Multimed Tools Appl
DOI 10.1007/s11042-016-3729-z

Designing and evaluating the usability of an API for real-time multimedia services in the Internet

Luis López-Fernández[1] · **Boni García**[1] · **Micael Gallego**[1] · **Francisco Gortázar**[1]

Received: 27 January 2016 / Revised: 21 June 2016 / Accepted: 27 June 2016

Abstract In the last few years, multimedia technologies in general, and Real-Time multimedia Communications (RTC) in particular, are becoming mainstream among WWW and smartphone developers, who have an increasing interest in richer media capabilities for creating their applications. The engineering literature proposing novel algorithms, protocols and architectures for managing and processing multimedia information is currently overwhelming. However, most of these results do not arrive to applications due to the lack of simple and usable APIs. Interestingly, in this context in which APIs are the critical ingredient for reaching wide developer audiences, the scientific literature about multimedia APIs and their usability is scarce. In this paper we try to contribute to fill this gap by proposing the RTC Media API: a novel type of API designed with the aim of making simple for developers the use of latest trends in RTC multimedia including WebRTC, Video Content Analysis or Augmented Reality. We provide a specification of such API and discuss how it satisfies a set of design requirements including programming-language agnosticism, adaptation to cloud environments, support to multisensory multimedia, etc. After that, we describe an implementation of such an API that has been created in the context of the Kurento open source software project, and present a study evaluating the API usability performed in a group of more than 40 professional developers distributed worldwide. In the light of the obtained results, we conclude that the usability of the API is adequate across the main development activities (i.e. API learning, code creation and code maintenance), with an average usability score of 3.39 over 5 in a Likert scale, and that this result is robust with respect to developers' profiles, cultures, professional experiences and preferred programming languages.

✉ Luis López-Fernández
luis.lopez@urjc.es

[1] Universidad Rey Juan Carlos, Camino del Molino S/N, 28943 Fuenlabrada, Spain

Published online: 13 August 2016

Keywords Media server · Real-time multimedia communications · Application Programming Interfaces · WebRTC · Multimedia processing · Multimedia tools and applications · Cognitive dimensions of notations

1 Introduction

Analysts such as Marc Andreessen claim that "software is eating the world" stressing the importance of software-centered models into the economy and the transition of traditional business to software-based organizations [7]. This trend is permeating into all areas of IT (Information Technologies) including also multimedia industries. In the last few years, we have witnessed how multimedia technologies have been evolving toward software-centered paradigms embracing cloud concepts through different types of XaaS (Everything as a Service) models [17].

More recently, another turn of the screw is taking place thanks to the emergence and popularization of APIs (Application Programming Interfaces). This is perfectly summarized by Steven Willmott with his claim "software is eating the world and APIs are eating software" [81]. Software developers worldwide are getting used to create their applications as a composition of capabilities exposed through different APIs. These APIs are typically accessible through SDKs (Software Development Kits) and expose in an abstract way all kind of capabilities including device hardware, owned resources and remote third party infrastructures. This model, applied to cloud concepts, is quite convenient for individual developers and small companies, which have now the opportunity of competing with large market stakeholders without requiring huge effort investments and without needing to acquire hardware infrastructure or software licenses. Thanks to this, in the last few years, we are experiencing an explosion of innovation with thousands of new applications and services both for WWW and smartphone platforms that are being catalyzed by the rich and wide ecosystems of APIs made available to developers.

This trend towards the "APIfication" is also invading the multimedia arena and, very particularly, the RTC (Real-Time multimedia Communications) area. Initiatives such as WebRTC [44] are bringing audiovisual RTC in a standard and universal way to WWW users. The main difference between WebRTC and other popular video-conferencing applications is that WebRTC is not a service, but a set of APIs enabling WWW developers to create their customized applications using standard WWW development techniques.

WebRTC belongs to the HTML5 ecosystem and has awakened significant interest among the most important Internet and telecommunication companies. As opposed to other previous proprietary WWW multimedia technologies, it has been conceived to be open in a broad sense, both by being based on open standards and by providing open source software implementations. Currently, a huge standardization effort on WebRTC protocols is taking place at different IETF working groups (WGs), being the RTCWeb WG the most remarkable one [41]. In turn, WebRTC APIs are being defined and consolidated at the W3C WebRTC WG [82]. WebRTC standards are still under maturation stage and they might take some time to consolidate. In spite of this, most of the major browsers in the market already support WebRTC and it is currently available in billions of devices providing interoperable multimedia communications.

Hence, WebRTC is an opportunity for the creation of a next generation of disruptive and innovative multimedia services catalyzed worldwide through those emerging APIs. However,

to reach this goal, the WebRTC ecosystem needs to evolve further. Basing on WebRTC browser capabilities, services can only provide peer-to-peer communications, which restrict use-cases to simple person-to-person calls involving few users. In order to enhance this model, server side infrastructures need to be involved. This is not new: as it is well known, the traditional WWW architecture is based on a three tier model [31] involving an application server layer and a service layer, this latter typically reserved to databases. In the same way, rich media applications also base on an equivalent three tier model where the service layer provides advanced media capabilities. The media component in charge of providing such capabilities is typically called media server in the jargon.

There is not a formal definition of what a media server is and different authors use the term with different meanings. In this paper, we understand that a media server is just the server side of a client-server architected media system. We concentrate our attention on RTC media servers, which are specialized in RTC media problems. Commonly, RTC media server capabilities consist on the following [50]:

- Group communication capabilities: These include mixing and forwarding. This type of media servers is called MCU (Multipoint Control Unit) [69] following the H.323 terminology and usually takes the form of Mixing Mixers or Selective Forwarding Units (SFU) [80].
- Media archiving capabilities: These are related to the recording of the audiovisual streams into structured or unstructured repositories and the ability to recover them later for visualization.
- Media bridging capabilities: This refers to attaining interoperability among networks or domains having incompatible media formats or protocols. Transcoders and IMS (IP Multimedia Subsystem) Gateways [34] are among the most popular in this area.

Media servers are a critical ingredient for transforming WebRTC into the next wave of multimedia communications, and the availability of mature solutions exposing simple to use yet powerful APIs is a necessary requirement in that area. However, most standardization and implementation efforts are still concentrated at the client side, and server side technologies are still quite fragmented. Although there are a relevant number of WebRTC media servers available they do not provide coherent APIs compatible with WWW development models. Developing solutions with them typically requires expertise with low level protocols such as SIP [63], XMPP [64] or MGCP [6], on which average WWW developers do not have any experience. In addition to this, most state-of-the-art WebRTC media servers just provide the three basic capabilities specified above and are extremely hard to extend with further features. However, nowadays, many RTC services involve person-to-machine and machine-to-machine communication models and require richer multimedia processing capabilities such as computer vision, augmented reality, speech analysis and synthesis, etc.

In this paper we propose an evolution on current state-of-the-art RTC media servers presenting a new type of RTC API for media server control, which has been designed for usability. This API addresses many current state-of-the-art limitations, such as the ones described above, and is aligned with WWW development principles, architectures and methodologies. The main contributions of this paper are threefold. First, we introduce the main concepts of the above mentioned API. Second, we present how developers may leverage it and create applications providing transparent interoperability among heterogeneous formats and protocols through a modular and extensible architecture. Third, we present an evaluation of the

proposed API usability based on the Cognitive Dimensions of Notations (CDs) [35], which is a lightweight framework created for describing and analyzing the usability of notational systems, such as user interfaces, programming languages and APIs.

The remainder of this paper is as follows. Section 2 summarizes some approaches of RTC media servers and APIs available in the literature. Section 3 presents the proposed RTC Media API and illustrates how to create applications with it. Section 4 describes a survey in which our API is evaluated by means of a research questionnaire following the CDs framework. The last section concludes this research with discussion, contributions of the study and suggestions for further work.

2 Related work

2.1 RTC media server control APIs

Media server technologies emerged in 90's catalyzed by the popularization of digital video services. Initial media servers were specialized into specific functions such as streaming [48], transcoding [71] and RTC for audio and video conferencing [7]. In this paper we concentrate on this latter category.

The popularization of video and audio conferencing made RTC media servers to evolve through different types of standards, which include H.323 [73], where the media server role is played by elements such as the MCU (Multipoint Control Unit) and the IMS (IP Multimedia Subsystem), where media servers are generically called MRF (Media Resource Function) [46]. These standards were conceived by operators and corporate communications solution vendors, who concentrated on the specificities of their infrastructures and not on the needs of developers. As a consequence, the involved media control interfaces were designed based on low level protocols and not on high level friendly APIs. Among such protocols we can find the IETF MGCP [6], which evolved later into the 3GPP H.248 [72] recommendation. These are based on binary formats, which are hard to understand, implement, debug and extend. Probably due to this, these protocols did not have much impact out of telecommunication providers.

More recently, the commoditization of RTC media server technologies brought increasing interest on more flexible mechanisms for media control. Several IETF WG emerged with the objective of democratizing them among common developers. As a result, further protocols such as MSCML [75], MSML [65] emerged providing the ability of controlling media server resources through technologies understandable and familiar to average developers such as XML [16].

Although these protocols are simpler to understand and integrate, developing application on top of them is still a cumbersome, complex and error prone process. Due to this, many stakeholders noticed that the natural tools used by developers are not protocols but APIs and SDKs. Hence, a number of initiatives emerged trying to transform the protocol-based development methodology into an API-based development experience providing seamless media server control through interfaces adapted to programming languages specificities and not to infrastructure characteristics. In particular, the Java platform was one of the first on integrating this philosophy by trying to reproduce the WWW development experience and methodology for the creation of RTC media enabled applications. A relevant activity in this area is JAIN (Java API for Integrated Networks), which issued several APIs for the signaling,

control and orchestration of media capabilities. These include the JAIN SIP API [53], the JAIN SLEE API [29] and the JAIN MEGACO API [9]; this latter being specifically devoted to control media servers through the H.248 protocol. JAIN APIs did not permeated much out of operators, but their ideas inspired more popular developments such as the SIP Servlet APIs [47] for the signaling plane, and the Media Server Control API (aka JSR 309) [27] for the media plane, which have been more widely used for the development of RTC solutions for voice and video.

Among all these APIs, this paper is especially interested in the JSR 309. JSR 309 concepts were quite revolutionary at the moment because the API tried to fully abstract the low level media server control protocols and media format details. The objective was to enable developers to concentrate on application logic. JSR 309 defined both a programming model and an object model for media server control through a northbound interface, but independent of media server control protocols and hence, without requiring any specific southbound protocol driver. JSR 309 does not make any kind of assumption in relation to the signaling protocol or to the call flow, which are left to the application logic.

From a developer's perspective, probably the most innovative concept of JSR 309 was the introduction of a mechanism for defining the media processing logic in terms of a topology. This mechanism is based on an interface called `Joinable`. In JSR 309, all objects having the ability to manipulate media (e.g. send, receive, process, archive, etc.) implement such interface, which has a `join` method enabling interconnecting such objects following arbitrary dynamic topologies. Hence, a specific media processing logic can be implemented by developers just joining the appropriate objects. As an example, if you want to create an application mixing two RTP (Real-time Transport Protocol) streams and recording the resulting composite into a file, you just need to join the appropriate objects with the appropriate topology. Taking into consideration that in JSR 309 the `NetworkConnection` is the class of objects capable of receiving RTP streams, that `MediaMixer` is the class of objects with mixing capability and that `MediaGroup` is the class with be ability of recording; the above mentioned media topology can be achieved just by joining two `NetworkConnection` instances to a `MediaMixer` instance, which in turn, is joined with a recording `MediaGroup`. This approach makes possible for developers to conceive their media processing logic as graphs of "black-box" *joinables*, which is a quite modular and intuitive mechanism for working in abstract terms with the complex concepts involved in RTC multimedia applications.

Another relevant innovation of JSR 309 is the introduction of media events. Thanks to this mechanism, the media processing logic held by a media server can fire events to applications through a publish/subscribe mechanism. This is very convenient for enabling applications to become media-aware meaning that complex processing algorithms at the media server can provide asynchronous information dealing with things happening inside the media, for instance DTMF (Dual-Tone Multi-Frequency) tones being detected, voice activity being present, and so on.

JSR 309 permeated into mainstream developer audiences as a suitable API for media server control following the typical three tier model [27]. However, in the last few years, the emergence of novel technologies and computation paradigms have made JSR 309 to show relevant limitations. For example, nowadays group videoconferencing services are evolving from Media Mixing models, which require relevant media processing, towards SFU (Selective Forwarding Unit) models, which are based on media

routing [80]. JSR 309 is heavily adapted to Media Mixing and, due to this, most of its APIs assume that participants send/receive only one media stream to/from the media server. As a consequence, SFU models do not fit nicely into JSR 309 APIs. This is particularly a problem when all the streams of a group videoconference are multiplexed into a single RTP session, as happens typically on modern WebRTC SFU media servers supporting bundle RTP [43], because JSR 309 APIs do not provide any kind of mechanism for demultiplexing streams from a `NetworkConnection`. Moreover, in JSR 309 the API specification explicitly forbids several input `NetworkConnections` to be joined to a single output `NetworkConnection`, as an SFU router would require. Instead, they need to be joined first to a `MediaMixer`, which, in turn, can be joined to the output `NetworkConnection`.

When looking to other modern RTC technologies, we notice again that the JSR 309 design has limitations. For example, if we consider WebRTC W3C APIs [82], we may understand that they split endpoint capabilities into different functional blocks each of which is exposed through an abstract interface (e.g. `RtpSender`, `RtpReceiver`, `PeerConnection`, etc.) However, if we want to expose WebRTC media server capabilities through JSR 309 we need to accept that endpoints can only be represented through the `NetworkConnection` interface, which is extremely limited to support rich WebRTC capabilities such as `DataChannels` [11], Trickle ICE [42], simulcast [79], etc.

JSR 309 shows also drawbacks in relation to its extensibility. In JSR 309 it is possible to support new media object types using `MediaGroups`, however, configuration of these new types have to be done with Media Server specific descriptions as strings, which cannot be validated by the compiler. It is important to note that these new media object types cannot be `NetworkConnection`, only `MediaGroups`. This is a hard limitation because no other network protocol different than RTP (negotiated through SDP) can be incorporated. The ideal would be to allow supporting the creation of new object types in a similar way than the core types, with factory methods in `MediaSession` (e.g. `createNetworkConnection`, `createMediaGroup`, etc.), but this is not possible as `MediaSession` is an interface defined in JSR 309 API and hence it cannot be modified by the API user.

Further limitations about JSR 309 include:

- A counter-intuitive asynchronous development model based on an obscure `joinInitiate` primitive, which is incompatible with modern Java mechanism for managing asynchrony such as futures, continuations or lambdas. This lack of clean asynchronous programming model makes JSR 309 difficult to adapt to reactive programming frameworks and languages that are very demanded by developers today such as Node.js or Scala.
- A complete lack of mechanisms for monitoring and gathering quality stats on media sessions. This is an essential ingredient for production systems.
- JSR 309 is designed specifically for the Java language. It would be desirable a portable API that can be used in as more languages as possible.
- This API is specifically designed to control Media Servers for phone communications because it exposes concepts like Dialogs (Prompt and record, DTMF, VoiceXML dialog, etc.). For example, it is mandatory for an implementation to provide a player with the capability to detect audio signals in DTMF, but this kind of functionality is not very useful in web applications.

2.2 Foundations of API evaluation and characterization

APIs are critical, non-optional and cross-cutting in the construction of modern software systems [39]. Programming is a hard mental work and developers need to deal with large amounts of information for writing satisfactory code. In that duty, APIs are the most critical ingredient, especially when dealing with distributed systems and enterprise frameworks. For example, recent works [40] show that API misuse is the single most prevalent cause of software defects.

Designing APIs consists on conceiving abstractions through types and interfaces so that they can be consumed seamlessly, efficiently and safely by application developers. This is quite a complex topic for which very little is known and which requires interdisciplinary knowledge combining cognitive psychology and software engineering. However, the responsibility of API design is typically assigned to development team members who often do not have expertise or training in this area and who, typically, are more concerned with implementation details than with usability.

In spite of the well-known importance of APIs, API design and evaluation has not been a mainstream research topic and only recently some light has been shed on this area. Early attempts to investigating APIs typically followed unstructured and ad-hoc approaches concentrating on the specificities of given technologies. For example, works have been published with guidelines and recommendations for API design in C# [78], Java [15] or C++ [60] and for ad-hoc evaluation of new programming languages [18].

Using another perspective, some authors concentrated on specific problems transversal to all APIs with independence on their underlying technologies. Some remarkable efforts on this area enabled to understand that, for instance, the factory pattern tends to generate usability problems [26] and that there is a systematic set of questions that developers have when learning new APIs [24]. All these efforts are relevant due to the talent of their authors to detect and isolate common patterns and practices, but they do not make possible to build a consistent and reusable methodology for the area.

During the last decades further authors have tried to systematize the problem of API design and usability evaluation from a holistic perspective. Different approaches have been created for this [2, 12, 21, 28]. However, in this area, the one which has gained highest popularity is the Cognitive Dimensions of Notations (CDs) framework [13, 36]. CDs is a framework for describing the usability of notational systems. In this context, a notational system typically consists of a collection of symbols made on some medium and which define a behavior (i.e. meaning) through some kind of structured interactions. Examples of notational systems include English text on paper, buttons on a WWW GUI or programing with API calls on an IDE. CDs allow designers of notational systems to evaluate their designs with respect to the impact they have on the users of those designs.

The CDs framework is not an analytic method. Rather, it is a set of discussion tools for use by designers and people evaluating designs whose main aim is to improve the quality of discussion. CDs emerged because, at the end of the day, API design is more of an engineering craft than a scientific discipline. It is subject to elements of affect, of fashion and of social acceptance, in addition to technical considerations. For these reasons, we can learn from studies of other design disciplines where the same craft elements apply. For example, a study comparing knitwear designers and helicopter designers [25] observed that designers' communities develop their own vocabulary for design criteria that is created through practice and tradition. The CDs framework aims to provide the same kind of vocabulary for API designers.

As a result, the CDs main objective is to enable API designers to reason consistently in relation to how well an API supports the intended activities of its users. Simply stated, CDs make possible to discuss in a coherent way about the extent to which an API supports application developers at the time of performing typical activities such as API learning and understanding, application design and creation, application maintenance and evolution, etc. For this, the framework considers a set of dimensions each of which describes an aspect of the API usability. These dimensions constitute a vocabulary of terms that can be used to characterize cognitive artifacts and which makes possible to establish comparisons and to discuss and investigate about the implications of design decisions on those artifacts. It is important to remark that these dimensions are not good or bad in themselves but that they simply describe properties of the system with respect to developers' activities.

In the context of API evaluation, the CDs framework is a powerful tool because it allows to compare users' expectations and designer's views of the APIs with what the system actually provides. For example, early users of the HTML notation probably expected to be able to modify their pages headings easily, whereas the language required one action per heading for doing so. This signaled an imperfection in HTML API's usability which probably drove to the introduction of CSS (Cascading Style Sheets). In CDs terms such resistance to changes is characterized through a dimension called *viscosity*. Hence, in this case, we would say that CSS decreased the viscosity of HTML APIs. A comprehensive description of CDs dimensions can be found in Blackwell et al [13]. For the sake of completeness, here we introduce a brief (and incomplete) description of the 13 main dimensions of the CDs framework:

Viscosity: resistance to change.

A viscous system needs many user actions to accomplish one goal. Changing all headings to upper-case may need one action per heading. (Environments containing suitable abstractions can reduce viscosity.) We distinguish repetition viscosity, many actions of the same type, from knock-on viscosity, where further actions are required to restore consistency.

Visibility: ability to view components easily.

Systems that bury information in encapsulations reduce visibility. Since examples are important for problem-solving, such systems are to be deprecated for exploratory activities; likewise, if consistency of transcription is to be maintained, high visibility may be needed.

Premature commitment: constraints on the order of doing things.

Self-explanatory. Examples: being forced to declare identifiers too soon; choosing a search path down a decision tree; having to select your cutlery before you choose your food.

Hidden dependencies: important links between entities are not visible.

If one entity cites another entity, which in turn cites a third, changing the value of the third entity may have unexpected repercussions. Examples: cells of spreadsheets; style definitions in Word; complex class hierarchies; HTML links. There are sometimes actions that cause dependencies to get frozen, e.g. soft figure numbering can be frozen when changing platforms; these interactions with changes over time are still problematic in the framework.

Role-expressiveness: the purpose of an entity is readily inferred.

Role-expressive notations make it easy to discover why the author has built the structure in a particular way; in other notations each entity looks much the same and

discovering their relationships is difficult. Assessing role-expressiveness requires a reasonable conjecture about cognitive representations.

Error-proneness: the notation invites mistakes and the system gives little protection.

Enough is known about the cognitive psychology of slips and errors to predict that certain notations will invite them. Prevention (e.g. check digits, declarations of identifiers, etc) can redeem the problem.

Abstraction: types and availability of abstraction mechanisms.

Abstractions (redefinitions) change the underlying notation. Macros, data structures, global find-and-replace commands, quick-dial telephone codes, and word-processor styles are all abstractions. Some are persistent, some are transient. Abstractions, if the user is allowed to modify them, always require an abstraction manager (i.e. a redefinition sub-device). It will sometimes have its own notation and environment (e.g. the Word style sheet manager) but not always (for example, a class hierarchy can be built in a conventional text editor). Systems that allow many abstractions are potentially difficult to learn.

Closeness of mapping: closeness of representation to domain.

How closely related is the notation to the result it is describing?

Consistency: similar semantics are expressed in similar syntactic forms.

Users often infer the structure of information artifacts from patterns in notation. If similar information is obscured by presenting it in different ways, usability is compromised.

Diffuseness: verbosity of language.

Some notations can be annoyingly long-winded, or occupy too much valuable "real-estate" within a display area. Big icons and long words reduce the available working area.

Hard mental operations: high demand on cognitive resources.

A notation can make things complex or difficult to work out in your head, by making inordinate demands on working memory, or requiring deeply nested goal structures.

Provisionality: degree of commitment to actions or marks.

Even if there are hard constraints on the order of doing things (premature commitment), it can be useful to make provisional actions such as recording potential design options, sketching, or playing "what-if" games. Not all notational systems allow users to fool around or make sketchy markings.

Progressive evaluation: work-to-date can be checked at any time.

Evaluation is an important part of a design process, and notational systems can facilitate evaluation by allowing users to stop in the middle to check work so far, find out how much progress has been made, or check what stage in the work they are up to. A major advantage of interpreted programming environments such as BASIC is that users can try out partially- completed versions of the product program, perhaps leaving type information or declarations incomplete.

The CDs framework has been criticized due to its theoretical and practical limitations. For example, Moody et al [51] claim that CDs do not provide a scientific basis because of several reasons:

- The dimensions are vaguely defined often leading to misinterpretation in applying them.
- The theoretical and empirical foundations of the dimensions are poorly defined.
- The dimensions lack clear operationalization (i.e. evaluation procedures and metrics), which mean they can be only applied in a subjective manner.

- It does not support evaluation, as the dimensions simply define properties of notations and are not meant to be either "good" or "bad".
- It does not support design: the dimensions are not design guidelines and issues of effectiveness are excluded from its scope.
- Its level of generality precludes specific predictions meaning that it is unfalsifiable and, hence, it cannot be considered to provide a scientific basis for evaluating anything.

In spite of these criticisms, most authors accept that, although CDs need further evolutions and improvements, they are today the most suitable tool for performing comparative API evaluation and that their methodological principles allow analyzing real-world development problems on controlled lab studies in quite an efficient and lightweight manner [36]. The advantages and main motivations why many authors prefer the CDs over other usability techniques include the following:

- They offer a comprehensive, broad-brush evaluation mechanism which does not suffer the 'death by details' symptom of other techniques.
- They offer a set of discussion tools and a common vocabulary helpful for evaluating designs.
- They are based on terms that are comprehensible by non-specialists.
- They are directly applicable, without requiring customizations or reinterpretations, to all types of notations including APIs.
- Although they are not theoretically complete, they are theoretically coherent, which makes possible to analysts to generate consistent analyses.
- They describe a set of necessary, though not sufficient, conditions for usability, which enable deriving usability predictions from the structural properties of a notation, the properties and resources of an environment and the type of activity.

2.3 Quantitative evaluation of API usability

CDs are used by designers for performing quantitative evaluation of API usability. The common practice for this is to use a questionnaire [14] requesting users to evaluate, through a Likert scale [4], how they experience CDs dimensions when performing their development activities. There is a broad literature illustrating how to create such reliable questionnaires [58]. When questionnaires target unsupervised and open audiences through the WWW, as it is the case on this paper, a critical aspect for attaining reasonable answer rates and acceptable accuracy is simplicity [33]. Hence, without a full and complete understanding of the questions, developers under evaluation might not be willing to provide any information on the API usability at all, or might be giving incomplete or mistaken answers.

As stated above, the CDs framework defines dimensions as a vocabulary that can be used by designers when investigating the cognitive implications of their design decisions, so that designers might be able to express any properties of their information artifacts as a composition of these basic dimensions. As an analogy, this is somehow similar to the way vector spaces work: any vector in the space can be expressed as a composition of the base vectors. From this perspective, the base CDs dimensions are designed for independence (i.e. they do not overlap) and not for clarity and simplicity. As a result, questionnaires addressing the complete set of CDs dimensions in the context of all common development activities are too

complex, long and impractical for our objectives [14]. Using them might decrease the aim of the target population to provide answers as well as the overall usefulness of the resulting research.

Due to this, some authors propose an adaptation to the CDs framework based on transforming the dimensions into another base that is more meaningful for developers and that is compatible with shorter and simpler questionnaires [57]. These new dimensions are called Clarke's dimensions and concentrate on 5 specific aspects of API usability: understandability, abstraction, expressiveness, reusability and learnability. All these high-level dimensions can be expressed in terms of the original CDs dimensions, but their operationalization for quantitative research is more practical due to a number of reasons. First, Clarke's dimensions illustrate the user's perspective (i.e. developer) and not the designer's one, as happens with the plain CDs dimensions. As a result, they allow to optimize the questionnaire for API users more than for API designers. Second, Clarke's dimensions are simpler to understand as they are fewer and as they refer to intuitive and positive usability properties (i.e. the highest the evaluation of each dimension the better the API usability). This is in opposition to CD dimensions that are not associated to a specific notion of goodness. Thanks to this, we are able to scale down the number of questions and to state them in a simpler and more straightforward way. Third, each of Clarke's dimensions refers to a specific developers' activity, which further simplifies the questionnaire and enables a more direct analysis of the results. For illustration, these activities include exploratory learning (i.e. learning how to use the API for creating applications), exploratory design (i.e. the process of using the APIs for designing and creating applications) and maintenance (i.e. corrective modifications, evolutionary modifications, etc.) Clearly, understandability and learnability are applicable to exploratory learning activities, abstraction and expressiveness to exploratory design activities and maintainability to maintenance activities. Let's explain in details the meaning and value of each of Clarke's high-level dimensions (see Table 1 for further details).

Understandability deals with evaluating the effort required for understanding how to use the API for achieving a desired functionality. This dimension encompasses aspects such as

Table 1 This table shows the relation of Clarke's dimensions of API usability with CDs dimensions and illustrates the meaning of each of these dimensions for developers. As it can be seen, Clarke's dimensions are, in all cases, more intuitive and simpler to understand than the original CDs dimensions

Clarke's dimension	Main related CDs dimensions	Assertion
Understandability	Closeness of mapping Hidden dependencies Hard mental operations	The higher, the simpler the process of learning and understanding the API is
Abstraction	Abstraction	The higher, the less the low-level details developers need to manage for creating their applications
Expressiveness	Role expressiveness Visibility Consistency Premature commitment Provisionality Error proneness	The higher, the simpler is the process of translating application requirements into code
Reusability	Viscosity Diffuseness	The higher, the simpler the maintenance and evolution of applications are
Learnability	Progressive evaluation	The higher, the more incremental and progressive the API learning process is

whether the API names are descriptive and the relation among API types and constructs are clear and unambiguous. This relates to the base CDs dimension called *closeness of mapping*. It also includes the ability of the API to avoid developers to manage hidden information not explicitly represented in the API, which is called *hidden dependencies* in terms of the base CDs dimensions. In addition, the base CDs dimension called *hard mental operations* also affects understandability. In brief, this dimension addresses how simple is to access API features through object creation, primitive invocations or other means.

Abstraction, which is itself a base CDs dimension, relates to the ability of the API to guarantee that programmers can use the API proficiency without requiring specific knowledge or assumptions in relation to its implementation details. Abstractions should match the conventions and practices of programmers, without being elegantly abstract at the expense of understandability or other practical concerns. Abstraction is typically correlated with the degree of comfort developers feel when using the API. Summarizing with a slogan, this research question asks whether the API "makes simple things simple, and complex things possible".

Expressiveness can be seen as the ability of inferring readily the purpose of an entity. This is related to the base CD dimension called *role-expressiveness*. Expressiveness is also related to how easy is for the programmer to build her code without needing to assume any specific cognitive model about API use. Intuitively, code written using expressive APIs tend to be simpler to read and transforming requirements into code is typically more efficient in expressive APIs. In terms of base CDs dimensions these properties are related to *visibility* and *consistency*. Moreover, expressive APIs do not impose constraints neither in the order or creation nor in the definiteness of the components comprising the code, which is related to the CDs dimensions called *premature commitment* and *provisionality*. We also consider the CDs base dimension called *error-proneness* to be part of the expressiveness properties of our API.

Reusability determines whether the client code is maintainable and extensible. In particular, this dimension addresses the typical concern on how hard is to modify pre-existing code and adapt it to slightly, extended or more general requirements. The main related base CDs dimensions is *viscosity*, understood as resistance to change, but it also involves other base dimensions such as *diffuseness* (i.e. the verbosity of the notation).

Learnability address the ability of the API learning process to be incremental. Learnable APIs enable developers to understand APIs in a gradual way without requiring initial disproportionate efforts, which is related to the CDs base dimension called *progressive evaluation*. Learnability also deals with whether performing a certain programming task using the API has a positive impact on performing other related but different tasks. This dimension might have some overlap with understandability, but emphasizes specifically the learning process rather than its practical outcomes.

2.4 Contributions of this paper: the RTC Media API requirements

APIs are always designed for satisfying requirements that are implicitly or explicitly assumed by the designer. The creation of the API proposed in this paper, that we call not surprisingly the RTC Media API, was founded also on a set of commonly accepted implicit requirements [15] plus a number of explicit ones. Among the former we have simplicity, usability, security, self-documenting and consistency. The latter were identified in the course of several large research projects devoted to RTC media [30, 52] as essential needs that should be provided by any modern RTC media API but, as discussed above, are not available in any state-of-the-art

technology. The creation of an API complying with these requirements, and the validation of its usability properties basing on the CDs framework, are the main contributions of this paper. Among the above mentioned requirements, we may include the following:

Seamless API extensibility through custom modules

We want developers to be able to plug additional capabilities to the API (e.g. processing algorithms, protocols, etc.) and to enable their consumption as if they were native API capabilities (i.e. without requiring different syntax or language constructs.) The mechanism we require for this is based on modules in the sense that every extension takes the form of a module artifact (e.g. a `.jar` file in the Java language, a `.js` file in JavaScript language, etc.) and that developers may plug the modules they wish at development time without requiring any further modification or configuration. Remark that, for the reasons specified in sections above, JSR 309 does not comply with this requirement.

Adaptation to WWW technologies and methodologies

This requirement has two aspects. The first, and most important, is the need of our API to be adapted to novel RTC WWW technologies and very particularly to WebRTC [44]. The WebRTC architecture, based on heavy use of RTP bundle [43] and RTCP demultiplexing mechanisms [56] and requiring complex ICE [62] management techniques such as Trickle ICE [42], makes complex to comply with this requirement. Also as specified in sections above, JSR 309 is not compatible with this as the `NetworkConnection` is based on plain RTP. The second, is the need of the API to adapt to the typical WWW three tier development model [73]. This means that the RTC Media API should be usable for WWW developers with their common development, deployment and debugging techniques and tools. To some extent, this means that the RTC Media API should be perceived by WWW developers as any other of the APIs consumed in the application logic, such as database APIs or ESB (Enterprise Service Bus) APIs.

Full abstraction of media details (i.e. codecs and protocols)

Media representation and transport technologies are complex and require specialized knowledge that is not typically available for common developers. For maximizing productivity and minimizing development and debugging complexity the RTC Media API should hide all the low level details of such technologies through the appropriate abstractions. In doing so, these abstractions must maintain the appropriate expressiveness enabling the API semantics to provide to developers the ability of performing the required operations onto protocols and formats including payloading, depayloading, decoding, encoding, re-scaling, etc.

Programming language agnostic

In today's Internet, developers use a multiplicity of programming languages for creating their applications. In fact, the majority of applications are called "polyglot" because they use different languages. The specific choice depends on factors such as the previous experience, the personal preferences, the tasks to be accomplished, the target platform or the required scalability. In this context, tying developers to a specific programming language may be perceived as inflexible and unfriendly. For this reason, the RTC Media API needs to be language agnostic and adapt to the most common programming languages used nowadays. Of course, the specific syntax of the API calls may differ depending on language specificities. However, this requirement indicates that, somehow, the constructs, basic mechanisms and programming experience needs to be the

same across different languages. This means, for example, that a developer having the appropriate expertise for creating applications with a Java RTC Media API implementation should be able of doing so with a JavaScript implementation as long as the subtleties of the two languages are known.

RTC media topology agnostic

One of the main objectives of RTC Media Servers is to provide group communication capabilities to applications. Due to this, any useful RTC media API must consider this as a central aspect of its design by exposing the appropriate constructs for group communications. When looking to how RTC group communications are technically implemented, we can notice that they are based on a set of well-known RTP interconnecting topologies [80] among which the most common ones are Media Mixing Mixers (MMM), Media Switching Mixers (MSM) and Selective Forwarding Units (SFU). In short, MMMs are based on the principle of composing a single output media stream out of N input media streams, so that the final composite stream represents the addition of the N input streams. MMMs require decoding of the N input streams, the generation of the composite (e.g. linear adding in audio or matrix layout for video) and encoding to generate the output stream. Due to the performance cost of these operations MMM do not scale nicely. On the other hand, MSMs and SFUs do not perform any heavyweight processing and they just forward and route N incoming streams to M outgoing streams, reason why they have better scalability properties. Their only difference is that MSMs enable the N to M mapping to change dynamically while on SFUs it is static and the only possible operation is switching on/off forwarding on any of the output M streams.

Understanding the differences and appropriate usage scenarios of these topologies is complex and a source of extra complexity for application developers. Due to this, we include a requirement for our RTC Media API to transparently manage all the subtleties of this problem so that the most appropriate solution is provided transparently by the API. Remark that JSR 309 also tried to comply with this requirement through the "Joinable" mechanism making possible for developers to establish topologies just by joining sources with sinks. However, as explained above, both JSR 309, and equivalently JSR 79, are only compatible with MMM topologies and cannot manage the, by the way most popular, MSM or SFU models.

Advanced media QoS information gathering

QoS is critical in multimedia services. Some milliseconds of latency or jitter can be the difference between successful and unsuccessful applications [77]. For this reason, RTC media developers need to have the appropriate instrumentation mechanisms enabling seamless debugging, monitoring and optimization of applications. These requirements guarantees that our RTC Media API developers are able to access advanced QoS metrics of the streams including relevant information such as packet loss, bandwidth, latency or jitter. Remark that none of the above mentioned RTC media server APIs, including the JSR 309, provide this kind of capability.

Compatibility with advanced media processing capabilities

So far, most RTC media technologies and APIs have been concentrated on the problem of transport (i.e. taking media information on one place and moving it to other places.) This happened because the most prevalent use case for RTC is

person-to-person communications, where end-users expect from technology to eliminate distance barriers (i.e. to maintain a conversation as if it were face-to-face.) However, during the last decade novel use cases involving person-to-machine and machine-to-machine communications are gaining popularity in different verticals such as video surveillance, smart cities, smart environments, etc. In all these verticals, going beyond plain transport is a relevant requirement. As an example, the number of low latency RTC video applications being used in security scenarios is skyrocketing. In all these applications the ability to integrate Video Content Analysis (VCA) capabilities through different types of computer vision algorithms is an unavoidable requirement [37]. In addition, modern media applications in areas such as gaming or entertainment complement VCA with another trending technology: Augmented Reality (AR), which is also having high demand from users [84]. As a result, we include our RTC Media API to provide full compatibility with these advanced processing techniques enabling their seamless integration and use.

Context awareness

In RTC media services, as in other types of services, context is becoming a relevant ingredient for providing added value to applications [1]. Context is somehow an ambiguous concept for which there is not yet a formal definition. However, most authors accept context as any kind of information that can be used for characterizing the situation of an entity [22]. The OMA (Open Mobile Alliance) has generated a formal definition of context through the NGSI standard [10] as a set of attributes that can be associated to an entity. When working with RTC media, the entity is most typically a RTC media session (e.g. a media call).

Considering this context definition, this requirement means that our RTC media API needs to be capable of consuming context for customizing and adapting end-user experience but, most important, it needs to be capable of extracting context attributes from the media communication itself. In other words, the part of the context dealing with the media itself (i.e. what the media content is and what it represents at any time) needs to be manageable by the proposed API.

Adapted to multisensory multimedia

Traditionally RTC media has referred to simple audiovisual streams comprising typically one video track and one or two (i.e. stereo) audio tracks. However, modern trends and technologies extend this to a new multisensory notion [55], where multisensory streams may comprise several audio and video tracks (e.g. Multi-view and 3D video) but may also enable the integration of additional sensor information beyond cameras and microphones (e.g. thermometers, accelerometers, etc.) [70]. Hence, we establish a requirement for our RTC Media API to be capable of managing such multisensory multimedia in as seamless and natural way.

Adaptation to cloud media servers

Cloud computing is permeating in all IT domains, including multimedia, as the de-facto standard for system deployment and management [85]. This trend is also permeating into the RTC media server arena, reason why we need to consider it in the definition of our API. Adapting the RTC Media API to cloud environments basically means to make it compatible with how a PaaS (Platform as a Service) media server works [76]. In other words, our API needs to be compatible with a new notion of distributed media server, which in opposition with traditional

monolithic media servers, is distributed through a cloud environment and can elastically scale to adapt to end-users generated load.

3 Description of the proposed API: the RTC Media API

3.1 API specification

3.1.1 MediaObjects: MediaElements and MediaPipelines

Before providing a formal description of the RTC Media API, which is probably too harsh, let's introduce some simple initial concepts that might be helpful for the understanding of the basic mechanisms and philosophy behind our API. The RTC Media API is built on top of an object oriented model where the root of the inheritance hierarchy is the `MediaObject`. The `MediaObject` is only a holder providing utility members (it is abstract and cannot be instantiated). The two main types inheriting from `MediaObject` are `MediaElement` and `MediaPipeline`.

The `MediaElement` is the main abstraction of the RTC Media API. Intuitively, a `MediaElement` can be seen as a black box implementing a specific media capability. In general, `MediaElements` receive media streams through sinks, send media streams through sources and, in the middle, do "something" with the media. There are two main subclasses of `MediaElements`: Endpoints and Filters. An Endpoint is always a `MediaElement` with the ability of communicating media with the external world. All media streams coming into an Endpoint sink are send out of the `MediaElement` through some kind of external interface (e.g. network interface, file system interface, etc.) In the same way, all media streams received from the external interface are published and made available to other `MediaElements` through the Endpoint source. Filters, on the other hand, do not communicate media streams with the external world. Their only function is to implement some kind of media processing. This can be simple transport (e.g. a pass-through filter) or may involve complex processing algorithms including computer vision or augmented reality.

`MediaElements` can be connected among each other by means of a `connect` primitive. When a `MediaElement` (let's call it A) is connected to other `MediaElement` (say B), the media streams available at A's source are feed to B's sink. The connectivity of `MediaElements` works following quite intuitive and natural rules. First, a `MediaElement` source can be connected to as many `MediaElement` sinks as you want (i.e. a `MediaElement` can provide media to many `MediaElements`). Second, a `MediaElement` sink can only receive media from a connected source. Hence, connecting a source to a sink that is previously connected makes that sink to first disconnect from its previous source before being connected to the new one. Hence, application developers create their media processing logic just by connecting media elements following the desired topology.

Another interesting feature of `MediaElements` is that the `connect` primitive is overloaded to provide the ability of connecting just one track of those available on a media stream. The RTC Media API distinguishes three types of tracks: AUDIO,

VIDEO and DATA. The two former correspond with the typical audio-visual component of a stream. The latter represents arbitrary sensor data whose semantics is application-dependent. The DATA component makes possible to integrate any kind of sensor data into media applications.

Just for illustration, some example of `MediaElements` follows:

- `RtpEndpoint`: it represents an Endpoint having the capability of sending and receiving media streams based on standards such as the RTP protocol [68], the AVP and AVPF RTP profiles [54, 67], and the SDP media session negotiation mechanisms [38].
- `WebRtcEndpoint`: it represents an Endpoint having the capability of sending and receiving WebRTC streams complying with the appropriate standards and drafts [5].
- `PlayerEndpoint`: it represents an Endpoint with the ability of reading streams from different sources, such as a file system, an HTTP resource or RTSP [66].
- `RecorderEndpoint`: it represents an Endpoint with the ability of storing media out of the pipeline, typically on the media server file system or in a media repository through HTTP.
- `FaceOverlayFilter`: it consists of a Filter using the Haar [49] computer vision algorithm for detecting faces on a stream and overlying on top of them images with customized scales and offsets.

`MediaPipelines`, in turn, are just containers of `MediaElement` graphs. A `MediaPipeline` holds `MediaElements` that can connect among each other following an arbitrary and dynamic topology. `MediaElements` owned by one `MediaPipeline` cannot connect to `MediaElements` owned by another `MediaPipeline`. Hence, the `MediaPipeline` represents an isolated multimedia session from the perspective of the application.

To illustrate these concepts, let's create a simple application. This application performs a full-duplex back-to-back call between two users and records their streams into a repository. Figure 1 shows the corresponding pipeline.

This pipeline can be implemented in Java with the code shown in Table 2.

3.1.2 RTC Media API IDL specification

One of the main requirements of the RTC Media API is that it should be available in different programming languages. Due to this, RTC Media API capabilities are specified through an IDL (Interface Definition Language) which is language agnostic. From an implementation perspective that IDL is compiled later to different programming languages in order to generate the appropriate SDKs. In this way, RTC Media API capabilities are defined only once but the corresponding implementations can be generated for a variety of languages.

For simplicity, we have decided the RTC Media API IDL to be based on a JSON notation. In an RTC Media API file there are four sections: `remoteClasses`, `complexTypes`, `events` and `code`:

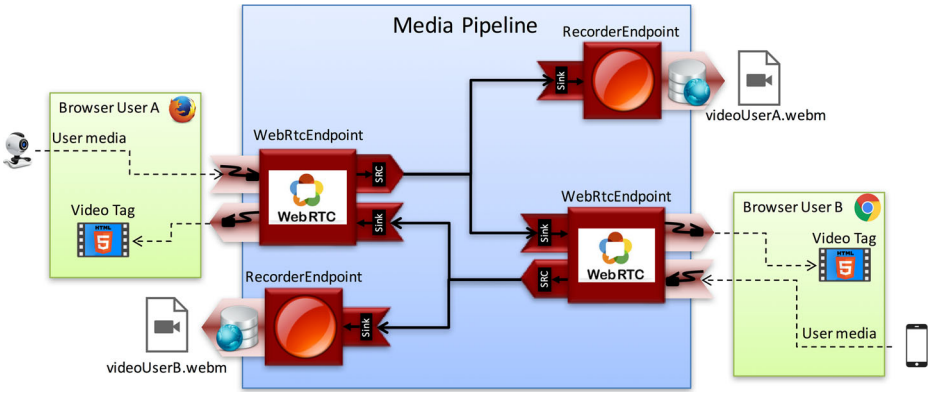

Fig. 1 Architectural diagram of an example application performing a back-to-back call between two users where their corresponding streams are recorded

- The `remoteClasses` section is used to define the interface to media server objects. We call it "remote" because these objects are remote from the perspective of the API consumer, as they are hosted into the RTC media server. For example, `PlayerEndpoint` and `ImageOverlayFilter` are defined in this section in their corresponding IDL file.
- The `complexTypes` section is used to define enumerated types and registers used by remote classes or events. For example, the enumerated type `MediaType` with possible values `AUDIO`, `DATA` or `VIDEO` may be defined in this section.

Table 2 Code snippet for developing the application specified in Fig. 1 in Java with the Kurento Client API, our reference implementation of the RTC Media API. Media from each WebRtcEndpoint is recorded in the file system of the media server (files videoUserA.webm and videoUserB.webm respectively)

```
KurentoClient rtcMediaAPI = KurentoClient.create();

MediaPipeline pipeline = rtcMediaAPI.createMediaPipeline();

WebRtcEndpoint userA = new WebRtcEndpoint.Builder(pipeline).build();
WebRtcEndpoint userB = new WebRtcEndpoint.Builder(pipeline).build();

userA.connect(userB);
userB.connect(userA);

String videoUserA = "file:///path/to/videoUserA.webm";
String videoUserB = "file:///path/to/videoUserB.webm";

RecorderEndpoint userARecorder =
    new RecorderEndpoint.Builder(pipeline,videoUserA).build();

RecorderEndpoint userBRecorder =
    new RecorderEndpoint.Builder(pipeline,videoUserB).build();

userA.connect(userARecorder);
userB.connect(userBRecorder);
```

- The `events` section is used to define the events that can be fired when using RTC Media API. For example, `EndOfStream` may be defined in the `events` section of the IDL file describing a `PlayerEndpoint`, so that the event is fired when the end of the stream is reached by the player.
- The `code` section is used to define properties to control the code generation phase for different programming languages. For example, in this section we can specify the package name in which all artifacts are generated for the Java language.

The code snippet shown in Table 3 outlines an example of an IDL file. For the sake of simplicity, we have replaced with dots (…) some parts of it.

As it can be observed, to define a remote class in Media API IDL it is mandatory to assign it a name. In addition, the following fields can be incorporated:

- **Extends:** A remote class may extend another remote class. In this case, all properties, methods and events of the superclass are available in objects of the subclass. Note that constructors of the superclass are not inherited. That is, they cannot be used to create objects of the subclass.
- **Constructor:** A remote class constructor is defined with a parameter list. Every parameter has a name and a type. The available types are: primitive types (`String`, `boolean`, `float`, `double`, `int` and `int64`), remote classes or complex types. Parameters can be defined as optional.
- **Properties:** A property is a value associated with a name. To define a remote class property it is necessary to specify its name and type. Properties can be defined as "read only".
- **Methods:** Methods are named procedures that can be invoked with or without parameters. Every parameter is specified by its name and type. Parameters can be defined as optional. A return type can be specified if the method returns a value.
- **Events:** If a remote class declares an event it means that events of this type can be fired by objects of this remote class. It depends on the target programming language how this events are processed.

Remote classes are used mainly to define the `MediaElements` of the RTC Media API. To define a new `MediaElement` the only requirement is to define a new remote class that extends the built-in `MediaElement` remote class. This super class define the properties, methods and events of all `MediaElements`. The `MediaElement` class extends the `MediaObject` class, creating the class hierarchy represented in Fig. 2

To define an event, it is mandatory to assign it a name. In addition, an event can have properties. Every property must be defined with a name and a type. In the same way than remote classes, events can also extend a parent event type inheriting all its properties.

Regarding complex types, they can have two formats: enumerated or register. If a property or `param` is defined with an enumerated complex type, it can only hold a value from the list of specified values. For example, properties based on the enumerated complex type `MediaType` of Table 3 must have the value `AUDIO`, `DATA` or `VIDEO`. On the other hand, register complex types can hold objects with several properties. For example, the register complex type `Fraction` has two `int` properties: `numerator` and `denominator`.

Table 3 Example of an RTC Media API IDL file defining a `PlayerEndpoint` media element capability

```
{
  "code": {
     "api": {
         "java": {
             "packageName": "org.kurento.client",
             ...
         }
     }
  },
  "remoteClasses": [
     {
        "name": "PlayerEndpoint",
        "extends": "UriEndpoint",
        "constructor": {
           "params": [
              { "name": "mediaPipeline", "type": "MediaPipeline" },
              { "name": "uri", "type": "String" }
           ]
        },
        "properties": [
           { "name": "position", "type": "int64" }
        ],
        "methods": [
           { "name": "play", "params": [] }
        ],
        "events": [ "EndOfStream" ]
     },
     ...
  ],
  "events": [
     {
        "name": "EndOfStream",
        "extends": "Media",
        "properties": []
     },
     ...
  ],
  "complexTypes": [
     {
        "name": "MediaType",
        "typeFormat": "ENUM",
        "values": [ "AUDIO", "DATA", "VIDEO" ]
     },
     {
        "name": "Fraction",
        "typeFormat": "REGISTER",
        "properties": [
           { "name": "numerator", "type": "int" },
           { "name": "denominator", "type": "int" }
        ]
     },
     ...
  ],
  ...
}
```

To conclude, the code section is used to specify language-dependent configurations to the IDL compiler. Every programming language has its own section to avoid collisions. For example, the Java package name of the generated code has only sense in Java, while the name of the node module has only sense in JavaScript.

3.1.3 Compiling the RTC Media API IDL

The IDL format described above makes possible to define the RTC Media API modules in a language-agnostic way. However, this needs to be translated into programming-language-

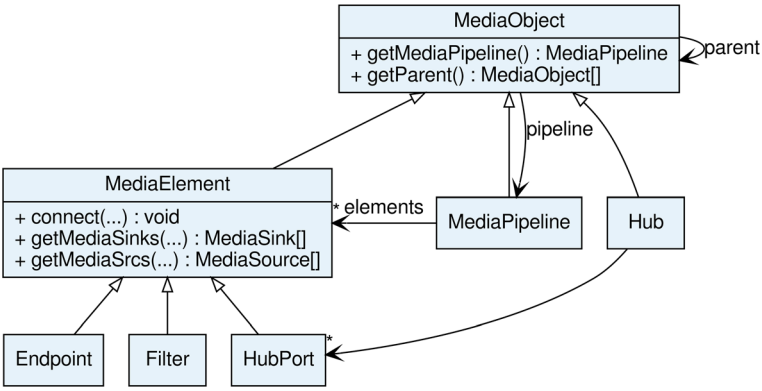

Fig. 2 `MediaObject` UML (Unified Modeling Language) inheritance diagram as defined in the RTC Media API IDL specification

dependent interfaces in order to have the real APIs to be used by application developers. The IDL compiler performs that task. Hence, we need to specify how this compilation happens so that all compiler implementations maintain compatibility on the generated code. For illustration, we have created such specification and as well as the compilers for the two most popular programming languages in the WWW: Java and JavaScript.

The Java IDL compiler works in the following way:

- Package: all artifacts (i.e. classes, interfaces and enums) are generated in the package specified in `code.api.java.packageName` section of JSON IDL file.
- Remote classes: For every remote class there are two generated artifacts: an interface and a builder class:
  - Interface: For every remote class a Java interface is generated. This interface has the remote class methods defined in the IDL. In addition, for every property, a getter method is also included. The name of the method is the string "get" followed by property name. If the property is not read only, a setter method is also generated following the same approach. Finally, for every event declared in the remote class, a method to subscribe listeners to it is generated. For example, the `PlayerEndpoint` has the event `EndOfStream` declared in the IDL so the method `String addEndOfStreamListener(Listener <EndOfStream> listener)` is generated. The complementary method to remove the subscription is also generated. `Listener<E>` is a generic interface with only one method: `onEvent(E event)`.
  - Builder class: We use the builder pattern [32] to create new remote class instances. A Builder is generated for each remote class. All mandatory `params` in the remote class constructor are mapped to parameters to the only constructor of the builder class. In this way, the compiler enforces that all mandatory parameters have a value. Optional constructor parameters are generated in builder class as fluent setter methods (prefixed with "with" instead of "set" or not prefixed if the method starts with "use"). The builder class is generated as an internal type of the above-mentioned interface to associate easily the class and the interface. The code snippet on Table 4 shows the creation of a `PlayerEndpoint` with the optional constructor parameter `useEncodedMedia` set to true.

Table 4 Code snippet showing how to instantiate a `PlayerEndpoint` in Java

```
PlayerEndpoint player =
   new PlayerEndpoint.Builder(pipeline,"video.webm").useEncodedMedia().build();
```

– Complex types: Depending on the complex type format (enum or register) the code generation is different:

  - Enumerated complex type: A Java `enum` class is generated.
  - Register complex type: A basic Java bean class is created. For every property, getter and setter methods are generated. In addition, a constructor with all properties as parameters is also generated. The code snippet in Table 5 shows a sample code using a register (`WindowParam`) as a constructor parameter of a `PointerDetectorFilter` remote class.

– Events: For each event defined in a RTC Media API IDL file a new Java class is generated. "Event" is appended to the name of the class. This class is very similar to the generated classes for register complex types. That is, a getter and a setter method is included for each property. In addition, all event classes extend from the `RaiseBaseEvent` base class. This base class contains properties for holding the source of the event (`source`) and the timestamp in which the event was generated (`timestamp`). The code snippet in Table 6 shows an example illustrating how to work with events.

When working with JavaScript IDL compilers, equivalent rules have been created:

– Package: We base on the NPM (Node Package Manager) [74] JavaScript packaging system. NPM mandates that a `package.json` file is generated. The following values are used:

  - package name: `code.api.js.nodeName`
  - package description: `code.api.js.npmDescription`

– Remote classes: For every class a new JavaScript prototype based class is generated. This class has all methods defined in the IDL file. In addition, for every property, a getter method is generated. Also setter methods are generated for non-read only properties. All generated methods have the parameters defined in the IDL plus a callback function. That callback parameter is used to implement the asynchronous execution of the method given that the API primitive may require to communicate with the RTC media server and, hence, cannot be synchronous. To create an object from a remote class, a factory method called `create` available at the pipeline object needs to be executed. The first parameter of the

Table 5 Example illustrating how to instantiate a register complex type (`WindowParam`) as a Java bean

```
PointerDetectorFilter pointer = new PointerDetectorFilter.Builder(pipeline,
     new WindowParam(5, 5, 30, 30)).build();
```

Table 6 Example illustrating how to work with events both in Java 7 and Java 8

```
PlayerEndpoint player =
   new PlayerEndpoint.Builder(pipeline,"video.webm").useEncodedMedia().build();

//Java 7
player.addEndOfStreamListener(new EventListener<EndOfStreamEvent>() {
  public void onEvent(EndOfStreamEvent e) {
     System.out.print("EOS Player "+e.getSource().getId());
  }
});

//Java 8
player.addEndOfStreamListener(
  e -> System.out.print("EOS Player "+e.getSource().getId())
);
```

method is the name of the remote class to create as a string. The second is an options bag used when constructor parameters are required. The third, and last, is the async callback to receive the new object handler or an error. The code snippet in Table 7 shows the creation of a `PlayerEndpoint` with the mandatory parameter `uri` and optional constructor parameter `useEncodedMedia` set to true. As it can be observed, media element creation is an async operation.

- Complex types: For enumerated complex types, there is no code generation. Enum values are simply strings. On the other hand, register complex type are generated as JavaScript classes based on prototypes. Also, for every registered complex type a factory function is generated to allow the creation of objects. The code snippet in Table 8 shows the creation of a `PointerDetectorFilter` using a complex type `WindowParam` as parameter.
- Events: There is no classes generated for events in JavaScript. When an event is raised, a new object is created and populated with all relevant information as properties. In Table 9 a Player is created and a listener is registered for its event `EndOfStream`. When this event is generated, a function is executed with the event as parameter. This event parameter can be used to obtain the relevant information such as timestamp, source of the event, etc.

3.1.4 Creation and deletion of media capabilities

Java and JavaScript have notable differences in media object creation. This is due to the differences in the type safety of both languages. Java is strongly typed. Hence, it is important that the compiler enforces typing in several contexts: mandatory parameters, optional parameters, media object signature, etc. On the other hand, in JavaScript there is no type checking until runtime and this is why we do not enforce any kind of protection.

Table 7 Code snippet showing how to instantiate a `PlayerEndpoint` in JavaScript

```
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true},
 function(error, player) {
    if (error) return console.error(error);

    //use player here
});
```

Table 8 Example illustrating how to instantiate a register complex type (`WindowParam`) as a JavaScript object

```
var options = {
 "windowParam":WindowParam({
    "topRightCornerX":5,
    "topRightCornerY":5,
       "height":30,
    "width":30
  })
};
pipeline.create("PointerDetectorFilter",options,
 function(error, pointer) {
    if (error) return console.error(error);

});
```

The releasing of media objects is simple. We consider that a media object is released when the `release` method is invoked. In Java, the `release` method can be executed in a synchronous way, blocking the invoking thread until a response is received. That response can be successful or fail. In the latter case, an exception is thrown. In JavaScript, it is executed asynchronously. For this reason, a callback parameter is necessary so that failures can be notified. The following piece of code shows the release of a media object in Java and JavaScript (Table 10).

3.1.5 Synchronous and asynchronous programming models in the RTC Media API

One of the most critical design decisions when designing APIs is how they behave in relation to threads. When performing I/O (Input/Output) operations, there is a common agreement that asynchronous APIs are more scalable than synchronous ones [8]. Synchronous I/O typically block threads until a response is received or a timeout is reached. Hence, given that there is a practical limit on the number of threads in a system (mainly due to memory constraints), synchronous API models tend to generate thread starvation and decrease performance due to the overload they generate into the operating system task scheduler. To solve this problem, many modern APIs provide asynchronous I/O operations. In this case, the thread executing the I/O is not blocked after the invocation and can be used to execute other tasks. However, asynchronous APIs are more complex to use and are susceptible of suffering a problem called "callback hell" [45]. This is a well-known problem that arises when asynchronous calls are invoked in the callbacks of another asynchronous calls, creating a deep nesting of callbacks.

When we designed the RTC Media API we decided to provide developers the flexibility of choosing between the synchronous and the asynchronous models so that they were not limited by any of their corresponding drawbacks. Due to this decision, our Java IDL compiler generates two methods for each I/O operation: the synchronous and the asynchronous

Table 9 Example illustrating how to work with events in JavaScript

```
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true},
 function(error, player) {
    if (error) return console.error(error);

    player.on("EndOfStream", function(e){
       console.log("EOS player "+e.source.id)
    });
});
```

Table 10 Code snippets showing how to release a `PlayerEndpoint` both in Java and JavaScript

```
//Media object release in JavaScript
player.release(function(error){
  if (error) return console.error(error);
});

//Media object release in Java
player.release();
```

versions. Synchronous methods block the calling thread until a response is received. This can be appreciated, for example, in the code snippet shown in Table 7. After that, the execution continues. The asynchronous primitives, in turn, include a continuation as last parameter, that is, an object that have two methods: `onSuccess`, that is executed when the response is received, and `onError`, that is executed when an error or timeout occurs. The code snippet in Table 11 shows an example.

When going to JavaScript, things are more complex. Due to the characteristics of the JavaScript language both in the browser and in Node.js [74], only asynchronous I/O operations are possible. Due to this, and as it can be seen in the code snippets shown in Table 12, our IDL compiler includes a callback as the last parameter that is executed asynchronously when the operation is resolved. However, providing only this mechanism reduces the flexibility of developers to avoid the callback hell. This is why we designed novel mechanisms for simplifying developers' work. The first one is based on Promises [45]. A Promise represents an operation that has not completed yet, but is expected to do so in the future. Hence, an asynchronous method can return a promise object instead of expect a callback as last parameter. The developer specifies the code to be executed when the promise is fulfilled, executing a method called "`then`" with the callback as parameter. Table 12 shows a

Table 11 Example illustrating the creation of a `PlayerEndpoint` using the asynchronous Java API

```
new PlayerEndpoint.Builder(pipeline, "video.webm")
    .buildAsync(new Continuation<PlayerEndpoint>() {

      @Override
      public void onSuccess(PlayerEndpoint player) throws Exception {

        player.play(new Continuation<Void>() {

          @Override
          public void onSuccess(Void result) throws Exception {
             log.info("Play started");
          }

          @Override
          public void onError(Throwable error) throws Exception {
            log.error("Exception invoking play in player", error);
          }
       });
      }

      @Override
      public void onError(Throwable error) throws Exception {
        log.error("Exception creating player", error);
      }
    });
```

Table 12 Different forms of `PlayerEndpoint` creation in JavaScript (with callbacks, with promises, and with ES6 arrow functions)

```
//Asynchronous API with callbacks
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true},
 function(error, player) {
    if (error) return console.error(error);

    player.play(function(error){
      if (error) return console.error(error);
      console.log("Play started");
    });
});

//Asynchronous API with promises
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true})
 .then(function(player){ return player.play(); })
 .then(function(){ console.log("Play started"); })
 .catch(function(err){ console.error(error); });

//Asynchronous API with promises and ES6 arrow functions
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true})
 .then(player => player.play())
 .then(() => console.log("Play started"))
 .catch(err => console.error(error));
```

JavaScript code creating a player and invoking the `play` method on it, comparing the traditional implementation based on callbacks with an implementation using promises.

Moreover, if promises are combined with generators [59], a new ES6 (ECMAScript 6) feature, the asynchronous code can look like synchronous one. For illustration, remark that Table 13 code snippet implements the same logic than the one on Table 12, but using generators. As it can be observed, the improvement on code readability is noticeable.

The next version of JavaScript, ES7, which is still under standardization, has a proposal to simplify this: the async/await keyword, which marks when a call is asynchronous but accepts synchronous API syntax. Using it, the code in Table 13 can be written as shown in Table 14 with ES7. As it can be observed, the `yield` keyword is replaced by await and the `co` function is no longer necessary.

3.1.6 RTC Media API capabilities

Once we have presented the formal aspects of the RTC Media API, we can switch to a more practical perspective and introduce its media capabilities. These capabilities comprise specific

Table 13 Creation of a `PlayerEndpoint` using generators in JavaScript. As it can be observed, the code readability improves significantly and the callback hell is fully avoided

```
co(function*(){
 try{
   var player = yield pipeline.create('PlayerEndpoint');
   yield player.play();
   console.log("Play started");
 } catch(e) {
   onError(e);
 }
})();
```

Table 14 Creation of a `PlayerEndpoint` in ES7

```
try{
   var player = await pipeline.create('PlayerEndpoint');
   await player.play()
   console.log("Play started");
} catch(e) {
  onError(e);
}
```

media objects that are made available to application developers to create their RTC media enabled applications following the above-described API guidelines. These capabilities can be grouped into two main categories: media elements, which inherit from the `MediaElement` class and manage a single media stream, and hubs, which inherit from the `Hub` class and have been specifically designed for the management of groups of streams.

As specified in sections above, media elements have two flavors: Endpoints and Filters. Endpoints are in charge of the I/O media operations in the media pipeline. Figure 3 shows the RTC Media API endpoint inheritance hierarchy, which comprises the following capabilities:

- The `WebRtcEndpoint` is an I/O endpoint that provides full-duplex WebRTC media communications compatible with the corresponding protocol standards [5]. It is important to remark, that among `WebRtcEndpoint` capabilities, the RTC Media API defines as mandatory the DataChannel support. DataChannels are a mechanism for receiving media information beyond audio and video given their ability to accommodate arbitrary sensor data that is transported in the same ICE connection than the audio and the video and, hence, may maintain synchronization with them.
- The `RtpEndpoint` is equivalent but with the plain RTP protocol.
- The `HttpPostEndpoint` is an input-only endpoint that accepts media using HTTP POST requests. This capability needs to support HTTP multipart and chunked encodings, so that it is compatible with the HTTP file upload function exposed by WWW browsers. This endpoint must support the MP4 and WebM media formats.
- The `PlayerEndpoint` is an input-only endpoint that retrieves content from the local file system, HTTP URLs or RTSP URLs and injects it into the media pipeline. This

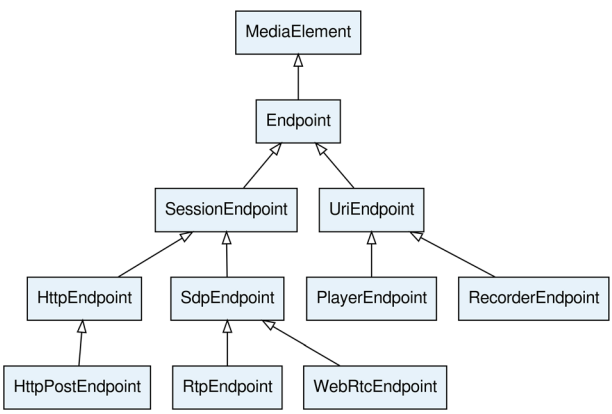

Fig. 3 UML class diagram of the Endpoints specified by the RTC Media API

endpoint must support the MP4 and WebM media formats for all input mechanisms as well as RTP/AVP/H.264 for RTSP streams.
- The `RecorderEndpoint` is an output-only endpoint that provides function to store contents in reliable mode (doesn't discard data). This endpoint may write media streams to the local file system, or to HTTP URLs using POST messages. This endpoint must support MP4 and WebM media formats.

Filters, in turn, are used for processing media streams. Filters are useful for integrating different types of capabilities such as Video Content Analysis (VCA), Augmented Reality (AR) or custom media adaptation mechanisms. The RTC Media API does not specify any kind of mandatory filter and it is let to API implementers to define their filters following the RTC Media API extensibility mechanisms.

To conclude, hubs follow the inheritance scheme depicted in Fig. 4. Hubs work in coordination with `HubPorts`: a special type of media element, which provides sinks and sources to hubs. The RTC Media API only defines as mandatory hub type the `Composite`, which implements a MMM media topology, as described in previous sections. Developing with `Composites` is simple as long as the following rules are taken into account.

- `Composites`, as all hubs, act as a factory of `HubPorts`. This means that at a `Composite` instance we can create as many `HubPorts` as we want. These `HubPorts` are media elements having sources and sinks, which makes possible to connect other media elements to them and get media into and out of the hub.
- A `Composite` mixes all streams received from its `HubPort`'s sinks and exposes the resulting mixed stream at the sources. The audio of the mixed stream obtained at a HubPort's source includes all the inputs except the one of its own `HubPort`'s sink. The video, on the other hand, combines all `HubPort`'s sinks into the resulting composite matrix.

3.1.7 Extending the API

One of the main requirements of the RTC Media API is extensibility: developers should be able to include new `MediaElements` into the API so that they maintain compatibility with other `MediaElements` defined natively by the API or by third parties. In order to support extensibility, we have created the notion of the RTC Media Module. A module is a bundle composed by:

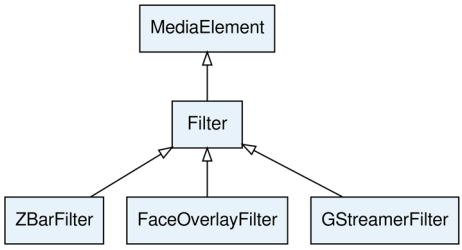

Fig. 4 UML class diagram of main Hub types in the RTC Media API

- A module definition: the MediaElement interfaces and related types defined in the RTC Media API IDL.
- The corresponding software libraries: the specific language-dependent SDK enabling developers to use the module in their software projects.

According to this, imagine that you have created a new capability in your RTC media server involving some kind of computer vision algorithm to process a video stream and mark some relevant regions on it. The details about how this is implemented are out of the scope of this paper. The point is that, for exposing this new feature through the RTC Media API, the best choice is to create a `Filter`. Without loss of generality imagine we call it `CompuVisionFilter`. This filter interface needs to be defined in a RTC Media Module Definition file, which contains the RTC Media API IDL. If we suppose that the filter requires an `int` parameter upon construction for tuning the behaviour of the algorithm and that it has a method that can be invoked at any time to enable or disable the processing (the "enable" method); the resulting module definition is the one shown in Table 15.

From this file, the Java IDL compiler should be able to generate the `CompuVisionFilter` SDK library. When generated, the filter can be incorporated into any pipeline and interoperate with any of the rest of RTC Media API capabilities. Table 16 shows a code snippet illustrating how to create an application processing a media clip with the filter and exposing the resulting stream in real-time to a WebRTC capable browser. Remark that the example requires the filter to interoperate with built-in RTC Media API capabilities such as the `PlayerEndpoint` or the `WebRtcEndpoint`.

Table 15 Example of module definition for a new Filter called `CompuVisionFilter`

```
{
  "code": {
     "api": {
         "java": {
             "packageName": "com.company.filter"
         }
     }
  },
  "remoteClasses": [
     {
        "name": "CompuVisionFilter",
        "extends": "Filter",
        "constructor": {
           "params": [
              { "name": "mediaPipeline", "type": "MediaPipeline" },
              { "name": "param", "type": "int" }
           ]
        },
        "methods": [
           {
              "name": "enable",
              "params": [{ "name" : "enabled", "type": "boolean" }]
           }
        ]
     }
  ]
}
```

Table 16 Using the filter `CompuVisionFilter` previously defined

```
import com.company.ComputVisionFilter;

...

    MediaPipeline p = rtcMediaApi.createMediaPipeline();

    WebRtcEndpoint webrtc = new WebRtcEndpoint.Builder(p).build();
    PlayerEndpoint player = new PlayerEndpoint.Builder(p,"file://...").build();

    CompuVisionFilter filter = new CompuVisionFilter.Builder(p,10).build();

    player.connect(filter);
    filter.connect(webrtc);

    //At any time…
    filter.enable(true);
```

3.1.8 Implementing the RTC Media API: the Kurento Client API

In order to implement the RTC Media API and making it expose useful capabilities to developers we just need two ingredients. The first is an RTC media server. This media server needs to expose at its northbound some kind of control interface or protocol enabling the management of RTC media capabilities in a compatible way with the semantic requirements of the RTC Media API. The details about how to create such RTC media server and control protocol are out of the scope of this paper. The second, is to implement an RTC Media IDL compiler suitable for translating the RTC Media IDL into the corresponding programming-language-dependent SDKs. Remark that this compiler is not protocol agnostic, in the sense that it needs to translate the RTC Media API invocations into the appropriate messages of the RTC media server control protocol. In other words, each specific media server control protocol needs to have a custom IDL compiler.

In the context of the Kurento open source software project (http://www.kurento.org), we have created an example implementation of the RTC Media API. This implementation follows the architectural scheme depicted in Fig. 5. As it can be observed, our implementation provides the two above-mentioned ingredients. The Kurento Media Server plays the role of the RTC media server. Observe that Kurento Media Server exposes its capabilities though a JSON-RPC over WebSocket control protocol called the Kurento Control Protocol. This protocol has been designed to be compatible with the RTC Media API semantics. In addition, we have created an RTC Media IDL compiler capable of translating the IDL specifications into the appropriate API implementations both in Java and JavaScript. The resulting programming-language-dependent SDKs are called Kurento Client APIs in the Kurento jargon. Remark that the Kurento Client API is just a specific implementation of the RTC Media API suitable to interoperate with the Kurento Media Server.

Kurento is a complex technological stack and the interested reader can check the community documentation (http://www.kurento.org/documentation) and source code repositories (https://github.com/kurento) to have full information about the project. For the objectives of this paper, the interesting aspects are twofold. First, the Kurento Client API provides a full implementation of the RTC Media API as described in this paper. This implementation is specified through the RTC Media IDL and stored in files with the `.kmd.json` extension

(KMD for Kurento Module Description).[1] Second, the Kurento project provides a bunch of extensions to the RTC Media API in the form of custom filters, hubs and endpoints.[2] These extensions have been created following the RTC Media Module mechanism described above. Just for illustration, some of these extensions are presented here:

- The `ZBarFilter` filter detects QR and bar codes in video streams. When a code is found, the filter publishes a `CodeFoundEvent`. Application developers can add a listener to this event to execute some logic.
- The `ImageOverlayFilter` filter inserts still images in the video stream. The filter makes possible to select the position, scaling and rotation coordinates of the image.
- The `FaceOverlayFilter` filter detects faces in a video stream and overlays custom images onto the face coordinates. The filter makes possible to select specific scaling and offsets for the image position.
- `CrowdDetectorFilter` filter implements a computer vision algorithm suitable for detecting crowds of people into video streams. The level of crowdedness is published through a custom event that contains information about the direction and speed of movement of the crowd.
- `PlateDetectorFilter` filter detects European car plates and publishes the detected plate number as a custom event.
- `AugmentedRealityFilter` filter wraps the Alvar library [3] to provide marker and markerless Augmented Reality capabilities.
- `AlphaBlending` hub is a special type of hub that makes possible to mix different video streams using alpha transparency. This hub is useful for producing chroma blended videos in real time.

Thanks to all these capabilities, the Kurento software stack has been used for creating hundreds of applications combining different types of features which include WebRTC and RTP transports, media recording, Video Content Analysis, or Augmented Reality.[3] All in all, Kurento provides a full working test-bed where the RTC Media API constructs described in this paper can be used, evaluated and improved.

[1] Latest versions of the IDL files specifying the Kurento RTC Media API in the *kmd.json* format can be found at the Kurento GitHub repository at the following locations:

- https://github.com/Kurento/kms-core/blob/master/src/server/interface
- https://github.com/Kurento/kms-elements/tree/master/src/server/interface

[2] Latest versions of the IDL files specifying the Kurento RTC Media API extensions in the *kmd.json* format can be found at the Kurento GitHub repository in different locations, including the following:

- https://github.com/Kurento/kms-filters/tree/master/src/server/interface
- https://github.com/Kurento/kms-crowddetector/tree/master/src/server/interface
- https://github.com/Kurento/kms-platedetector/tree/master/src/server/interface

[3] Demos showing different Kurento applications created using the Kurento RTC Media API implementations are accessible at the Kurento YouTube channel: https://www.youtube.com/channel/UCFtGhWYqahVlzMgGNtEmKug

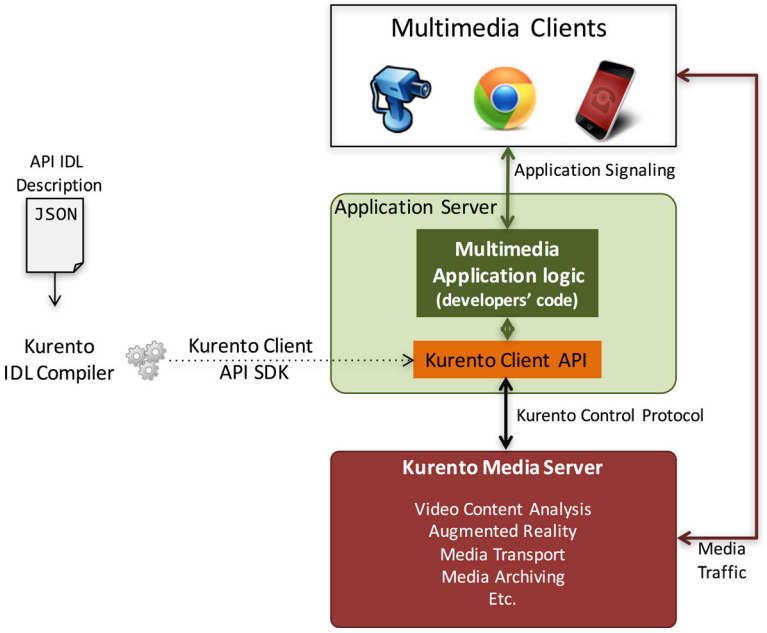

Fig. 5 Architecture of a Kurento application. As it can be observed, the Kurento IDL compiler generates the Java and JavaScript Kurento Client SDKs from the Kurento API IDL following the RTC Media API specifications described in this paper. Using these SDKs, developers can create their applications following the traditional three tiered WWW architecture just using the Kurento Client API as any other of their service APIs. The Kurento Client API implementation provides semantics to the API invocations by issuing the appropriate Kurento Control Protocol messages

3.1.9 Matching the RTC Media API requirements

To conclude with the RTC Media API presentation, we would like to come back to the list of requirements exposed in Section 2.4 to validate that they are fulfilled:

- **Seamless API extensibility through custom modules:** As it can be seen in Section 3.1.7, the RTC Media API can be extended in a seamless way by using the RTC Media Module mechanism, which provides full flexibility and no restrictions other than extending from the base RTC Media API classes.
- **Adaptation to WWW technologies and methodologies:** As shown in Section 3.1.8, RTC Media API implementations fully comply with the traditional WWW three tiered development model and enable developers to create applications leveraging novel WWW RTC media technologies such as WebRTC in a seamless and direct way.
- **Full abstraction of media details (i.e. codecs and protocols):** As it can be appreciated in the discussions and code examples in Sections 3.1.1, 3.1.3 and 3.1.7, the RTC Media API `connect` primitive exposed by all `MediaElements` makes possible to fully abstract codecs, protocols and formats. In all our examples there are no explicit references to codecs or formats, but many of the examples require specific transcodings to work. This is due to the fact the semantics of the `connect` primitive mandates the underlying media server capabilities to perform all the appropriate adaptations in a fully transparent way.

- **Programming language agnostic:** Discussions in Section 3.1.2 demonstrate the full agnosticism of the RTC Media API IDL in relation to programming languages. The only requirement for supporting a given programming language is to specify how the IDL is transformed into it and to implement the appropriate compiler following that specification. In Section 3.1.3 and 3.1.8 we provide such specifications and describe their implementations in Java and JavaScript in the context of the Kurento open source software project.
- **RTC media topology agnostic:** Following the discussions on Sections 3.1.1 and 3.1.6 one can appreciate that the RTC Media API makes possible to interconnect media elements following arbitrary and dynamic topologies thanks to the `connect` primitive. This means that developers do not need to be aware of the low level details of MMM, MSM or SFU technologies: they just need to interconnect their endpoints, filters and hubs accordingly to their needs. The RTC Media API semantics shall translate these interconnections into the appropriate low level mechanisms using MMMs, MSMs or SFUs in a fully transparent way.
- **Advanced media QoS information gathering:** As it can be observed in the discussions in Section 3.1.2, the RTC Media API IDL does not restrict in any way the information a media object may expose through its properties and methods. We have leveraged such flexibility for creating QoS metrics gathering mechanisms in all endpoints based on the RTP protocol. In particular, the `WebRtcEndpoint` exposes primitives fully compliant with the standard WebRTC "inboundrtp" and "outboundrtp" stats [83].
- **Compatibility with advanced media processing capabilities:** As it can be observed in the discussions in Section 3.1.8, the Kurento software project has created a bunch of modules providing advanced capabilities such as Video Content Analysis, Augmented Reality, Computer Vision, etc. This demonstrates the ability of the RTC Media API `Filter` concept to hold all kind of extensions for advanced media processing.
- **Context awareness:** The notion of context emerges in quite a seamless way from the discussions on Section 3.1.2. As it can be observed, the RTC Media API event mechanism makes possible for media capabilities to publish events to applications. These events may contain semantic information about the media content itself as shown, for example, in the `CrowdDetectorFilter` mentioned in Section 3.1.8. Hence, creating multimedia context-aware applications is straightforward: the application logic just needs to subscribe to the relevant events and publish them into a context database based on NGSI or any other equivalent standard.
- **Adapted to multisensory multimedia:** The RTC Media API can manage seamlessly arbitrary sensor data beyond audio and video. This can be achieved through the combination of two features. The first is the support for `DataChannel` that, as specified in Section 3.1.6, makes possible for any media pipeline to exchange multisensory multimedia with the external world using the WebRTC protocol stack. The second is the fact that, as described in Section 3.1.1, all streams exchanges among `MediaElements` may have a DATA track. In particular, any information received using `DataChannels` into a `WebRtcEndpoint` is published to the rest of the pipeline through the endpoint's source DATA track. In the same way, any information received through the DATA track at a `WebRtcEndpoint`'s sink is sent to the network using `DataChannels`. As the `MediaElement` interface enables all the information received through the DATA to be used by the element internal logic, this mechanism makes possible, for example, to create Augmented Reality filters that leverage sensor information for customizing the augmentation logic.

– **Adaptation to cloud media servers:** As it can be observed in the discussion of Section 3.1.2, the RTC Media API does not specify how media pipelines are placed into media server instances. The API implementer has full freedom for selecting how newly created media pipelines are scheduled. This flexibility can be leveraged by API implementers to adapt their code to all kinds of cloud architectures. For example, as shown in the code snippet in Table 2, at the Kurento Client RTC Media API Java implementation, we decided that the RTC Media API is represented by a specific class (i.e. `KurentoClient`) that is built though a static `create` factory method. This method may accept as a parameter a single IP, in which case all pipelines are instantiated into the media server listening at that IP; a list of IPs, which causes media pipelines to be round-robin distributed on the corresponding media servers; or a media server scheduling interface, which can provide arbitrary logic for scheduling media pipeline creation into media servers. It may also accept no parameters and let the developer specify the behavior in a configuration file. All this flexibility makes possible for our RTC Media API to work seamlessly in cloud clusters of Kurento Media Server instances. Just as an example, this scheme is currently used in the NUBOMEDIA [52] and FIWARE [30] clouds. The complex details on how this happens are out of the scope of this paper. The point is to remark that the RTC Media API does not constraint the API implementer in any way when adapting to complex cloud scheduling and placement logic**.**

3.2 Some real-world example applications

The RTC Media API is currently being used in the context of the Kurento open source software community by hundreds of developers for creating RTC applications. Just for illustration, we can briefly describe here two of these applications.

The first implements a functionality that we call “Crowd Detection” and is useful in Smart Cities scenarios. This application is convenient for public safety given that, when there are problems in a public space (e.g. a person having a heart-attach, a robbery, an accident, etc.) a crowd of people (i.e. a group of people who do not move or move slowly) tends to concentrate around in quite a fast way. To this aim, we use the `CrowdDetectorFilter` mentioned above. Upon reception of a video stream, the `CrowdDetectorFilter` generates events indicating the degree of crowdedness on specific areas of the scene (e.g. `NONE`, `LOW`, `MEDIUM`, `HIGH`). Assuming that we have video feeds coming from RTSP street cameras, and that, upon alarm, policemen receive the streams in a WebRTC capable device, the required pipeline for our application is the one shown in Figs. 6 and 7. The application logic can just subscribe to the crowdedness level and, when a `HIGH` level is received, the application may generate an alarm sending, for example, an instant message to a policeman device providing an URL where the camera video can be seen. The notion of multimedia context awareness, as mentioned above, emerges in quite a natural way as the crowdedness can be seen as a context attribute characterizing the camera’s video status. In the same way, the alarm generation logic may be influenced by other context attributes (e.g. the closest policemen, the time of day, etc.)

The second application shows how to implement a typical WebRTC group videoconferencing service. We call this the “room application” because every group of communicating users is connected to a virtual shared space called a “room”. As shown in Fig. 8 the media pipeline for this application can be created in a seamless way using the RTC Media API. Each participant maintains a WebRTC session with the media server through a

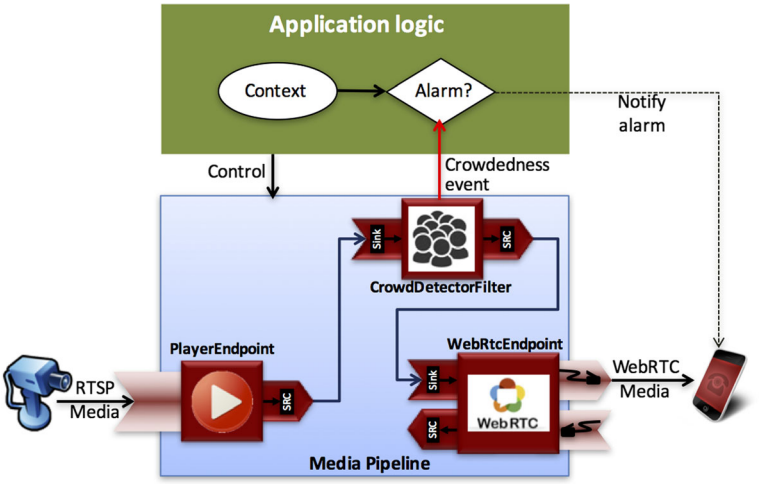

Fig. 6 Conceptual representation of the crowd detector application pipeline. An RTSP camera injects a street video scene into the pipeline through a `PlayerEndpoint` Media Element. This passes the media to the `CrowdDetectorFilter`. The application code can then subscribe to the crowdedness events and generate alarms accordingly. Upon reception of an alarm the application logic can take different actions such as, for example, sending an instant message to a policeman who can connect to a `WebRtcEndpoint` service to the processed stream. Given the RTC Media API flexibility, further actions could be taken for dynamically modifying the pipeline in order to, for example, recording the stream through a `RecorderEndpoint` whenever the crowdedness level is over a threshold

`WebRtcEndpoint`. All the WebRTC incoming streams are connected to a MMM Hub Media Element called `Composite`. This `Composite` creates an outgoing media flow mixing all the incoming video and audio streams following the scheme described above in this paper. This resulting flow is fed to the sinks of the `WebRtcEndpoints`, which send it back to the participants' browsers. Figure 9 shows the resulting user interface of this application.

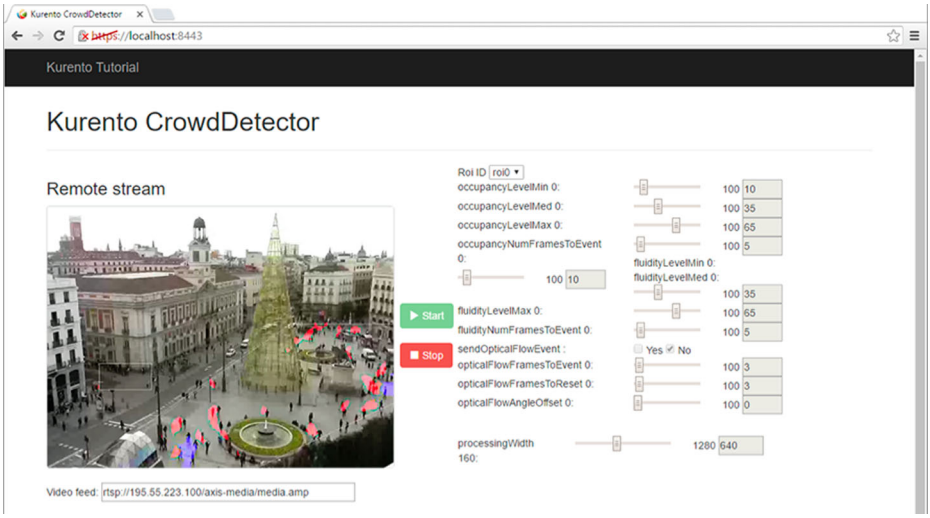

Fig. 7 Web user interface of "Crowd detector" application. It shows a real-time video stream from IP Camera augmented with colors, which depend on the detected level of crowdedness

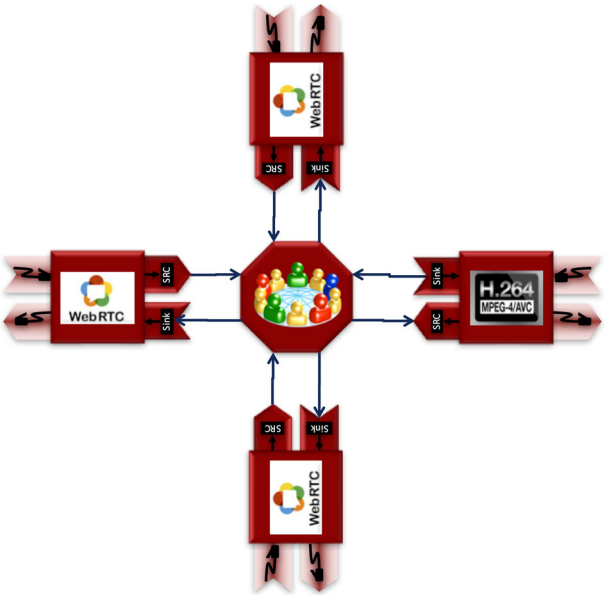

Fig. 8 Media pipeline associated to the "room" application assuming the presence of 4 participants into the videoconferencing session. All participants connect a WebRTC feed to the media server through a `WebRtcEndpoint`. The WebRTC received streams are mixed into a `Composite` Hub that generates a single output stream. This is fed to the `WebRtcEndpoint` sinks, which in turn send it back to the end-users' browsers

4 API evaluation

In the sections above we have presented the RTC Media API and we have introduced a specific implementation of it. The rest of this paper is devoted to describing a study we performed for evaluating the RTC Media API usability in the context of the Kurento open source software community.

4.1 Study design: methodology and hypotheses

For performing the study of the RTC Media API usability we decided to follow the methodology presented in Section 2.3 above. Hence, our study is based on a questionnaire that

Fig. 9 Web user interface of the "room" application. As it can be observed, all the incoming streams are mixed into a grid by the Composite that detect the dominant speaker and depicts it bigger

evaluates developers' experiences in terms of Clarke's dimensions in a Likert scale. The final objective of the study is to validate the main hypothesis of this paper, namely:

H1: The RTC Media API herein presented enables the creation of rich RTC applications consuming advanced media capabilities with full abstraction of low-level details and in a programming language agnostic way.

The term *creation* must be understood here in a wide sense to cover all the activities developers perform in relation to the API. As also stated in Section 2.3, these activities include exploratory learning (i.e. the process of learning how to use the API), exploratory design (i.e. the process of creating application code consuming the API) and maintenance (i.e. the process of debugging and evolving the application code after the application has been first created).

Following this, the validation of this hypothesis requires finding answers to questions such as the following:

- Do developers feel that the API can be learnt in a simple, incremental and seamless way?
- Do developers feel that the API is helpful for the creation of clean and error-free application code without needing to manage low-level complexities?
- Do developers feel that maintaining and evolving code consuming the API is smooth and uncomplicated?
- Do developers have the same perception of the API usability with independence on their demographic characteristics (i.e. years of experience, nationality, etc.) and on the types of applications they create?
- Do developers have the same perception of the API usability with independence on their programming language?

4.1.1 Research questionnaire

Following the methodology presented in Section 2.3 above, we created a questionnaire comprising 28 assertions which characterize the 5 target dimensions (understandability, abstraction, expressiveness, reusability, and learnability). For every assertion, users provide their degree of agreement or disagreement in a Likert scale from 1 (I fully disagree) to 5 (I fully agree). For assessing the internal consistency of the data, and following common practices in psychological research, some of the assertions are generated in negative terms. For example, if a respondent expressed agreement with the claim "I feel this API is simple" and disagreement with "I find it's hard programming with this API" this would be an indication of internal consistency. We call these N-assertions. For the statistical analysis, answers of N-assertions are inverted (i.e. 1 is transformed into 5, 2 into 4, 4 into 2 and 5 into 1), so that consistency and coherence is maintained. The complete list of assertions is provided in Table 17.

In addition to this, and with the objective of characterizing different aspects of participants, a number of questions were included for profiling demographic data and for evaluating their degree of experience. These questions are shown on Table 18. As it can be observed, most of the questions are self-explanatory with these exceptions:

Table 17 Research questionnaire used for evaluating developer's perception of API usability on the 5 target dimensions. Every assertion in the questionnaire is identified with a unique ID for further reference (e.g. U.1 refers to the first question of the Understandability dimension). Assertions generated in negative terms (i.e. N-assertions) start with an (N) mark. These assertions are useful for evaluating the consistency of the research. Participants are asked to provide their degree of agreement with every assertion in a scale from 1 (I fully disagree) to 5 (I fully agree). For the statistical analysis, N-assertions are inverted so that the coherence of the questionnaire is maintained

Dimension	ID	Assertion
Understandability	U.1	Kurento APIs are, in general, easy to understand
	U.2	In Kurento APIs object names are descriptive and unambiguous
	U.3	(N) I need to keep track of hidden information not represented by the APIs to create my applications
	U.4	(N) Kurento APIs are obscure and it takes a huge effort to use them, even for creating simple applications
	U.5	Kurento API objects, types, and primitives represent appropriately the underlying media-related concepts
	U.6	I understand the difference between a Media Pipeline and a Media Element
Abstraction	A.1	Kurento APIs make simple to create applications without needing to worry about the low level media details
	A.2	(N) I needed to adapt the API (e.g. inheriting, overriding, etc.) for having it meet my needs
	A.3	(N) It's necessary to understand how codecs and protocols work for being able to use Kurento APIs
	A.4	I like writing applications with Kurento APIs. I'm familiar with their programming model
	A.5	I feel appealing and attractive the general approach of Kurento APIs
	A.6	Creating simple applications with Kurento APIs is simple. Creating complex applications is possible
Expressiveness	E.1	Developing with Kurento fully matches the expectations I had
	E.2	I can translate my media application requirements into code in an easy way
	E.3	Reading an application code, I can understand what the application is doing in a simple way
	E.4	After creating an application, I can explain seamlessly to other people what I have done in terms of media elements and their interconnections
	E.5	(N) There are missing features in Kurento APIs that make not possible to implement interesting applications
	E.6	(N) Programming with Kurento APIs is error prone. You need to take into consideration a lot of details for having an application working
Reusability	R.1	(N) Creating applications requires too long and verbose code specifying too many things
	R.2	(N) Adding a recording capability to an non-recording application requires modifying a lot of code
	R.3	My code using Kurento APIs can be maintained and evolved easily
	R.4	I can re-use Kurento related code in a simple way
	R.5	(N) When using Kurento APIs, there are many different ways of doing the same thing and I need to take too many decisions in the process

Table 17 (continued)

Dimension	ID	Assertion
	R.6	Adapting my Kurento-based application to new media requirements is quite simple
Learnability	L.1	I learned how to use Kurento in an incremental way, starting with simple concepts and progressing towards complex applications
	L.2	(N) Programming with Kurento requires learning a lot of classes and dependencies, even for applications.
	L.3	When I create a complex application, I can start by simple example and evolve it later in a seamless way
	L.4	(N) I needed to read all the documentations and tutorials to be able to create my first application
	L.5	Reading simple tutorials made me possible to understand better the complex ones and to create later applications complying with my requirements

– For the question "Type of application being developed", we provided participants the possibility of selecting multiple items among the following options (the corresponding encoding token is provided between <> signs):

  - I'm creating video applications with recording capability (<Recording>)
  - I'm creating my own filters and extending Kurento APIs (<Filter>)

Table 18 This table shows the additional questions made to participants in order to characterize them. The first column shows the type of data to be gathered through the question. The second column shows the question itself (summarized for the sake of readability), the third column shows the value type accepted by the web form. The mark [] indicates users are given the choice of choosing one item in a list. The mark []* indicates that multiple items may be selected. In this table, list items are tokenized for simplicity

Question type	Question	Value type
Demographic data	Nationality	[Country select]
	Age	Integer
	Gender	[Male | Female]
	Main computer language used	[Java | JavaScript | Other]
	Type of application being developed	[Recording | Filter | Video surveillance | Videoconferencing | Broadcasting | Integration | Other]*
Development experience	Years of experience as developer	Integer
	Hours learning or programming with Kurento	Integer
	Self-assessment of expertise as WebRTC developer	[1 to 5]
	Self-assessment of expertise in media technologies	[1 to 5]
	Self-assessment of expertise with Kurento technologies	[1 to 5]
	Learning stage on Kurento technologies	[Tried | Installed | SimpleApp | ComplexApp | ProductionApp]

- I'm creating video surveillance applications (<Video surveillance>)
- I'm creating videoconferencing applications (<Videoconferencing>)
- I'm creating broadcasting applications for distributing media among large groups of receivers (<Broadcasting>)
- I'm using Kurento for integrating with other types of technologies beyond WebRTC (<Integration>)
- I'm creating other types of applications (<Other>)

– All questions dealing with "self-assessment of expertise" were expressed in the poll as assertions in the form "I'm an expert in …", where answers are in the above mentioned 1 (I fully disagree) to 5 (I fully agree) format.
– The question dealing with "Learning stage on Kurento technologies" provided the ability of selecting one option among the following items:

- I tried to install Kurento unsuccessfully
- I installed Kurento and executed some of the provided demos
- I developed a simple application
- I developed a complex application
- I developed a complex application which is in production

4.1.2 Participants and protocol

Most API usability studies are performed by recruiting students or researchers who are trained on the API through lectures or exercises and who are later interviewed for performing the evaluation [57]. These types of protocols are sensitive to many different types of bias that may affect the study reliability. In particular, their main weakness is that participants are typically not professional developers and are not faced with real world programming tasks. Hence, their perception of the API limitations and usability problems can be severely biased by their own background and by the nature and contents of the training contents and of the proposed exercises. In addition, those contents and exercises are typically created by the API designers, which increases significantly the risk of introducing the designers' cognitive models and hiding API limitations that might not be known even by designers themselves. Many API evaluation research works are aware of these limitations, but solving them is not trivial given the difficulty of reaching a statistical significant population of professional developers being independent of designers, having the time of learning API concepts and working on solving real-world tasks with them.

In order to avoid these problems, we leverage the fact that the RTC Media API has been implemented as part of the Kurento project. More specifically, the Kurento Client API is an almost complete implementation of it. This is a significant advantage because Kurento has been released as an Open Source Software and a community of developers has emerged around it. The size of the community is unknown but its main communication channel, the Kurento Public mailing list, has, at the time of this writing, 432 subscribers most of which are professional developers at different stages of the API learning process.

In this context, the survey protocol is simple. The questionnaire is designed for Kurento community members (i.e. in all assertions the API is referred to as the "Kurento API"). This questionnaire is published as a web form and participants are invited to participate through an aseptic e-mail invitation sent to the Kurento Public mailing list. This mail is written trying to avoid any kind of bias on participants so that it just presents the survey objectives and exposes a privacy policy guaranteeing that no personal data is to be disclosed or used for other objectives than the ones of the survey. Participants are incentivized to participate only once by requiring logging into the web form system through a valid e-mail. The form makes mandatory to fill answers on all assertions for making submission possible (i.e. partially answered questionnaires are not considered). The web form system stores in a persistent database each participant's answers and makes possible to edit them during the duration of the survey period, which is limited to 2 weeks' time.

4.2 Results and analysis

4.2.1 Analysis of participants

The survey was activated following the protocol described above. After one week from the initial e-mail invitation a total of 17 participants had answered. Several reminders were sent to the Kurento Public mailing and the announcement was also published through different social channels, such as the Kurento Twitter account. In two weeks, 42 answers were received, which represents 9.7 % of the number of Kurento Public mailing list subscribers. This is aligned with typical answer rates in surveys.

From a demographic perspective age of participants were distributed between 20 and 50 years old, being the more numerous group the one in the thirties, which account for 50 % of the total participants, as can be seen in Fig. 10. It is worth noting that 100 % of participants were male, and their main programming language, the language in which API usage was involved, was JavaScript, totaling a 62 % of respondents.

In relation to nationality, and as can be seen in Fig. 11, the poll was answered by developers from 20 nationalities on 4 different continents, being USA the country with more participants.

In Fig. 12 we show the types of applications being developed. WebRTC video broadcasting applications (for distributing a media stream among a group of developers) and videoconferencing services are the most popular with 27 % and 26 % respectively. Applications involving recording and WebRTC integration are also quite popular (19 % and 15 %). Media processing services, such as video surveillance applications and services requiring custom filters are less popular accounting only for 7 % and 5 % respectively.

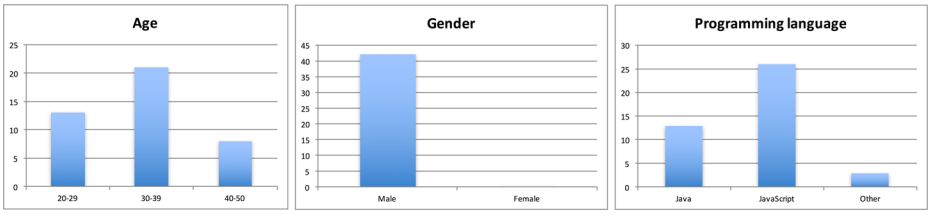

Fig. 10 Figures showing the total number of participants for different demographic data including Age (*left*), Gender (*middle*) and Main Programming Language (*right*)

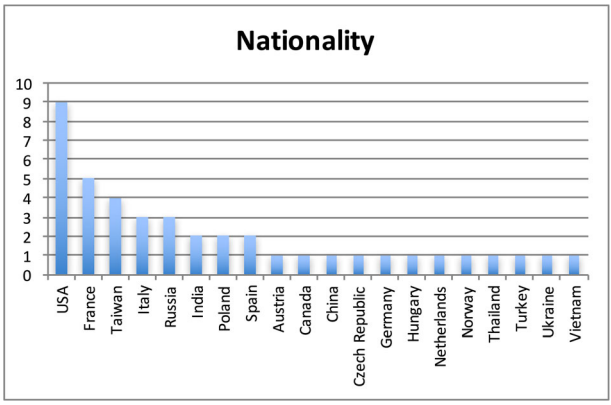

Fig. 11 Total number of participants per nationality

When coming to expertise evaluation, as shown in Table 19, participants accumulate, on average, 9.7 years of development experience, but with a relevant diversity (i.e. 1 year as minimum and 30 as maximum). Participants also declare to have invested, on average, more than 61 h for learning and programming with Kurento technologies, which provides a reasonable guarantee on their ability to evaluate the API. Regarding the self-assessment of competences, a relevant expertise on WebRTC technologies is clearly declared (mean 2.6 and median of 3). However, participants seem to have more uncertainty in relation to their knowledge of general video technologies (mean 2.6 and median of 2) and Kurento technologies (mean 2.5 and median of 2). On the latter, no participants declare to feel as a full featured expert (maximum is 4).

Regarding the stage of learning on Kurento technologies an interesting surprise emerges given than 38 % of participants declare to have already an application in production, while

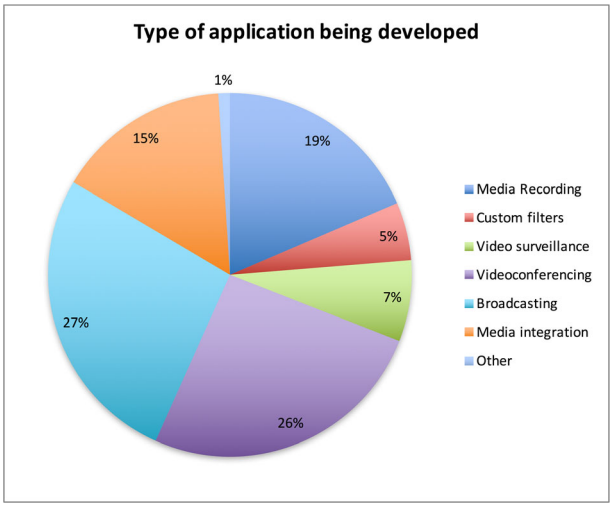

Fig. 12 Types of applications being developed with the Kurento API. The classification is based on the types of consumed features. In the poll, developers were able to select several classes of features for their applications

Table 19 Summary of answers in relation to participant's expertise as developers and in the different involved technological areas

Expertise question	Mean	Median	Min	Max	STDDEV	Type of data
Years of experience as developer	9.7	5	1	30	8	Integer
Hours learning or programming with Kurento	61.6	20	2	500	127	Integer
Self-assessment on WebRTC expertise	2.7	3	1	5	0.9	1 to 5
Self- assessment on video technologies expertise	2.6	2	1	5	1.2	1 to 5
Self- assessment on Kurento expertise	2.5	2	1	4	1.1	1 to 5

19 % and 33 % claim to have developed one complex and one simple application respectively. Only 3 % of participants has not been able to install and test Kurento and its APIs. These results are shown in Fig. 13.

4.2.2 Analysis of dimensions

The results of the poll are summarized in Table 20, where the main statistics for each of the assertions and their corresponding dimensions are depicted. As specified above, all N-assertions were inverted previous to the statistical analysis. Hence, magnitudes represent perceptions on API usability in positive terms (i.e. the higher the magnitude the better the developer's impression on the APIs). As it can be seen in Fig. 14, on average, participants feel API usability properties are adequate, being reusability the dimension with highest rank and expressiveness the one with the lowest score. A detailed analysis for each dimension is presented in the following paragraphs.

To complete our discussion, we perform an additional analysis to assess how participant experience might be influencing API perception. In this sense, we compute the average of all the scores provided by participants along all assertions and calculate its correlation coefficient with the available experience-related variables. Table 21 show the numerical results of this analysis, which evidence that higher development experience tends to be associated with better usability perception.

For completeness, we also evaluated the correlation of the perception of the different dimensions with other demographic data including nationality (consolidated in a per-continent manner) and

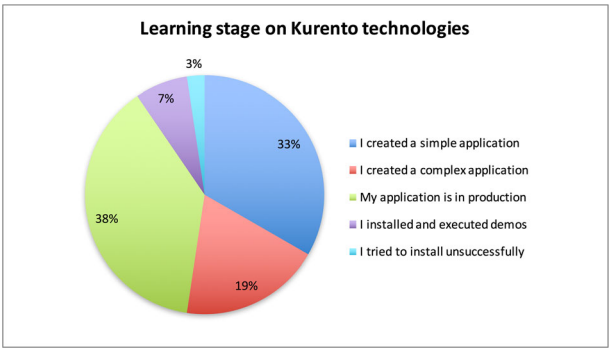

Fig. 13 Pie chart showing the level of expertise in the Kurento technologies of the participants

Table 20 Results of the research showing, for each assertion of the poll, the main statistics of the provided answers. Notice that N-assertion have their results inverted for maintaining coherence. For each dimension, the statistics are computed on the average values along all assertions on that dimension for each user

Dimension	ID	Mean	Median	Min	Max	STDDEV
Understandability		3.46	3.58	2	4.67	0.58
	U.1	3.33	3	1	5	0.98
	U.2	4.26	4	2	5	0.86
	U.3	2.69	3	1	5	1.12
	U.4	3.26	3	1	5	1.17
	U.5	3.52	3.5	2	5	0.71
	U.6	3.71	4	1	5	1.20
Abstraction		3.39	3.42	1.83	4.23	0.57
	A.1	3.40	3.50	1	5	1.04
	A.2	3.24	3	1	5	1.28
	A.3	3.29	3	1	5	1.11
	A.4	3.26	3	1	5	0.86
	A.5	3.69	4	1	5	0.84
	A.6	3.48	4	2	5	0.92
Expressiveness		3.17	3.17	2	4.33	0.57
	E.1	3.29	3	2	5	0.71
	E.2	3.17	3	1	5	0.82
	E.3	3.43	4	1	5	0.97
	E.4	3.45	4	1	5	1.02
	E.5	2.76	3	1	5	1.19
	E.6	2.9	3	1	4	1.05
Reusability		3.47	3.5	1.67	4.67	0.52
	R.1	3.48	4	1	5	0.99
	R.2	3.67	4	1	5	1.16
	R.3	3.31	3	1	5	0.98
	R.4	3.6	4	1	5	0.7
	R.5	3.62	4	1	5	0.7
	R.6	3.14	3	1	4	0.75
Learnability		3.43	3.4	1.4	5	0.65
	L.1	3.88	4	2	5	0.77
	L.2	3.07	3	1	5	1.35
	L.3	3.67	4	2	5	0.95
	L.4	2.86	3	1	5	1.26
	L.5	3.69	4	1	5	0.95

type of application being created. The corresponding results are illustrated in Figs. 16 and 17. As it can be observed, there are no significant dependencies on the API usability with these variables.

We also evaluated the correlation between the API usability perception and the main programming language being used. Results are summarized in Fig. 18. As it can be observed, JavaScript developers have a better perception of API usability than Java developers

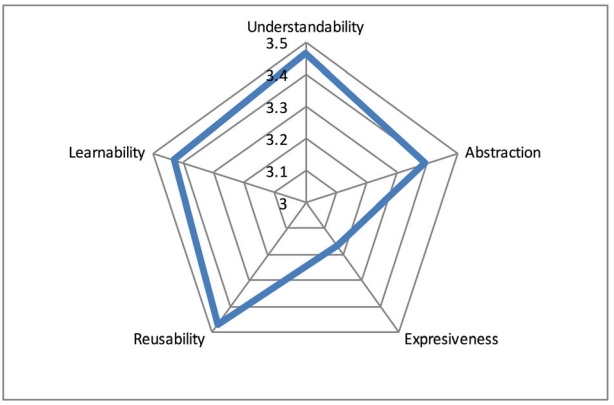

Fig. 14 Radar chart showing average rankings on the 5 target dimensions of our questionnaire. Scale is set between 3 and 3.5 for evidencing the differences among dimensions

4.3 Validity of the analyses

Following commonly accepted techniques for evaluating assessment data [23], we discuss the main threads to the validity of our research as well as the measures we deployed to minimize their impact.

4.3.1 Construct validity

Construct validity is the degree to which a test measures what it claims, which in our case, is the degree of usability of our RTC Media API. As introduced in sections above, evaluating API usability is a very complex multifaceted problem where it is difficult to get objective measures. To minimize threads, we performed a careful design of our research protocol which included the following protection mechanisms:

- We used a well-established methodology based on the CDs framework, the most widely accepted technique for this objective which has been already used successfully on a number of usability studies worldwide.
- We performed a careful design of the questionnaire basing on high-level usability dimensions more adapted to participants needs than to API designer's needs. Each of the high-level dimensions was measured through groups of 5 to 6 assertions facing the problem from different perspectives which minimizes effects of assertion misinterpretation.

Table 21 Correlation between the different parameter data captured through the questionnaire against the scores of API usability perception averaged across all assertions

Parameter to be correlated with overall average score along all answers	Correlation
Years of experience as developer	0.16
Hours learning or programming with Kurento	−0.07
Self-assessment of expertise as WebRTC developer	0.43
Self-assessment of expertise in media technologies	0.22
Self-assessment of expertise with Kurento technologies	−0.04

- The questionnaire contained complementary questions digging into the different components of each of the high-level dimensions and combining positively and negatively formulated assessments. This should enhance the consistency guarantees of the answers.
- The protocol avoided to introduce any kind of bias by enabling participants to answer assertions basing only on their own knowledge about the API artifacts (i.e. documentation, code, etc.) and not on previous information provided by designers (e.g. training courses) or on specific artificial exercises which could be associated to specific cognitive models about the API.

4.3.2 Internal validity

Internal validity is a property associated to the extent to which a study minimizes systematic errors and avoid introducing bias into measurements. For enhancing our internal validity, in our research we tried to avoid any kind of selection bias by enabling Kurento Open Source Community members to answer freely to the poll. This strategy was clearly successful given the wide spectrum of participants we had, comprising developers of different ages, expertise degrees, nationalities and cultures. This enhances significantly our internal validity in relation to other previous similar studies [57] where API designers and participants have tight relationships (e.g. professors and students, workers of the same company, etc.) The risk of statistical effects in the data is also low given the fact that the poll was answered by 42 participants, which is a population sample significantly over the ones of other similar studies [57].

To formalize our internal validity analysis, we perform an additional test based on Cronbatch's alpha [19, 20], which is the most commonly used estimate for assessing the reliability of psychometric tests in social sciences. As Cronbatch's alpha is a measure of the internal consistency of data it needs all test items to measure the same construct. Due to this, its computation needs to be performed for each of our high-level dimensions separately. Cronbatch's alpha evaluation for our data is shown in Table 22, while the thresholds for interpreting it are depicted in Table 23.

As it can be observed, the reliability of the obtained data is inside acceptable margins, which is reasonable for our type of questionnaire and open research search methodology where there is no control on the who's, how's and why's of participants.

4.3.3 External validity

External validity refers to the extent to which a study can be generalized to other situations or populations. In relation to this, the main threads to external validity come from the research

Table 22 Cronbach's alpha computed for all the high-level dimensions of our test

Dimension	Cronbatch's alpha
Understandability	0.77
Abstraction	0.73
Expressiveness	0.76
Reusability	0.76
Learnability	0.71

Table 23 Commonly accepted rule of thumb for describing internal consistency in terms of Cronbatch's alpha

Cronbatch's alpha	Internal consistency
alpha > = 0.9	Excellent
0.9 > alpha > = 0.8	Good
0.8 > alpha > = 0.7	Acceptable
0.7 > alpha > = 0.6	Questionable
0.6 > = alpha > = 0.5	Poor
0.5 > = alpha	Unacceptable

protocol, which was designed basing on specificities of our API implementation. In particular, the fact that we leveraged the Kurento Open Source software community for obtaining the test population is quite a strong restriction for the generalizability of our findings given that most newly designed APIs might not be Open Source and, even if they are, they do not need to have an active international community of more than 400 developers. Otherwise, the rest of our methodology as well as the analysis we performed and the conclusions we gathered from it do not assume any kind of specific requirements. This suggests that the gist of our findings are also applicable to different contexts and populations.

4.4 Discussion

Based on the analysis shown above, we come back to the hypothesis H1, as stated in Section 4.1, and analyze its degree of fulfillment. For this, we use the results of our study for answering the therein stated questions, namely:

Do developers feel that the API can be learnt in a simple, incremental and seamless way?

The evaluation of the learning ability of developers emerges mainly from two of the dimensions under analysis: understandability and learnability.

Understandability.

As illustrated in Table 20, and as a general perception, participants feel the API understandability to be fine, with an average of 3.46 over all answers on this dimension. Through U.1 (3.33) we find a general declaration that the API is easy to understand. In particular, and as shown in assertion U.2 answers, participants feel outstanding (4.26) how object and primitive names are descriptive. On the other hand, and through the U.3 N-assertion (2.69), we find that developers detect the presence of hidden dependencies that make the API more complex to understand.

Learnability.

As also illustrated in Table 20, the learnability of the API is evaluated positively by participants (3.43 in average). The improvement area in this topic emerges from L.2 (3.07) and L.4 (2.86), which evidence the impressions of developers of needing to learn about a lot of API constructs and to read a relevant amount of documentation before being able to start using the API for useful things. On the other hand, as L.1 (3.88) evidences, the learning process seems to be compatible with an incremental approach where complexity is introduced in progressive steps.

Based on this, we can confirm that the API can be understood and learnt by developers in a seamless and incremental way. The main area of improvement is in

the initial learning curve, which seems to be too abrupt, and in the presence of hidden information and dependencies among the API constructs. Probably, both topics are related and caused by the inherent complexity of RTC technologies. Our guess is that better and more complete documentation might be helpful for minimizing both problems.

Do developers feel that the API is helpful for the creation of clean and error-free application code without needing to manage low-level complexities?

The process of exploratory design, understood as the creative activity of creating application code consuming the API, is mainly related with two of our target dimensions: abstraction and expressiveness.

Abstraction.

As shown in Table 20, When coming to abstraction, we also find out a general positive evaluation (the overall average is 3.39). Answers on all questions are quite uniform, being A.5 the top ranked one (3.69), which shows that developers feel appealing the API approach; and A.2 the bottom ranked one (3.24) indicating that some developers feel the need of adapting the API to their needs. It is remarkable that A.2 has the largest standard deviation (1.28) of A assertions, which evidences some degree of controversy. This is confirmed when looking closely to answers: in A.2 answers, there are 12 % of ones and 19 % of fives, while considering all Abstraction answers the ratios for ones and fives are 4.3 % and 12.6 % respectively.

Expressiveness.

The Expressiveness analysis also reflects positive evaluation but shows improvement areas. This is the less successful dimension with an overall average ranking of 3.17. Expressiveness limitations seem to emerge on assertions E.5 and E.6 (2.76 and 2.90 respectively). In particular, E.5 reveals that developers miss features that are relevant for their applications. E.6, in turn, manifests that the API gives not enough protection against failures. On the other hand, as demonstrated through E.3 and E.4, our API is easy to read (3.43) and is consistent when dealing with explaining code logic in terms of the API constructs (3.45).

Hence, our API is suitable for being used in the process of creating application code. However, as Fig. 14 illustrates, abstraction, and more significantly expressiveness, are the two dimensions with lower usability score. This evidences that, although the API ideas are appealing and intuitive (learnability and understandability get very high scores), leveraging them for creating real-world RTC applications still presents some difficulties. These seem to be related with the lack of further desirable features (i.e. richer extensions to the API might be necessary) and with the lack of protection against failures. This latter topic is a pervasive problem in most RTC media APIs due to its distributed and real-time nature and, to the best of our knowledge, there are no simple solutions for fixing it.

Do developers feel that maintaining and evolving code consuming the API is smooth and uncomplicated?

Corrective and evolutive maintenance of the code is related to the dimension we call reusability:

Reusability.

As shown in Table 20, reusability is the dimension with the highest ranking (3.47 in average). This is illustrated through results in assertions R.2, R.3 and R.6, which average to 3.67, 3.31 and 3.14 respectively. Also the API demonstrates nice properties in relation to verbosity, as shown by the 3.48 exhibited by R.1. Our API is considered concise and it is not too much verbose.

Hence, we may conclude that, once the application code using the API has been created, it can be modified, maintained and evolved without much efforts.

Do developers have the same perception of the API usability with independence on their demographic characteristics (i.e. years of experience, nationality, etc.) and on the types of applications they create?

In our main hypothesis H1, as stated in Section 4.1, we assume that API perception of usability is fine for all developers, with independence on their origin, culture or experience. For validating this assertion, we have performed several statistical analyses whose outcome is the following:

Correlation between API usability and programming experience.

As Table 21 illustrates, there is a tendency to positive correlation between the API usability scores and the developers' previous experience and knowledge. This is particularly true for the self-assessment expertise on Kurento technologies: developers having better perception on their expertise on Kurento feel the Kurento API to be more usable for their objectives. However, as Fig. 15 shows, the negative effect is concentrated on users with very low expertise on Kurento technologies. In other words: as soon as a developer feels to have some initial knowledge on the API, his perception of the usability increases to be the same as the one of experts, which is a good symptom. Interestingly enough, this effect seems not to be correlated with the number of hours declared in learning or programming with Kurento. This means that the time invested by developers in learning or programming with the API does not have a strong influence on their perception of API usability. This may be related with the fact that the self-assessment of expertise on WebRTC technologies and on video technologies are much correlated with the self-assessment of expertise on Kurento (the correlation coefficient is 0.55 and 0.34 respectively). This evidences that having a previous understanding on WebRTC and video technologies, enables our API developers to improve their API usability perception in a faster and seamless way. In addition, this effect might also be related with the excessively abrupt initial learning curve mentioned above.

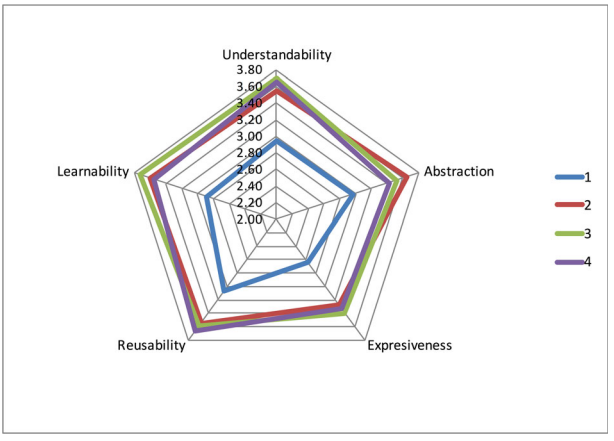

Fig. 15 Radar chart showing the average API usability for each of the analyzed dimensions particularized for the 4 levels of expertise declared for the question "Self-assessment of Kurento expertise" (notice that no participants declared a value of 5). As it can be observed, users declaring very low level of expertise (i.e. 1) tend to perceive the API as less usable in all dimensions. However, users declaring 2 or more perceive the API in quite a similar way

Hence, we can conclude that, as long as an initial knowledge about the API and its foundations is known, the API usability perception does not depend significantly on the proficiency of developers.

Relation between API usability and nationality/culture.

For analyzing this, we have consolidated developer's nationalities in continents. The results are illustrated in Fig. 16. Surprisingly, there is a clear tendency of USA developers to evaluate the API usability with less score. We do not have a consistent explanation for this, but we believe it might be caused by some specific cultural bias that may be related to the fact that the API (and its documentation) creators are not native English speakers, which may decrease the perception of "quality" from native English speakers. In any case, this tendency is not quantitative significant and we do not consider it invalidates our independence-of-culture hypothesis.

Relation between API usability and the type of application being created.

As shown in Fig. 17, the API usability scores do not seem to have a clear dependency on any of the analyzed dimensions with the type of application being created. Hence, we may conclude that the API usability perception does not depend on the specificities of the features a given application consumes.

Do developers have the same perception of the API usability with independence on their programming language?

This is a relevant question that deserves a separated analysis given that one of the main novelties of our API is to be programming-language-agnostic. For addressing it, we can observe Fig. 18, where the average API usability scores are represented as a function of the used programming language. As it can be seen, developers using programming languages other than Java and JavaScript (i.e. "Other") have clearly under-scored the API usability and, very particularly, in what refers to its understandability. This is natural as the only official implementations of the Kurento Client API are the Java and JavaScript ones, which means that such "Other" developers are either using non-official API implementations or directly consuming the JSON-RPC over WebSocket protocol exposed by Kurento Media Server, which enables to access the same functionalities than the Java and JavaScript SDKs, but which is clearly more complex and cumbersome to use as developers need firstly to understand the protocol to, secondly, create the appropriate SDKs for accessing it.

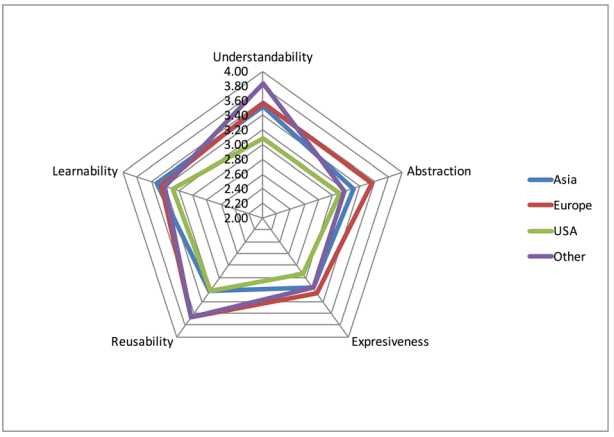

Fig. 16 Radar chart showing the average API usability perception as a function of developer's nationality, consolidated per continent. As it can be observed, there are not relevant differences, being USA developers the only ones showing a subtle tendency of considering lower usability perception in all dimensions

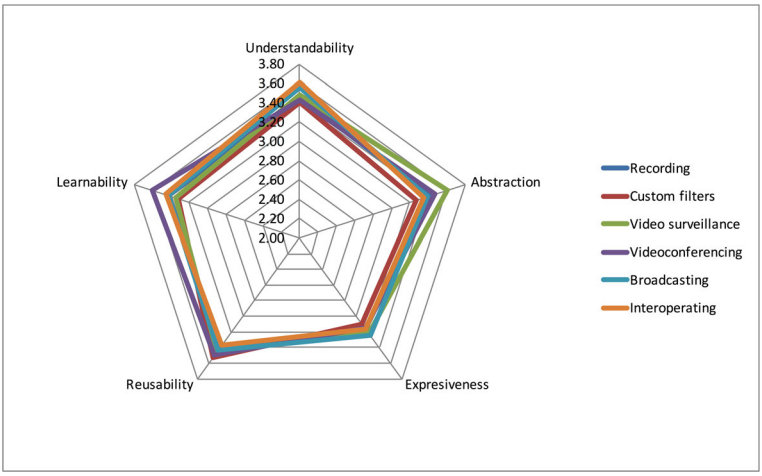

Fig. 17 Radar chart showing the average API usability perception as a function of the type of application being created by developers. As it can be observed, the type of application does not have a relevant impact on any of the usability dimensions under evaluation

However, which is more surprising is that JavaScript developers clearly over-score the API usability with respect to Java ones. The difference is not quantitatively significant and we believe it does not break our API programming-language-agnosticism claim, but it is remarkable enough for deserving our attention. We have several hypotheses for justifying this effect. First of all, we believe it might be caused by the higher simplicity of the JavaScript language. This would mean that this effect should be noticeable in all types of APIs, and not only on our RTC Media API, which may be plausible given the increasing success of JavaScript derived technologies such as node.js, that are stealing developers to other programming languages in a consistent and constant way. The second is that this effect might also have a relation with the

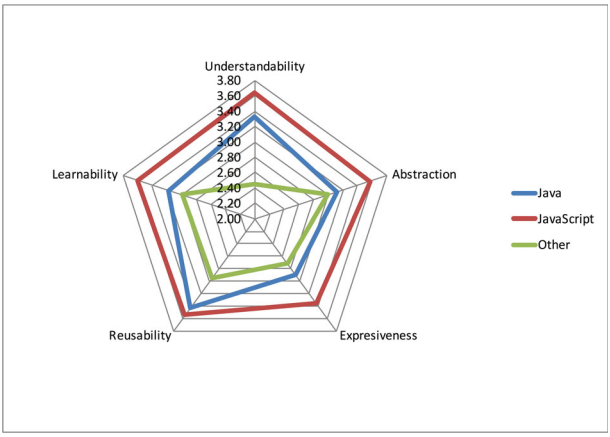

Fig. 18 Radar chart showing the average API usability perception as a function of the main programming language being used: Java, JavaScript or Other. The axis are re-scaled for clarity. Remark that the Kurento Client API is only available in Java and JavaScript. Hence, users in the category "Others" are directly consuming the JSON-RPC over WebSocket protocol exposed by Kurento Media Server, which is significantly more complex

Table 24 `PlayerEndpoint` creation using RTC Media API for Java and JavaScript

```
//Java
MediaPipeline pipeline = ...
PlayerEndpoint player =
   new PlayerEndpoint.Builder(pipeline,"video.webm").useEncodedMedia().build();
//use player here


//JavaScript
var pipeline = ...
pipeline.create("PlayerEndpoint",{uri:"video.webm",useEncodedMedia:true},
  function(error, player) {
    if (error) return console.error(error);
    //use player here
});
```

RTC Media API extensibility mechanism that becomes much more complex in strong typed languages such as Java. This makes Java to require the use of builders, while dynamic typed languages such as JavaScript can instantiate and manipulate the RTC Media API objects in a more seamless way. This is illustrated in Table 24: in order to create, for example, a `PlayerEndpoint` using the Java flavor of RTC Media API it is necessary to instantiate a Builder with the mandatory constructor parameters and then use its fluent API to configure the optional ones (like `useEncodedMedia` in the example). When the builder is fully configured, the "build" method may be invoked to create the `PlayerEndpoint` with the appropriate configuration. Using the JavaScript RTC Media API is more streamlined because the builder design pattern is not necessary. In JavaScript objects literals are often used as option bags when calling to methods. This technique is simpler than builder pattern. The drawback is that JavaScript leaks type safety, but some developers don't miss this extra protection when programming.

To finish, and basing on the answers to these questions, we can conclude that the main hypothesis of the paper is validated by our study and that, with the exception of some collateral effects that are not quantitatively relevant, the API usability is confirmed to be good across all application creation activities and high usability scores are robust with respect to developers' profiles, cultures, experience and preferred programming languages.

5 Conclusions

In this paper we have presented the RTC Media API: a new type of application programming interface complying with a number of stringent requirements answering to latest trends and needs in the area of real-time multimedia. We have introduced a specification of the API showing how it can be formally defined in a programming-language agnostic way through an IDL. We have also demonstrated how the IDL can be compiled into different programming languages, including Java and JavaScript, while complying with latest trends on API usability and performance. We have presented a specific RTC Media API implementation, the Kurento Client API, which includes a bunch of advanced processing capabilities such as Video Content Analysis, Augmented Reality, Media Blending, etc. We have evaluated the API usability through a research study basing on the CDs framework to demonstrate the API suitability in terms of dimensions such as understandability, abstraction, expressiveness or learnability. Thanks to this analysis, we have detected the main strengths and weaknesses of our RTC

Media API and we have been able to optimize the API future roadmap to address the specific issues that are decreasing developers' usability perception. These include the creation of improved documentation enabling a softer initial learning curve and the need of understanding better real-world applications requirements for adding missing features that developers demand.

Along the paper, we have tried to stress the importance of listening to developers' needs and solving developers' problems. We defend software is eating RTC multimedia technologies. Hence, for pushing them to a next level we need to create novel APIs and SDKs suitable for their democratization among wider developer audiences. In current state-of-the-art, there are a huge amount of algorithms and technologies for transporting, analyzing and enriching media, but there are very few APIs and SDKs making possible for average WWW and smartphone developers to use them in a seamless and effortless way. Our RTC Media API brings a whole new concept by incorporating WWW development methodologies to the multimedia arena.

The RTC Media API in general, and the Kurento Client API implementation in particular, are still research artifacts, which are under maturation, and miss many relevant ingredients. In particular, the adaptation to latest trends in WebRTC technologies, including the incorporation of ORTC (http://ortc.org/) concepts, would benefit significantly the API flexibility and its ability to adapt to latest trends in WebRTC. The API would also benefit from having richer support for complex media streams suitable for including multiple audio and video tracks, so that 3D or MVC multimedia is supported. Improvements are also possible from the perspective of development tools beyond the API itself so that seamless mechanisms for debugging, diagnosing and optimizing applications would be more than welcome by developers. To conclude, further efforts should be invested in the future to perform a consistent and complete evaluation of the API performance suitable for illustrating the main QoS metrics of the different media elements and of the media pipeline mechanism on real-world operational conditions following the scheme of pervious research on this area [61].

Acknowledgments This work has been supported by the European Commission under projects FI-WARE FP7-2011-ICT-FI GA-285248, FI-CORE FP7-2014-ICT-FI GA- 632893 and NUBOMEDIA FP7-ICT-2013-1.6 GA-610576; by Spanish Ministerio de Educación under project Reactive Media (TIN2013-41819-R); and by the Regional Government of Madrid (CM) under project Cloud4BigData (S2013/ICE-2894) co-funded by FSE & FEDER.

References

1. Abowd GD, Dey AK, Brown PJ, Davies N, Smith M, Steggles P (1999) Towards a better understanding of context and context-awareness. In: Handheld and ubiquitous computing. Springer, pp 304–307
2. Afonso LM, Cerqueira RFG, de Souza CS (2012) Evaluating application programming interfaces as communication artefacts. System 100:8–31
3. Ajanki A, Billinghurst M, Gamper H, Järvenpää T, Kandemir M, Kaski S, Koskela M, Kurimo M, Laaksonen J, Puolamäki K et al (2011) An augmented reality interface to contextual information. Virtual Reality 15(2–3):161–173
4. Allen IE, Seaman CA (2007) Likert scales and data analyses. Qual Prog 40(7):64
5. Alvestrand H (2015) Transports for WebRTC draft-ietf-rtcweb-transports-10. https://tools.ietf.org/html/draft-ietf-rtcweb-transports-10. Accessed 20 Dec 2015
6. Andreasen F, Arango M, Huitema C, Kumar R, Pickett S, Elliott I, Foster B, Dugan A (2003) Media gateway control protocol (MGCP) version 1.0. Tech. rep., internet engineering task force, request for comments (RFC) 3435

7. Andreessen M (2011) Why software is eating the world. Available online: http://www.wsj.com/articles/SB10001424053111903480904576512250915629460 , Wall Street Journal 20
8. Bainomugisha E, Carreton AL, Cutsem T, Mostinckx S, Meuter W (2013) A survey on reactive programming. ACM Comput Surv 45(4):52:1–52:34. doi:10.1145/2501654.2501666
9. Bajaj V (2004) JAIN MEGACO API Specification. Tech. rep., Java community process, java specification request (JSR) 79
10. Bauer M, Kovacs E, Schülke A, Ito N, Criminisi C, Goix LW, Valla M (2010) The context API in the oma next generation service interface. In: Intelligence in next generation networks (ICIN), 2010 14th International Conference on, IEEE, pp 1–5
11. Becke M, Rathgeb EP, Werner S, Rungeler I, Tuxen M, Stewart R (2013) Data channel considerations for rtcweb. IEEE Commun Mag 51(4):34–41
12. Bertoa MF, Troya JM, Vallecillo A (2006) Measuring the usability of software components. J Syst Softw 79(3):427–439
13. Blackwell AF, Britton C, Cox A, Green TR, Gurr C, Kadoda G, Kutar M, Loomes M, Nehaniv CL, Petre M, et al (2001) Cognitive dimensions of notations: Design tools for cognitive technology. In: Cognitive technology: instruments of mind. Springer, pp 325–341
14. Blackwell AF, Green TR (2000) A cognitive dimensions questionnaire optimised for users. In: proceedings of the twelfth annual meeting of the psychology of programming interest group, pp 137–152
15. Bloch J (2006) How to design a good API and why it matters. In: Companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems, languages, and applications. ACM, pp 506–507
16. Bray T, Paoli J, Sperberg-McQueen CM, Maler E, Yergeau F (1998) Extensible markup language (XML). World Wide Web consortium recommendation REC-xml-19980210 http://www.w3.org/TR/1998/REC-xml-19980210
17. Catherine MR, Edwin EB (2013) A survey on recent trends in cloud computing and its application for multimedia. Int J Adv Res Comput Eng Technol (IJARCET) 2(1):304–309
18. Clarke S (2001) Evaluating a new programming language. In: 13th Workshop of the psychology of programming interest group, pp 275–289
19. Cortina JM (1993) What is coefficient alpha? An examination of theory and applications. J Appl Psychol 78(1):98
20. Cronbach LJ (1951) Coefficient alpha and the internal structure of tests. Psychometrika 16(3):297–334
21. Daughtry III JM, Carroll JM (2012) Perceived self-efficacy and APIs. Programming interest group p 42
22. Dey AK (2001) Understanding and using context. Pers Ubiquit Comput 5(1):4–7
23. Downing SM (2003) Validity: on the meaningful interpretation of assessment data. Med Educ 37(9):830–837
24. Duala-Ekoko E, Robillard MP (2012) Asking and answering questions about unfamiliar APIs: An exploratory study. In: proceedings of the 34th international conference on software engineering. IEEE Press, pp 266–276
25. Eckert C, Stacey M (2000) Sources of inspiration: a language of design. Des Stud 21(5):523–538
26. Ellis B, Stylos J, Myers B (2007) The factory pattern in API design: a usability evaluation. In: Proceedings of the 29th international conference on software engineering, IEEE Computer Society, pp 302–312
27. Ericson T, Brandt M (2009) Media server control API. Tech. rep., Java community process, Java specification request (JSR) 309
28. Farooq U, Zirkler D (2010) API peer reviews: a method for evaluating usability of application programming interfaces. In: Proceedings of the 2010 ACM conference on computer supported cooperative work, ACM, pp 207–210
29. Ferry D, Lim S (2004) JAIN SLEE API specification. Tech. rep., Java community process, Java specification request (JSR) 22
30. FIWARE Consortium (2015) Future internet core platform. Available online: http://www.fiware.org, FP7-2011-ICT-FI (GA-285248))
31. Fraternali P (1999) Tools and approaches for developing data-intensive web applications: a survey. ACM ComputSurv (CSUR) 31(3):227–263
32. Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns. elements of reusable object-oriented software. Addison-Wesley
33. Ganassali S (2008) The influence of the design of web survey questionnaires on the quality of responses. Surv Res Methods 2:21–32
34. Gouveia F,Wahle S, Blum N, Magedanz T (2009) Cloud computing and EPC/IMS integration: new value-added services on demand. In: Proceedings of the 5th international ICST mobile multimedia communications conference, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), p 51

35. Green TR (1989) Cognitive dimensions of notations. People and computers V pp 443–460
36. Green TR, Blandford AE, Church L, Roast CR, Clarke S (2006) Cognitive dimensions: achievements, new directions, and open questions. J Vis Lang Comput 17(4):328–365
37. Gualdi G, Prati A, Cucchiara R (2008) Video streaming for mobile video surveillance. IEEE Tran Multimedia 10(6):1142–1154
38. Handley M, Perkins C, Jacobson V (2006) SDP: session description protocol. Tech. rep., internet engineering task force, request for comments (RFC) 4566
39. Henning M (2007) API design matters. Queue 5(4):24–36
40. Hovemeyer D (2005) Simple and effective static analysis to find bugs. PhD thesis, College Park, MD, USA, AAI3184274
41. Internet Engineering Task Force (2015) Real-time communication in web-browsers (rtcweb). https://datatracker.ietf.org/wg/rtcweb. Accessed 9 Dec 2015
42. Ivov E, Marocco E, Holmberg C (2014) A session initiation protocol (SIP) usage for Trickle ICE draft-ietf-mmusic-trickle-ice-sip-03. https://tools.ietf.org/html/draft-ietf-mmusictrickle-ice-sip-03. Accessed 12 Dec 2015
43. Jennings C, Holmberg C, Alvestrand HT (2015) Negotiating media multiplexing using the session description protocol (SDP) draft-ietf-mmusic-sdp-bundle-negotiation-23. https://tools.ietf.org/html/draft-ietf-mmusic-sdp-bundle-negotiation-23. Accessed 12 Dec 2015
44. Johnston AB, Burnett DC (2012) WebRTC: APIs and RTCWEB protocols of the HTML5 real-time web. Digital Codex LLC
45. Kambona K, Boix EG, De Meuter W (2013) An evaluation of reactive programming and promises for structuring collaborative web applications. In: Proceedings of the 7th workshop on dynamic languages and applications. ACM, New York, NY, USA, DYLA '13, pp 3:1–3:9. doi 10.1145/2489798.2489802
46. Koukoulidis V, Shah M (2006) The IP multimedia domain: service architecture for the delivery of voice, data, and next generation multimedia applications. Multimedia Tools Appl 28(2):203–220
47. Kristensen A (2003) SIP Servlet API. Tech. rep., Java community process, Java specification request (JSR) 116
48. Laursen A, Olkin J, Porter M (1994) Oracle media server: providing consumer based interactive access to multimedia data. SIGMOD Rec 23(2):470–477. doi:10.1145/191843.191933
49. Lienhart R, Maydt J (2002) An extended set of HAAR-like features for rapid object detection. In: Image processing. 2002. Proceedings. 2002 International Conference on, vol 1, pp I-900–I-903 vol. 1, doi: 10.1109/ICIP.2002.1038171
50. Melanchuk T (2009) An architectural framework for media server control. Tech. rep., internet engineering task force, Request for comments (RFC) 5567
51. Moody DL (2009) The physics of notations: toward a scientific basis for constructing visual notations in software engineering. IEEE Trans Softw Eng 35(6):756–779
52. NUBOMEDIA Consortium (2015) NUBOMEDIA: an elastic Platform as a Service (PaaS) cloud for interactive social multimedia. Available online: http://www.nubomedia.eu, ICT-2013.1.6 (GA-610579)
53. O'Doherty P, Ranganathan M (2003) JAIN SIP API specification. Tech. rep., Java community process, Java specification request (JSR) 32
54. Ott J, Wenger S, Sato N, Burmeister C, Rey J (2006) Extended RTP profile for real-time transport control protocol (RTCP)-based feedback (RTP/AVPF). Tech. rep., internet engineering task force, request for comments (RFC) 4585
55. Park S, Park NS, Kim JT, Paik EH (2009) Provision of the expressive multisensory adaptation platform for heterogeneous multimedia devices in the ubiquitous home. IEEE Transact Consum Electron 55(1):126–131
56. Perkins C, Westerlund M (2010) Multiplexing RTP data and control packets on a single port. Tech. rep., internet engineering task force, request for comments (RFC) 5761
57. Piccioni M, Furia C, Meyer B, et al (2013) An empirical study of API usability. In: Empirical software engineering and measurement, 2013 ACM/IEEE International Symposium on, IEEE, pp 5–14
58. Pickard A (2012) Research methods in information. Facet publishing, London
59. Prusty N (2015) Learning ECMAScript 6. Packt Publishing Ltd, Birmingham
60. Reddy M (2011) API design for C++. Elsevier, Amsterdam
61. Roccetti M, Ghini V, Pau G, Salomoni P, Bonfigli ME (2001) Design and experimental evaluation of an adaptive playout delay control mechanism for packetized audio for use over the Internet. Multimedia Tools Appl 14(1):23–53
62. Rosenberg J (2010) Interactive connectivity establishment (ICE): a protocol for network address translator (NAT) traversal for offer/answer protocols. Tech. rep., internet engineering task force, request for comments (RFC) 5245

63. Rosenberg J, Schulzrinne H, Camarillo G, Johnston A, Peterson J, Sparks R, Handley M, Schooler E (2002) SIP: session initiation protocol. Tech. rep., internet engineering task force, request for comments (RFC) 3261
64. Saint-Andre P (2011) Extensible messaging and presence protocol (XMPP): Core. Tech. rep., internet engineering task force, request for comments (RFC) 6120
65. Saleem A, Xin Y, Sharratt G (2010) Media server markup language (MSML). Tech. rep., internet engineering task force, request for comments (RFC) 5707
66. Schulzrinne H (1998) Real time streaming protocol (RTSP). Tech. rep., internet engineering task force, request for comments (RFC) 2326
67. Schulzrinne H (2003) RTP profile for audio and video conferences with minimal control. Tech. rep., internet engineering task force, request for comments (RFC) 3551
68. Schulzrinne H, Casner S, Frederick R, Jacobson V (2003) RTP: A transport protocol for realtime applications. Tech. rep., internet engineering task force, request for comments (RFC) 3550
69. Schulzrinne H, Rosenberg J (1999) The IETF internet telephony architecture and protocols. IEEE Netw 13(3):18–23
70. Seema A, Schwoebel L, Shah T, Morgan J, Reisslein M (2015) WVSNP-DASH name-based segmented video streaming. IEEE Trans Broadcast 61(3):346–355
71. Tang X, Zhang F, Chanson ST (2002) Streaming media caching algorithms for transcoding proxies. In: Parallel processing, 2002. Proceedings. International Conference on, IEEE, pp 287–295
72. Taylor T (2000) Megaco/H.248: a new standard for media gateway control. IEEE Commun Mag 38(10): 124–132
73. Thom G et al (1996) H.323: the multimedia communications standard for local area networks. IEEE Commun Mag 34(12):52–56
74. Tilkov S, Vinoski S (2010) Node.js: using javascript to build high-performance network programs. IEEE Internet Comput 14(6):80–83. doi:10.1109/MIC.2010.145
75. Van Dyke J, Burger E, Spitzer A (2007) Media server control markup language (MSCML) and protocol. Tech. rep., internet engineering task force, request for comments (RFC) 4722
76. Vaquero LM, Rodero-Merino L, Caceres J, Lindner M (2008) A break in the clouds: towards a cloud definition. ACM SIGCOMM Comput Commun Rev 39(1):50–55
77. Vogel A, Kerherve B, von Bochmann G, Gecsei J (1995) Distributed multimedia and QoS: a survey. IEEE MultiMedia 2(2):10–19. doi:10.1109/93.388195
78. Wagner B (2010) Effective C# (Covers C# 4.0): 50 specific ways to improve your C#. pearson education
79. Westerlund M, Burman B, Nandakumar S (2015) Using simulcast in RTP sessions draft-westerlund-avtcore-rtp-simulcast-04. https://tools.ietf.org/html/draftwesterlund-avtcore-rtp-simulcast-04. Accessed 12 Dec 2015
80. Westerlund M, Wenger S (2016) RTP internet draft topologies draft-ietf-avtcorertp-topologies-update-10. https://tools.ietf.org/html/draft-ietf-avtcore-rtp-topologies-update-10. Accessed 7 Jan 2016
81. Willmott S, Balas G (2013) Winning in the API economy. Available online: http://www.3scale.net/wp-content/uploads/2013/10/Winning-in-the-API-EconomyeBook-3scale.pdf
82. World Wide Web Consortium (2011) Web real-time communications working group. http://www.w3.org/2011/04/webrtc/. Accessed 11 Dec 2015
83. World Wide Web Consortium (2015) WebRTC stats. Available online: http://w3c.github.io/webrtc-stats/
84. Zhou F, Duh HBL, Billinghurst M (2008) Trends in augmented reality tracking, interaction and display: A review of ten years of ISMAR. In: Proceedings of the 7th IEEE/ACM International Symposium on Mixed and Augmented Reality, IEEE Computer Society, pp 193–202
85. Zhu W, Luo C, Wang J, Li S (2011) Multimedia cloud computing. IEEE Signal Process Mag 28(3):59–69

Dr. 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.

Boni García has a PhD degree in Information and Communications Technology from Universidad Politécnica de Madrid (Spain) in 2011. His main interests are Software Engineering and Computer Networking. Currently he is working as a Researcher and Assistant Professor at Universidad Rey Juan Carlos (Spain). He is member of the project Kurento, an open source framework devoted to simplify the development of multimedia applications.

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 has a PhD degree on Computer Science. He is Associate Professor at Universidad Rey Juan Carlos. His main research activities are related to Computer Networking and Operations Research. He is member of the Kurento project, where he is focused on developing infrastructure support for testing distributed applications.