Design and Implementation of a Multimedia Control and Processing Framework for IoT Application Development

Daijiro Komaki*, Shunsuke Yamaguchi*, Masako Shinohara*, Kenichi Horio*, Masahiko Murakami*, and Kazuki Matsui*

* Fujitsu Laboratories Ltd., Japan
{komaki.daijiro, yamaguchi.shun, m-shinohara, horio, mul, kmatsui}@jp.fujitsu.com

Abstract - When creating Internet-of-Things (IoT) applications, it is difficult to deal with multimedia data captured from cameras and microphones installed at field sites since it requires a wide variety of knowledge of topics such as codecs, protocols and image processing. To solve this problem, therefore, we propose a framework that makes it easy to deal with multimedia stream data in IoT application development. Our framework has three main features as follows: (1) Virtualization of multimedia input/output devices; (2) Distributed execution of multimedia processing pipeline between gateways and a cloud; and (3) Simple service description using a graphical flow editor. In this paper, we present some prototype applications we created and discuss the effectiveness of our framework from the perspective of complexity, productivity and ease of trial and error.

Keywords: Internet-of-Things, Multimedia, Framework, Web API

1 INTRODUCTION

We have entered the era of the *Internet-of-Things* (IoT), where not only computers but also physical objects (i.e., *things*) such as vehicles and home appliances are connected to the internet and interact with each other, or with systems, services and people. When creating such IoT applications, it is important to make devices such as sensors and actuators already installed at field sites (e.g., classrooms, concert halls, building entrances) available for various applications, rather than to install devices at a field site for a specific purpose [2, 5].

In addition, not only sensory data (e.g., temperature and acceleration) but also multimedia data (i.e., audio and video) captured from devices such as cameras and microphones installed at field sites are important for IoT application development, since we can offer many beneficial applications that utilize multimedia data as sensory data by using computer vision technologies (e.g., *detecting a suspicious person at a building entrance*), or that utilize sensory data as an input to process multimedia data (e.g., *adding effects to a live video stream of a concert event according to the mood of audiences there*).

On the other hand, there are some multimedia frameworks such as Kurento[1] [4] and Skylink[2] that make it easy to create applications utilizing multimedia data. These frameworks provide multimedia server programs and client libraries, and developers can easily create their applications without worrying about the differences in codecs and formats of audio/video contents by using the provided libraries. For example, developers can easily create applications equipped with multimedia features (e.g., VoIP, augmented reality) simply by connecting multimedia processing blocks as a pipeline. However, even when using these frameworks, problems remain when considering the characteristics of IoT application development, as follows:

- Since there may be many different types of devices at field sites, developers need to know the detailed specifications (e.g., an interface to start/stop capturing media data, to establish a media session between a device and a media server) in advance, and need to create applications according to the specifications.
- If all multimedia stream data generated at field sites are continuously transferred to a media server, they consume large amounts of network bandwidth.

To solve these problems, we designed and implemented a framework to simplify the process of multimedia IoT application development. Our framework has three main features as follows:

Virtualization of Multimedia Input/Output Devices

Our framework provides a mechanism to virtualize multimedia input/output devices (e.g., cameras and speakers) to obscure the differences in heterogeneous device specifications. This mechanism means that developers no longer need to consider the details of devices already installed at field sites, and applications once created can be adapted for another field site.

Distributed Execution of Multimedia Processing Pipeline between Gateways and a Cloud

Our framework provides a mechanism to distributedly process single multimedia stream data by coordinating multiple multimedia servers running independently on gateways (installed at field sites) and on a cloud, respectively. This mechanism enables developers to easily create multimedia IoT applications that can save network bandwidth usage and can serve immediate detection and response to field sites.

Simple Service Description using a Graphical Flow Editor

Our framework provides a web-based graphical flow editor tool to simply define a distributed multimedia processing

[1] Kurento: https://www.kurento.org/
[2] Skylink: http://skylink.io/

pipeline by connecting input/output device blocks and filter blocks. This tool enables developers to easily use trial and error by replacing and relocating each block.

In this paper, we present some of the prototype applications we created and discuss the effectiveness of our framework from the perspective of complexity, productivity, and ease of using trial and error.

2 RELATED WORK

2.1 Multimedia Frameworks

There are several frameworks that make it easy to create applications that utilize multimedia stream data. GStreamer [3] is an open-source framework for creating multimedia applications that handle audio, video and any kind of data flow in a modular way. The basic idea of GStreamer is to link together various plug-in elements (e.g., sinks/sources, encoders/decoders, filters) on provided pipeline architecture to obtain a stream that meets the desired requirements. This seems to be effective for developers who are not familiar with multimedia processing or multimedia networking. However, even when using GStreamer, developers are required to know which type of devices, which protocols, and which codecs to use in advance, in order to define a pipeline.

Kurento Media Server is an open-source multimedia server based on GStreamer that supports WebRTC. Developers can easily create web-based multimedia streaming applications (e.g., VoIP, video conference, augmented reality) using provided APIs. Since Kurento Media Server provides the mechanism to absorb the differences in media codecs and formats, even developers unfamiliar with multimedia processing (e.g., web application developer) can create multimedia streaming applications by linking media processing modules (e.g., image processing, event detection) via the provided web APIs. In addition, this framework provides a way to implement a new media processing module using OpenCV [4](Open-Source Computer Vison). Computer vision experts can create their new modules independently from the application development process.

Although it becomes easy to deal with multimedia stream data by using such framework technologies, these frameworks are not necessarily suitable for creating IoT applications (that make use of devices already installed at field sites) since developers need to create their applications according to device type, protocols, codecs and so on.

2.2 IoT Application Development Platforms

On the other hand, there are several cloud-based platforms for creating and deploying IoT applications that utilize multiple sensors and actuators installed at field sites. Kii Cloud[5], a *Backend-as-a-Service* for IoT application development, provides the functionalities to virtualize devices on the cloud. IoT application developers can create their IoT applications by combining multiple virtualized device functionalities by using provided APIs, so they need not be concerned about the differences in detailed specifications such as communication protocols.

IBM Bluemix[6] also provides a way to create IoT applications using virtualized device functionalities on the cloud. Bluemix provides a graphical flow editor (called Node-RED[7]) to create interactive, near real-time IoT applications by simply connecting things and services. Blackstock [3] focused on the fact that many IoT scenarios require the coordination of computing resources across networks: on servers, gateways, and devices, and extended Node-RED in order to create distributed IoT applications that can be portioned between servers and gateways. MyThings [8] (provided by Yahoo! Japan) enables users to create IoT applications that link various devices to various web services by simple IF-THEN rules.

Owing to such platforms, it becomes easy to create IoT applications that connect multiple devices and services to each other. However, if developers attempt to create an IoT application that deals with multimedia streams generated from field sites, they have to use multimedia frameworks such as those mentioned above.

There have been a few efforts to simplify the creation of IoT applications that can handle both sensory data and multimedia data in combination. ThingStore [1] provides the mechanisms to virtualize any type of device as a thing that generates Boolean data (i.e., Boolean value represents whether a certain event occurs or not). Owing to this abstraction, application developers can deal with media input devices as sensor devices and can simply create IoT applications that coordinate both sensory and multimedia stream data. However, since ThingStore abstracts multimedia stream data as Boolean data, it is not suitable for dealing with end-to-end multimedia stream data transferred from a device to another device.

2.3 Kurento Media Server

In this section, we focus on Kurento Media Server, which is the basis of our framework implementation. Kurento Media Server is an open-source software media server that makes it simple to create web applications equipped with multimedia features (e.g., VoIP, video conference, augmented reality). Kurento Media Server provides endpoint modules (i.e., elements to input or output the multimedia stream) and filter modules (i.e., elements affecting the media stream or detecting events from the media stream) shown in Table 1. Application developers are simply required to take the modules needed for an application and to connect them, without worrying about differences in codecs and formats of audio/video data. Figure 1 shows an application example: the video stream captured by the web browser is sent to the media server, then *FaceOverlayFilter* detects faces from frames of the video stream and puts a specified image on top of them, and finally the face-overlaid video stream is sent

[3] GStreamer: https://gstreamer.freedesktop.org/
[4] OpenCV: http://opencv.org/
[5] IoT Cloud Platform Kii: https://en.kii.com/
[6] IBM Bluemix: https://www.ibm.com/developerworks/cloud/bluemix/
[7] Node-RED: http://nodered.org/
[8] myThings: http://mythings.yahoo.co.jp/

back to the web browser, while recording it on the media server.

Kurento Media Server provides Java and JavaScript client libraries. Developers can use the functionalities Kurento Media Server offers on their applications. Figure 2 shows the required procedures to construct the pipeline shown in Fig. 1. Firstly, a web browser-side script creates a Session Description Protocol[9] (SDP) offer and sends it to the web application server. Then, a server-side script creates the pipeline by using the provided client library, makes *WebRtcEndpoint* process the SDP offer in order to get an SDP answer, and sends the SDP answer back to the client. Finally, the client processes the received SDP answer to establish a WebRTC session with the *WebRTCEndpoint* on the media server. When the client starts sending the video stream, the processed video stream data is sent back to the client. Our framework also uses the provided client library to control Kurento Media Server in the same manner as described.

The filters and endpoints that Kurento Media Server provides have their own methods for clients to change inner parameters. Moreover, events raised by filters are subscribable by client-side scripts. For example, in Fig. 3, an application can change the image path to overlay and can subscribe an event that a barcode is detected in a frame of video stream. Our framework makes use of such features of Kurento Media Server and implemented some additional functionalities from the perspective of the IoT scenario.

3 DESIGN OF THE FRAMEWORK FOR IOT APPLICATION DEVELOPMENT

3.1 Target

We aim to make it easier to deal with not only sensory data but also multimedia stream data among devices in such cloud-based IoT application development platforms. Our framework focuses on IoT applications where sensory stream data and multimedia stream data affect each other interactively. Typical scenario cases we assume as multimedia IoT applications are as follows.

[Case 1] *Capture live video stream from a certain camera at a field site and transfer it to a screen installed at the same place.*

[Case 2] *Obtain text segment data from live audio stream captured from a certain microphone by speech recognition, overlay the text on live stream video captured from a certain camera, and project the text-overlaid video stream on a nearby screen.*

[Case 3] *Count the number of people from live video stream captured by a certain camera, and detect an event according to the change of that number.*

[9] RFC 4566 - SDP: Session Description Protocol: http://tools.ietf.org/html/rfc4566.html

Table 1: Modules provided by Kurento Media Serer

Endpoints (Inputs/Outputs)	
WebRtcEndpoint	Send and receive WebRTC media flow
RtpEndpoint	Send and receive RTP media flow
PlayerEndpoint	Read media from a file or URL
RecorderEndpoint	Store media flow to a file or URL
Filters (Processing/Detecting…)	
FaceOverlayFilter	Recognize face areas and overlay picture on that area.
ZBarFilter	Detect barcode and QR code
GStreamerFilter	Use filters of GStreamer

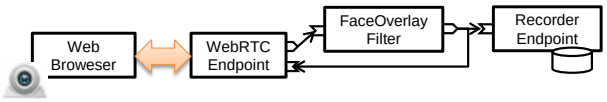

Figure 1: Example of connecting endpoints and filters

[Case 4] *Record live stream video captured from a certain camera installed at a field site only when the temperature there is higher than a threshold value.*

3.2 Functional Requirements

In order to make it easy to create such above multimedia IoT applications, we extract the requirements of functionalities that the framework should provide as follows:

(1) Virtualization of Multimedia Input/Output Devices

Considering the above scenarios, it is desirable to deploy an IoT application once created to many various field sites rather than to create an IoT application for a specific field site. However, there may be different types of devices (e.g., IP camera that supports RTSP, USB camera) at field sites and they may communicate using different protocols (e.g., WebRTC, RTP, HTTP). Such heterogeneity does not become a problem when creating conventional web applications since developers already know which type of device to use (i.e., devices are virtualized on the HTML5 layer on the browser side). However, from the perspective of IoT scenarios, it is assumed that many IoT applications utilize the same devices already installed at field sites together. Therefore, the framework should obscure such device heterogeneity so that developers do not need to consider it.

(2) Distributed Execution of Multimedia Processing Pipeline between Gateways and the Cloud

Since multimedia stream data is far larger than sensory data, it consumes a large amount of network bandwidth if transferring all multimedia stream data generated at field sites to the cloud. Considering the case where the results of event detection from video stream data captured from a field site are fed back to the same field site (Case 2), or the case where an IoT application needs only metadata extracted from multimedia streams (Case 3), Processing multimedia stream data on a computational resource near the field site is

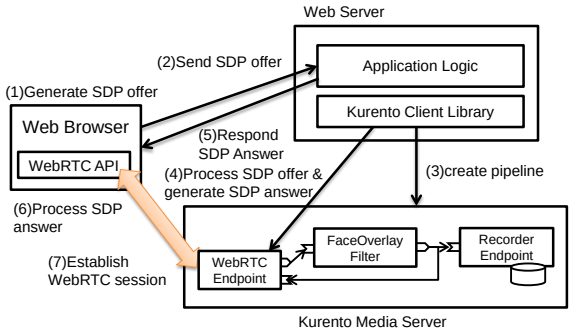

Figure 2: Procedure to establish media session

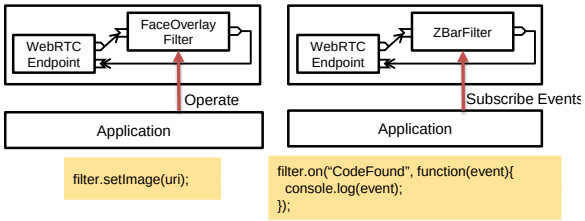

Figure 3: Interaction between application and filters

effective in saving network bandwidth usage and responding to the field site quickly.

To realize this, it is effective to process a single multimedia stream distributedly between a gateway (installed at the field site) and a cloud. However, in order to do that, it requires laborious procedures such as opening ports for sending/receiving multimedia stream on multiple media servers and establishing a media session between them. Therefore, the framework should enable the processing of single multimedia stream distributedly without considering media session establishment.

(3) Cooperation with External System

We assume not only the case where the framework control and process end-to-end multimedia stream are sent from one device to another (Case 1 and 2) but also the case where the framework detects an event using time series metadata extracted from a multimedia stream (Case 3) and the case where the framework controls the media stream according to the changes in sensory data (e.g., temperature) (Case 4). To do that, the framework should provide a way to easily cooperate with an existing IoT platform that provides the functionalities of time series data analysis or complex event processing.

(4) Simple Description of Media Processing Pipeline

By using media framework technologies such as Kurento Media Server, developers can easily create multimedia web applications by connecting multiple endpoints and filters as a pipeline, and can easily use trial and error by replacing or reconnecting each block. The framework should inherit this feature to simply implement a media processing pipeline, while satisfying the above three requirements ((1)-(3)).

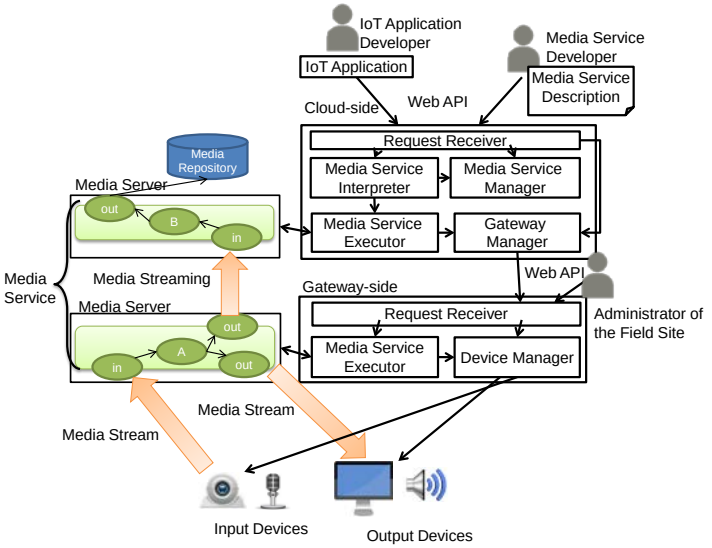

Figure 4: Architecture of our framework

4 IMPLEMENTATION OF THE FRAMEWORK

4.1 Architecture

In order to meet the above requirements, we designed the architecture of our framework (Fig. 4). Here, we define the term media service as a set of an entity to process multimedia stream data running on media servers, and an IoT application as an entity to use media service(s) by using the APIs that our framework provides. Our framework uses multiple media servers running independently on the gateway(s) and the cloud, respectively, and deploys corresponding modules to cooperate with multiple media servers. The framework forms star topology, where the cloud-side module aggregates all gateway-side modules. Noted that we adopted Kurento Media Server as the media server, but other media servers are adaptable to realize such architecture. In the following, we describe the behavior of each component.

[Request Receiver]

This component receives the requests from the clients (e.g., registration of devices, gateway, media services, and operation of media services) via web APIs (shown in Table 2). The person who installs the devices uses the web APIs on the gateway side, while media service developer and IoT application developer use those on the cloud side.

[Media Service Manager]

This component manages the media service descriptions written in JavaScript Object Notation (JSON) format. Since media services to be executed on the gateway side are deployed at the time of execution, media service descriptions are centrally managed on the cloud side.

[Media Service Interpreter]

Table 2: Web APIs

URI	Method		parameters
/service	GET	Get a list of registered media service descriptions	
			id
			service
/service/:id	GET	Get the media service description specified by id	id
			id
			params
/pipeline	GET	Get a list of executed media service instances	
/pipeline	POST	Create media service instance	service:
/pipeline/:id	GET	Get the media service instance specified by ID	id
			id
			method
/gw	GET	Get a list of registered gateways	
			key
			uri
/device	GET	Get a list of registered devices	
			key
			uri

This component converts media service descriptions into executable ones for the **Media Service Executor**; this component divides media service description into cloud-side and gateway-side media services.

[Media Service Executor]

Based on converted media service descriptions (described above), this component initializes and controls media processing modules on the corresponding media server. In addition, this component establishes media sessions between devices and gateways and between gateways and the cloud. The cloud-side **Media Service Executor** cooperates with the **Gateway Manager** in order to deploy a media service to the specified gateway and establish the session between media services, while the gateway-side one cooperates with the **Device Manager** in order to establish a media session between the device and the endpoint on the gateway-side media server.

[Gateway Manager]

This component manages the relationships between ID of each gateway (i.e., keyword to specify a field site) and their URLs and provides an interface to control gateways. When receiving the requests from **Media Service Executor**, this component forwards it to the specified gateway-side module.

[Device Manager]

This component manages the relationships between ID of each devices and connection information (e.g., URL, socket ID in the case of using HTTP, WebSocket, respectively) and provides an interface to negotiate SDP and to start/stop and sending/receiving multimedia stream data.

In the following, we describe the procedure to create a multimedia IoT application using our framework functionalities. The main players in this scenario are *Field Site Administrator*, *Media Service Developer*, and *IoT Application Developer*. These players may be either the different persons respectively or the same person.

(1) Registration of the Gateway

The *Field Site Administrator* edits the configuration file in the gateway module to define the field site ID. When starting up the gateway-side module, a request for registering this gateway is automatically sent to the cloud.

(2) Registration of the Devices

The *Field Site Administrator* registers devices to the gateway-side module via using gateway-side APIs by specifying the device ID and device type. Device IDs needs to be identifiable only in the same field site since they are managed by each gateway.

(3) Registration of the Media Service

The *Media Service Developer* writes the media service description (such as shown in Fig. 5) and registers it to the cloud-side module using cloud-side APIs.

(4) Execution of the Media Service

The *IoT Application Developer* connects his/her application to the specified media service using server-side APIs to operate media services.

4.2 Details of Functionalities

In this section, we describe the detailed behaviors of the functionalities of each component above.

Interpretation/Execution of Media Services

The *Media Service Developers* describe their services in JSON format (shown in Fig. 5). This example shows a media service where the video stream captured from a specified camera at a specified field site is processed to put a specified image on the face area on the gateway side and sent to cloud side to be recorded. Here, **type** is used to specify the type of filter/endpoint, **place** is used to specify the execution place (i.e., gateway or cloud), **front_id** is used to specify the field

```
[
 {
    id: 0,
  //type of  device
  type: “DeviceEndpoint”,
  //specify the device id to use
  key:“camera001”,
  //specify a place to execute media service
  place: “gw”,
  //specify the field site id ($0 is variable)
  front_id: “$0”,
  out:[0]
 },
 {
    id: 1,
  type: "FaceOverlayFilter",
  //filter’s own property (specify image url to overlay)
  img: “http://XXXX/hat.jpg”,
  place: "gw",
  in:[0],
  out:[1]
 },
 {
    id: 2,
  type: "RecorderEndpoint",
  place: "cloud",
  in:[1]
 }
]
```

Figure 5: Example of media service description

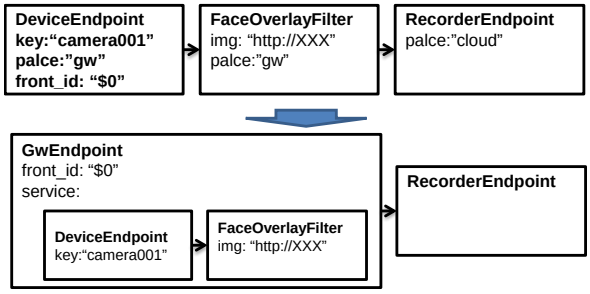

Figure 6: Transformation of media service description

site where the media service is applied, and **in/out** is used to specify the relationship between elements.

The **Media Service Interpreter** divides the received media service description and creates a *GwEndpoint* that includes a partial media service description that should be executed on the gateway side (shown in Fig. 6). Based on this converted media service description, the **Media Service Executor** initializes filters and endpoints on the media server and connects them.

In addition, the media service description can accept variable definition. For example, in Fig. 5, **front_id** (i.e., the ID that specifies where the device is installed) is defined as a variable (**“$0”**) so that it can be set when this media service is executed.

Virtualization of Media Devices

As an endpoint of multimedia stream data via a network, Kurento Media Server has three different types of endpoints: *RTSPEndpoint*, *WebRTCEndpoint* and *RTPEndpoint*. When using a camera that supports RTSP, it is required to simply specify the resource URL to establish a media session between the device and a media server, while it is necessary to

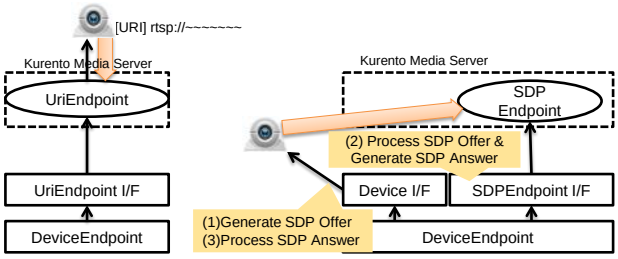

Figure 7: Obscuring the initialization procedure that varies according to device type

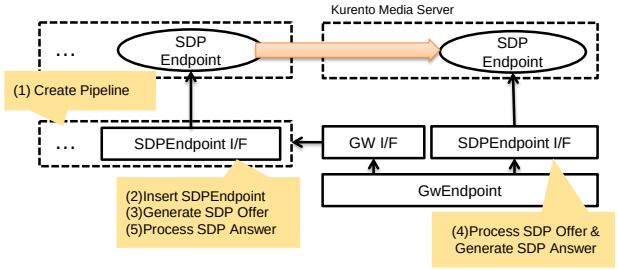

Figure 8: Session establishment between gateway and cloud

manually negotiate SDP when using a camera that supports WebRTC or RTP. Moreover, since each device may provide its own interface to operate (start, stop), developers need to take care of how to establish a session and how to operate devices that vary according to device type.

Therefore, the framework provides a set of classes, each of which implements required procedures to establish a session according to the corresponding device type (Fig. 7). *Field Site Administrators* are required to specify the device type when registering a new device. Owing to this, *Media Service Developers* do not need to be concerned about such differences in devices. A media session is automatically established when executing the media service.

Cooperation between Gateways and Cloud

To execute a media service distributedly on gateways and a cloud, it is necessary to establish a media session between divided partial media services. Therefore, the framework automatically inserts an *SDPEndpoint* (i.e., either *RTPEndpoint* or *WebRTCEndpoint*) at the end of the gateway-side *media service* description. The framework also creates an *SDPEndpoint* on the cloud-side media server and establishes a media session between gateway-side and cloud-side *SDPEndpoints* (as shown in Fig. 8). Thereafter, when the cloud-side module receives the request to operate a *media service*, it propagates this request to the corresponding gateway-side module.

Management of Events

The framework enables developers to describe event subscription between two filters in a media service description. As shown in Fig. 9, three attributes are required to define an event subscription: **target** (to specify which filter or endpoint publishes the event), **event** (to specify which type of

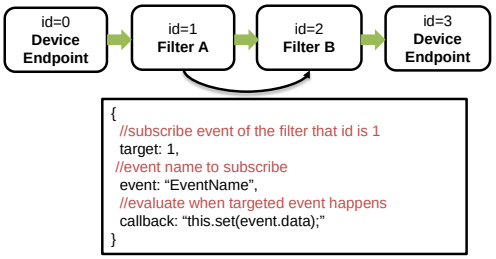

Figure 9: Event description between elements

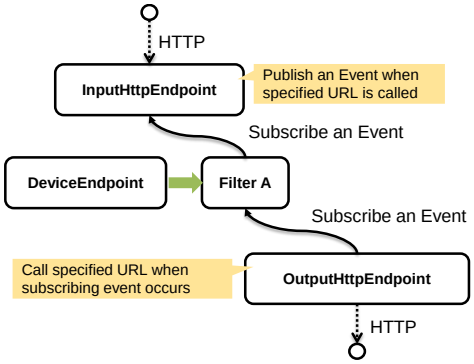

Figure 10: Input/output endpoint to external systems

event to subscribe), and **callback** (to describe the callback function that is evaluated when the specified event occurs). In the callback function, variable **this** is bound as the subscriber element itself.

Additionally, as shown in Fig. 10, the framework provides two endpoints in order to cooperate with external systems: *InputHttpEndpoint* publishes the event when a specified URL is called by an external system and *OutputHttpEndpoint* subscribes inner events and calls the external URL (specified in advance) when a specified event occurs. By using these endpoints, developers can easily create a media service that can process media stream data according to environmental changes or can store time series of metadata extracted from media stream data into external databases.

Graphical Editor for Media Service Description

Developers are able to create media service by following JSON format as shown in Fig. 5 without coding complicated logic. Furthermore, we implemented a web-based graphical flow editor for easily creating media service descriptions (Fig. 11-13). In the following, we describe how to use this client.

Figure 11 shows an example of the screen for creating and editing the media service, which is implemented using a SVG-based JavaScript library, JointJS[10]. When a user selects an item from the left-side list, a new node appears on the center area. The user can make a link from an input port of a node to an output port of another node by dragging and dropping. When a selected item requires some properties (e.g., image URL path for *FaceOverlayFilter*), input forms corresponding to each property appear on the right side of the screen. Event subscription can be defined in this area.

Figure 12 shows the screen for executing specified media service. The user can select which field sites to apply the specified media service to. Figure 13 shows a list of executed media services and the user can operate (i.e., start, stop, pause, release) each media service.

5 PROTOTYPE APPLICATIONS

In order to verify the effectiveness of our framework, we created three prototype IoT applications. In this section, we explain these IoT applications and discuss the features of each application.

[Prototype 1] Supporting Lectures in the Classroom

In the lectures at universities, teachers often use the projector to present their documents on the screen display. In such lectures, a teacher may use a stick or laser pointer to specify the focus area of the screen display. However, students may not clearly see the specified area in a large classroom. Therefore, we implemented an application that supports such lectures. We implemented it by connecting *TrapezoidCorrectorFilter* (which transforms a trapezoid-shaped area to square), *FingerDetectorFilter* (which detects the coordinates of fingertips and raises an event), and **ScalerFilter** (which expands the area around a specified point) as shown in Fig. 14.

There are three devices registered to the framework in the classroom: a camera (which captures video stream data including screen display area for detecting fingertips), a screen capturer (which captures video stream data from the teacher's PC screen), and a display screen (which displays the video stream process by *ScalerFilter*). Using this combination, the area of the screen the teacher points is scaled so that students can look at the focused area clearly. Since this media service is executed on the gateway-side, immediate response (i.e., followability of finger motion) can be expected compared with executing on the cloud-side.

[Prototype 2] Monitoring Suspicious Person

Suspicious person monitoring services using networked cameras are now widely used in various areas. However, when operating such monitoring services on the cloud, it consumes a large amount of network bandwidth and storage. Therefore, we created an IoT application that transfers a video stream data to the cloud while detecting a moving object and records it on the cloud.

We realized this by connecting *MotionDetectorFilter* (which detects moving objects using background subtraction) and *SwitchFilter* (which can be switched to drop or pass-through received buffer) as shown in Fig. 15. Since video stream data is transferred to the cloud only when moving objects are detected on the gateway side, network bandwidth and cloud storage can be saved.

[Prototype 3] Preventing Workers from Heatstroke

In summer, outdoor manual laborers are exposed to a risk of heatstroke due to both high temperatures and high-

[10] JointJs: http://www.jointjs.com/

Figure 11: Media service description screen

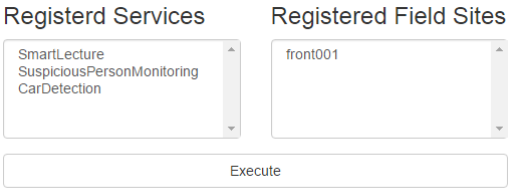

Figure 12: Media service execution screen

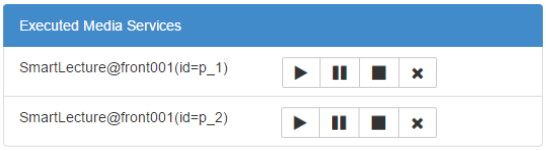

Figure 13: List of executed media services

humidity. To prevent this, there is a rule on restricting continuous work according to the heat index called WBGT[11]; however, it is difficult for the field overseer to know the WBGT of the corresponding field site and the health conditions of all workers at all time. Therefore, we implemented a monitoring application which records video stream data that captures a specified field site when WBGT is above a threshold and reports to the field overseer when a worker stops moving.

Here, we adopted an existing IoT platform that can detect events according to the changes in time series data. The WBGT value, calculated from temperature and humidity using sensors installed at the field, is continuously registered to the IoT platform. Whenever the WBGT value goes above a specified threshold, the IoT platform calls the web API defined by *InputHttpEndpoint*. This cooperation makes it possible to control media service (e.g., start recording video stream data, start detecting moving objects from video stream data) according to the changes in sensory data (e.g., WBGT). At the same time, our framework notifies the result of moving object detection to the IoT platform using *OutputHttpEndpoint*, and the IoT platform can send warnings to the overseer and workers according to the result. *InputHttpEndpoint* and *OutputHttpEndpoint* make it easy to create multimedia IoT applications that cooperate with existing IoT platforms.

[11] National Weather Service Weather Forecast Office: http://www.srh.noaa.gov/tsa/?n=wbgt

6 EVALUATION

We evaluated the effectiveness of our framework based on the above prototype applications from the perspectives as follows:

6.1 Complexity

To process single multimedia stream data cooperatively using multiple media servers, developers are required to establish a media session using RTP or WebRTC in addition to implementing originally required media processing. In addition, developers are required to take care of which endpoint to use and how to establish a media session, which varies with the device type installed at the field site. Our framework spares developers from such complexity, so IoT application developers can be dedicated to connecting devices at field sites with media processing modules.

6.2 Productivity

Table 3 shows the comparison of the number of program lines between cases using our framework and not. These numbers are counted without brackets. Although this comparison may not be fair since there is a difference between using or not using our framework (i.e., declarative description using JSON and procedural description using JavaScript), we confirmed that our framework works effectively since developers are not required to write logic to establish the media session both between the gateway and the cloud

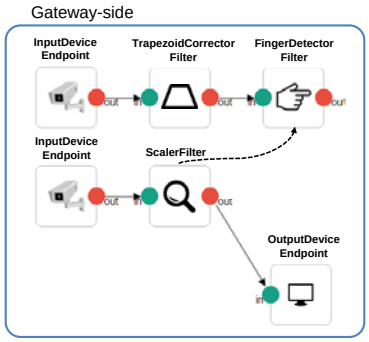

Figure 14: **[Prototype 1]** Media service description for lecture support

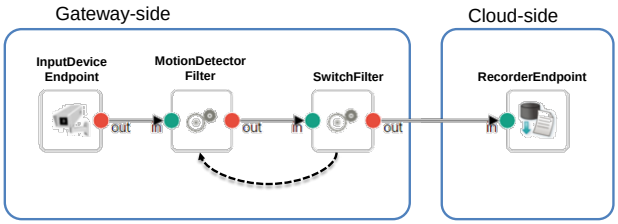

Figure 15: **[Prototype 2]** Media service description for suspicious person monitoring

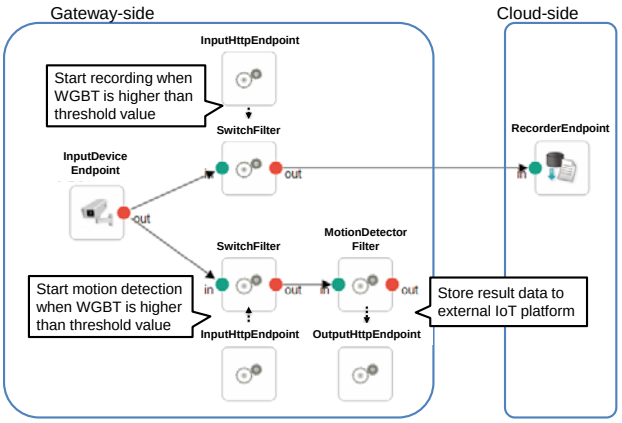

Figure 16: **[Prototype 3]** Media service description for heatstroke prevention

Table 3: Comparison of the number of program lines

	Used	Not used
Prototype 1	38	50
Prototype 2	26	42
Prototype 3	50	48

and between the media server and the device in the case of Prototype 1 and Prototype 2 (24% and 38% reduction, respectively).

On the other hand, considering the case of Prototype 3, i.e., the case where media processing pipeline topology changes according to an event, we defined a single static media service description that includes *SwitchFilter* in the case of using our framework, while we implemented this switching as IoT application-side logic in the case not using our framework. As a result, the number of lines not using our frameworks is fewer in spite of writing logic to establish a session with the gateway and the device. Therefore, the current media service description format is not suitable for describing a dynamically changing pipeline. We need to consider an effective way to describe event-driven media services as a future work.

6.3 Ease of Trial and Error

It is important to repeatedly use trial and error for creating IoT applications. However, in multimedia application, logic to handle multimedia input and output are tightly-coupled with multimedia processing logic itself. As a result, when application developers modify their application, they are required to rewrite programming logic itself. For example, in order to change the behavior of Prototype 1 to put a circle on the fingertips, developers are required only to replace ScalerFilter with another (i.e., a filter to put marker) and do not need to deploy the application again by using our framework.

6.4 Division of Labor

Considering the case of creating IoT applications such as **Prototype 1** using only Kurento Media Server, the application may not be adaptable to other classrooms, since the installed device type may differ from classroom to classroom. On the other hand, by using our framework, an IoT application once created can be adapted for any other classroom only if each device is registered in the same name owing to our framework's device virtualization mechanism. This makes it possible to create IoT applications independently from device installation.

In addition, not only cameras, microphones and screens, but also any software modules implemented to meet the specification of a virtualized device interface can be registered as a device. Developers can equally treat both cameras and screen capturer modules on our framework.

6.5 Cooperation with External Systems

Our framework can deal with not only the case where a media stream data captured from a device is processed, recorded and transferred to another device, but also the case where an external system affects multimedia stream data using *InputHttpEndpoint* and notifies the external system when an event occurs using *OutputHttpEndpoint*. In the case of Prototype 3, we adopted an existing IoT platform, but we are not limited to such IoT platforms. For example, cooperating with the existing complex event processing system enables more advanced event detection by using both sensory stream data stream and multimedia stream data.

In other words, our framework can be used as an extension to make it easy to deal with media stream data on an existing IoT platform rather than a substitute. Currently, our framework supports cooperation using only HTTP requests, but we plan to implement endpoint modules for protocols other than HTTP (e.g., MQTT, WebSocket) to make it easier to create IoT applications that deal with both multimedia and sensory data streams.

6.6 Performance

Our proposed framework can process single multimedia stream data by coordinating multiple multimedia servers running on gateways (installed at field sites) and the cloud, respectively. In typical IoT scenario, low-cost and not high-performance gateway devices are often used for each field site since it is necessary to distribute gateway devices to a large number of filed sites. In this section, we rather focus on gateway side where computational resources are restricted. We conducted a performance evaluation to confirm that our framework can work sufficiently when using a general gateway device used for IoT. Table 4 shows the specification of the gateway used for the following evaluation.

First, we discuss how many services our framework can handle when changing the quality of multimedia data and type of multimedia processing. Performance evaluations were conducted in the configuration shown in Fig. 17. A multimedia processing filter was deployed between WebRTC input and output endpoints on the gateway, while two Web browsers (Google Chrome) were used for capturing and showing video stream data using a USB camera on another machine. In this evaluation, two different quality of video streams were used; high quality (resolution: 640px x 480px, framerate: 30fps) and low quality (resolution: 320px x 240px, framerate: 10fps). Table 5 shows the result of CPU and memory usage rates when ScalerFilter, FingerDetectorFilter, and MotionDetectorFilter were used as a multimedia processing filter in the above configuration, respectively.

Compared with the case not using filters, CPU and memory usage rates increased when using filters, and these increased rates varied depending on the type of multimedia processing and video quality (11.9% - 84.30%). Our framework can handle 8-9 services simultaneously when using light-weight processing filters to low–quality video stream, while only one service when using heavy-weight processing filters to high-quality video stream.

Next, we discuss the overhead of our framework. As described in the section 4, our framework consists of two layers; multimedia processing layer and control layer (Fig. 18). Multimedia processing layer (written in C++) processes incoming multimedia stream data, while control layer (written in JavaScript (node.js)) receives requests from clients, manages registered devices, and coordinates multimedia servers and so on. We evaluated the overhead of control layer. The results were 0.2% of CPU usage rate and 4.6% of memory usage rate regardless of the type of multimedia processing filters and video quality. This result shows our extension have little effect on CPU usage at the time of execution.

Moreover, as shown in Fig. 18, our framework can handle event publishing and subscribing among multiple filters. In the example of Fig. 18, control layer subscribes an event of FingerDetectorFilter (a coordinate of a fingertip) and set the parameter (a coordinate to expand around that point) of ScalerFilter. Table 6 shows the result of CPU and memory usage rates of control layer using different framerate video streams (10fps and 30fps).

Table 4: Specification of the gateway

OS	Ubuntu 14.04 LTS 64 bit
CPU	Intel Atom E3826 @ 1.46 GHz x 2
Memory	1.8GB

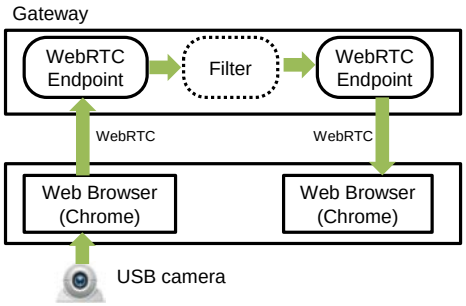

Figure 17: Configuration of evaluation environment

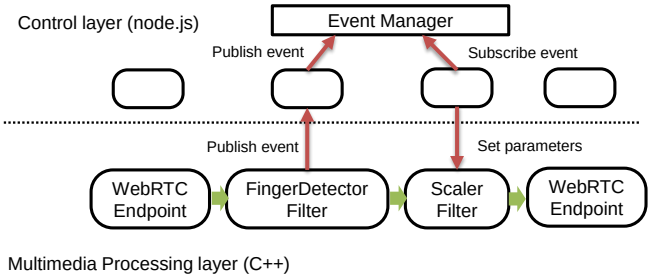

Figure 18: Proposed framework consists of two layers; multimedia processing (C++) and control (node.js) layer

The result shows that passing event data among filters had a little effect on CPU usage, and this rate was proportional to the framerate of the video stream, while memory usage rate was almost constant regardless of event subscription. From the result, we confirmed that the load of event publishing and subscribing was far smaller than that of multimedia processing.

From the above results, we confirmed that our framework worked sufficiently despite using a general gateway device for IoT when the quality of video stream was not so high or multimedia processing was not so heavy. Our framework can handle 8-9 services simultaneously when using light-weight processing filters to low–quality video stream. Besides, while our framework extends an existing multimedia server program to manage virtualized devices, to coordinates multiple multimedia servers and so on, processing overhead of this extension was sufficiently small.

On the other hand, however, our framework can handle at most one service simultaneously when using heavy-weight processing filters to high-quality video stream. If we need to use more high-quality video stream, this gateway device cannot process sufficiently. In order to provide a stable service, as a future work, we need to develop a mechanism to dynamically determine whether each multimedia processing filter should be run on the cloud or the gateway according to the performance of the gateway device and processing load of multimedia processing filters, and a mechanism to lively migrate a multimedia processing filter from the gateway to

Table 5: CPU and memory usage rates

	Low Quality (320 x 240, 10fps)		High Quality (640 x 480, 30fps)	
	CPU	Memory	CPU	Memory
None	7.4%	3.2 %	12.1%	3.3%
ScalerFilter	19.3%	5.9%	72.3%	7.1%
FingerDetector Filter	26.1%	6.4%	63.5%	7.5%
MotionDetector Filter	47.4%	6.8%	96.4%	9.0%

Table 6: CPU and memory usage rates of control layer

10fps		30fps	
CPU	Memory	CPU	Memory
1.7%	5.1%	3.0%	5.1%

the cloud if the processing load of the gateway becomes large.

7 SUMMARY

We designed and implemented a framework that makes it easy to deal with multimedia data such as audio and video generated from devices installed at field sites. The features of our framework are as follows:

- Virtualization of multimedia input/output devices
- Distributed execution of media service between gateways and a cloud
- Simple media service description using a graphical flow editor.

In this paper, we presented the three prototype applications we created and discussed the effectiveness of our framework from the perspective of complexity, productivity, and ease of trial and error.

As future work, we need to improve the media service description format and create endpoints other than HTTP. We plan to offer our framework to workshops and hackathons to verify the effectiveness of our framework from both qualitative and quantitative perspectives.

REFERENCES

[1] K. Akpinar, K. A. Hua, and K. Li., “ThingStore: A Platform of Internet-of-Things Application Development and Deployment,” ACM International Conference on Distributed Event-Based Systems (ACM DEBS 2015), pp.162-173, 2015.

[2] S. Alam , M. Chowdhury , and J. Noll., “SenaaS: An Event-Driven Sensor Virtualization Approach for Internet of Things Cloud,” IEEE International Conference on Networked Embedded Systems for Enterprise Applications (IEEE NESEA 2010), pp. 1-6, 2010.

[3] M. Blackstock, and R. Lea., “Toward a Distributed Data Flow Platform for the Web of Things (Distributed Node-RED),” International Workshop on Web of Things (WoT 2014), pp.34-39, 2014.

[4] L. Fernandez, M. P. Diaz, R. B. Mejias, and F. J. López, "Kurento: A Media Server Technology for Convergent WWW/Mobile Real-Time Multimedia Communications Supporting WebRTC," IEEE International Symposium and Workshops on World of Wireless, Mobile and Multimedia Networks (IEEE WoWMoM 2013), pp. 1-6, 2013.

[5] D. Munjin, and J. H. Morin, "Toward Internet of Things Application Markets," IEEE International Conference on Green Computing and Communication (IEEE GreenCom, 2012), pp. 156-162, 2012.

(Received October 8, 2016)
(Revised February 9 , 2017)

Daijiro Komaki received his B.E., M.E., and Ph.D. degrees from Osaka University, Japan in 2008, 2009, and 2012, respectively. He joined Fujitsu Laboratories Ltd. in 2012. His current research interests include multimedia processing framework and multimedia networking system.

Shunsuke Yamaguchi received his B.E. degree from The University of Electro-Communications, Japan in 2005, and Master of Information Science and Technology degree from The University of Tokyo, Japan in 2007. He joined Fujitsu Laboratories Ltd. in 2007. His current research interests include multimedia processing framework and multimedia networking system.

Masako Shinohara received the B.E., M.E., and Ph.D. degrees from Osaka University, Japan in 2004, 2006, and 2009. She joined Fujitsu Laboratories Ltd. in 2009. Her research interests include human-centric computing for smartly supporting user’s behavior, and multimedia networking system for real time communication.

Kenichi Horio received the B.E and M.E. degree from The University of Tokyo, Japan in 1999 and 2001, respectively. He currently works for Fujitsu Laboratories Ltd. His research interests include mobile communications and multimedia communications.

Masahiko Murakami received the B.E. and M.E. degrees in electrical engineering from Kyoto University, Japan, in 1990 and 1992. He joined Fujitsu Laboratories Ltd. in 1992. He is currently researching about human-centric connections for providing user-friendly timely services tailored for individuals, including multimedia networking system.

Kazuki Matsui received his B.E. and M.E. dgrees from Keio University, Japan in 1990, 1992, respectively. He joined Fujitsu Laboratories Ltd. in 1992. His current research interests include Virtual Reality Computing System.