An API proposal for integrating Sensor Data into Web Apps and WebRTC

João Azevedo
INESC-ID / Instituto Superior Técnico
Av. Professor Cavaco Silva
2744-016 Porto Salvo, Portugal
joao.ramos.de.azevedo@tecnico.ulisboa.pt

Ricardo Lopes Pereira
INESC-ID / Instituto Superior Técnico
Av. Professor Cavaco Silva
2744-016 Porto Salvo, Portugal
ricardo.pereira@inesc-id.pt

Paulo Chainho
PT Inovação
Rua Eng. José Ferreira Pinto Basto,
3810-106 Aveiro, Portugal
paulo-g-chainho@telecom.pt

Abstract

Today we use *smartphones* and personal computers everywhere to communicate in ways made possible by the ubiquity of the Internet: we exchange emails and instant messages, make voice and video calls and participate in social media. WebRTC furthers the possibilities of audio and video communications and true peer-to-peer communications as it brings these capabilities also to an universal platform: the browser. But communication is more than just audio or video sharing, communication is about providing features capable of filling the distance void. Today we have sensors that can provide data to enrich communication by enabling the usage of information about the context of the remote peer, e.g.: providing temperature information, speed or acceleration. Mobile devices, being equipped with a range of sensors, such as accelerometers, gyroscopes and magnetometers are prime terminals to enrich communications with context data, but all types of devices can make use of external sensing devices or even reach out to sensor networks. As the browser becomes a fundamental platform, sensor data is also relevant to non WebRTC web applications.

In this paper, we present an API proposal to enable web applications to access sensor data and an extension to WebRTC to enable peer-to-peer exchange of sensor data, bringing nearby sensor streams to web applications and multimedia communication over the web. Telemedicine, Meteorology or Seismology are prime examples of the applicability of this technology, that also enables new types of context-aware and context-based applications and communications.

Categories and Subject Descriptors H.4.3 [*Information systems applications*]: Communications Applications - *Computer conferencing, teleconferencing, and videoconferencing*; J.7 [*Computers in other systems*]: Real time

Keywords Sensor Data, API, WebRTC

AWeS'15, April 21, 2015, Bordeaux, France
Copyright 2015 ACM 978-1-4503-3477-8/15/04. . . $15.00
http://dx.doi.org/10.1145/2749215.2749221

1. Introduction

We live in a connected world, where the ability to share and interact with others fights the distance and enhances our capability to solve problems. We want to spend less time, energy and other scarce resources to achieve higher productivity. We use mobile devices and personal computers, each with its strengths and weaknesses. We have different Operating Systems (OSs), different manufacturers, different form factors. The web browser is the common denominator, offering the promise of an universal platform.

At the same time, sensors are evolving, becoming smaller, more portable and easier to interact with, via well defined Application Programming Interfaces (APIs) and using communication standards like Bluetooth [6] or WiFi [10]. Today's *smartphones* and wearable devices come equipped with a set of sensors (e.g. Global Positioning System (GPS), accelerometers, gyroscopes and magnetometers) that have enabled the creation of innovative native applications. CenceMe is an example of an application that embeds data collected from sensors into social networks [13]. However, web applications have been left out of this revolution as there are no standard, portable methods for accessing sensor data. Only GPS [17], Ambient Light [12] and accelerometer [5] data are made available. While location-aware web applications are now possible, the same does not hold true for context-based and context-aware applications, which require access to sensor data.

On the other hand, Web Real-Time Communication (WebRTC) [1, 14] has enabled browsers to access and exchange audio and video streams and data using a peer-to-peer architecture. This has lowered the barrier of entry into the complex audio and video communications, providing developers easy access to state of the art technologies. We now have access to real-time audio and video communication on the browser. WebRTC's success has prompted the adoption of WebRTC outside browsers, e.g. it is possible to find the WebRTC stack both on node.js [21], thanks to the node-webrtc module[1], and native Android applications using WebView [2].

The integration of sensor data into day to day communications opens the door to new collaboration possibilities, enhancing video or voice in Human-to-Human (H2H) communications and enabling new uses for Machine-to-Machine (M2M) communications, explored in projects such as reTHINK, and Machine-to-Human (M2H), such as in Telemedicine [8, 16].

Taking the Telemedicine example, imagine an individual, who lives in a remote location, far away from a hospital or healthcare center, starts to feel a cold. Thanks to its mobile device, he quickly accesses the webpage of its insurance company and without having

[1] http://js-platform.github.io/node-webrtc

[2] https://developer.chrome.com/multidevice/webview/overview

to download some plugin or browser extension, a multimedia session is established between him and an operator at the company's contact center. After some preliminary questions, the communication is redirected to an available doctor, which answers using a WebRTC video conference system. The patient informs the doctor that he carries an healthcare sensor kit with a thermometer, a blood pressure sensor, stethoscope and multiple electrocardiogram (ECG) channels and makes sensor data available to the doctor, improving her diagnose capability. This scenario could include other data sources or entities, like a nurse located with the patient, using a different terminal, assisting the doctor in preparing the patient for the examination.

What currently prevents this scenario from happening is the absence of a modular and interoperable solution to integrate sensor streams originated from native or nearby sensors, into web applications or web multimedia communications. In this paper, we propose an extension to the JavaScript MediaStream API to enable web applications to manipulate sensor streams with a certain abstraction level and making possible to read data to different consumers. These consumers include media elements like Hyper Text Markup Language (HTML) 5 [4], WebRTC or media recording (MediaRecorder).

Furthermore, we propose a solution for transmitting such sensor data streams using WebRTC. The proposal builds on existing APIs, namely WebRTC and MediaStream [15], and includes provision to access local sensors as well has remote sensors, to which the mobile device will act as a gateway.

The rest of this paper is organized as follows. Section 2 presents the related work, focusing on WebRTC, the MediaDevices API and sensor discovery. Section 3 presents our reference architecture proposal. Section 4 discusses some of the implementation challenges and alternatives. We present the conclusions and future work plans in Section 5.

2. Related Work

In this section we present the current state of WebRTC, particularly the MediaStream API, and discuss different techniques applicable to the discovery and configuration of sensors.

2.1 WebRTC

The arrival of WebRTC and specifically its appearance in mobile devices, created the conditions for JavaScript Engines, like web browsers, to become relevant entities on a multimedia flow between peers. Transmission protocols like Secure Real-time Transport Protocol (SRTP) [3] and Stream Control Transmission Protocol (SCTP) [20] offer a suitable layer for direct peer-to-peer communication, fulfilling the requirements for a low-latency streaming process, or requirements for guaranteed delivery or message ordering.

Developers manage video and audio streams as simple objects, abstracting away their direct manipulation but at the same time, allowing developers to define restrictions by simply creating a JavaScript Object Notation (JSON) object. Presentation is also facilitated by the ability to associate a stream to a HTML element with the capability to reproduce its content, like the video and audio HTML5 tags. The possibility to create multiple communication channels between two peers supports the existence of other multimedia streams whose type is not video or audio but, for example, sensor data.

Over the last couple of months, both the WebRTC 1.0: Real-time Communication Between Browsers and the Media Capture and Streams W3C Working Group Drafts, have evolved with some interfaces redesign or clarification and the inclusion of new entities [14, 15]. The high level APIs suffered an addition, particularly interesting to our problem: the introduction of the MediaDevices interface, defined in Listing 1, able to aggregate a set of devices, either native to the system or plugged/connected. This inclusion opens the possibility of adding more cameras, microphones or speakers, but also opens the door to the introduction of other media sources. The difference between an internal and an external webcam should be clarified on the *label* field of the MediaDeviceInfo interface and so, the object whose information this interface revels must be responsible for handling the consequent differences.

```
interface MediaDevices : EventTarget {
  attribute EventHandler ondevicechange;
  Promise<sequence<MediaDeviceInfo>>
      enumerateDevices();
};
```

Listing 1. MediaDevices Interface

A MediaStream is defined as a set of MediaStreamTracks that might be one of two kinds: video or audio, accordingly with the transported media type. Both kinds have type-oriented getters (*getVideoTracks()* and *getAudioTracks()*) and a general one (*getTracks()*) that returns all the tracks of a MediaStream object. On the MediaStreamTrack interface, four methods require a special attention: *getCapabilities()*, *getConstraints()*, *getSettings()* and *applyConstraints()*.

The three getters, apply the Constrainable Pattern, which "allow applications to inspect and adjust the properties of objects implementing it" [15]. All of them rely on the concept of capability of a given track. A Media Device should have the capacity to ensure a certain kind of constrainable properties and the set of its possible values, which may be specified either as a range or as an enumeration. The *applyConstraints* method accepts a set of parameters, MediaTrackConstraintSet presented on Listing 2, that constrain the configuration of the media source.

```
dictionary MediaTrackConstraintSet {
  ConstrainLong width;
  ConstrainLong height;
  ConstrainDouble aspectRatio;
  ConstrainDouble frameRate;
  ConstrainDOMString facingMode;
  ConstrainDouble volume;
  ConstrainLong sampleRate;
  ConstrainLong sampleSize;
  ConstrainBoolean echoCancellation;
  ConstrainDOMString deviceId;
  ConstrainDOMString groupId;
};
```

Listing 2. MediaTrackConstraintSet Dictionary

On the network transport side, protocol definition has reached an large consensus. Multimedia streams must be transmitted using SRTP and data, like strings, blobs or array buffers, must rely on SCTP to perform peer-to-peer transportation. SRTP goals are to satisfy security considerations while maintaining the low bandwidth cost. These goals are achieved by the low computational cost, small footprint, limited packet expansion and independence from the underlying transport, network and physical layers used by Real-time Transport Protocol (RTP) [19].

SCTP presents a set of relevant features to make usage of a Transmission Control Protocol (TCP) friendly congestion control, thanks to its Datagram Transport Layer Security (DTLS) [18] over User Datagram Protocol (UDP) configuration. This congestion control is also modifiable for integration with the SRTP media stream congestion control. SCTP is capable of supporting multiple unidirectional streams with ordered or out-of-order message delivery and reliable or partially reliable message transport. The flexibility behind SCTP allows developers to define different modes to fulfill the requirements of the transported data.

2.2 Resources Discovery and Configuration

Many sensors may be available in or to a device. In order to handle resource discovery and configuration, there are two possible approaches that need to be taken into consideration: proactive discovery and driver-oriented service configuration.

Proactive discovery does not require users interaction. The system via multicasting, accessing a resource directory, like with service location protocol [23], or hosting a mobile agent can find the required services on its disposal.

This method could be called in real-time and it is characterized by some kind of systems proactivity, since no Input/Output (I/O) level configuration would be managed.

Driver-oriented service configuration shows itself as a more sophisticated process. Usually this mode requires service offline configuration, using some file or script to define the communication process between the device and the local operative system's kernel, with the goal to provide the mechanisms to connect, configure, start or stop reading and writing from, for example, a certain data pipe [9].

3. Architecture

The architecture presented on this section aims to import sensor data to the peer-to-peer communications environment or simply to the JavaScript Engine, becoming available for local applications. This will enable, e.g., making mobile devices such as *smartphones* or *tablets* work as aggregation gateways of multiple sensor-generated streams. To achieve this goal, the WebRTC standard will be established as the communication technology, being the web-browser its agent. This approach will led us to the introduction of a new type of multimedia stream, sensor data, whose source could be native (on-board sensors) or peripheral (external and connected) sensors.

We propose an increment on WebRTC's architecture, in order to include new features such as a sensor source type MediaDevices and a new *kind* of MediaStreamTrack targeted to transport general data streams. Application level interpretation will be covered last.

The addition of these entities will provide WebRTC Agents with the ability to easily add external data sources, i.e. sensors, to the peer-to-peer multimedia session. The architecture will cover two different system layers. The JavaScript API layer for WebRTC must be the object of some increments in order to expose the new features provided by the native layer bellow.

We will leave out of our architecture definition the discovery and configuration protocols and methods to grant in-browser access to both native or nearby sensors. For now we will assume that these are available and configured in the browser. The subject will be revisited in Section 4.

3.1 MediaDevices

The MediaDevices interface revels to the applicational level the list of media sources available for a certain UserMedia. On our browser environment we already have access to the NavigatorUserMedia but an external entity should be defined to expose the sensors accessible by the mobile device. For simplicity, it will be considered just one entity, the ExternalUserMedia, but it should be possible to consider each sensor as a device. This new object assumes that all the external sensors are joined into a MediaDevices Object. Listing 3 shows the interface declaration for this UserMedia. By accessing an ExternalUserMedia instance, the developer will be able to discover which constraints are supported and call the *getUserMedia* method to get access to the MediaStream from that MediaDevice. Therefore, the enumeration of devices showed in Listing 1 will allow developers to identify the accessible sources during the session. The MediaDeviceInfo interface will identify the *kind* of MediaDevices, where the inclusion of a new type like, for instance, temperature, pressure or humidity, will help to determine which application requested sensors could be accessible.

```
interface ExternalUserMedia : MediaDevices {
  static Dictionary getSupportedConstraints(
      DOMString kind);
  void getUserMedia(
    MediaStreamConstraints constraints,
    ExternalUserMediaSuccessCallback
        successCallback,
    ExternalUserMediaErrorCallback
        errorCallback);
};
```

Listing 3. ExternalUserMedia interface

3.2 MediaStreamTrack

The MediaStreamTrack interface represents the primary component of a media stream, exposing the concept of an individual stream track. Its formal definition includes attribute declarations to identify the kind, id or label of a certain track and flags to query if the track is enabled, muted or its source (local or remote). Besides the introduction of the new *kind* of media, which requires to set *MediaStreamTrack.kind* to the *"data"* string value, the attributes will not behave differently. The *muted* and *enable* attributes will expose if the track is turned on or off and the difference between not being available (muted) or enabled/disabled by the applicational level.

On the other hand, the methods *getCapabilities()*, *getConstraints()*, *getSettings()* and *applyConstraints()* will imply some increments to their pattern declaration: *ConstrainablePattern*. The core concept behind *ConstrainablePattern* is the presence or absence of a certain capability. For instance, if we take a set of electrodes spread over the body, connected to an "electrode sensor", it will be necessary to tune the sensor for the small and slow signals from brain activity (electroencephalography (EEG)) or the large and quick generated by superficial muscles (electromyography (EMG)). This procedure is ensured by the use of the *applyConstraints()* method with the argument {*electrodeSource: 'brain'*}. Therefore, a new set of MediaTrackConstraints should be considered, allowing developers to add new fields to the dictionary on Listing 2, such as *electrodeSource* in the previous scenario, that could be expressed as discrete values but also as continuous values.

The evaluation of the device capabilities and constraint definition will give the developer the power to manage different tracks, but will at the same time allow the correct adaptation to environmental conditions, like network congestion or power saving mode.

A different approach is also required for peer-to-peer transmission of data. The introduction of this new track *kind*, that is data oriented, will require to rethink which protocols should handle Multimedia Streams. According with the protocol definition for Multimedia Streams by Internet Engineering Task Force (IETF), media should be handled making use of the SRTP protocol. However, when we deal with data the system is not so sensible to delays since the receiver can evaluate the data progressively but not necessarily at a certain instant, like our brain does. Also, gaps in the sensor data stream may render it useless. The SCTP protocol must be employed to wrap the sensor data streams in order to preserve guarantees such as order of the messages or assured delivery. Figure 1 shows us the increment on MediaStream API to lay over both SRTP and SCTP, considering if it is audio/video media streams or sensor data streams.

3.3 Application Level Interpretation

The power to export some functionality to user's applicational level must be correctly ensured by the infrastructure and application

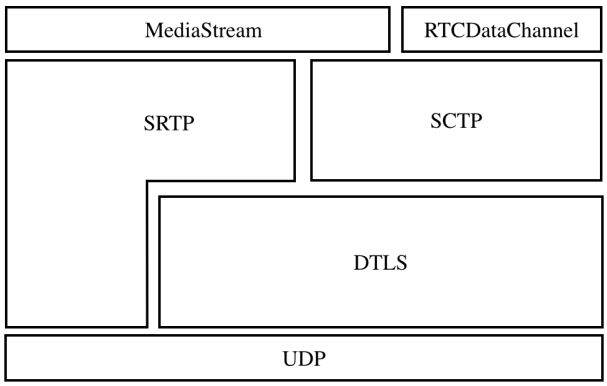

Figure 1. WebRTC Stack Redesign

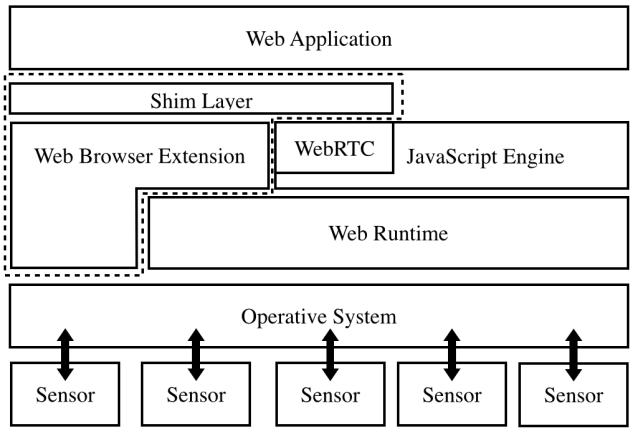

Figure 2. Prototype implementation architecture

developers. On one hand, developers want a simple and object oriented interface with the ability to abstract complex data mapping and representation. On the other hand, users just want to see a video on their screen or listen to some audio. Both audio and video have been the subjects of study and understanding in order to choose and define the correct codecs to interpret and represent them uniformly.

WebRTC defined two audio codecs and two video codecs to safeguard its interoperability. For audio, Opus and G.711 [22] where defined as Mandatory to Implement (MTI). For video, the MTI codecs are H.264 and VP8 [2]. The definition of these codecs as MTI was not a simple process. It required negotiations between different vendors.

Imagine we want to standardize a communication data representation for temperature sensors. First, we had to find an agreement between a large group of vendors and implement it on every thermometer. If we take this a little further, and try to define a unified representation among different kind of sensors, the situation will become chaotic. Data codecs do not raise the same issues as video and audio codecs raise at the representation level. We can easily define a method that translates every raw data representation into a well defined set of variables, that could be represented on a graph or a simple scale.

HTML5 and Cascading Style Sheets (CSS) 3 allow us to represent data on a friendly way and the introduction of tools like chart.js [3] and canvas.js [4] increase the capacity to concede extra functionality to applications. We may focus only on the layer between raw data from a stream and its application level representation. This translation layer should rely on the definition of a data codec that must be loaded by the application developer.

The notion of Codec-On-The-Fly, introduced in Wonder [7] and reutilized on project reTHINK, allows an application developer to include an external translator script to the application. This mechanism will grant the developer the capacity to build a sensor-oriented translation entity that will exhibit streamed data on a more friendly way. In order to achieve it, some *display()* function must be defined by the programmer.

3.4 Prototype implementation architecture

The implementation of a system like the one we propose is an ambitious process and needs to be conceived in detail. We intend to implement our architecture through a prototype on a mobile environment, where access to a richer set of sensors is available. This will enable us to validate the API by developing a set of applications that represent major use cases. Android will be the target platform as it presents the best WebRTC support. We will need to combine a WebRTC platform with native code to access sensors. The two alternatives we are evaluating are: the use of the Firefox Browser (which provides a Plugin extension mechanism), or the use of a native application where the native Android code is binded with JavaScript using WebView. The scope of intervention for either case is presented in Figure 2, inside the doted area.

In order to avoid having to perform complex changes to the WebRTC implementation, we plan to create a shim layer to complement it. This will provide access to the new functionality while passing through the rest of the calls to the underlying WebRTC implementation.

Access to the sensors will be provided by a native code layer in the form of a Web Browser Extension. This layer will detect, configure and access the sensors, be it in-device of off-device. It is also in this layer that the sensor data streams are created. We will use an open-source WebRTC implementation (most likely the one from the browser) to transmit streams to peers using SCTP.

While native code can use the OS APIs for discovering the available in-device sensors, the same will not hold true for accessing off-device sensors. On subsection 2.2 two solutions were presented to deal with the issue of discovery and configuration of a connection between two devices. In our prototype, we will follow the driver-oriented approach, where off-device sensors will be configured in the Web Browser Extension layer. These will then be presented to the web application through the modified MediaDevices interface.

4. Discussion

In this section we discuss some of the choices made in our architecture and prototype implementation plans.

We choose to provide access to sensor data using streams, while this could have been done in different ways, such as providing a method for reading current sensor values. The introduction of sensor data streams to the MediaStream API expresses the importance that non-audio/non-video streams have on today's systems. This enables a uniform access to sensor devices, inline with what already exists for audio and video devices. For the transmission of sensor data, data channels could have been used. Instead, by handling sensor generated data as streams, developers can count on

[3] http://www.chartjs.org

[4] http://canvasjs.com

timestamped channels that easily allow synchronization with video or audio streams.

Sensor devices may be very complex. They may range from in-device accelerometers to a Wireless Sensor Network (WSN) where each node measures multiple values. *Which attributes should characterize a certain device?* If we consider a restricted set of attributes, the developer will have difficulties to fully understand the capabilities of the device but, on the other hand, if we specify each detail of the device, the process of configuration will be too complex. A successful API should abstract this complexity from the developer while preserving the possibility of configuring complex devices. We made a compromise between flexibility and ease of use. Web developers should only have to choose from the set of sensors made available by the browser. Configurations by the web application should be restricted to the minimum, such as defining the sampling rate or the example provided in the Section 3.2. Device configuration and tuning may be performed by the user by resorting to native applications or an extension configuration interface in the browser, both provided by the sensor manufacturer.

Another factor stands on the disassociation of MediaStream API from WebRTC, that allow the integration with different consumers such as new HTML tags or other communication mechanisms such as WebSockets [11].

In our current architecture proposal, we assume that sensor data samples are too important to be lost when being transmitted over the Internet to another peer. The use of SCTP ensures this, but can lead to large delays if the sampling frequency is to high. However, there may be scenarios where a high sampling rate with eventual gaps is preferable to a slower sampling rate without loses. Offering the developer the possibility to chose between using SRTP or SCTP would enable him to adapt the system's behavior according to his usage scenario. For simplicity, in this first proposal we have decided to restrain stream transmission to SCTP pending real-world validation.

5. Conclusion and Future Work

The amount of different sensors that equip our mobile devices, our homes and in the future even our cloths, have already begun to spark new revolutionary applications. However, web developers have been left out, as despite all its many innovations, HTML5 does not yet contemplate access to sensors other than GPS and accelerometers.

In this paper we propose an extension to the WebRTC APIs to enable general access to sensors. Our extension proposes to extend the MediaDevices interface to allow web applications to discover and access sensor devices both in-device and off-device configured into the browser. WebRTC is leveraged to enable to peer-to-peer transmission of sensor data. Our proposal will be evaluated through an implementation we are starting to build and of which we present the architecture. We will use the implementation to validate some use cases, such as Telemedicine.

This paper also aims to foster the discussion about *what do we need during a conference call?* Taking WebRTC's architecture as a communication vehicle, sensors were added to enriched our ability to communicate and enable support for new use cases.

Acknowledgments

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013. This work has received funding from the European Union's Horizon 2020 research and innovation program under grant agreement No 645342; project reTHINK.

References

[1] H. Alvestrand. Overview: Real Time Protocols for Browser-based Applications. Internet-Draft draft-ietf-rtcweb-overview-13, IETF Secretariat, November 2014.

[2] J. Bankoski, J. Koleszar, L. Quillio, J. Salonen, P. Wilkins, and Y. Xu. VP8 Data Format and Decoding Guide. RFC 6386, RFC Editor, November 2011.

[3] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711, RFC Editor, March 2004.

[4] R. Berjon, S. Faulkner, T. Leithead, S. Pfeiffer, E. O'Connor, and E. D. Navara. HTML5. Candidate Recommendation, W3C, 2014.

[5] S. Block and A. Popescu. DeviceOrientation Event Specification. Last call WD, W3C, 2011.

[6] S. Bluetooth. Specification of the Bluetooth System, version 1.1. *http://www. bluetooth. com*, 2001.

[7] P. Chainho, K. Haensge, and S. Druesedow. Signalling-On-the-fly: SigOfly. WebRTC Interoperability tested in contradictive Deployment Scenarios, November 2014.

[8] P. Y. Chau and P. J.-H. Hu. Investigating healthcare professionals' decisions to accept telemedicine technology: an empirical test of competing theories. *Information & management*, 39(4):297–311, 2002.

[9] R. Chaudhri, W. Brunette, M. Goel, R. Sodt, J. VanOrden, M. Falcone, and G. Borriello. Open data kit sensors: mobile data collection with wired and wireless sensors. In *Proceedings of the 2nd ACM Symposium on Computing for Development*, page 9. ACM, 2012.

[10] B. P. Crow, I. Widjaja, J. G. Kim, and P. T. Sakai. IEEE 802.11 wireless local area networks. *Communications Magazine, IEEE*, 35(9):116–126, 1997.

[11] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011.

[12] A. Kostiainen and D. Turner. Ambient Light Events. Last call WD, W3C, 2014.

[13] E. Miluzzo, N. D. Lane, K. Fodor, R. Peterson, H. Lu, M. Musolesi, S. B. Eisenman, X. Zheng, and A. T. Campbell. Sensing meets mobile social networks: the design, implementation and evaluation of the cenceme application. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 337–350. ACM, 2008.

[14] A. Narayanan, C. Jennings, A. Bergkvist, and D. Burnett. WebRTC 1.0: Real-time Communication Between Browsers. W3C working draft, W3C, 2013.

[15] A. Narayanan, C. Jennings, D. Burnett, and A. Bergkvist. Media Capture and Streams. W3C working draft, W3C, 2013.

[16] C. C. Poon, Y.-T. Zhang, and S.-D. Bao. A novel biometrics method to secure wireless body area sensor networks for telemedicine and m-health. *Communications Magazine, IEEE*, 44(4):73–81, 2006.

[17] A. Popescu. Geolocation API Specification. W3C recommendation, W3C, 2013.

[18] E. Rescorla and N. Modadugu. Datagram Transport Layer Security. RFC 4347, RFC Editor, April 2006.

[19] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. STD 64, RFC Editor, July 2003.

[20] R. Stewart. Stream Control Transmission Protocol. RFC 4960, RFC Editor, September 2007.

[21] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*, 14(6): 80–83, 2010. ISSN 1089-7801. .

[22] J.-M. Valin and C. Bran. WebRTC Audio Codec and Processing Requirements. Internet-Draft draft-ietf-rtcweb-audio-05, IETF Secretariat, February 2014.

[23] J. Veizades, E. Guttman, C. E. Perkins, and S. Kaplan. Service Location Protocol. RFC 2165, RFC Editor, June 1997.