xu Y (2011) Vp8 data format and decoding guide. Tech. rep., available: http://tools.ietf.org/html/rfc6386 (accessed: 2016-07-07)
3. Claypool M, Finkel D (2014) The effects of latency on player performance in cloud-based games. In: 2014 13th annual workshop on Network and systems support for games (netgames). IEEE, pp 1–6
4. Cozzolino A, Flammini F, Galli V, Lamberti M, Poggi G, Pragliola C (2012) Evaluating the effects of mjpeg compression on motion tracking in metro railway surveillance. In: Advanced concepts for intelligent vision systems. Springer, pp 142–154
5. De Jong T, Linn MC, Zacharia ZC (2013) Physical and virtual laboratories in science and engineering education. Science 340(6130):305–308
6. Deshpande H, Bawa M, Garcia-Molina H (2001) Streaming live media over a peer-to-peer network. Tech. rep
7. Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, Berners-Lee T (2006) Hypertext transfer protocol–HTTP/1.1, 1999. RFC2616
8. Garcia-Zubia J, López-de-Ipiña D, Orduña P (2008) Mobile devices and remote labs in engineering education. In: 2008 eighth IEEE international conference on advanced learning technologies. IEEE, pp 620–622
9. García-Zubia J, Orduña P, López-de-Ipiña D, Alves GR (2009) Addressing software impact in the design of remote laboratories. IEEE Trans Ind Electron 56(12):4757–4767
10. Golparvar-Fard M, Bohn J, Teizer J, Savarese S, Peña-Mora F (2011) Evaluation of image-based modeling and laser scanning accuracy for emerging automated performance monitoring techniques. Autom Constr 20(8):1143–1155
11. Grange A, Rivaz P, Hunt J (2016) Draft vp9 bitstream and decoding process specification. Tech. rep., available: http://www.webmproject.org/vp9/ (accessed: 2016-07-07)
12. Grois D, Marpe D, Mulayoff A, Itzhaky B, Hadar O (2013) Performance comparison of H.265/mpeg-hevc, vp9, and H.265/MPEG-AVC encoders. In: Picture coding symposium (PCS), 2013. IEEE, pp 394–397
13. Harward VJ, Del Alamo JA, Lerman SR, Bailey PH, Carpenter J, DeLong K, Felknor C, Hardison J, Harrison B, Jabbour I et al (2008) The ilab shared architecture: a web services infrastructure to build communities of internet accessible laboratories. Proc IEEE 96(6):931–950
14. Hashemian R, Riddley J (2007) Fpga E-lab, a technique to remote access a laboratory to design and test. In: Microelectronic systems education, 2007. IEEE International Conference on MSE'07. IEEE, pp 139–140

15. Hei X, Liang C, Liang J, Liu Y, Ross KW (2007) A measurement study of a large-scale P2P IPTV system. IEEE Trans Multimed 9(8):1672–1687
16. Html5 specification (2016) Tech. rep., W3, available: https://www.w3.org/TR/html5
17. ISO/IEC (2015) Iso/iec 11172:1993. coding of moving pictures and associated audio for digital storage media at up to about 1.5 mbit/s. Tech. rep., available: http://www.iso.org/iso/catalogue_detail?csnumber=19180 (Accessed: 2016-07-07)
18. ITU-T (2015) High efficiency video coding. Tech. rep., available: https://www.itu.int/rec/t-REC-h.265-201504-i/en (accessed: 2016-07-07)
19. ITU-T (2016) Advanced video coding for generic audiovisual services. Tech. rep., available: http://www.itu.int/rec/T-REC-H.264-201602-I/en (Accessed: 2016-07-07)
20. Jara CA, Candelas FA, Torres F (2008) Virtual and remote laboratory for robotics e-learning. Comput Aided Chem Eng 25:1193–1198
21. Kaspar M, Parsad NM, Silverstein JC (2010) Cowebviz: interactive collaborative sharing of 3D stereoscopic visualization among browsers with no added software. In: Proceedings of the 1st ACM international health informatics symposium. ACM, pp 809–816
22. Latency in live network video surveillance (2015) Tech. rep., axis communications, available: http://bit.ly/2izYVOb (accessed: 2016-07-07)
23. Kim K, Cho BY, Ro WW (2016) Server side, play buffer based quality control for adaptive media streaming. Multimed Tools Appl 1–19
24. Ma J, Nickerson JV (2006) Hands-on, simulated, and remote laboratories: a comparative literature review. ACM Comput Surv (CSUR) 38(3):7
25. Martinez G, Angulo I, Garcia-Zubia J (2016) Weblabmicroscope: A remote laboratory for experimenting with digital microscope. In: 2016 13th international conference on remote engineering and virtual instrumentation (REV). IEEE, pp 159–162
26. Nedic Z, Machotka J, Nafalski A (2003) Remote laboratories versus virtual and real laboratories. In: FIE 2003 33rd annual, vol 1. IEEE
27. Nielsen J (1994) Usability engineering. Elsevier
28. Nishantha D, Hayashida Y, Hayashi T (2004) Application level rate adaptive Motion-JPEG transmission for medical collaboration systems. In: Proceedings of 24th international conference on distributed computing systems workshops, 2004. IEEE, pp 64–69
29. Orduña P, Bailey PH, DeLong K, López-de-Ipiña D, García-zubia J (2014) Towards federated interoperable bridges for sharing educational remote laboratories. Comput Hum Behav 30:389–395
30. Quax P, Liesenborgs J, Barzan A, Croonen M, Lamotte W, Vankeirsbilck B, Dhoedt B, Kimpe T, Pattyn K, McLin M (2016) Remote rendering solutions using web technologies. Multimed Tools Appl 75(8):4383–4410
31. Rodriguez-Gil L, Orduña P, García-Zubia J, Angulo I, López-de-Ipiña D (2014) Graphic technologies for virtual, remote and hybrid laboratories: Weblab-fpga hybrid lab. In: 2014 11th international conference on remote engineering and virtual instrumentation (REV). IEEE, pp 163–166
32. Salkintzis A, Passas N (2005) Emerging wireless multimedia: services and technologies. Wiley
33. Schauer F, Lustig F, Ožvoldová M (2009) Ises-internet school experimental system for computer-based laboratories in physics. Innovations 109–118
34. Schauer F, Krbecek M, Beno P, Gerza M, Palka L, Spilaková P (2014) Remlabnet-open remote laboratory management system for e-experiments. In: 2014 11th international conference on remote engineering and virtual instrumentation (REV). IEEE, pp 268–273
35. Schulzrinne H, Rao A, Lanphier R (1998) Rtsp: real time streaming protocol. IETF RFC2326, april
36. Shea R, Liu J, Ngai ECH, Cui Y (2013) Cloud gaming: architecture and performance. IEEE Netw 27(4):16–21
37. Soares J, Lobo J (2011) A remote fpga laboratory for digital design students. In: 7th portuguese meeting on reconfigurable systems (REC 2011). pp 95–98
38. Sripanidkulchai K, Ganjam A, Maggs B, Zhang H (2004) The feasibility of supporting large-scale live streaming applications with dynamic application end-points. In: ACM SIGCOMM computer communication review, vol. 34. ACM, pp 107–120
39. Tan WL, Lam F, Lau WC (2008) An empirical study on the capacity and performance of 3g networks. IEEE Trans Mob Comput 7(6):737–750
40. Ueberheide M, Klose F, Varisetty T, Fidler M, Magnor M (2015) Web-based interactive free-viewpoint streaming: a framework for high quality interactive free viewpoint navigation. In: Proceedings of the 23rd ACM international conference on multimedia. ACM, pp 1031–1034
41. Van Lancker W, Van Deursen D, Mannens E, Van de Walle R (2012) Implementation strategies for efficient media fragment retrieval. Multimed Tools and Appl 57(2):243–267

42. Vargas H, Farias G, Sanchez J, Dormido S, Esquembre F (2013) Using augmented reality in remote laboratories. Int J Comput Commun Control 8(4):622–634
43. Wang B, Zhang X, Wang G, Zheng H, Zhao BY (2016) Anatomy of a personalized livestreaming system. In: Proceedings of the 2016 ACM on internet measurement conference. ACM, pp 485–498
44. Webgl specification (2014) Tech. rep., khronos webGL working group, available: https://www.khronos.org/registry/webgl/specs/1.0/
45. Wiegand T, Sullivan GJ, Bjontegaard G, Luthra A (2003) Overview of the h. 264/avc video coding standard. IEEE Trans Circuits Syst Video Technol 13(7):560–576
46. Yazidi A, Henao H, Capolino GA, Betin F, Filippetti F (2011) A web-based remote laboratory for monitoring and diagnosis of ac electrical machines. IEEE Trans Ind Electron 58(10):4950–4959
47. Youtube now defaults to HTML5 video (2016) https://youtubeeng.googleblog.com/2015/01/youtube-now-defaults-to-html5_27.html (accessed: 2016-07-07)
48. Zhang C, Liu J (2015) On crowdsourced interactive live streaming: a Twitch.TV-based measurement study. In: Proceedings of the 25th ACM workshop on network and operating systems support for digital audio and video. ACM, pp 55–60

Luis Rodriguez-Gil is a PhD student at DeustoTech Internet group. He finished his studies of a double degree in Computer Eng. and Industrial Org. Eng. in 2013, and he completed a MSc in Information Security in 2014. Since 2009, he has been involved in the WebLab-Deusto Research Group, collaborating in the development of the WebLab-Deusto RLMS. He has published several peer-reviewed publications and contributed to some Open Source projects.

Pablo Orduña is a full time researcher and project manager at the MORElab Research Group at DeustoTech Internet. He finished Computer Engineering in 2007 and his PhD in 2013 in the University of Deusto. During his PhD he was a visiting researcher twice for 6 weeks each, in the MIT CECI in 2011 and UNED DIEEC in 2012. Since 2004, he has also been involved in the WebLabDeusto Research Group, leading the design and development of WebLab-Deusto.

Javier García-Zubia holds a PhD in Computer Sciences by the University of Deusto. He is a full professor in the Faculty of Engineering of the University of Deusto, Spain. His research interest is focused on remote laboratory design, implementation and evaluation. He is the leader of the WebLab-Deusto research group.

Diego López-De-Ipiña is an associate prof. and P.R. of MORElab group and director of DeustoTech Internet unit, and of the PhD program within the Faculty of Eng. of the University of Deusto. He received his PhD from the University of Cambridge in 2002. Responsible for several modules in the BSc and MSc in Comp. Eng. degrees, he is interested in pervasive computing, IoT, semantic service middleware, open linked data and social data mining. He is taking and has taken part in several big consortium-based research european (IES CITIES, MUGGES, SONOPA, CBDP, GO-LAB, LifeWear) and Spanish projects, and has more than 70 publications in relevant int. conf. and journals, including more than 25 JCR-indexed articles.