Garcia, B., Lopez-Fernandez, L., Gallego, M. & Gortazar, F. (2017). Kurento: The Swiss Army Knife of WebRTC Media Servers. *IEEE Communications Standards Magazine*, 1(2), pp. 44–51.

DOI: 10.1109/mcomstd.2017.1700006

Kurento: the Swiss Army Knife of WebRTC Media Servers

Boni García, Luis López-Fernández, Micael Gallego, Francisco Gortázar
Universidad Rey Juan Carlos
{boni.garcia, luis.lopez, micael.gallego, francisco.gortazar}@urjc.es

Abstract

In this article we introduce Kurento, an open source WebRTC media server and a set of client APIs aimed to simplify the development of applications with rich media capabilities for the Web and smartphone platforms. Kurento features include group communications, transcoding, recording, mixing, broadcasting and routing of audiovisual flows, but also provides advanced media processing capabilities such as computer vision and augmented reality. It is based on a modular architecture, which makes it possible for developers to extend and customize its native capabilities with third party media processing algorithms. Thanks to all this, Kurento can be a powerful tool for Web developers who may find natural programming with its Java and JavaScript APIs following the traditional three-tiered Web development model.

Introduction

WebRTC is the umbrella term for a number of emerging technologies and APIs having the ambition of bringing Real-Time Communications (RTC) to the Web. WebRTC standardization bodies are investing huge efforts on addressing most of the problems of creating client-side WebRTC implementations for the Web [1]. Thanks to this, both Chrome, Firefox, and Opera browsers support it. This means that, at the time of this writing, around 2 billion devices enjoy built-in WebRTC capabilities. This number is expected to grow during the next few years as Edge and Safari browsers are creating their WebRTC implementations.

Originally, most WebRTC applications were based on peer-to-peer communication models. However, the use of WebRTC media servers and infrastructures for making richer applications is becoming common practice. There is not a formal definition of what a media server is and different authors use the term with different meanings. In this article, we understand that a media server is just the server side of a client-server architected media system. In the same way database servers provide data persistence capabilities to Web applications, media servers provide multimedia capabilities to media-enabled applications. Following this definition, media server technologies emerged in the 90's catalyzed by the popularization of digital video services. Most of these early media servers were specifically conceived for providing multimedia content distribution and transport in two main flavors: streaming media servers and RTC media servers.

Streaming media servers [2] provide media distribution capabilities through transport protocols designed for reliability that tend to relax latency requirements. This means, among other things, that these protocols typically recover from packet loss in the network through iterative retransmissions. The QoS provided through them is convenient for one-way applications such as VoD (Video on Demand) or live streaming, where the media information travels from sources to destinations but there is no real-time feedback communication.

RTC media servers [3], in turn, are designed for bidirectional communications. Due to this, the transport protocols they implement are designed for low latency and packet retransmissions are not always useful

for recovering form losses. In these services, the full duplex media information exchange is used to provide conversational interactions among users. Due to this, this type of media servers is typical in audio and video conferencing systems. Most available RTC media servers are designed and optimized for providing one of the following capabilities:

- Group communication capabilities: These refer to the ability of enabling groups of users to communicate synchronously. Most group videoconferencing services require them.
- Media archiving capabilities: These are related to the ability of recording multimedia streams into media repositories and of recovering them later for visualization. Services requiring communication auditability or persistence use them.
- Media bridging: This refers to providing media interoperability among different network domains having incompatible media formats or protocols. These are used, for example in WebRTC gateways, which interconnect WebRTC browsers with and legacy VoIP systems.

Following this, in this article we concentrate our attention on RTC media servers introducing Kurento, an open source WebRTC media server and a set of client APIs making simple the development of advanced video applications for Web and smartphone platforms. Kurento is the Esperanto term for the English word 'stream'. We chose this name because we believe the Esperanto principles are inspiring for what the multimedia community needs: openness and universality. Kurento modular architecture contributes to the state-of-the-art by following a holistic approach, meaning that it has been designed for providing, in an integrated way, all types of server-side media capabilities. This includes the three-specified above but also further ones, such as augmented reality, media content analysis or arbitrary media processing; which enable novel use cases for rich person-to-person, person-to-machine and machine-to-machine communications.

The structure of this article is as follows. In the next section, we present an overview of the background of this work. Next section introduces the main features of Kurento. Then, a case study aimed to evaluate the scalability of Kurento is depicted. Finally, we summarize the main conclusions of this piece of research.

Related work

Modularity in media server architectures

In software development, the concept of module is typically used for referring to a specific functional block that is hidden behind an interface. However, beyond the intuitive idea of a module, we are interested in the stronger concept of modularity.

From a developer's perspective, an RTC media server is just a platform (i.e. a technology providing capabilities to a system designer for programming application on top.) When bringing the concept of modularity to platforms, the main idea is to split systems into smaller parts (i.e. modules) so that there is strong cohesion within modules and loose coupling among them. Moreover, for having full modularity they should comply with the following additional properties [4]:

- **Isolation**: when modules operate together in a system, the internal status of a module should not directly affect the status of others.
- **Abstraction**: the internal state of a module should be hidden for both the system designer and for other modules. In software, abstraction is typically achieved through interfaces that limit how the module interact with the external world through a limited set of primitives.

- **Composability**: those abstract interfaces should enable components to recombine and to reassemble in various combinations to satisfy specific users requirements. In other words, modules should behave as building blocks and the system designer's role is to create the appropriate interconnection topology among them to provide the desired logic.
- **Reusability**: if a module provides a specific capability, any system requiring such capability might reuse the module without the need of re-implementing again the capability.
- **Extensibility**: the platform should provide seamless mechanisms for creating and plugging additional modules extending the exposed capabilities.

Modularity brings relevant benefits for developers including cost reduction, shorter learning times, higher flexibility, augmentation (i.e. adding a new solution by merely plugging a new module), etc. On the other hand, a downside to modularity is that low quality modular systems are not optimized for performance. This is usually due to the cost of putting up interfaces between modules.

In this context, and to the best of our knowledge, there is a single scientific reference dealing with modular RTC media servers: the Janus WebRTC Gateway [5]. Being Janus indeed modular, it does not comply with all modularity requirements, in particular in what refers to composability, that is not supported at all (i.e. Janus modules provide specific application logic and cannot be assembled among each other) and reusability, which is only partially supported (i.e. in Janus, different modules need to provide similar capabilities due to the lack of composability). Hence, we can conclude that one of the most relevant contributions of this article is to present a full modular architecture complying with all the above-mentioned modularity requirements. Table 1 in next section provides a comparison of each modularity dimensions in several RTC media servers.

Media servers for group communications

Since their early origins, group communications were one of the most popular applications of videoconferencing services. Due to this, today many RTC media servers concentrate on providing such capability [6][7]. In current state-of-the-art, there are two widely accepted strategies for implementing group communications on RTC media servers: media mixing and media forwarding.

Media mixing is performed through the Media Mixing Mixer (MMM) Topology, in which multiple input media streams are decoded, aggregated into a single output stream and re-encoded. For audio, mixing usually takes place through linear addition. For video, the typical approach is to perform video downscaling plus composite aggregation to generate a single outgoing video grid. Hence, a participant in a group RTC session based on MMM sends one media stream (its own audio and/or video) and receives one media stream (the mixed audio and/or video of all participants). As MMM were the common topology on H.323, MMM are sometimes informally called MCUs in the literature [8].

Media forwarding is typically performed through a Selective Forwarding Middlebox topology. RTC media servers implementing this topology are sometimes called Selective Forwarding Units (SFU) in the literature [9]. An SFUs clones and forwards (i.e. routes) the received encoded media stream onto many outgoing media streams. This means that a participant in a group RTC session based on SFUs sends one media stream (its own audio and/or video) and receives N-1 media streams (the audio and/or video of the rest), being N the total number of participants. For the sake of simplicity, RTC media servers implementing the Media Switching Mixer (MSM) topology might also be called SFUs. SFUs and MSM share most of its properties being the only difference that each SFU outgoing stream is uniquely mapped to one of the incoming streams [10].

In current state-of-the-art, RTC most media servers just implement one of the above-mentioned topologies and do not provide flexible mechanism for using the rest. Some solutions [4] enable the ability to use different topologies as pluggable modules, so that, different applications may use different topologies by consuming different module capabilities. In this context, the contributions of this article are straightforward: following our holistic approach, our media server offers all topologies in an integrated way. This enables a novel feature that we could define as "topology as an API", in the sense that application developers do not need to be aware of the complexities or internal details of MMM, MSM or SFUs. They just need to use the media server API modularity features to interconnect the appropriate building blocks in the appropriate way for satisfying application requirements. The media server takes care of translating the API calls into the corresponding topologies combinations and of implementing the required optimizations and media adaptations enabling the appropriate mechanism to be used at the right places.

Transparent media interoperability

Transcoding media servers exist since long time ago due to the need of achieving interoperability among incompatible media systems [11]. Different media systems tend to have different requirements and, due to this, specific formats and codecs were conceived during the years for satisfying them. In addition, the existence of different commercial interests in the multimedia arena contributed to the emergence of a multiplicity of incompatible codecs [12]. As a result, transcoding RTC media servers have been traditionally necessary as soon as a service requires interoperability among different communication domains.

This situation has become even more complex with the arrival of WebRTC given that developers typically need to interoperate WebRTC services with legacy IP infrastructures and with the phone system [13]. Due to this, many RTC media servers provide transcoding capabilities. However, for managing them and create interoperable services, developers need to explicitly manage format and codec conversions. This is in general a very cumbersome process which is error prone and requires deep knowledge about low level details of media representation.

In this context, a very relevant contribution of this article is to introduce a novel capability, that we call the agnostic media, which performs fully transparent transcoding and format adaptations for developers over a wide variety of codecs. To understand how this happens, observe that the modularity requirements specified above mandate our modules to be composable. This means that application developers should be able to freely interconnect them for generating the desired media processing logic. From a practical perspective, this means that, for example, a WebRTC endpoint module receiving video encoded with the VP8 codec can be connected with an RTP endpoint sending H.264 encoded video. Of course, this type of module pipelining requires the appropriate transcodings which, in our case, take place without even requiring developers to know they are happening: the agnostic media detects the need of a transcoding and performs it transparently in the most optimal possible way.

Kurento: the WebRTC modular media server

As stated above, traditional WebRTC applications are peer-to-peer, so that browsers can directly communicate without the mediation of any kind of infrastructure. This is enough for creating basic applications but features such as group communications, media stream recording, media broadcasting or media transcoding are difficult to implement on top of it. For this reason, most interesting applications require using a media server.

Current state-of-the-art WebRTC media servers are typically monolithic pieces of technology that cannot be evolved beyond the above three mentioned capabilities. However, there are many interesting additional things we can do with real-time media such as augmented reality, computer vision, or alpha blending. These kinds of capabilities might provide differentiation and added value to applications in many scenarios such as e-Health, e-Learning, security, entertainment, games or advertising just to cite a few. Based on these ideas, we have created Kurento Media Server (KMS) as an evolution of current state-of-the-art technologies by providing a modular architecture where further capabilities can be plugged as modules. This architecture complies with the 5 main characteristics of modularity: isolation, abstraction, reusability, composability, and extensibility.

In order to provide isolation, abstraction, reusability, Kurento introduces the concept of Media Element. A Media Element is a module that holds a specific media capability. For example, the media element called *WebRtcEndpoint* holds the capability of sending and receiving WebRTC media streams, the media element called *RecorderEndpoint* has the capability of recording into the file system any media streams it receives, the *FaceOverlayFilter* detects faces on the exchanged video streams and adds a specific overlaid image on top of them, etc. As depicted in Figure 1, Kurento exposes a rich toolbox of media elements as part of its APIs based on the following classification:

- We call endpoint to any media element with the capability of communicating media with the external world. *WebRtcEndpoint*, *RecorderEndpoint*, *PlayerEndpoint* or *RtpEndpoint* are some examples of endpoints.
- We call filter to any media element with the capability of processing media. *FaceOverlayFilter*, *PointerDetectionFilter* or *CrowdDetectionFilter* are just examples of filters.
- We call hub to the media elements that make possible group communications: *Composite* and *Dispatcher* are examples of hubs

In Kurento jargon, a graph of connected media elements is called a Media Pipeline. In order to achieve composability within a Media Pipeline, in the heart of every Media Element there is an internal we find a special element called *AgnosticBin*, which manages the transformations and adaptations needed for enabling seamless interconnectivity among different types of Media Elements. The details of this element are discussed in next section.

Finally, in order to implement extensibility, Kurento has been designed as a pluggable media server. Developers can create new modules that can be easily installed on Kurento Media Server (such as the *Extra Filters* group in Figure 1). There are two main flavors of Kurento modules: based on GStreamer (low-level technology of Kurento Media Server) and OpenCV (recommended to create computer vision filters).

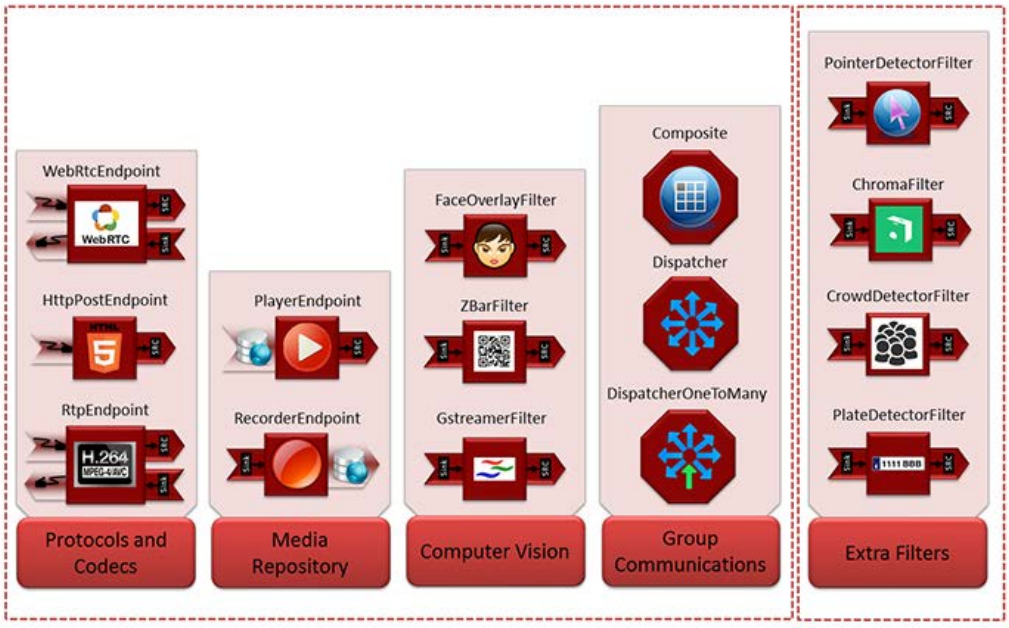

Figure 1. Kurento Media Elements toolbox.

As a result, Kurento Media Server behaves as a complete media middleware that developers can leverage for creating WebRTC applications combining real-time media communications and advanced media processing. There are other open source WebRTC media servers available that include Jitsi, Janus, Medooze and Licode. Kurento is quite unique, as the following comparisons illustrate their differences from the perspective of modularity (summarized in Table 1):

- Jitsi[1] is an open source videoconferencing application that is based on a SFU architecture enforced by a video router component called Jitsi Videobridge. It exposes a high-level API to access to its capabilities. Due to this, Jitsi can be consider to provide abstraction. However, this is the only modular aspect of Jitsi that has been designed more as an application than a framework. As a result, the Jitsi Videobridge cannot be extended easily nor combined with other types of media processing modules in a seamless way. Due to this, extending Jitsi or developing applications based on Jitsi beyond the plain room videoconferencing model is quite complex.
- Janus[2] is a general purpose WebRTC gateway released under the GPL license based on a modular architecture. Janus philosophy is similar to the one of Kurento and its abstraction and isolation properties are equivalent. However, the Janus modular system lacks some relevant modularity properties. For example, Janus modules are not composable at all and their reusability is very limited because modules provide typically high level functionalities comprising a set of capabilities that cannot be "split" and reused independently. In addition, the number of modules that Kurento exposes is significantly higher.
- Medooze[3] is a multiparty videoconferencing service based on a MCU. Medozee, as Janus and Kurento, has been conceived as a development framework that exposes an API to developers. However, it has not been designed with modularity in mind. Due to this, it is not composable and

[1] https://jitsi.org/
[2] https://janus.conf.meetecho.com/
[3] http://www.medooze.com/

cannot provide easily other types of group communication models such as SFU nor any kind of media processing modules (i.e. it is not extensible).

- Licode[4] is a videoconferencing application conceived basing in a very similar philosophy to the one of Jitsi but isolating further their components as different processes. Hence, as Jitsi, it exposes an abstract API for creating applications on top, but it is not really modular and cannot be extended seamlessly with further composable capabilities.

Table 1. Comparison of the modularity dimensions in different media servers.

	Isolation	Abstraction	Composability	Reusability	Extensibility
Jitsi	✗	✓	✗	✗	✗
Janus	✓	✓	✗	✓ (partial)	✓
Medooze	✗	✓	✗	✗	✗
Licode	✓ (partial)	✓	✗	✗	✗
Kurento	✓	✓	✓	✓	✓

As a result, we can conclude that Kurento Media Server implements a novel modularity concept in the area of RTC media servers and that its properties and features are quite unique and adapt to the needs of developers requiring a really modular WebRTC infrastructure.

Media Elements architecture

Kurento Media Server has been implemented in low level technologies based on GStreamer to optimize resource consumption. As described before, the key ingredient to provide modularity by Kurento is the concept of Media Element. In order to understand how these components internally work, it is worth to take a look to the low-level details of their architecture.

As illustrated in Figure 2, inside a Media Element we find a component called *KmsMediaElement*, which is the base class of all Media Element implementations. In short, a *KmsMediaElement* provides the ability of managing *pads* and *bins*. In the GStreamer jargon, a *pad* is the abstraction used to connect different elements, which means that a *KmsMediaElement* can be connected to other *KmsMediaElement*, thus creating a path through which the media flows. A *bin* is a GStreamer object representing a chain of low-level media processing components. The *KmsMediaElement* media processing capabilities (e.g. crowd detection, face overlay, and so on) are implemented as one of such *bins*. Finally, we find a special *bin* called *AgnosticBin*, which is in charge of providing agnostic media, i.e. managing the transformations and adaptations needed for enabling seamless interconnectivity of *KmsMediaElements*.

[4] http://lynckia.com/licode/

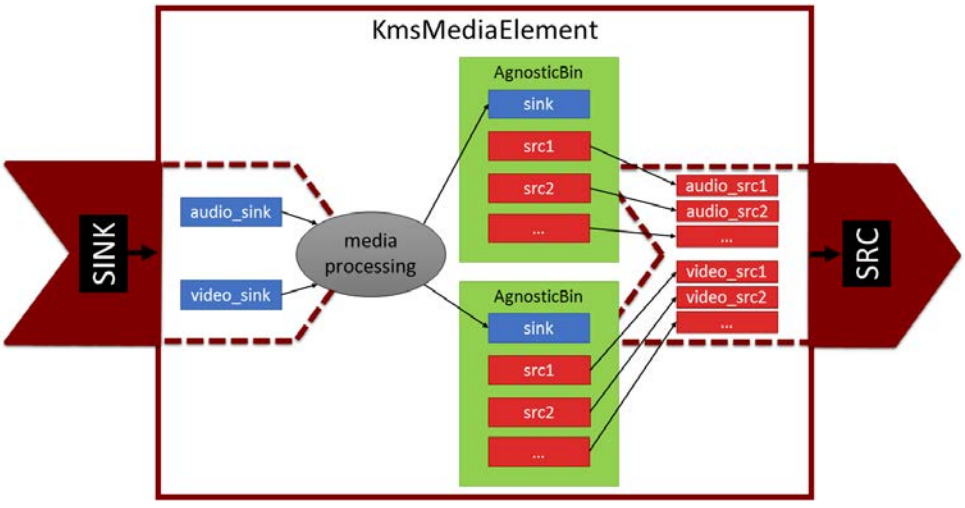

Figure 2. Internal structure of a Kurento Media Element.

The *AgnosticBin* is a tailor-made *bin* which is in charge of adapting between different media formats and performing the codec adaptation needed by the prospective receivers of the processed media. The source pads of the *AgnosticBin* are then connected with the source pads of the enclosing *KmsMediaElement* as new consumers of media. *AgnosticBin* is one of the main innovations of the Kurento Media Server in relation to GStreamer architecture given that it enables developers to interconnect *KmsMediaElement* with full composability as it clones and adapts the media flows to the specific requirements of each of the connected consumers.

Kurento client and protocol

Kurento Media Server capabilities are exposed by the Kurento API to application developers. This API is implemented by means of libraries called Kurento Clients. These clients exchange JSON-RPC over WebSocket messages to control Kurento Media Server (e.g. create a Media Pipeline, connect Media Elements, etc.). The format of these messages implement the Kurento protocol. All in all, as shown on Figure 3, a developer can use Kurento using the following scenarios:

a. Using the Kurento JavaScript Client directly in a compliant WebRTC browser.
b. Using the Kurento Java Client in a Java EE application server.
c. Using the Kurento JavaScript Client in a Node.js server.

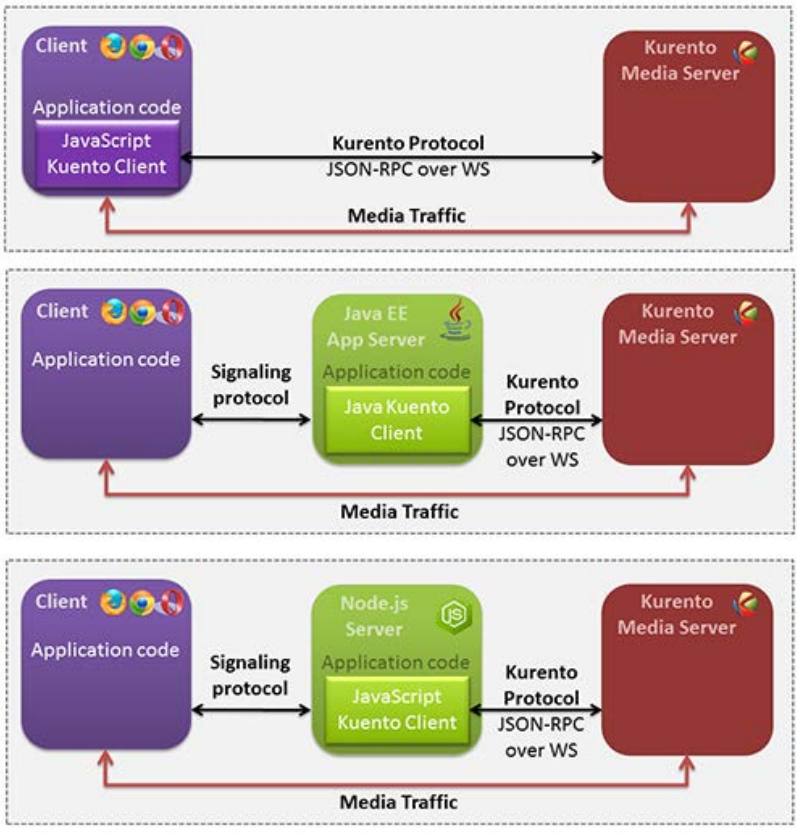

Figure 3. Out-of-the-box Kurento usage scenarios.

Creating Kurento applications

From the application developer perspective, and thanks to the composability properties, Media Elements are like Lego pieces: the developer just needs to take the elements needed for an application and connect them following the desired topology. When creating a Media Pipeline, developers need to determine the capabilities they want to use (the Media Elements) and the topology determining which media elements provide media to which other media elements (the connectivity). The connectivity is controlled through the *connect* primitive, exposed on all Media Elements. This primitive is always invoked in the element acting as source and takes as argument the sink following this scheme:

sourceMediaElement.connect(sinkMediaElement)

Following this approach, developers can create applications with media capabilities is a seamless way. Figure 4 shows several examples of Media Pipelines together with the screenshot of the application implementing each one. First, we can see a group communication application example. This application is a N-to-N (i.e. video room) WebRTC communication, supported in Kurento through SFU and MMM models. Developers can obtain the SFU model just by interconnecting their *WebRtcEndpoints* at wish. The second example is a 1-to-1 WebRTC application (i.e. a softphone app) with recording capabilities. Third, we find a simple example of augmented reality, in which the WebRTC user media is send to Kurento Media Server. This media feeds a *FaceOverlayFilter*, which overlays an image on the top of the detected faces in the media. Finally, a computer vision is illustrated with an application with receives the live RTSP media of a street cam using a *PlayerEndpoint* detecting crowds in the media and sent to a browser by means of a *WebRtcEndpoint* in receive-only mode.

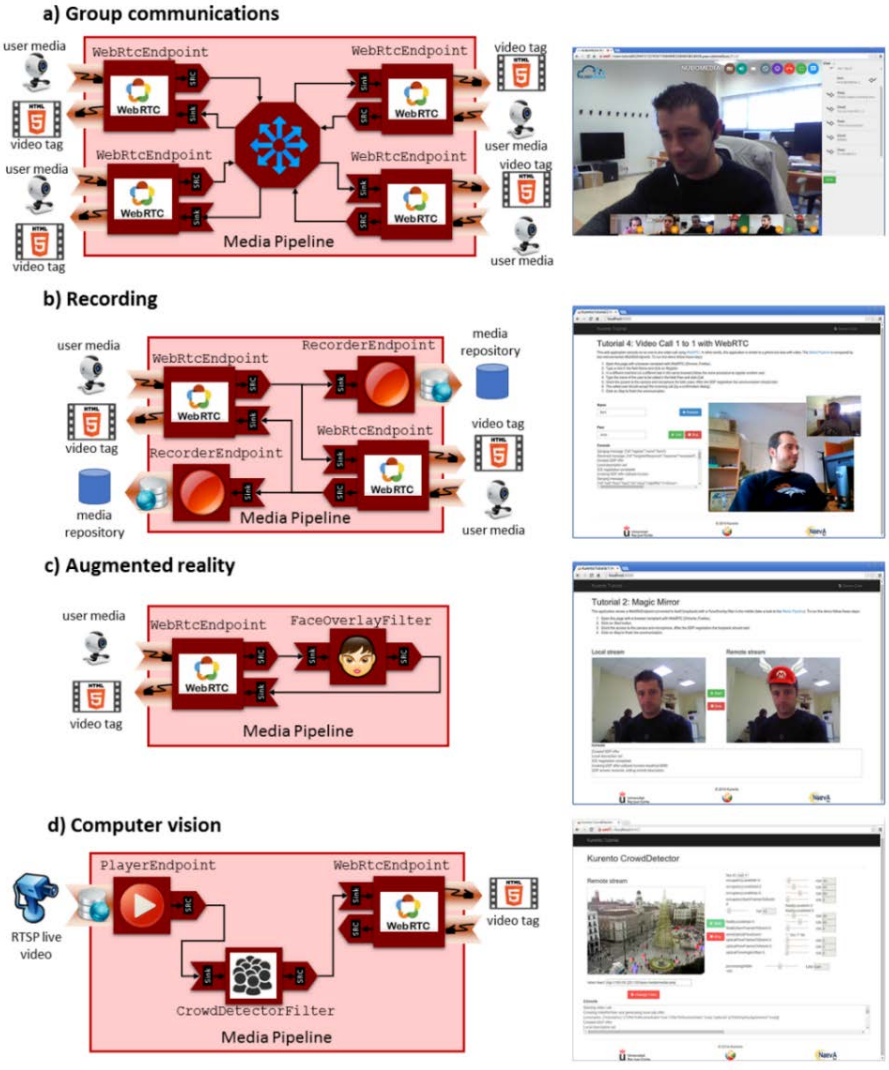

Figure 4. Example of applications created with Kurento (Media Pipelines and screenshots).

Kurento Media Server scalability evaluation

This section presents an experimental case study aimed to evaluate the scalability of the Kurento Media Server. The research question driving this study can be stated as follows: Is KMS capable to support large Media Pipelines while maintaining real-time capabilities to applications?

To address this question, a benchmark application using the Kurento Java client has been developed. The aim of this application is to implement a Media Pipeline with a configurable number of N Media Elements. As depicted before, the key component to provide interoperability among Media Elements is the *AgnosticBin*. Therefore, in this experiment we use a special kind of Media Element called *Passthrough*, a filter in which the media processing component (see Figure 2) is empty. In other words, this media element just provides connectivity (i.e. a couple of raw *AgnosticBin* components). Our application requires two browsers to work: one acting as presenter (i.e. feeding WebRTC user media to the KMS) and other one acting as viewer (i.e. receiving the WebRTC media after KMS processing). All

in all, the Media Pipeline implemented in the applications is the following: *WebRtcEndpoint* → N x *Passthrough* → *WebRtcEndpoint*.

The idea is to execute this application while different key performance indicators are gathered, namely: a) internal latency in each *Passthrough* (this metric is exposed by KMS and can be gathered using the client API just asking the Media Element object); b) Number of threads of KMS; c) CPU usage in machine hosting KMS; d) RAM memory usage in machine hosting KMS. For the later 3 metrics, a custom test component called *KmsMonitor* has been created to read the current state of the Linux kernel (i.e. the */proc* folder). In this experiment, the KMS has been hosted in a physical machine with the following features: Ubuntu 16.04 64 bits, 8GiB RAM, Intel Quad Core i7 2.00GHz.

We execute this application with an incremental value of N up to 50 counting 5 by 5. The results are depicted on Figure 5. We can see in chart a) of this picture that latency remains in an optimal range. Even in the worst case (N=50) the average total internal latency through all the *Passthrough* media elements is quite stable (around 6ms). Regarding the number of threads, the results shows a complete linear behavior with slope m=2. This confirms that each new *Passthrough* consumes a thread per *AgnosticBin*. Finally, both CPU and memory consumption remains stable during this experiment: up to 80% of CPU and 0.75GiB of RAM memory usage in the worst case (i.e. 50 chained *Passthroughs*).

RTC applications imply a hard restriction related to having a low latency (less than 300ms [14]) to allow a conversational (bidirectional) communication. Thus, we can conclude that the scalability of the KMS in terms of latency is probed due to the fact that very large Media Pipelines can be executed while maintaining real-time features to applications. Moreover, in the light of the results we confirm that KMS also presents linear performance for the physical parameters (CPU, memory, and number of threads). This behavior allows vertical scaling simply increasing the size of the machine hosting KMS (i.e. more CPU cores and more memory), and horizontal scalability by adding more KMS instances (typically in a cloud environment) [15].

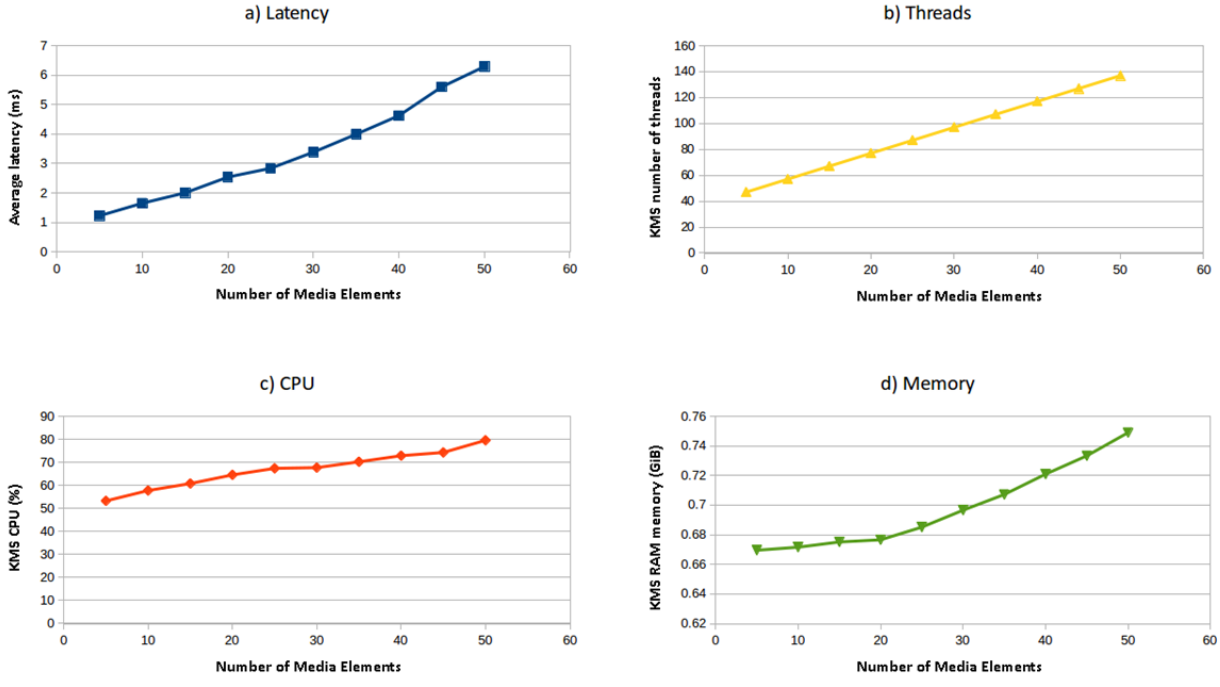

Figure 5. Kurento Media Server scalability evaluation results.

Conclusions

Kurento is an open source WebRTC media server that makes it possible to create media processing applications basing on the concept of Media Pipelines, which are created by interconnecting modules called Media Elements. Each Media Element holds a specific media capability, whose details are hidden to application developers. The Kurento toolbox contains Media Elements capable of, for example, recording streams, mixing streams, applying computer vision to streams, augmenting or blending streams, etc. In addition, thanks to the pluggable architecture of Kurento, developers can create and plug their very own Media Elements. From the application developer perspective, media elements are like Lego pieces: one just needs to take the elements needed for an application and connect them following the desired topology. This type of modularity is a novelty in the area of RTC media servers.

Kurento Media Server exposes the capabilities of creating Media Pipelines through a simple network protocol based on JSON-RPC. However, for simplifying further the work of developers, we have created an abstract client API implementing that protocol and directly exposing Media Elements and Pipelines as objects that developers can instantiate and manipulate on client programs. Currently there the Java and JavaScript client API are available out-of-the-box for developer.

Acknowledgments

This work has been supported by the European Commission under projects NUBOMEDIA (FP7-ICT-2013-1.6, GA-610576), and ElasTest (H2020-ICT-10-2016, GA-731535); and by the Regional Government of Madrid (CM) under project Cloud4BigData (S2013/ICE-2894) cofunded by FSE & FEDER.

References

[1] S. Loreto, and S. P. Romano, “Real-Time Communications in the Web: Issues, Achievements, and Ongoing Standardization Efforts”, *IEEE Internet Computing*, vol. 16, no. 5, 2012.

[2] B. Li, et al., “Two decades of internet video streaming: A retrospective view”, *ACM transactions on multimedia computing, communications, and applications (TOMM)*, vol. 9, no. 1, pp. 33, 2013.

[3] Y. Lu, et al., “Measurement study of multi-party video conferencing”, *International Conference on Research in Networking*, pp. 96-108. Springer Berlin Heidelberg, 2010.

[4] C. Baldwin, et al.: *The power of modularity*. Vol. 1. MIT press, 2000.

[5] A. Amirante, et al., “Janus: a general purpose WebRTC gateway”, *Conference on Principles, Systems and Applications of IP Telecommunications*, pp. 7, 2014.

[6] B. Grozev, et al., “Last n: relevance-based selectivity for forwarding video in multimedia conferences”, *25th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video*, pp. 19-24, 2015.

[7] N. Kwok-Fai, et al., “A P2P-MCU Approach to Multi-Party Video Conference with WebRTC”, *International Journal of Future Computer and Communication*, vol. 3, no. 5, pp. 319, 2014.

[8] H. Schulzrinne, and J. Rosenberg, “The IETF internet telephony architecture and protocols”, *IEEE network*, vol. 13, no. 3, pp. 18-23, 1999.

[9] M. Westerlund, and W. Stephan: *RTP topologies*. No. RFC 7667. 2015.

[10] H. Schulzrinne, et al.: *RTP: A transport protocol for real-time applications*. No. RFC 3550. 2003.

[11] A. Elan, S. McCanne, and R. Katz, “An active service framework and its application to real-time multimedia transcoding”, *ACM SIGCOMM Computer Communication Review*, vol. 28, no. 4, pp. 178-189, 1998.

[12] S. Harjit, et al., “VoIP: State of art for global connectivity - A critical review”, *Journal of Network and Computer Applications*, vol. 37, pp. 365-379, 2014.

[13] B. Emmanuel, et al., “WebRTC, the day after”, *17th International Conference on Intelligence in Next Generation Networks, ICIN*, 2013.

[14] W. Cellary and A. Iyengar, *Internet Technologies, Applications and Societal Impact: IFIP TC6 / WG6.4 Workshop on Internet Technologies, Applications and Societal Impact*. Springer, 2013.

[15] Rimal, B.P., et al, “A taxonomy and survey of cloud computing systems”, *INC, IMS and IDC*, pp. 44-51, 2009.

Biographies

Boni García has a PhD degree on Information and Communications Technology from Universidad Politécnica de Madrid (Spain) in 2011. His main research interests are software testing, web engineering and computer networking. Currently he is working as a Researcher at Universidad Rey Juan Carlos and Assistant Professor at Centro Universitario de Tecnología y Arte Digital (U-tad) in Spain. He is member of Kurento project, where he is in charge of the testing framework for WebRTC applications.

Luis Lopez is associated professor at URJC and lead of the Kurento.org open source software project. Dr. Lopez research interests are concentrated on the creation of advanced multimedia communication technologies and on the conception of Application Programming Interfaces on top of them. His ideas have generated more than 60 technical publications and have been included into important research and industrial projects including FIWARE and NUBOMEDIA.

Micael Gallego earned a PhD on Computer Science from Universidad Rey Juan Carlos (Spain) in 2008. Among other scientific publications of high impact, he is the coinventor with an AT&T researcher of an American patent. He has participated in three national research projects, from the Spanish Research Agency, and in two European research projects. He is software architect in the Kurento project, where he is focused on developing scalable and fault tolerance distributed systems.

Francisco Gortázar holds a PhD degree on Computer Science. He is Associate Professor at Universidad Rey Juan Carlos. His main research activities are related to Software Engineering and Software Testing. He is member of the Kurento project, where he is focused on developing infrastructure support for testing distributed applications. He leads the ElasTest project focused on end-to-end testing on large distributed software systems.