Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 07, 2014

R. Raymond
E. Lagerway
Hookflash
I. Baz Castillo
Versatica
R. Shpount
TurboBridge

July 06, 2013

WebRTC JavaScript Object API Rationale
draft-raymond-rtcweb-webrtc-js-obj-api-rationale-01

Abstract

This document describes the reasons why a JavaScript Object Model approach is a far better solution than using SDP [RFC4566] as a surface API for interfacing with WebRTC. The document outlines the issues and pitfalls as well as use cases that are difficult (or impossible) with SDP with offer / answer [RFC3264], and explains the benefits and goals of an alternative JavaScript object model approach.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

While the IETF RTCWEB WG is not specifically tasked with providing an API by the W3C, the group has effectively defined a surface API with the mandate to use SDP [RFC4566] with offer / answer [RFC3264].

SDP is a condensed text based format that typically describes all of the real-time media streams, networking properties, codecs, media state and media attributes. SDP is completely extensible and can be used to describe absolutely anything so long as it is formatted correctly within its minimally defined limitations.

The points for mandating SDP with an offer / answer API typically boils down to:

1. It's really easy to establish communication, especially with SIP [RFC3261].

2. The decision was already made.

3. SDP yields greater compatibility (especially with SIP networks).

4. We must have some kind of universal exchange format.

5. There is no alternative to this approach except destroying everything created and starting from scratch.

This document will explain why these reasons are insufficient to continue with an SDP with offer / answer mandate approach given strong logical arguments and reasons with real world scenarios where this approach fails and due in no small part to its lasting consequences (including negative consequences for SIP).

The document highlights the benefits and goals for a different "JavaScript Object Model" approach, which satisfies the RTCWEB WG charter's requirements, yields greater compatibility and offers a road-map where future potential extensions can be readily added without breaking existing implementations.

A "JavaScript shim" is described including details on how it can offer a wrapped API around a core WebRTC JavaScript Object Model. This Shim will provide the same level of "ease of use" as experienced with the current SDP WebRTC API. However, this JavaScript shim is not mandatory to use for those who do not require an "SDP with offer / answer" model.

1.1. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Issues with a Universal Session Description Format (and Offer / Answer)

The issue with SDP is not the expressiveness of the format but its usage as an arbitrary universal format and an API surface instead of providing JavaScript developers an object model they can readily understand. JavaScript could be used to control the plumbing of media objects using familiar JavaScript expressive concepts enshrined with methods, properties and events. Today, in many real-world use cases, controlling WebRTC requires modifying SDP directly.

Requiring JavaScript developers to serialize their API control requests into a text format (via modifications of SDP existing blobs) is only one aspect of the many issues the SDP approach creates for developers. Needlessly, an offer / answer state machine is imposed on JavaScript developers as well.

While the currently mandated SDP based API allows developers to quickly implement basic calling demos and interoperability with some SIP networks, it has many issues that will be explored and explained in this document and include (but not limited to):

1. Defining a standard universal all-encompassing session description format for use with WebRTC that describes all connections, media, constraints, streams and tracks for all scenarios is especially challenging.

2. Rather than focusing and defining the properties needed for communication, the focus is put on the best way to express the format where every nuance and behavior will need to be detailed for any browser vendor to capably implement the SDP based WebRTC specification.

3. The bar for browsers (or other applications with WebRTC engines) to produce a WebRTC engine is raised substantially by forcing the browser to implement an entire SDP offer / answer engine too, with little to no added benefit.

4. A universal format built into the browser's API is entirely unneeded and goes well beyond the RTCWEB chartered mandate for the RTCWEB Working Group.

5. A flexible and expendable universal exchange format leads to greater interpretations and mistakes in various implementations, which in turn leads to increased incompatibilities.

6. Given the format is entirely flexible and open to interpretation, resulting implementations will more likely be prone to errors relative to the other truly needed aspects of RTC (which have better defined boundaries, behaviours, and scope).

7. Mistakes in the format won't be fixed until a new browser binary update is released and deployed amongst users.

8. Mistakes in implementation of the session description format can become enshrined and difficult to deprecate (for the sake of compatibility).

9. Compatibility issues caused by the format will not be limited to browsers-only as many hybrid browser-engine based applications now exist too.

10. Using alternative signaling formats will require complete understanding of the universal format to be able to translate it into other alternative signaling formats.

11. JavaScript (or proxies) will need to parse and rewrite the output session description format with 100% precision and without loss. They will also require pre-knowledge of what each browser produces and expects, despite the likelihood of a multitude of outputted flavors, on various platforms, and from version to version and despite the inability to easily predict or detect the variants.

12. JavaScript developers trying to control WebRTC features will need to manipulate any defined universal format rather than interacting with JavaScript objects.

13. Offer / answer is mandated and the state machine is required but the exact rules and violations of the rules ill defined when used within WebRTC.

14. The rules of how a universal format can be modified before being delivered to remote parties need to be meticulously defined or compatibility issues will arise (including the allowed rules of post browser format regeneration as to what can be modified and fed back into the browser).

15. Due to the issues defined above, SIP compatibility will worsen, not strengthen.

An alternative to all of the issues caused by a universal format and state machine are described later in the document. This alternative allows JavaScript to control the behavior of the media engine's plumbing while providing extensible and modifiable shims written entirely in JavaScript that produce consistent signaling and exchange formats for the specific network where those formats operate.

2.1. Goal of Minimized Requirements

While the primary goal of WebRTC is to enable browser to browser communication, the definition of a "browser" is ever expanding. Beyond just traditional hand-held applications, hybrid applications that are part HTML-5 and part native code exist. Servers will become as much as part of the WebRTC infrastructure as browsers. Minimizing the requirements to the basic wire compatibility necessary to achieve RTC is essential for maximum compatibility, flexibility and varying usage scenarios.

The mandate for the RTCWEB charter is to simply define requirements, provide basic "on-the-wire" compatibility, and define security requirements (such as enforcing ICE connection agreements). The RTCWEB charter goals have been exceeded by going well beyond that scope by mandating an API that works fine for simple SIP interoperability demos but does not provide easy compatibility to the basic constructs needed as outlined from the charter for use with other on-the-wire signaling protocols (other than SIP). If SIP is the only end goal of the WG, then that goal must be specifically stated rather than effectively mandated by making alternative signaling approaches unreasonably difficult to achieve.

2.2. Offer / Answer State Machine

The current SDP approach requires an offer / answer state machine. Mandating an offer / answer state machine implies that:

1. SDP be generated by browser A and sent to browser B

2. Browser B must respond with the offer with an answer

3. If either party issues a new offer but the offer is rejected, the state must revert to the previous agreed SDP (or fail to none)

4. If one side receives an offer while the other side has an outstanding offer, a conflict occurs and both sides must reject and revert and perform SDP conflict resolution to issue an offer again

5. The only changes to the media that are allowed happens if both sides agree

6. Any change required to the SDP requires a network round trip where both sides mutually agree (at least as traditionally defined in offer / answer but the rules are in flux)

This offer / answer model is defined as required with the current implementation. Not only do the browser vendors have to enforce the rules, all JavaScript authors must also adhere to these rules of signaling. While WebRTC does not dictate the signaling mechanism between browsers, effectively it is imposing this signaling state machine on all implementations (which is not a mandate of the RTCWEB Working Group).

There are other models for signaling other than offer / answer. For example, one-sided constraints based negotiation is an alternative model. This type of negotiation requires each side to determine what it wants to receive independent of the other. This signaling is akin

to saying "if you plan to send anything, make sure it conforms to the following". Changes to the media may occur without agreement from the remote party where each side decides what is acceptable to receive without agreement from the other. The remote side can decide if it wants to send within those constraints or not. There is no round trip offer / answer required in this model to affect change.

Offer / answer introduces the unnecessary asynchronism to the API and JavaScript implementations. For example, changing the list of codecs expecting to receive or the current sending codec can be done immediately without the need for asynchronous calls.

Offer / answer is not required to achieve RTC wire compatibility but it is currently mandated when alternatives could exist.

2.2.1. Offer / Answer Violations

The offer / answer SDP state machine is already violated in WebRTC. Trickle ICE precludes offer / answer round trips and other proposed standards like NoPlan [I-D.ivov-rtcweb-noplan] suggest relaxing the offer / answer model even more. The rules of what offer / answer at this point is undefined and in clear violation of the strict previous rules without clear direction on what exactly constitutes offer / answer anymore and where it should and should not be used.

A new state for offer / answer called PRANSWER is now defined, which did not exist as part of the standard offer / answer state machine. Offer rollback is not adequately defined either should an offer / answer conflict occur.

Currently, switching codecs requires an SDP offer / answer should perform a round trip even though it is not technically needed for an RTC engine to change codecs. Should this be another exception to the offer / answer state machine?

2.3. Browser to Browser Format Compatibility Issue

SDP is a flexible format, and it allows many alternative methods to express the same intentions. The smallest change may alter the SDP's meaning.

This creates a parsing and SDP generation compatibility issues. If SDP is packaged by JavaScript and delivered to the remote browser then each browser must support every single possible variant of SDP for every browser version and platform in existence. They must do this without failure. To maximize compatibility, a browser should generate the SDP format in the variant expected by the remote party (despite not having sufficient knowledge about the remote party to provide the correct SDP).

2.4. Browser to JavaScript Compatibility Issues

Since WebRTC is not supposed to mandate the format on the wire for signaling, one supported use case for WebRTC must be allowing the browser generated SDP to be converted into alternative on-the-wire formats. This SDP conversion may be performed by JavaScript in the browser, or later by an intermediate gateway. In either case, the converter must be entirely aware of all variants to the SDP possible from every browser platform and version, despite browser version detection being heavily frowned upon by industry best practices. Likewise, the JavaScript or gateway must know how to generate the correct SDP for all browsers and versions before passing the serialize SDP blob into the browser. Generating compatible SDP may be impossible unless the exact formats and restrictions are unquestionably clear by all implementers of the specification (which is anything but clearly described in the current WebRTC SDP based API that developers are mandated to use).

2.5. SDP as a surface API for JavaScript developers

The current SDP based API is limited to placing a call and answering a call and adding media. To perform common edge cases or to utilize RTC features beyond the basic API typically requires SDP mangling.

Many of the operations from JavaScript to control or fetch properties from RTC will be through serialization to / from the SDP instead of a developer using familiar JavaScript language constructs (e.g. object methods, structures, properties and events). The JavaScript developer must learn an entirely new protocol called "SDP" and be able to parse and generate not only basic SDP but any SDP extensions without introducing a single compatibility issue.

Examples; A JavaScript developer wants to hold / un-hold media streams. The developer must use a widely adopted but hidden feature to parse the SDP from the browser, change it to add the appropriate "hold" state, send that hold state to the remote side, wait for the "answer" to accept the hold, parse the result on the return to see if the hold was accepted and feed the result to the browser.

Worse, a flood of extensions to SDP for WebRTC are being written to "enhance" and "extend" the functionality of the browser with new features. Many basic things are ill defined in the current SDP based API, for example, changing non-negotiated codec parameters, such as codec bandwidth.

There is no facility for JavaScript to detect what SDP the browser is currently using or capable of delivering. The developer has no idea of the extensions available, or what SDP will be produced, or what SDP is compatible. The developer's JavaScript code must be able to handle everything generated by the browser for any use case beyond basic call, answer and hang-up. This is a heavy burden to place on a JavaScript developer who is not familiar with the details of RTC concepts as expressed in SDP, and is a challenge even for those who are familiar.

Effective APIs are meant to be contracts between a producer and consumer, whereas this SDP methodology offers little in the form of any such contract.

If SDP is to become standardized for use with WebRTC then JavaScript developers must learn SDP to use RTC's available features and build new features. Alternatively, accessors will need to be provided to manipulate the SDP on behalf of the JavaScript (and if so, then why not move to an object model straight away and do away with SDP?).

2.6. Is SDP allowed to be mangled?

The choice must be made if SDP may be modified or not. If modifications are the only way to achieve RTC features available then what is allowed to be modified must be clearly defined in exact detail and the expected behavior of each feature (and modification of each feature), as expressed in SDP, must be defined. Anything short of exact specifications will cause incompatibility. Again, the implication is that Web / JavaScript developers must learn SDP to utilize the available RTC features and they must learn the rules of modification equally well, which virtually do not exist at all today.

If the choice is to not allow complete SDP modification at all, then the protocol becomes extremely tied to SDP based protocols like SIP. Yet, there is no mandate for SIP to be the standardized protocol in WebRTC. In fact, the mandate to require SIP was explicitly denied, which presents the argument that SDP manipulation must be allowed.

The SDP mangling issue isn't just an issue when the format is sent on-the-wire. If Browser A sends Browser B an SDP, the current philosophy is that the SDP is allowed to be modified. However, there is the possibility of modifying the SDP generated by Browser A and

giving that modified SDP back to Browser A to change it's behavior (i.e. a serialized text based API call) before the offer is given to Browser B (and likewise with Browser B when it responds with its SDP answer).

How much of the SDP is allowed to be modified before giving the SDP back to the local browser? SDP is a free-form format so anything can theoretically get changed, but should it be allowed? If not, what can and cannot be modified? CODECS? SSRC? SDES? Fingerprints? Transports? M-lines? And so on...

This issue becomes further compounded when extensions are factored in as well.

2.7. SDP errata and bugs compatibility issues

With the SDP baked into the browser binary, the only way SDP compatibility issues can be fixed is by releasing a new browser update, and the JavaScript developers must support or work around flaws until the browser vendors deliver the fix and the user base upgrades their browsers.

While it could be argued that any bug must be worked around, SDP is a unique problem. SDP is a free-form format. Being compatible isn't as easy as implementing a limited wire protocol for media transport or a API contract with well defined features and attributes. The likelihood of free-form SDP containing errors is far greater than a typical well defined API due to SDPs many flavors, interpretations and lack of strong definition.

2.7.1. SDP Bugs Become Enshrined

To illustrate a scenario:

1. Browser Vendor A has a bug

2. Browser Vendor B can't work with A because of the bug so it implements a "work around"

3. Browser Vendor A fixes the bug but implements a work around to be compatible with Browser Vendor B's "work around"

This situation demonstrates is how browser bugs can become enshrined as there's no way to update the SDP produced by the browser binary once it's released until the next update release cycle occurs. This would not be true if JavaScript was used via a shim to produce SDP as JavaScript can be dynamically updated as needed at any time and a service provider can choose to update their JavaScript implementation

to exacting expectations for their network regardless of the browser version.

The lower level RTC wire protocols that need to be mandated by the RTCWEB Working Group have limited scopes and well defined behaviors. Any mistakes are obvious, likely to present very rapidly, and easy to spot which party is doing something wrong and much easier to fix earlier as a result. This is not true with a free form highly descriptive language for sessions. The combinations are limitless and every scenario is difficult to test, especially in concert with every other browser vendor with every version released. The session description will be the likely place of failure across the browsers when the session description is generated inside the browser's binary.

2.8. SIP/SDP compatibility worsened

One of the main arguments for using SDP with offer / answer was supposed to be ease of compatibility with existing signaling networks, like SIP. Instead, variations in the browser's SDP will likely worsen SIP compatibility instead of enhance it.

A SIP provider must now be compatible with every browser's SDP on every platform and version and the browser's SDP must be compatible with every SDP from a SIP network. Alternatively, JavaScript or SBCs (Session Border Controller) must be used to re-write any incompatible SDP to be compatible. However, this moves the problem from the browser to JavaScript, or requires SBCs to "fix" the problem.

Had SDP been entirely generated by JavaScript rather than come from the browser engine, the JavaScript could create only SDPs compatible with a particular SIP provider under control of their own JavaScript and the SIP provider could chose which JavaScript SDP parsing / generation code to run, for maximum compatibility.

2.9. Increased surface API

By mandating SDP, the requirement for compatibility with WebRTC is increased substantially with little benefit. Instead of just supporting basic media RTP [RFC3550], STUN/ICE/TURN [RFC5389]/[RFC5245]/[RFC5766], DTLS [RFC6347] and CODECS an additional bar must be passed, i.e. a browser or other WebRTC compliant API must support SDP with a full offer / answer state machine (or a state machine with additional rules to make it flexible for various scenarios).

With an alternative approach, the entire requirement for SDP could be removed without any loss of compatibility or increase in complexity while achieving greater compatibility via the JavaScript shim.

2.10. Impossible API to implement to achieve browser compatibility

The current mandated SDP based API cannot be implemented as a standard by independent browser vendors in its current form. A list of subsequent behaviors regarding the usage, parsing, handling, extensions, behaviors, constraints and other such reference documents must be meticulously defined for SDP with the modified offer / answer state machine or no browser can ever claim to be "compliant". The current definition process is far from complete.

The current WebRTC SDP based API is far from achieving that goal due to the inclusion of free-form SDP with offer / answer and it is grounds for removing it as it goes beyond the RTCWEB's charter and limited scope.

Any incremental approach that does not remove the offer / answer model requirement yields a road block to achieving alternative WebRTC signaling protocols other than SIP.

An alternative WebRTC JavaScript object model approach that does not require an all-encompassing session description and related state machine is being proposed as an alternative solution so the RTCWEB charter can complete its defined goals in a timely fashion.

2.10.1. Example Oddities That Need Definition

There are many oddities in the SDP RFC [RFC4566] and the various related extensions.

For example; will RTP CODEC maps be required or not? They are not required for basic CODECs according to the SDP RFC. However, with all the flavors of CODECs being offered, defining a mapping between payloads is critical to compatibility and not just a good idea.

Another example; should "t=0 0" be respected? Is that allowed to be changed? Do the browser vendors need to enforce the attribute, or should the JavaScript layer enforce it? Should the streams wait to start until the NTP time stamp and close when the NTP time completes?

These are just small samples of questions that must all be completely addressed in detail. This could also cause a cascade of updated reference drafts and confusion as to which version is to be adhered by browsers as well as what each browser specifically supports. Nominally referencing the SDP RFC will not be sufficient, and deltas from the established standards when violated will need to be defined when the rules change.

2.11. Plan A, Plan B vs NoPlan

At the time of authoring this document, three plans on how to handle large number of media streams in SDP have emerged currently under consideration from the IETF, referred to as PlanA [I-D.roach-rtcweb-plan-a], PlanB [I-D.uberti-rtcweb-plan] and NoPlan [I-D.ivov-rtcweb-noplan].

PlanA and PlanB acknowledge that using SDP as it is historically defined in SIP is inefficient and problematic for large number of media streams, especially factoring in that each media line must have its own unique ports.

NoPlan allows for media to be described in a more JavaScript friendly way and goes a long way towards improving the situation from SDP by taking out the mapping of the streams from the SDP but does not remove the reliance upon SDP. This creates a dual format system where some information is initially carried over SDP and other information is signaled through an alternative approach (including the possibility of SDP offer/answer). NoPlan could have been the sufficient approach if it took one step further and removed SDP entirely.

PlanA, PlanB and NoPlan are a perfect example of why not to use SDP as the basis for WebRTC. SDP has some arbitrary limitations as a description protocol for multiple streams whereas no such limitations exist at the lower layer transports themselves. RTP allows for multiplexing multiple SSRCs. In other words, the problem is SDP, not the real time transportation technologies.

These drafts illustrate the limitations of SDP and attempt to solve it by introducing even more complex descriptions around SDP and / or by "relaxation" of the offer answer model combined with altering the description language of SDP.

None of these drafts address most of the concerns outlined in this draft. If anything, they further illustrate how divergent the SDP will become as more and more effort is put into working around problems inherent to the nature of utilizing SDP (or any universal format).

The issue that SDP implementers face should be isolated to those who require SDP for their signaling protocols (namely SIP) where they can choose the best practices for their networks for interoperability. These complex approaches do not have to be forced on other signaling protocols that do not have or require such limitations.

Certainly JavaScript programmers and the W3C should not be impacted by such limitations by introducing SDP (or any universal format) into the mix when it adds zero value and fails in its primary objectives, namely: interoperability with existing SIP vendors & networks.

This further illustrates why SDP baked into the browser binary is not beneficial for SIP vendors either. They will be forced to upgrade their SIP infrastructure to support SDP packets from browsers with these kinds of extensions or be forced to utilize a JavaScript SDP re-write of SDP approach to "fix" these incompatibilities.

With an object approach, newer signaling protocols could describe multiple media streams with ease and SIP providers could ensure they only generate compatible SDP with their networks and agree on their best practices and launch new features that incorporate approaches like as PlanA, PlanB or NoPlan in a manner they deem fit rather then when the browser vendors decide to upgrade the SDP arbitrarily.

2.12. SIP Forking Issue

The current SDP based API model does not allow for SIP parallel forking even though the RTC engine can allow for demuxing a media stream. The current model does not allow for one offer to be transmitted but accepts multiple answers, which is legal in SIP. A complex UPDATE process is described on how to work around the problem instead of fixing the original problem, i.e. the state machine being required.

A WebRTC JavaScript object model is designed to easily allow forking but does not care if an upper shim supports SDP / SIP style forking in the negotiation or not, so long as the basic rules of the RTC media engine is respected.

3. Alternatives to Fixing these Issues Now

3.1. Waiting for WebRTC 2.0

If we don't get WebRTC 1.0 correct, fixing the API in WebRTC 2.0 may become even more difficult.

At this stage, prototypes are underway but to our knowledge there are no major commercial services deployed by more that one major vendor using the current WebRTC API. Yet, the argument to even consider an alternative is that 'it's too late'. Imagine trying to argue fixing it after major networks are reliant upon specific browser implementation. Having a good but simple architecture from the start could alleviate a lot of pressure to fix a broken 1.0 in a 2.0 release before APIs become entrenched.

3.1.1. Cost now to fix versus fixing later

The cost of fixing the API issues today may pale in comparison to the cost of compatibility problems spread across entire sets of industries where constant fixes and work around may be required.

3.1.2. If starting over, would even SIP people want SDP as a surface API?

Even SIP providers and vendors have started to realize that baking SDP into the browser is not necessarily in their best interests, but they do have an interest in a simple API to use since they aren't specialized JavaScript developers but SIP integrators.

If an alternative approach provides SIP providers a simple JavaScript API shim they desire and achieves greater interoperability because of predictable, controllable and tailored SDP for their network, would they not prefer such a model over the current "baked in the browser" approach?

If the current WebRTC specification was ever rebooted, the current mandated SDP based API would undoubtedly be scrapped in favor of a better approach without its inherent design and use case flaws with negative long term compatibility consequences.

3.1.3. Incremental Approach may make Compatibility Worse

One argument put forward, to keep the current SDP model, proposes the current WebRTC SDP-based API must be completed soon and an incremental improvement approach can be used to gradually move away from these obvious problems.

The trouble with an incremental approach is that it may increase incompatibility further. Not all browser vendors will match the incremental improvements in unison nor will all customers upgrade simultaneously. This puts the onus on JavaScript developers to support multiple versions of the WebRTC API and increase the number of APIs they must learn and maintain. The JavaScript developers must still perform all the workarounds required for the current API even if they support the increments. This limits their willingness to use any additional APIs until all browsers universally support the incremental improvements. This will likely slow innovation and adoption of future improvements.

This will likely create a situation where browser vendors cannot easily achieve compliance because they too must support the existing API and incremental improvements along the way, or break those reliant upon the current methods.

Having a good solid simple foundation is key to ensuring basic compatibility while allowing for innovation to occur for those developers who are willing to give new APIs a trial without needing to support multiple sets of equivalent but incompatible APIs simultaneously.

3.2. Session Description Format Construction API

An alternative JavaScript model has in the past been floated around, other than the model advocated in this draft. That model creates a JavaScript session description format construction API in the browser. Such an API would use JavaScript objects to construct the session description format rather than allowing direct control of how media should be plumbed together from JavaScript.

While using SDP as the chosen format for WebRTC highlights the issues described in this draft particularly well, using an alternative format like JSON instead of SDP does not remove many of the issues presented in this draft. The issues expressed are not solely caused by the lack of expressiveness of the SDP format but the nature of creating a universal all-encompassing format to describe all transport, media, constraints, and negotiations with an attached inflexible state machine is the nature of the issue. This format must do everything and encompass all concepts and becomes the effective mandate for signaling even if not explicitly required to perform signaling.

A few years ago there was an attempt to create a new "SDP 2.0" format with a draft named Session Description and Capability Negotiation [I-D.ietf-mmusic-sdpng]. This effort to create the "ultimate" SDP format in XML was ultimately abandoned, in no small part because of the difficulties in coming up with a single solution that works for all scenarios.

Given the difficulty in creating a universal all-encompassing format that works for all scenarios, the idea that creating a JavaScript based API that constructs a similar flexible, but well defined universal session description format using JavaScript objects is highly suspect to fail equally. The reality is that such an effort is complex.

Even if successful, this format is not necessarily the format that will be sent on-the-wire, especially for existing alternative signaling protocols. As such, the format will still need to be transformed into alternative formats by JavaScript (or by a gateway). If the format must be parsed or interpreted by an intermediate then the format becomes an interaction point to the browser no matter how clever the JavaScript session description construction API

implementation. Whatever format is selected, each browser or alternative protocol format will have to decide how to convert and interpret the output and generate new compatible inputs and deal with the variations that will undoubtedly arrive from browser to browser and from version to version.

Even if JavaScript APIs are made available to simplify the construction or interpretation of a defined format, this format would still become a do-everything serialization access point for the browser and the defined exchange point for the local and remote browser. Therefore the format itself must be described in meticulous detail.

The standardization requirements for such an approach would increase substantially over the WebRTC JavaScript object model advocated by this draft since not only would such a JavaScript format construction API have to be standardized (as any JavaScript Model would) but the formatting rules and state machine it relies upon needs to become standardized in detail as well.

Every combination of this all-encompassing format would need to be outlined, rather than minimal definition of fixed properties needed on a scoped objects as used in the WebRTC JavaScript Object Model. Any slight variations would likely cause JavaScript developers or other browsers to break their implementations. Obtaining 100% stability in such an output equally across all browsers, on all platforms with all versions is highly doubtful.

While a JavaScript format construction API is merely hypothetical at the time of writing this draft, any proposal will need to be vetted to see if it addresses all the concerns and issues brought up in this draft.

This hypothetical JavaScript session description construction API still puts the emphasis in driving the developer towards building up a media signaling exchange format rather than in the logic of how the media should be controlled and pipelined.

The WebRTC JavaScript object model is being proposed as the alternative. In a follow-up to this draft the model will describe how the JavaScript developer gains control over the stream's pipelining for the browser's media/RTC engine and thus free the JavaScript developer to express signaling and state machines using whatever mechanism desired. A simplified shim implemented entirely in JavaScript will allow easier interpretation to any format desired by the JavaScript developer in a way that can be updated independently of a browser's binary release. Should any changes be needed in signaling, a JavaScript shim generating this custom format

is strictly under the control of the service provider and not the browser.

4. Example Difficult Usage Cases with Current Model

4.1. On / off hold example usage case

This is a typical scenario widely adopted SIP technique of an SDP attribute to place a stream on / off hold. This is the accepted methodology and performing alternative approaches would deviate from the expected practices for use with SIP and its manipulation of SDP. Although not officially documented as supported, it is effectively supported in WebRTC implementations. This is a typical use case need by media application:

1. Browser A establishes a connection with Browser B

2. Browser A and browser B are streaming media

3. JavaScript developer wants Browser A to put "on hold"

These are the steps that must be performed by a JavaScript developer:

1. createOffer to obtain the SDP from Browser A

2. Parse the SDP

3. Add "a=sendonly" or "a=inactive" to all media

4. Regenerate the SDP, feed back to browser

5. Send the SDP to Browser B

6. Receive the answer from Browser B (which should respond with a=recvonly if it still wishes media)

7. Parse the received SDP and modify with "a=recvonly" if it did not respond correctly (to ensure the local side hold back its media)

8. Pass the modified SDP answer back into Browser A

This also implies that:

1. All future SDP events received from Browser B must be mangled to ensure the "sendonly/recvonly/inactive" attribute is maintained while on hold

2. All future createOffer/createAnswer calls from Browser A must be modified to ensure the "sendonly" property is maintained

3. We need to handle alternative formats to describe hold, e.g. "c=0.0.0.0" from Browser B which may not utilize the latest SDP specifications depending on the remote device / platform

Ironically, hold is a very SIP and telephony specific concept. The better approach would be to allow the streams to be pause/unpaused at will as that does not require interaction with the SDP, and allow the higher layers to signal the desire to pause the session to the remote peer in whatever manner desired.

This is a very basic use case that is extremely complex for a JavaScript developer, but it is the only way to perform this particular action which is effectively supported by the browsers, except only via the "SDP surface API". Even if this particular use case ends up being an exposed JavaScript method to manipulate the SDP by the browser, there are countless other scenarios where tweaking a field to modify the behavior in the format will only be only available via SDP manipulation.

4.2. One-Sided Constraints Negotiation use Case Scenario

As WebRTC is a web API and not a SIP API, the API must be capable of allowing for alternative signaling methods without enforcing it's own signaling aspects (other than basic principles like ensure ICE agreement has been achieved for security reasons).

Consider the following scenario:

1. Browser A and Browser B establish a connection

2. Browser A and Browser B use one-sided constraints negotiation where each party independently decides what "it expects to receive"

3. Browser A decides that it wishes to alter the properties of the video it expects to receive

With this model, browser A must be capable of independently modifying its expectations without waiting for an answer from the remote side (as that's illegal by the nature of the offer / answer signaling), unless the rules are relaxed and special exceptions are made. For the model to work, browser A's receive constraints must be applied to the send constraints of the remote peer. This model does not require an SDP offer / answer exchange since the sending peer can monitor the expectations of the receiving peer and set its send constraints as appropriate.

To achieve this a for one-sided negotiation:

1. Browser A's JavaScript must respond to every SDP offer with an answer locally generated from JavaScript without a round trip, extracting out last known expectations from the remote SDP last received as part of the answer

2. The JavaScript must update the constraint signaling for the remote party

3. Browser B's JavaScript sees the constraints have changed from Browser A thus it initiates a fake offer from the remote party (generating the intentions of the constraint and generating an SDP format)

4. Browser B's JavaScript must examine the answer if any constraints have changed, and if so, it may trigger another reverse situation where step 1 is repeated, except with Browser A and B's role reversed.

Is this really doable? Maybe, with a great deal of difficulty and SDP mangling but it is unquestionably a hack and a violation of offer / answer (and relaxed rules create exceptions and exceptions require additional logic to handle). The offer / answer rules are violated because no round trip was performed at the time when the constraints were changed.

This is also fragile because if Browser B failed to accept the fake offer there is no way to enforce the constraint nor can the JavaScript rollback the expected constraint. Likewise if the state machine in Browser A expected an offer to be generated before a new offer would be accepted, the conflict resolution process would be extremely difficult and messy.

This offer / answer state machine is not even required to fulfill the mandate of the RTCWEB Working Group charter but it is currently mandated because it supposedly makes producing "SIP interoperability" easier (which is highly suspect at best).

A JavaScript shim approach on a WebRTC JavaScript object model and without offer / answer could achieve the same (or better) "SIP interoperability" without breaking other stateless negotiation models, such as one-sided negotiation.

4.3. Meet-me Negotiation Use Case Scenario

1. WebRTC client A generates an offer and sends to a server

2. WebRTC client B generates an offer and sends to a server

3. WebRTC client C generates an offer and sends to a server

4. The server returns all the exchanges to each of these clients simultaneously

5. WebRTC client A, B and C interconnect

Technically, there is no need for independent SDP offer / answer negotiation amongst all these peers to achieve a mesh scenario for this use case. Each client has enough information about the other clients to establish a peer connection. The current WebRTC SDP API imposes independent round trip negotiations that are not technically necessary. If WebRTC client D was added later, the original connection can be forked and re-use the same DTLS fingerprints to negotiate new encryptions keys for media or data. Fingerprint or identity signature reuse should not introduce any additional security concerns since identities will be verified and keys negotiated for each peer-to-peer connection.

A JavaScript object model approach would allow for this kind of scenario without independent round trip negotiations for each WebRTC client in the mesh.

4.4. Browser to Browser Compatibility Extension Compatibility Issue Scenario

Consider the following scenario:

1. Browser A has implemented an extension to SDP (which is allowed)

2. Browser B has no knowledge of such an extension

3. The JavaScript engine running on Browser A has no knowledge of the extension

4. The JavaScript engine packages up the SDP from Browser A and sends it to Browser B

Under this scenario, what should browser B do? To reject the offer means communication cannot occur. To accept the offer has ambiguous meaning because the answer might have misunderstood the extension's intention and does not allow for the appropriate behavior.

The exact rules of what is allowed in SDP and what is not and how extensions are treated must be defined clearly and non ambiguously. Even though current SDP offer / answer API can deal with some extensions, like new codecs being introduced, it is ambiguous on how to deal with more major extensions such as new SDP profiles, transports, or encryption methods.

Assuming that a lack of response to an extension is non-agreement to use the extension is not acceptable. For example, if the extension was security related dictating some security precondition to opening a stream, the offer must be rejected as the precondition cannot be met. Ignoring the extension would mean the offer was accepted where it cannot be accepted. Another example would be introduction of new SDP profile, like AVPF2. Offer/answer negotiation simply fails when it encounters an unknown profile even if it is backwards compatible, like for instance, most of the calls to current SIP devices will fail if AVPF is used instead of AVP. A better approach is to define the rules for how extensions can be made, whereas SDP has no such rules.

Currently, in SIP networks, such extensions are agreed upon in advanced and extensively tested before they are introduced. SBCs (Session Border Controllers) are often used to make devices with different feature sets work with each other. By allowing JavaScript control over the format generated on the wire, feature roll out is under strict control of the provider, and not whenever a browser vendor decides to produce an update.

4.5. Building Interoperability between WebRTC and a SIP Service Scenario

Consider the following scenario:

1. Developer takes SDP produced by browser and send to SIP gateway (which is supposed to be SIP "compatible")

2. Users happily use this service

3. Browser Vendor A updates the browser SDP generator and a slight variation in SDP changes

4. Users are now broken

5. SIP gateway must be updated to handle new SDP (and old SDP)

6. Browser Vendor B updates their browser SDP generator (with a different SDP variation)

7. Users are now broken again

8. SIP gateway must be updated to handle another variation of SDP (and maintain the old variations)

9. Repeat to step 3, but add Browser Vendor C, D and multiple platforms

This is not an unrealistic scenario by any stretch of the imagination. This currently happens in the SIP world, but at least in that world new devices are tested to ensure compatibility before roll outs occur on the network so issues can be addressed before the user's experience is broken. Since the SIP provider and gateway vendor do not have control over the update cycle of the browsers, their users are much more prone to breakage by taking the SDP from the browser and sending to their network.

Whereas this is what happens with a JavaScript Object API model with SDP shim written in JavaScript-only:

1. Developer uses shim to generate SDP by browser and sends to SIP gateway (with SDP that is compatible)

2. Users happily use this service

3. Browser Vendor A updates the browser with a new RTC feature.

4. Repeat to step 2

The reason why the browser update does not affect the gateway is because the SDP is generated entirely in JavaScript and thus updates to the browser do not change the SDP generation logic. The SDP is entirely in control of SIP network provider. Any bugs with SDP compatibility can be addressed by the SIP provider without changes in the browser's binary. Bugs, updates and improvements are completely within the boundary and control of the SIP network provider.

4.6. Bit-rate Change Scenario

Consider the follow scenario:

1. User is connected to a conference server

2. While user is listening, the user transmits a low bit-rate

3. The users starts to communicate and the bit-rate is adjusted to maximum quality

Using the current WebRTC API, this would require an offer / answer round trip to perform the change and thus the quality would be updated until the answer was acknowledged, although proposals have been made to alter the rules for offer / answer in this case and allow for an exception. This round trip is unnecessary technically since the bit-rate can be dynamically adjusted without remote acknowledgment. Yet, the current offer / answer model imposes a round trip (unless yet another exception to the SDP rules are adopted).

4.7. Video Codec Option Change Scenario

Consider the follow scenario:

1. JavaScript wishes to change a video codec option

Using the current WebRTC API, this would require parsing the entire SDP, isolating the video codecs for a particular video media line, figuring the mapping and then reconstructing the original SDP with the newly incorporated changes. Accessors have been suggested for these common use cases but do not exist yet. If such accessors are created then a more involved API cannot be avoided out of necessity. One of the main justifications given by SDP proponents for only having an API that creates and accepts SDP is due to its supposed simplicity, as opposed to providing a more involved API.

4.8. Video Upgrade Scenario

1. Alice and Bob are having an audio conversation

2. Alice presses the video button on her application and offers Bob video

3. Bob does not wish to see Alice's video, so the application rejects the media (e.g. using "a=inactive" or "m=video 0")

4. Alice's web application successfully parses and interprets Bob's rejection

5. As Alice's video window of herself is independent of the SDP negotiation, Alice's HTML5 application successfully renders Alice's video locally

The current WebRTC implementation offers no event to indicate the rejection, thus Alice is given no feedback of the rejection. She

incorrectly assumes she's in a video conversation. In order to solve this scenario, custom signaling must be added to indicate of Bob's rejection of Alice's video. Yet this is duplication of signaling as the video is already rejected in the SDP. This leaves the JavaScript developer with a choice: either parse the SDP, understand the SDP and derive meaning, or duplicate the SDP efforts by introducing custom signaling for a common scenario when upgrading from audio to video and providing appropriate user feedback.

5. Proposal: WebRTC JavaScript Object Model

5.1. Overview

The browser can expose simple object methods, properties and events representing the various RTC components at an abstracted level and provide a solid API for controlling how the media should be pipelined. The properties needed to be exchanged is separated into the appropriate object rather than meshed into an all-encompassing format.

A JavaScript-only shim can be layered on top of an object model to provide easy SDP offer / answer capability for those who want a similar "simple" API to the current WebRTC API for use with SIP. A developer can chose to use this shim or not if they do not need SDP. Likewise, the object model could be used to produce alternative formats to SDP if the same do-everything format is needed but in an alternative on-the-wire session description format.

The object model described in the solution is presented in a related draft. This solution will allow for the RTCWEB Working Group to complete its chartered mandate without starting from scratch. If adopted, all of the drafts proposed to solve issues in expressing SDP for WebRTC can be moved to more appropriate working groups. For example, SDP for SIP issues can be moved to the appropriate SIP working groups and multi-party SDP to the MMUSIC (e.g. drafts like PlanA or PlanB).

5.2. Benefits

5.2.1. Greater compatibility

By having a WebRTC JavaScript object model, the exact inputs, outputs, properties and events can be well defined on individual objects and each object will be designed to be a specific contract between browser vendors and JavaScript developers.

5.2.2. Easier to extend

New objects and methods can be added without breaking existing compatibility. Compliance can be verified with unit tests able to test each and every behavior across all browsers' versions on every platform. JavaScript developers can expect their version of the API object contract to remain fixed to expected behaviors and not break (unless through well planned deprecation).

Any extensions added to a JavaScript object model does not change the behavior expectation from JavaScript developers when using the current version of the API regardless of any extensions, unless explicitly deprecated. This is unlike SDP where extensions could be silently added into the SDP produced by the browsers at will, even in minor browser version changes, where any component that consumes the SDP may be unaware what those additional feature behaviors imply or require as a result.

5.2.3. Faster Reaction Time To Issues

Signaling related bugs produced by the JavaScript shims can easily be fixed and updated at any time regardless of the browser's release cycle. If a SIP provider discovers their SIP is not compatible within their JavaScript shim, the SIP provider can update the shim code to their own needs dynamically without lobbying the browser vendor and waiting for the browser to be patched and updated.

5.2.4. Decreased surface API

With a JavaScript object model, the features are well defined so the surface API is fixed to the agreed contract. Once agreed, a browser vendor only has to ensure their compatibility with well defined limited scope unit tests, and need not worry about some free-form format that may introduce untold compatibility issues should another vendor issue an update. This is also true of any non-browsers that may wish to implement and be compliant to the WebRTC API for JavaScript and provide their own JavaScript and WebRTC engines.

5.2.5. Greater compatibility for SIP

While SIP is not the main RTCWEB Working Group charter responsibility for WebRTC, SIP compatibility is highly desirable. By exclusively generating SDP from a JavaScript shim, the SDP produced will be identical across all platforms and all devices with every browser version and entirely under the control of the SIP provider. This increases compatibility for SIP providers. The SDP produced from the shim can be custom tailored to a SIP network without affecting any other SIP vendor or harming compatibility with other utilizing WebRTC.

5.2.6. Alternative formats

With a JavaScript shim approach on top of an object model, the information going over the wire can be transformed from the JavaScript object properties to alternative formats, including JSON, XML or SIP (or anything custom). As the JavaScript shim to use is under control of the service provider and identical regardless of the platform, the output from the JavaScript format generation is consistent and controllable, thus ensuring maximum compatibility within a network.

The party receiving this format can be sure the format is to an exacting specification of their choosing rather than relying on whatever format is produced by whatever browser vendor.

5.3. Design Goals and Considerations

5.3.1. Objects Model Kept Simple

The JavaScript developer should not need to understand the mechanics of RTC other than understanding how to plumb the objects together. Those who need extended properties or events for finer control can obtain them with simple method access to an object, but those extended attributes should not be required for simple use cases.

5.3.2. Simple to Gather Negotiation Information

The objects model should allow a simple method for collecting information that will be needed for various alternative negotiation models, highly focused to the object. One of the targets for negotiation must be SDP and SIP.

5.3.3. Offer / Answer

The proposed JavaScript object model should not require the offer / answer state machine but must not preclude this state machine being built in a layer above. The offer / answer state machine must be possible to implement as a JavaScript shim without any additional built-in browser services needing to be implemented.

5.3.4. Extensions

Extending the object model for the expected common extension use cases without breaking the JavaScript API should be possible. Such possible extension use cases should include items like local mixing and data synchronization, or extended properties, events or features.

As any design, there may be limitations but the design should hold up to various realistic scenarios that are likely to happen in the near future.

5.3.5. Well Defined Behaviors

An API must describe specific API behavior sets to the browser vendors so they have the appropriate guidelines for implementation, including the mapping to on-the-wire to RTC protocols. The API presented in the related draft may be the input to a W3C efforts to define specific and exact expected behavior sets for an object based JavaScript API for an official WebRTC 1.0 release.

5.3.6. Data Channel

The proposed WebRTC JavaScript Object model will provide a definition for basic JavaScript usage of the data channel.

5.3.7. Satisfy the expectations of the RTCWEB charter

The object model must adhere to the expectations of the RTCWEB charter either directly, via extensions that can be defined by the working group on top of the object model or possibly via a JavaScript shim written to utilize the functionality of the object model but it must not preclude the RTCWEB charter from fulfilling its previously stated goals.

5.3.8. SIP/SDP and current WebRTC API shim compatibility statement

The goal of the object model is to allow for a JavaScript shim that provides a simple mechanism for parsing and generating SDP for basic compatibility with SIP networks (capable of supporting the WebRTC wire protocols).

The goal of this object based model is not to provide working JavaScript shim on top that is a 1-for-1 matching of the current WebRTC API as a shim, including all behaviors, features, bugs and expectations since the definition of the current approach is not defined enough to be able to produce that level of compatibility. This would be an impossible goal as a result, and would add little value.

Extensions are beyond the scope of the JavaScript shim, but it is possible for others to fork and modify the shim to their own needs specific to their own SIP/SDP network infrastructure.

Compatibility with the SDP used in all SIP networks is not a stated goal for any JavaScript shim since not even SIP providers can agree on a common agreed definitive standard set of RFCs and drafts.

5.3.9. Greater Separation of RTCWEB Working Group and Other Working Groups

A JavasScript object model would remove much of the need for cross IETF working group coordination, which has become common place with the current movement because of utilizing SDP and its close ties to SIP. By limiting the RTCWEB technologies used to only those required for Real-Time Communication from the browser (e.g. RTP, ICE/STUN/TURN, DTLS), the RTCWEB Working Group is freed from tight couplings with other IETF working groups, each having their own charters, schedules, agendas and interests and thus ensures more rapid progress between RTCWEB Working Group the W3C and developers who are to use this technology.

6. Security Considerations

While RTCWEB has it's own security considerations for protocols, a JavaScript object model has no additional requirements other than those already established for use within RTCWEB, e.g. ICE connectivity permission check or DTLS fingerprint checks.

JavaScript as a browser language itself has security consideration but nothing inherent to using a JavaScript object model versus a JavaScript SDP API model, as any proposed implementations must have a JavaScript API. The specifics of any API must list their own specific security considerations to their defined model and API, should any exist.

Any specific issues for the proposed JavaScript object model will be outlined in the separated draft WebRTC JavaScript object model draft as needed and warranted.

7. References

7.1. Normative References

[RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, June 2002.

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, July 2006.

7.2. Informative References

[I-D.ietf-mmusic-sdpng]
Kutscher, D., Ott, J., and C. Bormann, "Session Description and Capability Negotiation", draft-ietf-mmusic-sdpng-08 (work in progress), February 2005.

[I-D.ivov-rtcweb-noplan]
Ivov, E., Marocco, E., and P. Thatcher, "No Plan: Economical Use of the Offer/Answer Model in WebRTC Sessions with Multiple Media Sources", draft-ivov-rtcweb-noplan-01 (work in progress), June 2013.

[I-D.roach-rtcweb-plan-a]
Roach, A. and M. Thomson, "Using SDP with Large Numbers of Media Flows", draft-roach-rtcweb-plan-a-00 (work in progress), May 2013.

[I-D.uberti-rtcweb-plan]
Uberti, J., "Plan B: a proposal for signaling multiple media sources in WebRTC.", draft-uberti-rtcweb-plan-00 (work in progress), May 2013.

[MediaCapture]
Burnett, D., "Media Capture and Streams", May 2013, <http://www.w3.org/TR/2013/WD-mediacapture-streams-20130516/>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, June 2002.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.

[RFC5245] Rosenberg, J., "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols", RFC 5245, April 2010.

[RFC5389] Rosenberg, J., Mahy, R., Matthews, P., and D. Wing, "Session Traversal Utilities for NAT (STUN)", RFC 5389, October 2008.

[RFC5766] Mahy, R., Matthews, P., and J. Rosenberg, "Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)", RFC 5766, April 2010.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012.

[WebRTC10] Bergkvist, A., "WebRTC 1.0 Real-time Communication Between Browsers", August 2012, <http://www.w3.org/TR/2012/WD-webrtc-20120821/>.

Authors' Addresses

Robin Raymond
Hookflash
436, 3553 31 St. NW
Calgary, Alberta T2L 2K7

Email: robin@hookflash.com

Erik Lagerway
Hookflash
436, 3553 31 St. NW
Calgary, Alberta T2L 2K7
Canada

Email: erik@hookflash.com

Inaki Baz Castillo
Versatica
Barakaldo
Basque Country
Spain

Email: ibc@aliax.net

Roman Shpount
TurboBridge
4905 Del Ray Ave Suite 300
Bethesda, MD 20814
USA

Email: rshpount@turbobridge.com