IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 4, 2017, accepted May 25, 2017, date of publication June 1, 2017, date of current version June 28, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2710328

An Open and Scalable Web-Based Interactive Live-Streaming architecture: The WILSP Platform

LUIS RODRÍGUEZ-GIL[1,2], JAVIER GARCÍA-ZUBIA[2], (Senior Member, IEEE), PABLO ORDUÑA[1,2], (Member, IEEE), AND DIEGO LÓPEZ-DE-IPIÑA[1,2]

[1]Faculty of Engineering, University of Deusto, 48007 Bilbao, Spain
[2]DeustoTech-Deusto Foundation, University of Deusto, 48007 Bilbao, Spain

Corresponding author: Luis Rodriguez-Gil (luis.rodriguezgil@deusto.es)

This work was supported by the Department of Education, Language Policy and Culture of the Basque Government through a Predoctoral Scholarship to Luis Rodriguez-Gil.

ABSTRACT Interactive live-streaming applications and platforms face particular challenges: the actions of the viewer's affect the content of the stream. A minimal capture-render delay is critical. This is the case of applications, such as remote laboratories, which allow students to view specific hardware through a webcam, and interact with it remotely in close to real time. It is also the case of other applications, such as videoconferencing or remote rendering. In the latest years, several commercial live-streaming platforms have appeared. However, the most of them have two significant limitations. First, because they are oriented toward standard live-streaming, their capture-render delay tends to be too high for interactive live-streaming. Second, their architectures and sources are closed. That makes them unsuitable for many research and practical purposes, especially when customization is required. This paper presents the requirements for an interactive live-streaming platform, focusing on remote lab needs as a case study. Then, it proposes an architecture to satisfy those requirements that relies on Redis to achieve high scalability. The architecture is based on open technologies, and has been implemented and published as open source. From a client-side perspective, it is web-based and mobile-friendly. It is intended to be useful for both research and practical purposes. Finally, this paper experimentally evaluates the proposed architecture through its contributed implementation, analyzing its performance and scalability.

INDEX TERMS Webcam, live streaming, live streaming platform, remote laboratories, online learning tools, open.

I. INTRODUCTION

Throughout the last few years many live-streaming platforms have emerged, such as YouTube Live,[1] TwitchTV,[2] Instagram Livestream,[3] and Facebook Live.[4] These platforms tend to be backed by large social media companies and be proprietary. They are designed for scalability and are able to provide live-streaming to a large number of users. However, for some purposes, they have significant limitations. Although they are effective for live-streaming non-interactive content, such as live sports, they tend to be unsuitable for interactive live-streaming. In interactive live-streaming, users react to the stream and affect it in close to real-time. Thus, a minimal capture-render delay is critical. Standard live-streaming platforms, such as the aforementioned ones, tend to have a relatively high delay of at least several seconds. This is by design. It is, among other reasons, because they rely on transcoding, buffering, and heavy interframe compression techniques to maximize scalability and minimize networking issues. The work in [1], for example, outlines the TwitchTV architecture, providing further detail in these aspects. In that case, the broadcast delay of the platform is measured to vary from 12 to 21 seconds. Other limitation is that these major platforms are proprietary, and their architectures and sources are closed. This makes it difficult to rely on them for learning and research purposes. They are also impractical for applications that would need to customize them, or integrate them as middleware.

Interactive live-streaming has particular requirements, limitations, and expectations. Figure 1 characterizes interactive live-streaming among other types of streaming: standard live-streaming and Video on Demand (VoD). There are

[1]https://www.youtube.com/live
[2]https://twitch.tv
[3]https://instagram.com/livestream
[4]https://live.fb.com

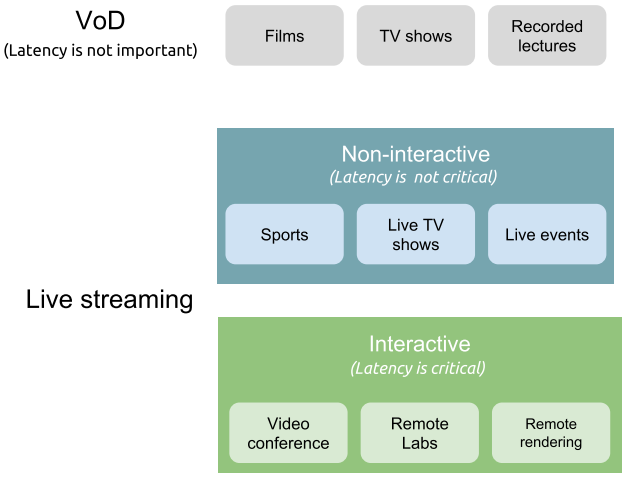

FIGURE 1. Characterization of the different types of streaming and some of its applications. Reproduced from [4].

several aspects that differ, but the most remarkable one is the capture-render delay. The videos in a VoD application are created far in advance. VoD platforms, such as Youtube or Netflix, can use the heaviest compression techniques, and convert the video in advance for different configurations and network conditions. Thus, a high-bandwidth client in a desktop will receive a high-quality and high-bandwidth stream; while a low-bandwidth client in a mobile device will receive a lower-quality but lower-bandwidth stream. They can rely on adaptive streaming to adapt to varying conditions, and they can use buffering to provide high quality in a relatively unstable network, avoiding network jitter [2]. Standard live-streaming platforms have more limitations at this respect. They no longer have nearly-unlimited time to convert the input streams to different formats, nor can they use as large a buffer. Most commercial platforms still allow a significant capture-render delay. This allows them to partially leverage the previously mentioned techniques. However, for interactive live-streaming applications, such as remote labs, collaborative tools, video conferencing applications or remote rendering applications, a low capture-render delay is critical. According to Human Computer Interaction (HCI) research, a second in delay is high. Beyond 0.1 seconds the user can notice a system does not react instantaneously. Beyond a second the user's flow of thought is interrupted [3].

In this context, this work proposes a novel architecture for interactive live-streaming. A platform, named WILSP,[5] has been designed and implemented according to that architecture. It is designed to overcome the aforementioned limitations in existing platforms. Firstly, it is designed for interactive live-streaming, ensuring a low capture-render delay. Secondly, it is open source and relies on open technologies. It can be used for research and practical purposes, customized freely and integrated as middleware. From a technical perspective, it is designed to be a distributed, scalable platform by relying on Redis.[6] It is extensible and supports different video formats and techniques, such as H.264 [5] and image-refreshing, which previous research shows as effective for this purpose [4]. Also, from a client-side perspective, it is fully web-based. This is remarkable because, traditionally, certain features such as multimedia have had more limited support on the Web [6], [7]. This trend has changed and applications no longer need to depend on non-standard plugins [8], such as Adobe Flash[7] or Java Applets.[8] Now they can rely on HTML5 [9] and other related Web standards such as WebGL [10]. Today, major platforms such as Youtube or Netflix[9] rely on HTML5 [11].

The proposed architecture is designed to be generalistic and integrable as middleware into different applications, but its initial and main use-case are remote labs. Remote labs are software and hardware tools that enable remote users to access real, remote equipment through a website [12]–[14]. They can view and interact with that equipment through a webcam. To evaluate the proposed architecture, this work analyzes whether certain requirements are met, focusing particularly on this use-case. Furthermore, a study to measure the performance and scalability of the platform is conducted.

The paper is organized as follows: Section II describes in more detail the purpose and contributions of this research, and the relevant state of the art on interactive live-streaming and remote labs. Section III analyzes the requirements for the proposed platform. Section IV presents an overview of the proposed architecture. Section V describes each layer in more detail. Section VI describes the methodology of the performance and scalability study. Section VII enumerates and explains the conducted experiments. Section VIII describes the results. Section IX discusses those results and potential applications. Section X summarizes the conclusions and proposes future lines of work.

II. MOTIVATION

A. INTERACTIVE LIVE-STREAMING

In an interactive live-streaming system viewers are expected to interact with the stream: they are not simply passive spectators [1], [4], [15]. This is not necessarily the case in a standard live-streaming system. In any live-streaming system there is a capture-render delay: an unavoidable delay between the moment a frame is captured by the source camera, and the moment it is rendered on the viewer's screen. However, the maximum delay that interactive live-streaming systems can allow while still providing an acceptable Quality of Experience for the user is much smaller than for standard live-streaming systems. Although it is not always noticeable for users, most popular standard live-streaming systems nowadays have a significant capture-render delay [16], [17].

[5]The WILSP platform has been released as Open Source and is available at: https://github.com/zstars/wilsp

[6]https://redis.io/
[7]http://www.adobe.com/products/flashplayer.html
[8]http://java.com
[9]https://www.netflix.com

For example, the TwitchTV[10] platform tends to have a higher than 10 seconds delay [1], and the YouTube[11] live-streaming platform seems to commonly have a 20-30 seconds delay in its low-latency configuration.[12]

This delay is purposefully built into the design of their architectures [18]. It allows them to leverage techniques such as buffering, heavy-compression codecs, and video segmentation to maximize scalability and performance. Through these, they can withstand higher network jitter and provide better quality at a lower bandwidth. The high capture-render delay that results is generally not a problem, due to the non-interactive nature of their live-streaming applications. Live sports streaming or live-shows are some common uses of these platforms. The content is indeed being produced while it is broadcast, but users interact very little with the stream, so they can withstand a high capture-render delay. For example, if users are viewing a football match with a 30 seconds delay, their user-experience is unlikely to be affected significantly.

As described above, however, interactive live-streaming applications allow only for a much smaller delay. Some applications of this kind are videoconferencing ones (e.g., Skype,[13] Google Hangouts),[14] surveillance systems (those which require real-time monitoring), remote rendering systems (e.g., [19], [20]), or remote laboratories. Remote labs are in fact the main use-case of the platform that is proposed in this work, and a use-case for this study. They will be described in further detail in later sections.

Currently, the better-known live-streaming platforms are thus not suitable for interactive live-streaming. Most of them are purpose-specific, proprietary, and closed-source, which hinders their use for learning or research purposes. Nonetheless, some proprietary engines which are designed to be used as middleware do exist, such as Wowza.[15] Some live-streaming open source projects can also be found. A remarkable one is the *nginx-rtmp-module*,[16] a module for the Nginx web server. It can live-stream through protocols such as RTMP, HLS or MPEG-DASH.

B. REMOTE LABORATORIES

Remote laboratories allow users to access remote equipment through the Internet [8], [12], [21], [22]. Nowadays, remote laboratories are often educational. Students can access, from anywhere, remote equipment that is located in institutions across the globe. Research has shown that when they are properly designed and implemented they can be as educationally effective as a standard hands-on lab [14], [23], [24].

[10] http://www.twitch.tv
[11] http://www.youtube.com
[12] YouTube provides no official figures or guarantees, but observations and informal tests can be found, such as those at http://blog.ptzoptics.com/youtube-live/low-latency-streaming/ or at the Google product forum (https://productforums.google.com/forum/).
[13] http://www.skype.com
[14] https://hangouts.google.com
[15] https://www.wowza.com
[16] https://github.com/arut/nginx-rtmp-module

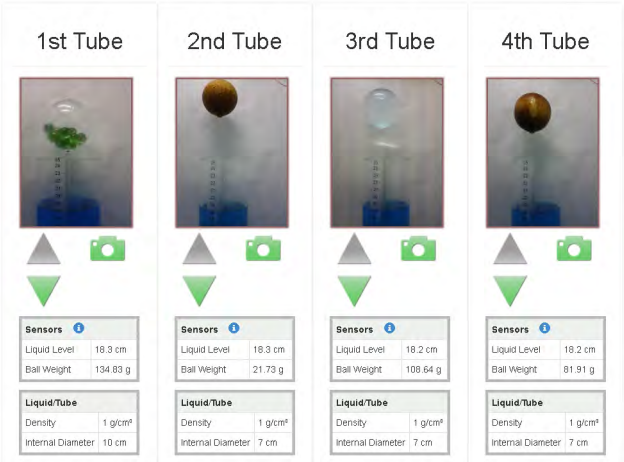

FIGURE 2. Archimedes remote lab. Users experiment with the principle of Archimedes by raising and lowering objects into different liquids and obtaining sensor readings. From http://weblab.deusto.es.

They have many advantages. By relying on remote laboratory technology, institutions that are geographically separated can share expensive equipment. This makes more equipment available for their students, reduces their costs and reduces underusing of equipment [25], [26].

Remote laboratories are formed by multiple hardware and software components [8]. Often they are developed by universities and other institutions as part of research projects. Often they are built on top of Remote Laboratory Management Systems, such as WebLab-Deusto [8] or MIT iLabs [21]. Those systems provide common features, such as authentication, user management, learning analytics, or laboratory federation.

Most remote laboratories feature one or more live-streams of the equipment. In a hands-on laboratory, students see the equipment through their own eyes and interact with it as needed. In a remote laboratory, the interaction needs to be different. Students view the equipment through one or more webcams and interact with it through virtualized controls [27]–[30]. For an interactive remote laboratory an interactive live-stream is required. When lab users conduct an action, such as pressing a button, they expect to see the result immediately.

Figure 2 shows the GUI of the Archimedes remote lab. In it, students can raise and lower objects into different liquids. Sensors provide them with the object weight and liquid height. They can thus verify how the Principle of Archimedes works in reality. As the figure shows, this remote lab includes different simultaneous live-streams. Figure 3 shows a different remote lab. In it, students can create a program using a visual programming language and run it on a real, remote, Arduino-based robot. They have access to several input peripherals (e.g., sensors, buttons) and output ones (LEDs, a serial terminal). In this case, a webcam provides a live-stream as well, and it is also key to the user experience. Users monitor the robot through it and check

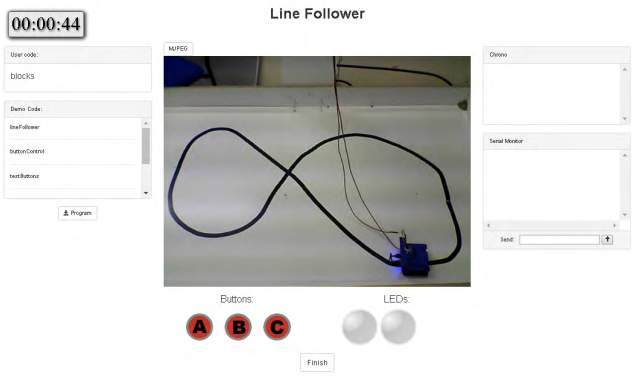

FIGURE 3. Arduino robotics remote lab. Users program their own Arduino robot remotely and are able to interact with it. From http://labsland.com.

whether their program is making the robot behave as they expect. They can interact with the robot in near real-time through the buttons and serial terminal, so the capture-render delay must be low. In this case, because the robot moves relatively fast, the video, ideally, needs to provide a relatively high FPS.

C. CHALLENGE AND CONTRIBUTIONS

As described above, an interactive live-stream is a key feature of most remote laboratories. The stream is the window through which remote students see and interact with the equipment, and is thus particularly important for their user experience. However, traditionally, in remote laboratory research and practice, little attention has been paid to this aspect [4]. The purpose of this work is to analyze the requirements for an interactive live-streaming architecture that is general-purpose but particularly suited for remote laboratories, to design and implement it, and to evaluate it. The platform architecture is designed to be general-purpose but optimal for interactive remote laboratories. It is web-based, based on open technologies, and open source. This is key because the goal is for the platform to be useful for remote laboratory researchers and developers. It is also designed to be distributed and scalable, so that it can manage a large number of input cameras, and so that these streams can be served to a high number of users.

The contributions of this work are the following:

- An analysis of the requirements for an interactive live-streaming platform that is optimized for remote laboratories.
- Architecture for an interactive live-streaming platform that is distributed, based on open technologies, and highly scalable.
- Implementation for the proposed architecture, which is made available as Open Source.
- Design and implementation of two supporting tools for performance evaluation (an IP webcam simulation and a *requester* load-testing script).
- Novel use of the Redis in-memory store engine as a middleware for interactive live-streaming.
- Experimental performance analysis and evaluation of the proposed architecture.
- Conclusions, based on the conducted experiments, on the performance and suitability of such an architecture for remote laboratory research and development.

III. PLATFORM GOALS AND REQUIREMENTS

The main target of the interactive live-streaming platform that is proposed in this work is to satisfy the requirements of remote laboratories, for both research and production contexts. Even though, it is also designed to be general-purpose, and should thus be suitable for additional applications that have similar interactive live-streaming needs.

The main general goals of the platform, which are in line with the needs described in the previous sections, are the following:

- **Universality:** The streams, from a technical perspective, should be available to as many end-user configurations as possible, independently of their platform or device type.
- **Efficiency and scalability:** The architecture should scale horizontally for a large number of camera sources and viewers.
- **Openness:** The architecture should rely on open technologies and be open itself, so that researchers and developers can learn from it, build on it, or use it for research purposes.

The main requirements are the following:

- **Interactive live-streaming:** The platform should, as previously described, be capable of interactive live-streaming. The live-streams it provides should have a small capture-render delay.
- **Supporting multiple input sources:** Being able to handle many video sources (IP cameras) with different stream formats.
- **Supporting multiple output schemes:** Being able to provide different types of stream, from a client-side perspective, depending on the particular needs.

In the following subsections, these goals and requirements will be described in further detail.

A. UNIVERSALITY

The meaning of universality varies across different contexts. In this work, we refer to the goal that, from the client-side perspective, the streams that the platform provides should be as broadly compatible as possible. Universality is particularly important for remote laboratories and similar applications [12], [31].

To promote universality the platform is intended to be fully web-based. Researchers and developers will thus be able to integrate the streams into any web-based application, and users will be able to view them through any web browser. Being web-based is the most significant step towards universality, but there are additional concerns that will be considered. The first of them is that non-standard browser plug-ins should be avoided. Traditionally, some web applications have

relied on technologies such as Java Applets [32] or Adobe Flash [33] to provide certain advanced features such as graphics or non-HTTP networking. For example, [34] describes a remote laboratory which relies on the YawCam[17] Java Applet for streaming its webcam. Using them was once necessary, and the networking access they provide makes protocols such as RTSP [35] available. However, they hinder universality because they are rarely supported in mobile phones and nowadays some popular browsers are even dropping support and discouraging their use on desktops. They can also pose a security risk [36]. For modern applications, generally, standard non-intrusive technologies are preferred. Remote laboratories are often deployed behind the institutional networks of universities. Their IT teams and policies often avoid offering intrusive technologies to students, because their institution could, in fact, be liable were them to be exposed to security issues [8].

Currently, certain technologies that are potentially useful for live streaming (e.g., MPEG-DASH, Media Source Extensions, WebRTC) are being standardized. To promote universality, the interactive live-streaming schemes described in this work rely only on approved standards, and technologies that are widely supported by all browsers. Nonetheless, the proposed architecture is designed to be extensible, and these might be considered in the future.

Finally, it is important to remark that a wide range of different user devices should be supported. This is critical for education: nowadays students from institutions across the world often rely on mobile and tablet devices [37]–[40].

B. EFFICIENCY AND SCALABILITY

In order for the platform to be useful for researchers and developers, it needs to be reasonably efficient and scalable. Institutions that host remote laboratories often host several different ones, each with potentially several live-streams to manage. For example, Figure 2 shows a single laboratory that has had up to 7 simultaneous ones. Other laboratories only require a single stream, such as the robot shown in Figure 3. However, even in that case, the institution often hosts several instances of the same laboratory. For example, in the case of the robot, users upload their own program into it, watch it execute, and interact with it. Only one user can thus have control over it at a given time. To support several simultaneous users, several instances of the laboratory exist. It is also noteworthy that the number of supported viewers per stream should also be scalable, especially for these cases where the laboratory is publicly viewable or collaborative. Currently, Remote Laboratory Management Systems are used to host a large number of different remote laboratories and instances. It is therefore convenient for a single platform to be able to handle all interactive streams.

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

C. OPENNESS

Most popular live-streaming middlewares and platforms are closed-source and proprietary. Nonetheless, they attract significant research attention. Those researchers deduce their architectures from practical experiments, reverse-engineering, and other sources. For example, works can be found on TwitchTV [1], YouTube Live [41], YouNow [42], Meerkat [43] or Periscope [43], [44].

Unfortunately, the closed-source nature of these platforms makes them unsuitable for certain research and practical purposes. As far as we know, for example, no remote laboratory has tried to use any of the above platforms, in a research or production context. Most of the proprietary platforms are not intended to be integrated into other systems, and, as explained in previous sections, do not generally satisfy the low-latency constraints that interactive live-streaming entails.

The goal of the proposed platform is thus to be open, customizable and flexible. This is in line with works such as [45], where it is also stated that the customizability and flexibility that being open-source ensures is particularly critical for academia. Researchers and developers are meant to be able to freely use and modify the platform, adapt it to their needs, and integrate it into their own systems (of which remote labs are a good example).

In line with this, the platform will rely on open technologies, such as Redis, Flask[18] and FFmpeg.[19]

D. INTERACTIVE LIVE-STREAMING

Although it has also been described in earlier sections, it is important to remark that the main requirement of the platform is being able to serve *interactive* live-streams. That is, they should have a low-enough latency, which guarantees that even for interactive applications, the user's Quality of Experience is satisfactory. The actual value for that satisfactory threshold will inevitably vary among users and applications. In cloud-based games, for example, the maximum delay is very low: for each 100 ms of latency, there is a 25% decrease in performance [46]. The threshold for videoconferencing applications is closer to the 300-400ms range [47]. For other contexts where specific research is not available, such as remote labs themselves, general guidelines can be considered. According to Nielsen [3], if a system's response has a delay higher than a second, the user's flow of thought is interrupted, and a 0.1 seconds delay is enough for users to notice.

The proposed architecture will take this into account, and avoid those techniques which tend to increase latency, such as heavy compression and long buffering times.

E. SUPPORTING MULTIPLE INPUT SOURCES

Multiple input sources should be supported, as mentioned in Subsection III-B. The main input sources will be IP webcams.

[18]http://flask.pocoo.org
[19]https://ffmpeg.org

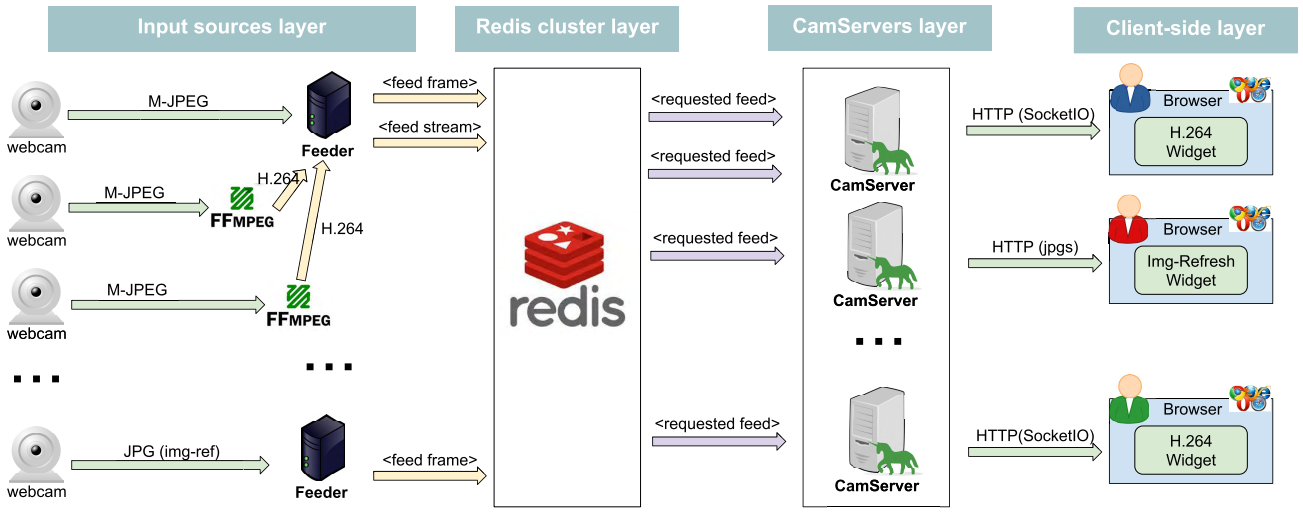

FIGURE 4. Architectural overview of the proposed platform and its components.

Those webcams support different output formats. Common ones are snapshots (which can be turned into a video by simply requesting them periodically fast enough) and M-JPEG, though some modern cameras also support formats such as H.264, and some models even protocols such as RTSP. The platform should be extensible, so that several formats can be supported and new ones added in the future.

F. SUPPORTING MULTIPLE OUTPUT SCHEMES

The output schemes are those that will be used to send the stream to the end user's browser, and actually render it. Traditionally, some applications, including remote laboratories, have dedicated little attention to this aspect [4]. However, it plays a very significant part in promoting universality (being accessible across platforms and devices), a high enough reliability, and, in general, a good enough Quality of Experience.

Previous research suggests that there is currently no single best scheme for all purposes [4]. Several components intervene in a scheme, including the communications protocol, the container format, the video codec and the rendering technology. Many valid combinations may exist. Due to this, the architecture is designed to support various schemes and to be easily extensible.

IV. ARCHITECTURE OVERVIEW

As described in Section III, the proposed architecture is designed to be highly scalable, both in respect to the number of input sources and to the number of end-users. In order to achieve close-to-horizontal scalability, the architecture has several layers. A standard Redis server is at the core of the architecture. Redis is a popular Open Source in-memory data store and message broker, which has successfully been used in many well-known projects to provide scalability.

Figure 4 shows an overview of the proposed architecture and its main components. The architecture is divided into several layers which are highly decoupled, in order to promote scalability and to better separate the concerns of each layer. Particular deployments may have any number of webcams, Feeders, CamServers and clients. Only a fixed number are represented in the picture.

It can be observed that the data flows from left to right. Firstly, the IP webcams, to the left, capture the initial stream. The architecture is designed to scale to an arbitrary number of source webcams and to support different input formats. M-JPEG and continuous snapshots are the most common ones, and the ones the contributed implementation provides. Note that the ones in the figure have been chosen arbitrarily: all Feeders support all formats. The one to be used will depend on the particular IP camera. Secondly, as the figure shows, the streams are directed to Feeder servers. In some cases, the streams are transcoded through FFmpeg, depending on whether the particular input and output format combinations require transcoding or not. The Feeder servers are also designed to scale horizontally, and they forward the streams into the Redis cluster layer.

The Redis layer contains a standard Redis cluster (or a single Redis instance, depending on the scalability needs), and acts as a central decoupling element. Redis is a well-known Open Source middleware designed for in-memory storage, message-brokering, and scalability. Its goal here is to receive the streams, store them in-memory very briefly and efficiently, and distribute them as requested by the CamServers.

The CamServers, which are also designed to be horizontally scalable, retrieve, from Redis, the streams that users are currently requesting. They support different output schemes. When a client requests a stream, they serve it in the expected one. The contributed implementation and the experiments conducted in this work focus on two: image-refreshing

and H.264. Previous research suggests that they are effective and have different advantages [4].

The client-side layer is fully web-based and is formed by different client-side technologies (mostly JavaScript-based) that are served by the CamServers themselves. It provides libraries and widgets that are able to request the streams in different formats and render them.

V. ARCHITECTURE LAYERS

In this section, each of the layers and components that form the architecture, and which were briefly introduced in Section IV, will be described in more detail. See again Figure 4 for a general overview of these layers.

A. INPUT SOURCES LAYER

The purpose of this layer is to encapsulate access to the input sources (webcams), forwarding the stream into the Redis cluster, after having transcoded it into an appropriate format if required.

This scheme has several advantages. Some applications and remote laboratories, for simplicity, access the stream provided by IP cameras directly. Experience shows, however, that this can lead to different significant issues which are not necessarily simple to foresee. Some of them are mentioned in the previous work [4]. From a technical perspective, the software and hardware of these cameras are often limited. While normally they will be able to provide a reasonable Quality of Experience for a single user (or a few), under higher loads their performance tends to be unpredictable. Also, they are heterogeneous. They support different stream formats and specifications, which are not always documented, and which may need to be handled in different ways.

Most IP cameras, for instance, provide an M-JPEG stream. This is a common format because it is simple to implement, requires little encoding, and adds little delay. However, the FPS rate tends to be fixed, either specified in the configuration of the IP webcam, or fixed in their firmware. When users are unable to process the images fast enough, due for example to network jittering or bandwidth constraints, a capture render delay builds up that can add up to many seconds. As a result, applications that rely on directly rendering the stream often become unusable under certain conditions, with no trivial way to notice programmatically or even warn the user. As mentioned, there are also differences between webcams. For example, some provide different qualities and video resolutions, others provide different formats such as an H.264 stream, others use specific non-standardized HTTP headers to transmit timestamps.

The Feeder servers abstract out these idiosyncrasies, providing well-known and reliable streams into the Redis cluster. The contributed implementation, for example, relies mostly on the different M-JPEG streams that the IP webcams provide. No matter how many simultaneous users the interactive live-streaming platform has, a single Feeder accesses a single IP webcam at any time. The IP webcam can thus safely stream reliably a high-quality stream into the Feeder, at a high FPS (generally at 30 FPS). Also each webcam is intended to be located in the same LAN as its Feeder, so there is very little risk of a delay building up, and the chance of webcam-side issues is minimized.

```
ffmpeg -r 30 -f mjpeg -i <src> -flags +low_delay \
  -probesize 32 -c:v libx264 -tune zerolatency \
  -preset:v ultrafast -r 30 -f h264 -s 480x640 \
  -b:v 1500k -g 100 -pix_fmt yuv420p pipe:1
```

FIGURE 5. Example of FFmpeg configuration for transcoding M-JPEG into H.264 with minimal latency.

When transcoding is necessary, the scheme works similarly. A single FFmpeg instance accesses the webcam at any one time. For example, one of the main output schemes that the contributed platform provides is based on H.264 [5]. This is a relatively strong, interframe compression codec. It is not possible to rely on webcams supporting it, and even those that do often do not support it at an interactive live-streaming level. To provide a very low latency it is necessary to configure the H.264 codec appropriately.[20] The FFmpeg instance takes a stream as input from the webcam, in a low-latency format that the webcam supports, such as M-JPEG. Then it transcodes it into the target format (e.g., H.264), sends it to the Feeder, and the Feeder forwards it to the Redis server through its publish-subscribe scheme. See Figure 5 for an example of the FFmpeg configuration that can be used to transcode M-JPEG into H.264 with minimal latency.

The Feeder server, in the contributed implementation, has been created in Python and relies on the Gevent[21] coroutine-based networking engine. It is designed to scale horizontally, supporting any number of instances. Each instance can handle many cameras. The source code is available in the repository.

B. REDIS CLUSTER LAYER

The Redis[22] [48] cluster is a central element of the architecture. It decouples the Feeders (which abstract out access to the input sources and transcoding) from the CamServers (which serve the appropriate stream to the end-users, in whatever format and framerate is required).

Redis is an Open Source in-memory data store. It is used as an in-memory cache and message broker. The proposed architecture makes use of a standard, unmodified Redis cluster (which can, in fact, be a single Redis instance up until a significantly high number of users and cameras, as the experiments in Section VII will show). Redis databases can be replicated through a master-slave model and current versions support a form of sharding [49].

Redis is not specifically targeted towards video streaming. However, it has certain characteristics that make it appropriate for this purpose. Redis is well-known to provide high

[20] Depending on the codec, the processing power, the buffering size, the H.264 mode, and many other configuration details, the capture-render delay, quality and bandwidth usage will vary widely.

[21] http://www.gevent.org/

[22] https://redis.io/

performance and to have been used to provide scalability for many applications. It is well-tested, well-maintained and there is a strong community of developers behind it.[23] Research works suggest that its performance is excellent. For example, [50] compares several NoSQL systems including MongoDB, ElasticSearch and OrientDB, and Redis shows the best performance.

The architecture relies on two different Redis subsystems. In case no transcoding is necessary, which is typically the case when the output format is based on image-refreshing or M-JPEG, it relies on Redis' standard key-value storage features to store individual image frames. The Feeder servers place frames into Redis in a particular Redis key for each active stream. Meanwhile, the CamServers read them as needed. Through this, the Feeders and the CamServers are fully decoupled, and can work at different framerates without issues. Normally the Feeder will work at the maximum framerate for the camera (e.g., 30 FPS). The CamServer can retrieve snapshots from Redis at a slower FPS (e.g., 25 FPS), or even at an adaptive FPS. Some frames will be skipped but no delay is built up. Simplistic as it might seem, it is a very suitable choice for many applications [4], especially since many remote laboratories today rely on those rendering schemes.

When transcoding is necessary, such as when a CamServer requests an H.264 stream, the architecture relies on Redis' message brokering features (Redis channels). FFmpeg transcodes into the Feeder servers, and the Feeder servers publish the output directly into a specific Redis channel for the stream configuration.

As previously described, Redis is not generally geared towards multimedia. However, it fits the proposed platform needs particularly well. In the case of the key-value storage system (for the image-refreshing scheme), the proposed architecture relies on short-lived, non-permanent, readily-replaced frames. These are key features of the Redis store system. Similarly, in the case of the channel-based system (for the H.264 scheme), Redis provides memory-only, short-lived, efficient messages. This is also what the platform requires. Therefore, an interesting contribution of this work is also this novel use of the Redis engine.

C. CamServers LAYER

CamServers obtain the stream data from Redis and serve them to each individual user. They are based on Python's Flask microframework and the Gevent coroutine engine. In order to provide horizontal scalability, an arbitrary number of CamServer instances can be deployed, relying on the WSGI and Gunicorn[24] technologies. Each CamServer instance can serve a number of clients. The architecture supports several types of schemes for this last streaming layer. The main ones are a simple image-refreshing scheme and an H.264 streaming scheme. Previous research suggests that those schemes are effective [4].

When a client is requesting image-refreshing the CamServers simply need to serve discrete frames. This scheme adapts automatically to the client's FPS and bandwidth requirements. The CamServer retrieves each frame directly from Redis, from the appropriate Redis key for the stream. That key's value, at any given moment, will be the latest frame pushed into Redis by the Feeder.When a client is requesting H.264 streaming the CamServer registers itself to listen to the appropriate Redis channel. Then, it forwards that channel's stream data to the client through SocketIO.[25] The architecture can be extended easily to support additional formats based on these schemes, and some additional ones are in fact supported by the contributed platform implementation.

The CamServer component, in the contributed implementation, has also been implemented in Python using Gevent. Each individual CamServer component instance can thus handle several clients, but, additionally, it relies on Gunicorn and WSGI to provide horizontal scalability. Through them, an arbitrary number of CamServer processes on an arbitrary number of hardware servers can be started. The source code is available in the repository.

D. CLIENT-SIDE LAYER

Evaluating potential interactive live-streaming schemes from a client-side perspective, and determining the ones with the most potential for remote laboratories, was the main focus of previous research [4]. As a result, the schemes that are mainly considered in this work are a simple image-refreshing scheme and a more complex H.264 based scheme.

Both the image-refreshing and the H.264 schemes rely only on standard HTML5 and JavaScript. Dependence on non-standard technologies such as Adobe Flash and Java Applets is purposefully avoided. The implementation includes all schemes as widgets, so that they can be easily integrated into other systems. The image-refreshing scheme is the simplest. The widget simply retrieves frames at a high-rate from the CamServer using standard HTTP requests. Though this is not particularly efficient, in relatively modern devices 30 FPS or even higher framerates can be achieved without issues.

The H.264 scheme relies on a customized version of the Broadway[26] H.264 decoder. The core of the decoder is programmed in C but compiled into highly-optimized JavaScript through the EmScripten[27] technology, and can render even more efficiently in WebGL-compatible devices. More details about this scheme and its client-side performance can be found in [4]. The CamServer forwards the stream data to the browser and the decoder through the SocketIO library. SocketIO relies on WebSockets in compatible environments, and falls back to less efficient transports if needed.

[23]The official repository can be found at: https://github.com/antirez/redis. At the moment of writing this, over 8,725 developers have forked the code, and over 23,145 have starred (are following) it.

[24]http://gunicorn.org/

[25]https://socket.io/

[26]https://github.com/mbebenita/Broadway

[27]https://github.com/kripken/emscripten

It is noteworthy that the source code of these widgets is also provided in the repository, in both the JavaScript and the TypeScript languages.

VI. METHODOLOGY

The architecture has been implemented as an Open Source platform. In order to evaluate the architecture, various experiments are conducted on the contributed implementation. The performance is experimentally analyzed through several benchmarks, and the results compared against the previously proposed goals and requirements.

The architecture is formed by several components, which are designed to scale horizontally. A Redis cluster is at the center. The number of Feeders can be increased as the number of camera sources increases, the number of CamServers can be increased as the number of end-users increases. In order to experimentally analyze the performance of the system, the CamServer and the Feeder components are analyzed through separate experiments. That way it is possible to gain insight on what kind of performance we can expect as the number of cameras increases, and what kind of performance we can expect as the number of end-users increases. Feeders and CamServers are decoupled through the Redis cluster. Thus, their performance can mostly be expected to be independent.

It is noteworthy that analyzing the behavior of the platform when a resource limit (e.g., CPU, memory, bandwidth) is reached is out of the scope of this study. Therefore, the ranges (from 1 to 50 cameras, and from 1 to 50 users) have been chosen to avoid this eventuality. It is expected that whenever such a limit is reached the performance of the streams can no longer be guaranteed to satisfy the requirements appropriately. In that case, deployers would be advised to add a new computer to the cluster hosting additional instances of the constrained component. The actual effect of an unsolved constraint would depend on the resource whose limit was reached, on the component that suffers it, and on the stream format. Additional detail on this, especially from a client-side perspective, is described in [4]. The most likely results would be an always-increasing delay, or, in the best-case, a reduced FPS.

The architecture supports several different video schemes. The ones which are most convenient according to previous research, and the ones which are the focus of this work, are image-refreshing and H.264. Because significantly different performance for each of those can be expected, all the experiments are also conducted for those two schemes. Therefore, four different experiments are conducted:

- **Experiment 1**: Feeder component performance in snapshots mode (image-refreshing).
- **Experiment 2**: Feeder component performance in stream mode (H.264).
- **Experiment 3**: CamServer component performance in snapshots mode (image-refreshing).
- **Experiment 4**: CamServer component performance in stream mode (H.264).

A. MATERIALS

All the experiments are conducted on a Gigabit LAN. Thus, the latency of this network is minimal and the bandwidth is much higher than the maximum amount required for the highest load in the experiments. This is critical so that the network does not add a significant amount of latency and so that the bandwidth does not become a bottleneck, affecting the measurements. Three different physical servers are used:

- **Experiment Server**: Intel i7-6700 CPU (3.40GHz, 4 cores, 8 threads), 16 GB RAM.
- **Support Server**: Intel Xeon E5-2630 v3 CPU (2.40GHz, 8 cores, 16 threads), 48 GB RAM.
- **GUI Server**: Intel Core 2 Duo CPU E8400 (3.00GHz, 2 cores), 4 GB RAM.

The Experiment Server, for each experiment, holds only the component being tested. Measurements are taken on it. The Support Server holds the components that are not being tested and also hosts the load simulation components. It is significantly more powerful than the Experiment Server so that, again, it does not act as a bottleneck or affect the results. The GUI server is specifically to support a real graphical browser and measure the capture-render delay.

B. SUPPORTING COMPONENTS

Supporting components have been developed to help benchmark. Firstly, A *FakeWebcam* component simulates an IP camera and provides a looping M-JPEG stream. This way it is possible to reliably simulate an arbitrary number of equal high-performing input sources. It also can embed a QR code with a timestamp into the image, so that the receiver can calculate the capture-render delay. This component has been implemented in Python and supports WSGI through Flask and WSGI. Therefore, an arbitrary number of worker processes can be started, and it can simulate an arbitrary number of cameras. Secondly, a *Requester* component can simulate client loads by requesting and reading a stream through SocketIO, as a real client's browser would do. In this case, it has been implemented in NodeJS. Similarly, several instances can be started, and each instance can simulate several clients.

The clip that the FakeWebcam loops, and that is used for the experiments, shows a remote laboratory. It has a 480x640 resolution and there is a total of 928 frames which loop continuously. Each frame is originally JPEG-compressed and is roughly 13 KB in size.

The source code for the supporting components and for the benchmarking scripts, and the video clip itself, are also available as Open Source.

C. MEASUREMENTS

For the Feeder component experiments the measured variables are CPU usage, the RAM usage, outgoing bandwidth usage and FPS. For the CamServer component experiments the measured variables are CPU usage, RAM usage, incoming bandwidth usage, FPS, and latency.

Measurements are taken on the Experiment Server, which holds the component that is under testing for each

FIGURE 6. Interactive live-stream with an embedded QR timestamp to measure the true capture-render delay.

experiment. Exceptions are the FPS and latency measurements for the CamServer experiments, which are taken in the GUI Server. Every measurement in the Experiment Server is taken 50 times and the average is computed. Every measurement in the GUI Server is taken 30 times and the average is computed.

Due to the nature of an interactive live-streaming system, measuring the capture-render delay is important. This is a challenge because not all streaming formats allow inserting a text-based timestamp into the frames. To work around this limitation, the servers used during the experiments are synchronized through the NTP protocol. A timestamp is calculated, and a QR code containing it is generated and embedded into each frame.[28] Figure 6 shows the live-stream with the embedded QR timestamp. When the frame is rendered, as in the image, the QR code can be read on the target computer. It can be compared to the current time to obtain an accurate capture-render delay. These measurements are taken on the GUI server, because a server with a window system (X Server in this case) is required to open a browser, use the platform's

[28]QR generation and embedding has been measured to take around 23 milliseconds in the Support Server. A small part of the latency measurement will thus be due to this.

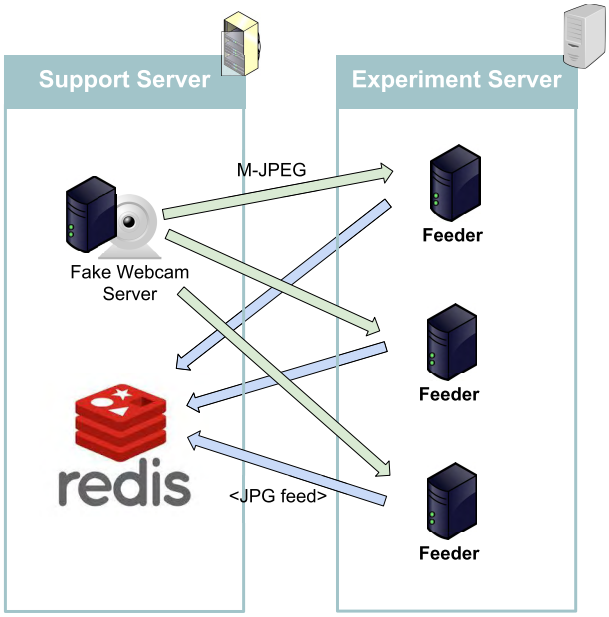

FIGURE 7. Setup and components for Experiment 1: Feeder components and image-refreshing.

JavaScript-based widgets to render the stream, and saving the image with the QR code timestamp.

VII. EXPERIMENTS

A. EXPERIMENT 1

Experiment 1 measures the performance of the Feeder component with the simple image-refreshing scheme. Figure 7 shows this setup. The Support Server hosts a Redis instance and FakeWebcam instances. The Experiment Server hosts the Feeder component. The experiment is run 50 times, with an increasing number of cameras, from 1 to 50. The Feeders read the stream from them and forward the frames into the Redis servers in the image-refreshing format. The number of Feeder instances is increased proportionally to the number of cameras (an instance is added for every 10 cameras). All the measurements are taken in the Experiment Server, which contains the Feeders. As described in Section VI, each measurement is taken 50 times, and the average is computed.

B. EXPERIMENT 2

Experiment 2 measures the performance of the Feeder component with the H.264 format scheme. Figure 8 shows this setup, which is similar to Experiment 1's. In this case, the H.264 format scheme is evaluated instead. The experiment is also run 50 times, with an increasing number of cameras (from 1 to 50). This time, however, transcoding is required, so FFmpeg instances to transcode into H.264 are used. These instances forward the stream into the Feeders and the Feeders into Redis, using the channel-based scheme. All the measurements are taken in the Experiment Server, which contains the Feeders and the FFmpeg instances. As described in Section VI, each measurement is taken 50 times, and the average is computed.

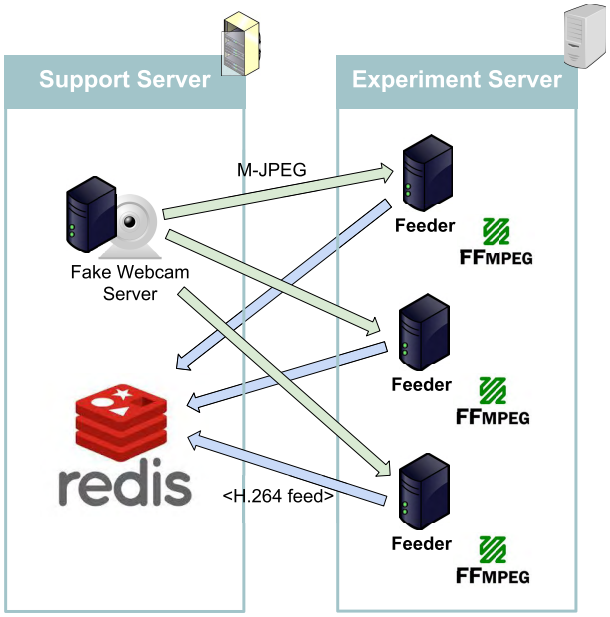

FIGURE 8. Setup and components for Experiment 2: Feeder components and H.264.

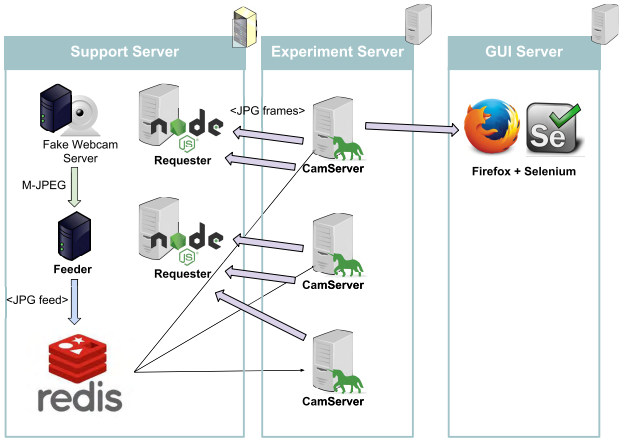

FIGURE 9. Setup and components for Experiment 3: CamServer components and image-refreshing.

C. EXPERIMENT 3

Experiment 3 measures the performance of the CamServer component with the simple image-refreshing scheme. Figure 9 shows this setup. It is significantly more complex than the setup for the previous experiments. The Support Server hosts the Redis instance, a single Feeder component, and a varying number of *Requester* components to simulate end-user load. The Experiment Server hosts several CamServer components (6 Gunicorn-initiated worker processes of Gevent type). The GUI Server hosts a Selenium script[29] that also simulates load for a single client. A script in the GUI servers opens a real browser, renders the stream, and automatically extracts the timestamp from the frame's QR code,

[29]http://www.seleniumhq.org/

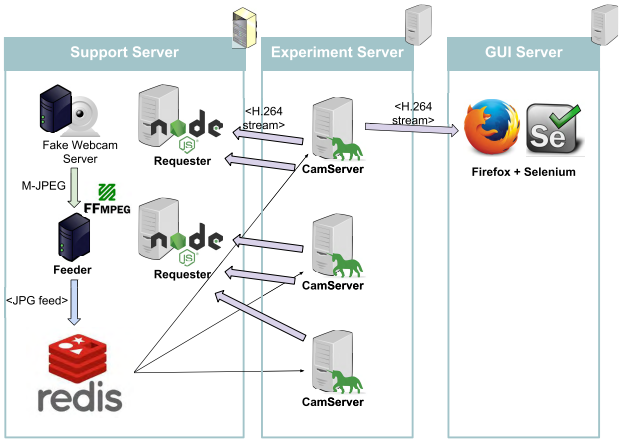

FIGURE 10. Setup and components for Experiment 4: CamServer components and H.264.

measuring the FPS and capture-render delay. The other measurements are taken from the Experiment Server. Note that in this case FPS and latency measurements are particularly realistic, because they are measured in a real browser while rendering the frames. As described in Section VI, each measurement in the Experiment Server is taken 50 times, and the average is computed. Measurements in the GUI Server (FPS and latency) are taken 30 times, and the average is computed as well.

D. EXPERIMENT 4

Experiment 4 measures the performance of the CamServer component with the H.264 format scheme. Figure 10 shows this setup. It is very similar to Experiment 3's, except that in this case the H.264 format scheme is evaluated instead. Transcoding is needed again, so the single Feeder instance is aided by an FFmpeg. Redis channels are used this time to transit the H.264 stream. In this case, the stream is transmitted using SocketIO. It is noteworthy that properly rendering the H.264 stream using the provided widgets requires WebGL, so for this experiment, relying on the GUI Server to open a real browser and measure the capture-render delay is particularly important. It can't easily be calculated without truly rendering the stream. As described in Section VI, each measurement in the Experiment Server is taken 50 times, and the average is computed. Measurements in the GUI Server (FPS and latency) are taken 30 times, and the average is computed as well.

VIII. RESULTS

The results of the experiments are presented in this section. Charts are provided for all of them. The charts combine different variables. Units have been chosen so that they are reasonably proportioned on the vertical axis. CPU is presented in usage percent. RAM is presented in GB. Bandwidth is presented in megabytes per second. Latency (capture-render delay) is presented in centiseconds (the unit is thus 10ms), so that the scales match.

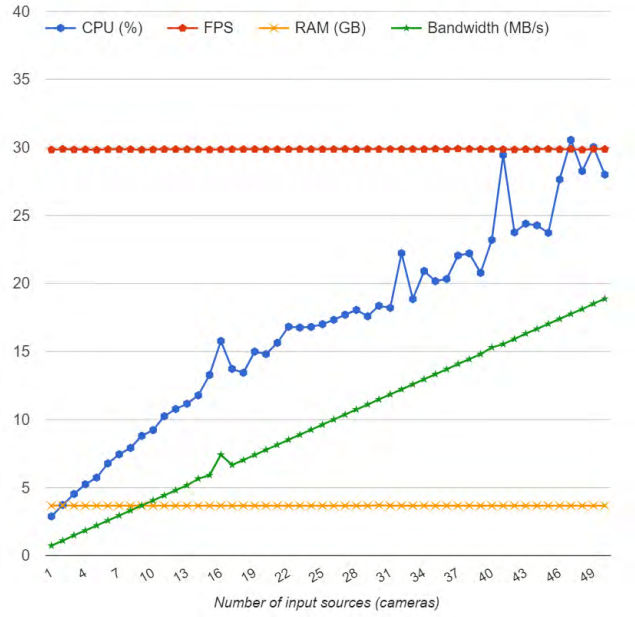

FIGURE 11. Results of Experiment 1. Feeder component using the image-refreshing technique.

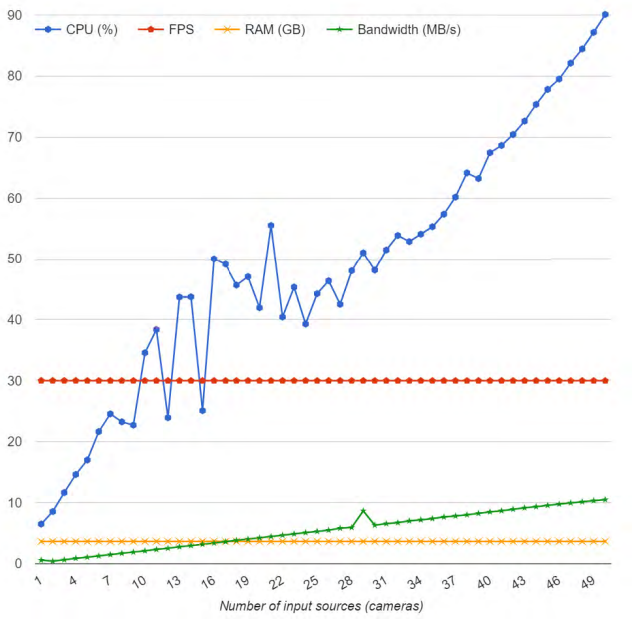

FIGURE 12. Results of Experiment 2. Feeder component using the H.264 format technique.

A. EXPERIMENT 1

The results of Experiment 1 are summarized in Figure 11. The system's RAM usage remains nearly constant. This is expected, because individual frames are small and the architecture is designed to replace old frames with updated ones: no frames are stored. The value itself (around 3.6 GB) is not particularly relevant (most of it is allocated by the system itself and not the platform). The target 30 FPS is maintained without issues. Bandwidth and CPU usage increase linearly with the number of cameras. CPU usage raises up to around 30% at 50 cameras. Bandwidth raises up to 28.8 MB/s at 50 cameras, which is roughly 576 KB/s per camera.

B. EXPERIMENT 2

The results of Experiment 2 are summarized in Figure 12. RAM usage, again, remains nearly constant. The target 30 FPS is maintained throughout. Bandwidth and CPU usage increase linearly with the number of cameras, using up to around 85% CPU at 50 cameras. This is significantly higher than the CPU usage in the previous experiment, which is understandable because in this case all the streams are being transcoded from M-JPEG into H.264. At the same time, nonetheless, the bandwidth requirements are significantly lower. This is expected, as the compression of H.264 is significantly more effective than that of image-refreshing, which is similar to M-JPEG. At 50 cameras it consumes around 10.8 MB/s, which is roughly 216 KB/s per camera.

C. EXPERIMENT 3

The results of Experiment 3 are summarized in Figure 13. RAM usage is mostly constant. The FPS is also mostly constant at 28 FPS. Bandwidth and CPU increase linearly with the number of users, with CPU usage at around 12% for 50 users, and bandwidth usage at around 28.8 MB/s (576 KB/s per stream). In this experiment, the latency (capture-render delay) is measured as well. To do this, as previously described, the stream is rendered in a real Firefox browser and the QR timestamping technique is used to determine the delay. In this case, the delay remains mostly constant. The average is 213 ms ($\mu = 95$), though there are some peaks of up to 574 ms. It is noteworthy that though there might seem to be a significant variation in latency in the chart due to the chosen scale, the absolute value is quite low, always below 0.6 seconds.

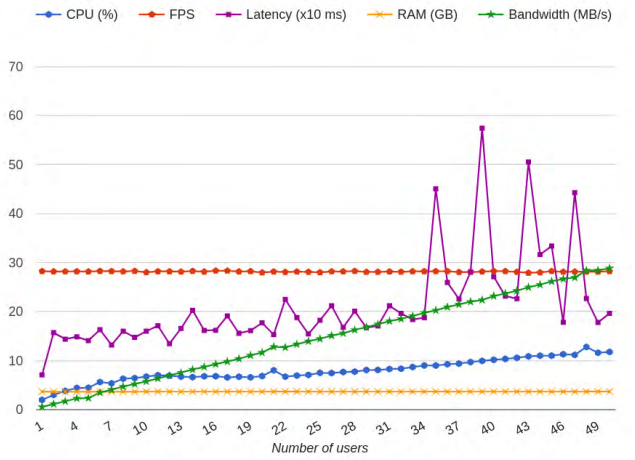

FIGURE 13. Results of Experiment 3. CamServer component using the image-refreshing technique.

D. EXPERIMENT 4

The results of Experiment 4 are summarized in Figure 14. RAM usage is again mostly constant. The target 30 FPS is

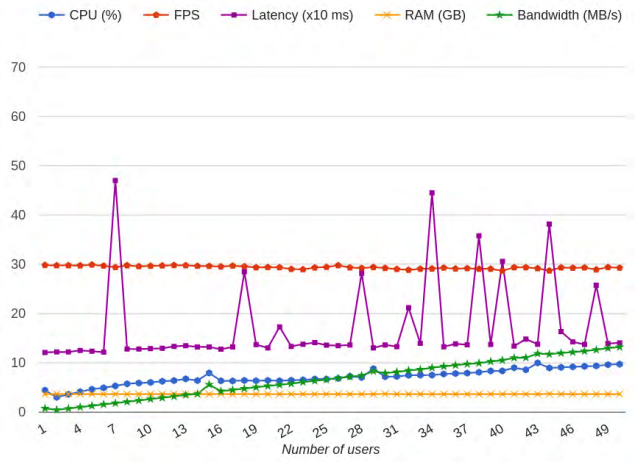

FIGURE 14. Results of Experiment 4. CamServer component using the H.264 format technique.

mostly maintained, sometimes decreasing very slightly to around 29 FPS. Bandwidth and CPU increase linearly with the number of users, with CPU usage at around 10% for 50 users, and bandwidth usage at around 19.9 MB/s (around 398 KB/s per stream). The capture-render delay remains mostly constant as well. The average is 171 ms ($\mu = 85$), though there are some peaks of up to 383 ms. It might again be noteworthy that the absolute value is nonetheless quite low (always below 0.4 seconds in this case).

IX. DISCUSSION

In Section III we described the goals and requirements of the proposed platform. The evaluation, through the contributed implementation and the described experiments, suggests that those goals and requirements are indeed met.

A. GOALS

The goals were the following:

- Universality
- Efficiency and scalability
- Openness

The platform, from a client-side perspective, can indeed be considered to provide high universality. The two main streaming approaches (which are image-refreshing and H.264), described in more detail in [4], are fully web-based. They rely only on JavaScript and HTML5 and are compatible with any modern device, including mobile phones and tablets. The communication protocols that are required are HTTP and optionally WebSockets.

The *efficiency and scalability* of the platform are satisfactory as well. The results suggest that both the Feeders and CamServers are efficient for both image-refreshing and H.264. As expected, there are some performance differences between them. The most significant one is that image-refreshing takes less CPU but takes significantly more bandwidth. This is understandable, because image-refreshing is essentially an M-JPEG variation: there is no interframe compression. No transcoding is needed, but, in exchange, the compression rate is worse than for H.264.

The capture-render delay that the platform provides, for both schemes, is similarly low. Some minor peaks are present but they are within reasonable bounds. It is noteworthy that this H.264 delay is particularly small, considering that it is an interframe-compression format. The codec has been configured to very aggressively minimize latency, reducing buffer sizes to a minimum and sacrificing compression. In exchange, as observed, the bandwidth usage, while not particularly large, is higher than that of a typical H.264 stream.

A single server can handle more than 50 cameras or users (in some cases many more, depending on the format). Moreover, both the Feeders and CamServers scale horizontally. This has not been explicitly tested during the experiments, but they are based on technologies such as WSGI which rely on multiple independent processes. And, indeed, multiple independent processes were used during the experiments.

The *openness* goal is also satisfied. The libraries and technologies that the platform relies on are Open Source. The implementation of the platform itself has also been released as Open Source.

B. REQUIREMENTS

The requirements that were stated were the following:

- Interactive live-streaming
- Supporting multiple sources
- Supporting multiple output schemes

These particular technical requirements are indeed met by the platform. The experiments show that the capture-render delay that the platform can provide is indeed low enough to be considered interactive-level. It supports an arbitrary number of input cameras. They also support multiple output schemes. These include the image-refreshing scheme and the H.264-based schemes that have been discussed throughout this work and evaluated through the experiments. They also include some additional schemes, such as rendering M-JPEG through JavaScript in the client-side. These have not been discussed but are present in the contributed source.

With all these goals and requirements met, we expect that this work will be useful for both research and practical purposes. It is our intention, particularly, that the remote laboratory community may benefit from an easy to deploy, integrable, interactive live-streaming system that improves the user experience of remote laboratories.

X. CONCLUSIONS AND FUTURE WORK

This work has proposed goals and requirements for an open, web-based, interactive live-streaming platform. These goals and requirements are particularly well-suited for the field of remote laboratories, though the platform is intended to be useful for any other applications which have similar requirements. Then, an architecture to meet those goals has been designed. An implementation of that architecture has been

created and released as Open Source.[30] That contributed implementation has been used to experimentally evaluate the architecture. Results suggest that the goals and requirements are indeed met and that the proposed architecture will indeed be useful for practical and research purposes. Most live-streaming platforms, and especially interactive live-streaming ones, are not open. It is expected that the proposed platform and its implementation, due to its open nature, can be integrated, used and customized as needed by other researchers or developers.

These results are promising. Nonetheless, some lines of work remain open and could be pursued in the future. To our knowledge, there are no other interactive live-streaming architectures that are specifically oriented towards remote laboratories. However, some alternative open source platforms, with different goals, do exist. An example is *nginx-rtmp-module*.[31] It would therefore be interesting to compare these architectures in terms of performance and in terms of remote laboratory requirements. It would also be interesting to analyze the architecture's performance under varying network conditions. Furthermore, in the future, new interactive live-streaming schemes may be added to the platform and evaluated. In the latest times, new promising standards such as MPEG-DASH have appeared. Although they are not necessarily suited for interactive live-streaming, as they are developed and become more mainstream they may be explored as additional live-streaming schemes. Also, live-streaming additions to the HTML5 *video* tag are expected to appear in the future. Once such additions are standardized and well-supported by modern browsers, they may be explored as well.

REFERENCES

[1] C. Zhang and J. Liu, "On crowdsourced interactive live streaming: A twitch. TV-based measurement study," in *Proc. 25th ACM Workshop Netw. Oper. Syst. Support Digit. Audio Video*, 2015, pp. 55–60.

[2] D. Nishantha, Y. Hayashida, and T. Hayashi, "Application level rate adaptive motion-JPEG transmission for medical collaboration systems," in *Proc. IEEE 24th Int. Conf. Distrib. Comput. Syst. Workshops*, 2004, pp. 64–69.

[3] J. Nielsen, *Usability Engineering*. Amsterdam, The Netherlands: Elsevier, 1994.

[4] L. Rodriguez-Gil *et al.*, "Interactive live-streaming technologies and approaches for Web-based applications," *Multimedia Tools Appl.*, pp. 1–32, 2017, doi: 10.1007/s11042-017-4556-6.

[5] *Advanced Video Coding for Generic Audiovisual Services*, document ITU-T Rec. H.264 & ISO/IEC 14496-10 AVC, v3:2005, Amendment 3: Scalable Video Coding.

[6] W. Van Lancker, D. Van Deursen, E. Mannens, and R. Van de Walle, "Implementation strategies for efficient media fragment retrieval," *Multimedia Tools Appl.*, vol. 57, no. 2, pp. 243–267, 2012.

[7] L. Rodriguez-Gil, P. Orduña, J. García-Zubia, I. Angulo, and D. López-de-Ipiña, "Graphic technologies for virtual, remote and hybrid laboratories: WebLab-FPGA hybrid lab," in *Proc. IEEE 11th Int. Conf. Remote Eng. Virtual Instrum. (REV)*, Feb. 2014, pp. 163–166.

[8] J. García-Zubia, P. Orduña, D. López-de-Ipiña, and G. R. Alves, "Addressing software impact in the design of remote laboratories," *IEEE Trans. Ind. Electron.*, vol. 56, no. 12, pp. 4757–4767, Dec. 2009.

[9] (Oct. 2014). "HTML5 specification," W3, Tech. Rep. [Online]. Available: https://www.w3.org/TR/html5

[30] Available at: https://github.com/zstars/wilsp

[31] https://github.com/arut/nginx-rtmp-module

[10] (Oct. 2014). "WebGL specification," Khronos WebGL Working Group, Tech. Rep. [Online]. Available: https://www.khronos.org/registry/webgl/specs/1.0/

[11] (2017). *YouTube Now Defaults to HTML5 Video*. [Online]. Available: http://youtubeeng.blogspot.com.es/2015/01/youtubenowdefaultstohtml5_27.html

[12] L. Gomes and S. Bogosyan, "Current trends in remote laboratories," *IEEE Trans. Ind. Electron.*, vol. 56, no. 12, pp. 4744–4756, Dec. 2009.

[13] J. G. Zubía and G. R. Alves, Eds., *Using Remote Labs in Education: Two Little Ducks in Remote Experimentation*, vol. 8. Bilbao, Spain: Universidad de Deusto, 2012.

[14] T. de Jong, M. C. Linn, and Z. C. Zacharia, "Physical and virtual laboratories in science and engineering education," *Science*, vol. 340, no. 6130, pp. 305–308, 2013.

[15] G. Paravati, C. Celozzi, A. Sanna, and F. Lamberti, "A feedback-based control technique for interactive live streaming systems to mobile devices," *IEEE Trans. Consum. Electron.*, vol. 56, no. 1, pp. 190–197, Feb. 2010.

[16] S. Akhshabi, A. C. Begen, and C. Dovrolis, "An experimental evaluation of rate-adaptation algorithms in adaptive streaming over HTTP," in *Proc. 2nd Annu. ACM Conf. Multimedia Syst.*, 2011, pp. 157–168

[17] X. Hei, C. Liang, J. Liang, Y. Liu, and K. W. Ross, "A measurement study of a large-scale P2P IPTV system," *IEEE Trans. Multimedia*, vol. 9, no. 8, pp. 1672–1687, Dec. 2007.

[18] B. Wang, X. Zhang, G. Wang, H. Zheng, and B. Y. Zhao, "Anatomy of a personalized livestreaming system," in *Proc. ACM Internet Meas. Conf.*, 2016, pp. 485–498.

[19] R. Shea, J. Liu, E. C.-H. Ngai, and Y. Cui, "Cloud gaming: Architecture and performance," *IEEE Netw.*, vol. 27, no. 4, pp. 16–21, Jul./Aug. 2013.

[20] M. Ueberheide, F. Klose, T. Varisetty, M. Fidler, and M. Magnor, "Web-based interactive free-viewpoint streaming: A framework for high quality interactive free viewpoint navigation," in *Proc. 23rd ACM Int. Conf. Multimedia*, 2015, pp. 1031–1034.

[21] V. J. Harward *et al.*, "The iLab shared architecture: A Web services infrastructure to build communities of Internet accessible laboratories," *Proc. IEEE*, vol. 96, no. 6, pp. 931–950, Jun. 2008.

[22] J. Ma and J. V. Nickerson, "Hands-on, simulated, and remote laboratories: A comparative literature review," *ACM Comput. Surv.*, vol. 38, no. 3, p. 7, 2006.

[23] J. J. Rodriguez-Andina, L. Gomes, and S. Bogosyan, "Current trends in industrial electronics education," *IEEE Trans. Ind. Electron.*, vol. 57, no. 10, pp. 3245–3252, Oct. 2010.

[24] J. R. Brinson, "Learning outcome achievement in non-traditional (virtual and remote) versus traditional (hands-on) laboratories: A review of the empirical research," *Comput. Edu.*, vol. 87, pp. 218–237, Sep. 2015.

[25] Z. Nedic, J. Machotka, and A. Nafalski, "Remote laboratories versus virtual and real laboratories," in *Proc. IEEE 33rd Annu. FIE*, vol. 1. Nov. 2003, pp. T3E-1–T3E-6.

[26] P. Orduña, P. H. Bailey, K. DeLong, D. López-de-Ipiña, and J. García-Zubia, "Towards federated interoperable bridges for sharing educational remote laboratories," *Comput. Human Behavior*, vol. 30, pp. 389–395, Jan. 2014.

[27] R. Hashemian and J. Riddley, "FPGA e-Lab, a technique to remote access a laboratory to design and test," in *Proc. IEEE Int. Conf. Microelectron. Syst. Edu. (MSE)*, Jun. 2007, pp. 139–140.

[28] C. A. Jara, F. A. Candelas, and F. Torres, "Virtual and remote laboratory for robotics e-learning," *Comput. Aided Chem. Eng.*, vol. 25, pp. 1193–1198, 2008.

[29] H. Vargas, G. Farias, J. Sanchez, S. Dormido, and F. Esquembre, "Using augmented reality in remote laboratories," *Int. J. Comput. Commun. Control*, vol. 8, no. 4, pp. 622–634, 2013.

[30] A. Yazidi, H. Henao, G.-A. Capolino, F. Betin, and F. Filippetti, "A Web-based remote laboratory for monitoring and diagnosis of AC electrical machines," *IEEE Trans. Ind. Electron.*, vol. 58, no. 10, pp. 4950–4959, Oct. 2011.

[31] L. Rodriguez-Gil, J. García-Zubia, P. Orduña, and D. López-de-Ipiña, "Towards new multiplatform hybrid online laboratory models," *IEEE Trans. Learn. Technol.*, 2016, doi: 10.1109/TLT.2016.2591953.

[32] (2017). *Java Website*, accessed on Apr. 30, 2017. [Online]. Available: https://www.java.com

[33] (2017). *Adobe Flash Website*, accessed on Apr. 30, 2017. [Online]. Available: http://www.adobe.com/es/products/flashplayer.html

[34] J. Soares and J. Lobo, "A remote FPGA laboratory for digital design students," in *Proc. 7th Portuguese Meeting Reconfigurable Syst. (REC)*, 2011, pp. 95–98.

[35] H. Schulzrinne, A. Rao, and R. Lanphier, *RTSP: Real Time Streaming Protocol*, document IETF RFC2326, Apr. 1998.
[36] McAfee, "McAfee Labs threats report, May 2015," Intel Secur., Santa Clara, CA, USA, Tech. Rep., May 2015.
[37] H. Crompton, D. Burke, K. H. Gregory, and C. Gräbe, "The use of mobile learning in science: A systematic review," *J. Sci. Edu. Technol.*, vol. 25, no. 2, pp. 149–160, 2016.
[38] L. J. Couse and D. W. Chen, "A tablet computer for young children? Exploring its viability for early childhood education," *J. Res. Technol. Edu.*, vol. 43, no. 1, pp. 75–96, 2010.
[39] S. N. Şad and Ö. Göktaş, "Preservice teachers' perceptions about using mobile phones and laptops in education as mobile learning tools," *Brit. J. Edu. Technol.*, vol. 45, no. 4, pp. 606–618, 2014.
[40] D. G. de la Iglesia, J. F. Calderón, D. Weyns, M. Milrad, and M. Nussbaum, "A self-adaptive multi-agent system approach for collaborative mobile learning," *IEEE Trans. Learn. Technol.*, vol. 8, no. 2, pp. 158–172, Apr./Jun. 2015.
[41] K. Pires and G. Simon, "YouTube live and Twitch: A tour of user-generated live streaming systems," in *Proc. ACM 6th ACM Multimedia Syst. Conf.*, 2015, pp. 225–230.
[42] D. Stohr, T. Li, S. Wilk, S. Santini, and W. Effelsberg, "An analysis of the YouNow live streaming platform," in *Proc. IEEE 40th Local Comput. Netw. Conf. Workshops (LCN Workshops)*, Oct. 2015, pp. 673–679.
[43] J. C. Tang, G. Venolia, and K. M. Inkpen, "Meerkat and periscope: I stream, you stream, apps stream for live streams," in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2016, pp. 4770–4780.
[44] M. Siekkinen, E. Masala, and T. Kämäräinen, "Anatomy of a mobile live streaming service: The case of Periscope," in *Proc. IMC*, 2016, pp. 1–8.
[45] C. Ceglie, G. Piro, D. Striccoli, and P. Camarda, "*3DStreaming*: An open-source flexible framework for real-time 3D streaming services," *Multimedia Tools Appl.*, vol. 75, no. 8, pp. 4411–4440, 2016.
[46] M. Claypool and D. Finkel, "The effects of latency on player performance in cloud-based games," in *Proc. IEEE 13th Annu. Workshop Netw. Syst. Support Games (NetGames)*, Dec. 2014, pp. 1–6.
[47] A. Salkintzis and N. Passas, Eds., *Emerging Wireless Multimedia: Services and Technologies*. Hoboken, NJ, USA: Wiley, 2005.
[48] J. L. Carlson, *Redis in Action*. Greenwich, CT, USA: Manning Publications Co., 2013.
[49] (2017). *Redis Clustering Tutorial*, accessed on Apr. 28, 2017. [Online]. Available: https://redis.io/topics/cluster-tutorial
[50] Y. Abubakar, T. S. Adeyi, and I. G. Auta, "Performance evaluation of NoSQL systems using YCSB in a resource austere environment," *Int. J. Appl. Inf. Syst.*, vol. 7, no. 8, pp. 23–27, 2014.

LUIS RODRÍGUEZ-GIL received the double degree in computer engineering and industrial organization engineering in 2013, and the M.Sc. degree in information security in 2014. He is currently pursuing the Ph.D. degree with the DeustoTech Internet Group, University of Deusto. Since 2009, he has been with the WebLab-Deusto Research Group, collaborating in the development of the WebLab-Deusto RLMS. He has authored a number of peer-reviewed publications and contributed to several Open Source projects.

JAVIER GARCÍA-ZUBIA (M'08–SM'11) received the Ph.D. degree in computer science from the University of Deusto, Spain. He is currently a Full Professor with the Faculty of Engineering, University of Deusto. He is the Leader of the WebLab-Deusto Research Group. His research interests include remote laboratory design, implementation and evaluation, and embedded systems design. He is the President of the Spanish Chapter of the IEEE Education Society.

PABLO ORDUÑA (M'05) received the degree in computer engineering in 2007, and the Ph.D. degree from the University of Deusto in 2013. He was a Visiting Researcher twice for six weeks each, with the MIT CECI in 2011 and UNED DIEEC in 2012. He has also attended two programs for entrepreneurship training with Singularity University: Global Solutions Program and Launchpad. Since 2004, he has been also with the WebLab-Deusto Research Group, University of Deusto, leading the design and development of WebLab-Deusto, and later a Researcher and the Project Manager with MORElab (DeustoTech Internet) until 2017. He is currently the CTO with LabsLand, spin-off of the WebLab-Deusto project, and an External Collaborator with DeustoTech.

DIEGO LÓPEZ-DE-IPIÑA received the Ph.D. degree from the University of Cambridge in 2002. He is currently an Associate Professor and Principal Researcher of the MORElab Group, and the Director of the DeustoTech Internet Unit, and the Ph.D. program within the Faculty of Engineering, University of Deusto. Responsible for several modules in the B.Sc. and M.Sc. degrees in computer engineering. He has over 70 publications in relevant International conferences and journals, including over 25 JCR-indexed articles. He is interested in pervasive computing, Internet of Things, semantic service middleware, open linked data, and social data mining. He is taking and has taken part in several big consortium-based research European, including IES CITIES, MUGGES, SONOPA, CBDP, GO-LAB, and LifeWear, and Spanish projects.