Abstract

This document outlines the codec and media processing requirements for WebRTC client application and endpoint devices.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, and it may not be published except as an Internet-Draft.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 13, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ................................................. 3
2. Terminology .................................................. 3
3. Codec Requirements .......................................... 3
   3.1. Audio Codec Requirements ................................. 3
   3.2. Video Codec Requirements ................................. 3
4. Audio Level ................................................... 4
5. Acoustic Echo Cancellation (AEC) ............................... 5
6. Legacy VoIP Interoperability .................................. 6
7. IANA Considerations .......................................... 6
8. Security Considerations ....................................... 6
9. Acknowledgements ............................................ 6
10. Normative References ......................................... 6
    Authors’ Addresses ........................................... 7
1. Introduction

An integral part of the success and adoption of the Web Real Time Communications (WebRTC) will be the voice and video interoperability between WebRTC applications. This specification will outline the media processing and codec requirements for WebRTC client implementations.

2. 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].

3. Codec Requirements

This section covers the audio and video codec requirements for WebRTC client applications. To ensure a baseline level of interoperability between WebRTC clients, a minimum set of required codecs are specified below. While this section specifies the codecs that will be mandated for all WebRTC client implementations, it leaves the question of supporting additional codecs to the will of the implementer.

3.1. Audio Codec Requirements

WebRTC clients are REQUIRED to implement the following audio codecs.

- PCMA/PCMU - 1 channel with a rate of 8000 Hz and a ptime of 20 - see section 4.5.14 of [RFC3551]
- Telephone Event - [RFC4734]
- Opus [draft-ietf-codec-opus]

For all cases where the client is able to process audio at a sampling rate higher than 8 kHz, it is RECOMMENDED that Opus be offered before PCMA/PCMU. For Opus, all modes MUST be supported, for all ptime values up to 120 ms. Clients MAY use the offer/answer mechanism to signal a preference for a particular mode or ptime.

3.2. Video Codec Requirements

The following feature list applies to all required video codecs.

Required video codecs:
- MUST support at least 10 frames per second (fps) and SHOULD support 30 fps
- If VP8 is supported, then it MUST support the bilinear and none reconstruction filters
- OPTIONALLY offer support for additional color spaces
- MUST support a minimum resolution of 320x240
- SHOULD support resolutions of 1280x720, 720x480, 1024x768, 800x600, 640x480, 640 x 360, 320x240

4. Audio Level

It is desirable to standardize the "on the wire" audio level for speech transmission to avoid users having to manually adjust the playback and to facilitate mixing in conferencing applications. It is also desirable to be consistent with ITU-T recommendations G.169 and G.115, which recommend an active audio level of -19 dBm0. However, unlike G.169 and G.115, the audio for WebRTC is not constrained to have a passband specified by G.712 and can in fact be sampled at any sampling rate from 8 kHz to 48 kHz and up. For this reason, the level SHOULD be normalized by only considering frequencies above 300 Hz, regardless of the sampling rate used. The level SHOULD also be adapted to avoid clipping, either by lowering the gain to a level below -19 dBm0, or through the use of a compressor.

AUTHORS’ NOTE: The idea of using the same level as what the ITU-T recommends is that it should improve inter-operability while at the same time maintaining sufficient dynamic range and reducing the risk of clipping. The main drawbacks are that the resulting level is about 12 dB lower than typical "commercial music" levels and it leaves room for ill-behaved clients to be much louder than a normal client. While using music-type levels is not really an option (it would require using the same compressor-limitors that studios use), it would be possible to have a level slightly higher (e.g. 3 dB) than what is recommended above without causing interoperability problems.

Assuming 16-bit PCM with a value of +/-32767, -19 dBm0 corresponds to a root mean square (RMS) level of 2600. Only active speech should be considered in the RMS calculation. If the client has control over the entire audio capture path, as is typically the case for a regular phone, then it is RECOMMENDED that the gain be adjusted in such a way that active speech have a level of 2600 (-19 dBm0) for an average speaker. If the client does not have control over the entire audio
capture, as is typically the case for a software client, then the client SHOULD use automatic gain control (AGC) to dynamically adjust the level to 2600 (-19 dBm0) +/- 6 dB. For music or desktop sharing applications, the level SHOULD NOT be automatically adjusted and the client SHOULD allow the user to set the gain manually.

The RECOMMENDED filter for normalizing the signal energy is a second-order Butterworth filter with a 300 Hz cutoff frequency.

It is common for the audio output on some devices to be "calibrated" for playing back pre-recorded "commercial" music, which is typically around 12 dB louder than the level recommended in this section. Because of this, clients MAY increase the gain before playback.

5. Acoustic Echo Cancellation (AEC)

It is plausible that the dominant near to mid-term WebRTC usage model will be people using the interactive audio and video capabilities to communicate with each other via web browsers running on a notebook computer that has built-in microphone and speakers. The notebook-as-communication-device paradigm presents challenging echo cancellation problems, the specific remedy of which will not be mandated here. However, while no specific algorithm or standard will be required by WebRTC compatible clients, echo cancellation will improve the user experience and should be implemented by the endpoint device.

SHOULD include an AEC and if not, SHOULD ensure that the speaker-to-microphone gain is below unity at all frequencies to avoid instability when none of the client has echo cancellation. For clients that do not control the audio capture and playback devices directly, it is RECOMMENDED to support echo cancellation between devices running at slight different sampling rates, such as when a webcam is used for microphone.

The client SHOULD allow either the entire AEC or the non-linear processing (NLP) to be turned off for applications, such as music, that do not behave well with the spectral attenuation methods typically used in NLPs. It SHOULD have the ability to detect the presence of a headset and disable echo cancellation.

For some applications where the remote client may not have an echo canceller, the local client MAY include a far-end echo canceller, but if that is the case, it SHOULD be disabled by default.

Call control event notification to connected devices such as headsets (what’s that exactly?)
6. Legacy VoIP Interoperability

The codec requirements above will ensure, at a minimum, voice interoperability capabilities between WebRTC client applications and legacy phone systems.

7. IANA Considerations

This document makes no request of IANA.

Note to RFC Editor: this section may be removed on publication as an RFC.

8. Security Considerations

The codec requirements have no additional security considerations other than those captured in [I-D.ekr-security-considerations-for-rtc-web].

9. Acknowledgements

This draft incorporates ideas and text from various other drafts. In particularly we would like to acknowledge, and say thanks for, work we incorporated from Harald Alvestrand.

10. Normative References

[I-D.ekr-security-considerations-for-rtc-web]

[I-D.webm]


[RFC4734] Schulzrinne, H. and T. Taylor, "Definition of Events for Modem, Fax, and Text Telephony Signals", RFC 4734,
Authors’ Addresses

Cary Bran
Plantronics
345 Encinial Street
Santa Cruz, CA  95060
USA

Phone: +1 206 661-2398
Email: cary.bran@plantronics.com

Cullen Jennings
Cisco
170 West Tasman Drive
San Jose, CA  95134
USA

Phone: +1 408 421-9990
Email: fluffy@cisco.com

Jean-Marc Valin
Mozilla
650 Castro Street
Mountain View, CA  94041
USA

Email: jmvalin@jmvalin.ca
RTCWeb Datagram Connection
draft-ietf-rtcweb-data-channel-00.txt

Abstract

The Web Real-Time Communication (WebRTC) working group is charged to provide protocol support for direct interactive rich communication using audio, video, and data between two peers’ web-browsers. This document describes the non-media data transport aspects of the WebRTC framework. It provides an architectural overview of how the Stream Control Transmission Protocol (SCTP) is used in the WebRTC context as a generic transport service allowing Web Browser to exchange generic data from peer to peer.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 7, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ..................................................... 3
2. Requirements ..................................................... 3
3. Use Cases ........................................................ 5
   3.1. Use Cases for Unreliable Datagram Based Channels ......... 5
   3.2. Use Cases for Reliable Channels (Datagram or Stream Based) ........................................... 5
4. Datagrams over SCTP over DTLS over UDP ......................... 6
5. The Envisioned Usage of SCTP in the RTCWeb Context .......... 8
   5.1. Association Setup ........................................... 8
   5.2. SCTP Streams ............................................... 9
   5.3. Channel Definition ........................................ 9
   5.4. Usage of Payload Protocol Identifier ...................... 9
6. Minor Protocol and Message Format ................................ 10
7. Security Considerations .......................................... 10
8. IANA Considerations ............................................. 10
9. Acknowledgments ................................................ 10
10. Informative References .......................................... 10
Authors’ Addresses .................................................. 11
1. Introduction

The issue of how best to handle non-media data types in the context of RTCWEB has reached a general consensus on the usage of SCTP [RFC4960] encapsulated on DTLS [RFC6347]:

```
+----------+
|   SCTP   |
+----------+
|   DTLS   |
|   ICE/UDP|
+----------+
```

Figure 1: Basic stack diagram

The encapsulation of SCTP over DTLS over ICE/UDP provides a NAT traversal solution together with confidentiality, source authenticated, integrity protected transfers. This data transport service operates in parallel to the media transports, and all of them can eventually share a single transport-layer port number.

SCTP provides multiple streams natively with reliable, unreliable and partially-reliable delivery modes.

The remainder of this document is organized as follows: Section 2 and Section 3 provide requirements and use cases for both unreliable and reliable peer to peer datagram base channel; Section 4 argues SCTP over DTLS over UDP; Section 5 provides an overview of how SCTP should be used by the RTCWeb protocol framework for transporting non-media data between browsers.

2. Requirements

This section lists the requirements for P2P data connections between two browsers.

Req. 1  Multiple simultaneous datagram streams must be supported. Note that there may 0 or more media streams in parallel with the data streams, and the number and state (active/inactive) of the media streams may change at any time.

Req. 2  Both reliable and unreliable datagram streams must be supported.
Req. 3   Data streams must be congestion controlled; either individually, as a class, or in conjunction with the media streams, to ensure that datagram exchanges don’t cause congestion problems for the media streams, and that the rtcweb PeerConnection as a whole is fair with competing streams such as TCP.

Req. 4   The application should be able to provide guidance as to the relative priority of each datagram stream relative to each other, and relative to the media streams. [ TBD: how this is encoded and what the impact of this is. ] This will interact with the congestion control algorithms.

Req. 5   Datagram streams must be encrypted; allowing for confidentiality, integrity and source authentication. See [I-D.ietf-rtcweb-security] and [I-D.ietf-rtcweb-security-arch] for detailed info.

Req. 6   Consent and NAT traversal mechanism: These are handled by the PeerConnection’s ICE [RFC5245] connectivity checks and optional TURN servers.

Req. 7   Data streams must provide message fragmentation support such that IP-layer fragmentation does not occur no matter how large a message the Javascript application passes to be sent.

Req. 8   The data stream transport protocol must not encode local IP addresses inside its protocol fields; doing so reveals potentially private information, and leads to failure if the address is depended upon.

Req. 9   The data stream protocol should support unbounded-length "messages" (i.e., a virtual socket stream) at the application layer, for such things as image-file-transfer; or else it must support at least a maximum application-layer message size of 4GB.

Req. 10  The data stream packet format/encoding must be such that it is impossible for a malicious Javascript to generate an application message crafted such that it could be interpreted as a native protocol over UDP - such as UPnP, RTP, SNMP, STUN, etc.

Req. 11  The data stream transport protocol must start with the assumption of a PMTU of 1280 [ *** need justification ***] bytes until measured otherwise.
Req. 12 The data stream transport protocol must not rely on ICMP or ICMPv6 being generated or being passed back, such as for PMTU discovery.

Req. 13 It must be possible to implement the protocol stack in the user application space.

3. Use Cases

3.1. Use Cases for Unreliable Datagram Based Channels

U-C 1 A real-time game where position and object state information is sent via one or more unreliable data channels. Note that at any time there may be no media channels, or all media channels may be inactive, and that there may also be reliable data channels in use.

U-C 2 Non-critical state updates about a user in a video chat or conference, such as Mute state.

3.2. Use Cases for Reliable Channels (Datagram or Stream Based)

Note that either reliable datagrams or streams are possible; reliable streams would be fairly simple to layer on top of SCTP reliable datagrams with in-order delivery.

U-C 3 A real-time game where critical state information needs to be transferred, such as control information. Typically this would be datagrams. Such a game may have no media channels, or they may be inactive at any given time, or may only be added due to in-game actions.

U-C 4 Non-realtime file transfers between people chatting. This could be datagrams or streaming. Note that this may involve a large number of files to transfer sequentially or in parallel, such as when sharing a folder of images or a directory of files.

U-C 5 Realtime text chat while talking with an individual or with multiple people in a conference. Typically this would be datagrams.

U-C 6 Renegotiation of the set of media streams in the PeerConnection. Typically this would be datagrams.
U-C 7 Proxy browsing, where a browser uses data channels of a PeerConnection to send and receive HTTP/HTTPS requests and data, for example to avoid local internet filtering or monitoring. Typically this would be streams.

4. Datagrams over SCTP over DTLS over UDP

The encapsulation of SCTP over DTLS as defined in [I-D.tuexen-tsvwg-sctp-dtls-encaps] provides a NAT traversal solution together with confidentiality, source authenticated, integrity protected transfers. SCTP provides also natively several interesting features for transporting non-media data between browsers:

- Support of multiple streams.
- Ordered and unordered delivery of user messages.
- Reliable and partial-reliable transport of user messages.

Each SCTP user message contains a so called Payload Protocol Identifier (PPID) that is passed to SCTP by its upper layer and sent to its peer. This value represents an application (or upper layer) specified protocol identifier and be used to transport multiple protocols over a single SCTP association. The sender provides for each protocol a specific PPID and the receiver demultiplexes the messages based on the received PPID.

The encapsulation of SCTP over DTLS, together with the SCTP features listed above satisfies all the requirements listed in in Section 2.

The layering of protocols for WebRTC is shown in the following Figure 2.

```
+------+
| WEBAPP|
+------+
    +-------------------+
    | SCTP               |
    +-------------------+
        +-------------------+
        | STUN | SRTP | DTLS |
        +-------------------+
            +-------------------+
            | ICE               |
            +-------------------+
                +-------------------+
                | UDP1 | UDP2 | ... |
                +-------------------+
```

Figure 2: WebRTC protocol layers
This stack (especially in contrast to DTLS over SCTP [RFC6083]) has been chosen because it

- supports the transmission of arbitrary large user messages.
- shares the DTLS connection with the media channels.
- provides privacy for the SCTP control information.

Considering the protocol stack of Figure 2 the usage of DTLS over UDP is specified in [RFC6347], while the usage of SCTP on top of DTLS is specified in [I-D.tuexen-tsvwg-sctp-dtls-encaps].

Since DTLS is typically implemented in user-land, an SCTP user-land implementation must also be used.

When using DTLS as the lower layer, only single homed SCTP associations can be used, since DTLS does not expose any address management to its upper layer. The ICE/UDP layer can handle IP address changes during a session without needing to notify the DTLS and SCTP layers, though it would be advantageous to retest path MTU on an IP address change.

DTLS implementations used for this stack must support controlling fields of the IP layer like the Don’t fragment (DF)-bit in case of IPv4 and the Differentiated Services Code Point (DSCP) field. This is required for performing path MTU discovery. The DTLS implementation must also support sending user messages exceeding the path MTU.

When supporting multiple SCTP associations over a single DTLS connection, incoming ICMP or ICMPv6 messages can’t be processed by the SCTP layer, since there is no way to identify the corresponding association. Therefore the number of SCTP associations should be limited to one or ICMP and ICMPv6 messages should be ignored. In general, the lower layer interface of an SCTP implementation has to be adapted to address the differences between IPv4 or IPv6 (being connection-less) or DTLS (being connection-oriented).

When protocol stack of Figure 2 is used, DTLS protects the complete SCTP packet, so it provides confidentiality, integrity and source authentication of the complete SCTP packet.

This protocol stack supports the usage of multiple SCTP streams. A user message can be sent ordered or unordered and, if the SCTP implementations support [RFC3758], with partial reliability. When using partial reliability, it might make sense to use a policy limiting the number of retransmissions. Limiting the number of
retransmissions to zero provides a UDP like service where each user messages is sent exactly once.

SCTP provides congestion control on a per-association base. This means that all SCTP streams within a single SCTP association share the same congestion window. Traffic not being sent over SCTP is not covered by the SCTP congestion control. Due to the typical parallel SRTP media streams, it will be advantageous to select a delay-sensitive congestion control algorithm or to at least coordinate congestion control between the data channels and the media streams to avoid a data channel transfer ending up with most or all the channel bandwidth.

5. The Envisioned Usage of SCTP in the RTCWeb Context

The appealing features of SCTP in the RTCWeb context are:

- TCP-friendly congestion control.
- The congestion control is modifiable for integration with media stream congestion control.
- Support for multiple channels with different characteristics.
- Support for out-of-order delivery.
- Support for large datagrams and PMTU-discovery and fragmentation.
- Reliable or partial reliability support.
- Support of multiple streams.

Multihoming will not be used in this scenario. The SCTP layer would simply act as if it were running on a single-homed host, since that is the abstraction that the lower layer (a connection oriented, unreliable datagram service) would expose.

5.1. Association Setup

The SCTP association would be set up when the two endpoints of the WebRTC PeerConnection agree on opening it, as negotiated by JSEP (typically an exchange of SDP) [I-D.ietf-rtcweb-jsep]. It would use the DTLS connection created at the start of the PeerConnection connection.

The application should indicate the number of simultaneous streams required when opening the association, and if no value is supplied,
the implementation should provide a default, with a suggested value of 16. If more simultaneous streams are needed, [RFC6525] allows adding additional (but not removing) streams to an existing association. There can be up to 65536 SCTP streams per SCTP association in each direction.

5.2. SCTP Streams

SCTP defines a stream as an unidirectional logical channel existing within an SCTP association one to another SCTP endpoint. The streams are used to provide the notion of in-sequence delivery. Each user message is sent on a particular stream, either order or unordered. Ordering is preserved only for all ordered messages sent on the same stream.

5.3. Channel Definition

The W3C seems to have consensus on defining the application API for WebRTC dataChannels to be bidirectional. They also consider the notions of in-sequence, out-of-sequence, reliable and un-reliable as properties of Channels.

A possible realization of a bidirectional Data Channel is a pair of one incoming stream and one outgoing SCTP stream.

Closing of a Data Channel can be signalled resetting the corresponding streams [RFC6525]. Resetting a stream set the Stream Sequence Numbers (SSNs) of the stream back to 'zero' with a corresponding notification to the application layer that the reset has been performed. Closed streams are available to reuse.

[RFC6525] also guarantees that all the messages are delivered (or expired) before resetting the stream.

It might be useful to use a specific pair of SCTP streams for transporting control information.

5.4. Usage of Payload Protocol Identifier

The SCTP Payload Protocol Identifiers (PPIDs) can be used to signal the interpretation of the "Payload data", like a string, ASCII or binary data.

RFC 4960 [RFC4960] creates the registry from which these identifiers have been assigned. Eventual PPIDs defined within the RTCWeb Context have to be registered with IANA.
6. Minor Protocol and Message Format

A separate draft (draft-jesup-rtcweb-data-protocol) is being submitted to define the minor protocol to set up and manage the bidirectional data channels needed to satisfy the requirements in this document for WebRTC.

Masking of the protocol is not needed if the lower layer always encrypts with DTLS.

7. Security Considerations

To be done.

8. IANA Considerations

This document does not require any actions by the IANA.

9. Acknowledgments

Many thanks for comments, ideas, and text from Cullen Jennings, Eric Rescorla, Randall Stewart, Justin Uberti, and Harald Alvestrand.

10. Informative References


Authors’ Addresses

Randell Jesup
Mozilla
USA

Email: randell-ietf@jesup.org

Salvatore Loreto
Ericsson
Hirsalantie 11
Jorvas 02420
Finland

Email: salvatore.loreto@ericsson.com
Internet-Draft             data P2P in RTCWEB                 March 2012

Michael Tuexen
Muenster University of Applied Sciences
Stegerwaldstrasse 39
Steinfurt  48565
Germany

Email: tuexen@fh-muenster.de
Javascript Session Establishment Protocol

draft-ietf-rtcweb-jsep-00

Abstract

This document proposes a mechanism for allowing a Javascript application to fully control the signaling plane of a multimedia session, and discusses how this would work with existing signaling protocols.

This document is an input document for discussion. It should be discussed in the RTCWEB WG list, rtcweb@ietf.org.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on July 26, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must
Table of Contents

1. Introduction ............................................ 4
2. JSEP Approach ........................................... 5
3. Other Approaches Considered .............................. 6
4. Semantics and Syntax ..................................... 7
   4.1. Signaling Model ....................................... 7
   4.2. Session Descriptions .................................... 7
   4.3. Session Description Format .............................. 8
   4.4. Separation of Signaling and ICE State Machines .......... 9
   4.5. ICE Candidate Trickling ................................. 9
   4.6. ICE Candidate Format ..................................... 10
5. Media Setup Overview ...................................... 10
   5.1. Initiating the Session ................................... 10
   5.1.1. Generating An Offer ................................. 11
   5.1.2. Applying the Offer ................................... 11
   5.1.3. Initiating ICE ......................................... 11
   5.1.4. Serializing the Offer and Candidates ................. 11
   5.2. Receiving the Session .................................... 12
   5.2.1. Receiving the Offer ................................... 12
   5.2.2. Initiating ICE ......................................... 12
   5.2.3. Handling ICE Messages ................................. 12
   5.2.4. Generating the Answer ................................. 12
   5.2.5. Applying the Answer ................................... 13
   5.2.6. Serializing the Answer ................................. 13
   5.3. Completing the Session ................................. 13
   5.3.1. Receiving the Answer ................................... 13
   5.4. Updates to the Session ................................... 13
6. Proposed WebRTC API changes ............................... 14
   6.1. PeerConnection API ...................................... 14
   6.1.1 MediaHints ........................................... 15
   6.1.2 createOffer ........................................... 16
   6.1.3 createAnswer .......................................... 16
   6.1.4 SDP_OFFER, SDP_PRANSWER, and SDP_ANSWER ............... 17
   6.1.5 setLocalDescription .................................... 17
   6.1.6 setRemoteDescription ................................... 18
   6.1.7 localDescription ....................................... 18
   6.1.8 remoteDescription ...................................... 18
   6.1.9 IceOptions ............................................. 19
   6.1.10 startIce ............................................... 19
   6.1.11 processIceMessage ..................................... 19
7. Example API Flows .......................................... 20
1. Introduction

The general thinking behind WebRTC call setup has been to fully specify and control the media plane, but to leave the signaling plane up to the application as much as possible. The rationale is that different applications may prefer to use different protocols, such as the existing SIP or Jingle call signaling protocols, or something custom to the particular application, perhaps for a novel use case. In this approach, the key information that needs to be exchanged is the multimedia session description, which specifies the necessary transport and media configuration information necessary to establish the media plane.

The original spec for WebRTC attempted to implement this protocol-agnostic signaling by providing a mechanism to exchange session descriptions in the form of SDP blobs. Upon starting a session, the browser would generate a SDP blob, which would be passed to the application for transport over its preferred signaling protocol. On the remote side, this blob would be passed into the browser from the application, and the browser would then generate a blob of its own in response. Upon transmission back to the initiator, this blob would be plugged into their browser, and the handshake would be complete.

Experimentation with this mechanism turned up several shortcomings, which generally stemmed from there being insufficient context at the browser to fully determine the meaning of a SDP blob. For example, determining whether a blob is an offer or an answer, or differentiating a new offer from a retransmit.

The ROAP proposal, specified in http://tools.ietf.org/html/draft-jennings-rtcweb-signaling-01, attempted to resolve these issues by providing additional structure in the messaging—in essence, to create a generic signaling protocol that specifies how the browser signaling state machine should operate. However, even though the protocol is abstracted, the state machine forces a least-common-denominator approach on the signaling interactions. For example, in Jingle, the call initiator can provide additional ICE candidates even after the initial offer has been sent, which allows the offer to be sent immediately for quicker call startup. However, in the browser state machine, there is no notion of sending an updated offer before the initial offer has been responded to, rendering this functionality impossible.

While specific concerns like this could be addressed by modifying the generic protocol, others would likely be discovered later. The main reason this mechanism is inflexible is because it embeds a signaling state machine within the browser. Since the browser generates the session descriptions on its own, and fully controls the possible
states and advancement of the signaling state machine, modification of the session descriptions or use of alternate state machines becomes difficult or impossible.

The browser environment also has its own challenges that cause problems for an embedded signaling state machine. One of these is that the user may reload the web page at any time. If this happens, and the state machine is being run at a server, the server can simply push the current state back down to the page and resume the call where it left off. If instead the state machine is run at the browser end, and is instantiated within, for example, the PeerConnection object, that state machine will be reinitialized when the page is reloaded and the JavaScript re-executed. This actually complicates the design of any interoperability service, as all cases where an offer or answer has already been generated but is now "forgotten" must now be handled by trying to move the client state machine forward to the same state it had been in previously in order to match what has already been delivered to and/or answered by the far side, or handled by ensuring that aborts are cleanly handled from every state and the negotiation rapidly restarted.

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. JSEP Approach

To resolve these issues, this document proposes the Javascript Session Establishment Protocol (JSEP) that pulls the signaling state machine out of the browser and into Javascript. This mechanism effectively removes the browser almost completely from the core signaling flow; the only interface needed is a way for the application to pass in the local and remote session descriptions negotiated by whatever signaling mechanism is used, and a way to interact with the ICE state machine.

JSEP’s handling of session descriptions is simple and straightforward. Whenever an offer/answer exchange is needed, the initiating side creates an offer by calling a createOffer() API on PeerConnection. The application can do massaging of that offer, if it wants to, and then uses it to set up its local config via a setLocalDescription() API. The offer is then sent off to the remote side over its preferred signaling mechanism (e.g. WebSockets); upon receipt of that offer, the remote party installs it using a setRemoteDescription() API.
When the call is accepted, the callee uses a createAnswer() API to generate an appropriate answer, applies it using setLocalDescription(), and sends the answer back to the initiator over the signaling channel. When the offerer gets that answer, it installs it using setRemoteDescription(), and initial setup is complete. This process can be repeated for additional offer/answer exchanges.

Regarding ICE, in this approach we decouple the ICE state machine from the overall signaling state machine; the ICE state machine must remain in the browser, given that only the browser has the necessary knowledge of candidates and other transport info. While transport has typically been lumped in with session descriptions, performing this separation provides additional flexibility. In protocols that decouple session descriptions from transport, such as Jingle, the transport information can be sent separately; in protocols that don’t, such as SIP, the information can be easily aggregated and recombined. Sending transport information separately can allow for faster ICE and DTLS startup, since the necessary roundtrips can occur while waiting for the remote side to accept the session.

The JSEP approach does come with a minor downside. As the application now is responsible for driving the signaling state machine, slightly more application code is necessary to perform call setup; the application must call the right APIs at the right times, and convert the session descriptions and ICE information into the defined messages of its chosen signaling protocol, instead of simply forwarding the messages emitted from the browser.

One way to mitigate this is to provide a Javascript library that hides this complexity from the developer, which would implement the state machine and serialization of the desired signaling protocol. For example, this library could convert easily adapt the JSEP API into the exact ROAP API, thereby implementing the ROAP signaling protocol. Such a library could of course also implement other popular signaling protocols, including SIP or Jingle. In this fashion we can enable greater control for the experienced developer without forcing any additional complexity on the novice developer.

3. Other Approaches Considered

Another approach that was considered for JSEP was to move the mechanism for generating offers and answers out of the browser as well. This approach would add a getCapabilities API which would provide the application with the information it needed in order to generate session descriptions. This increases the amount of work that the application needs to do; it needs to know how to generate session descriptions from capabilities, and especially how to generate the
correct answer from an arbitrary offer and available capabilities. While this could certainly be addressed by using a library like the one mentioned above, some experimentation also indicates that coming up with a sufficiently complete getCapabilities API is a nontrivial undertaking. Nevertheless, if we wanted to go down this road, JSEP makes it significantly easier; if a getCapabilities API is added in the future, the application can generate session descriptions accordingly and pass those to the setLocalDescription/setRemoteDescription APIs added by JSEP. (Even with JSEP, an application could still perform its own browser fingerprinting and generate approximate session descriptions as a result.)

Note also that while JSEP transfers more control to Javascript, it is not intended to be an example of a "low-level" API. The general argument against a low-level API is that there are too many necessary API points, and they can be called in any order, leading to something that is hard to specify and test. In the approach proposed here, control is performed via session descriptions; this requires only a few APIs to handle these descriptions, and they are evaluated in a specific fashion, which reduces the number of possible states and interactions.

4. Semantics and Syntax

4.1. Signaling Model

JSEP does not specify a particular signaling model or state machine, other than the generic need to exchange RFC 3264 offers and answers in order for both sides of the session to know how to conduct the session. JSEP provides mechanisms to create offers and answers, as well as to apply them to a PeerConnection. However, the actual mechanism by which these offers and answers are communicated to the remote side, including addressing, retransmission, forking, and glare handling, is left entirely up to the application.

4.2. Session Descriptions

In order to establish the media plane, PeerConnection needs specific parameters to indicate what to transmit to the remote side, as well as how to handle the media that is received. These parameters are determined by the exchange of session descriptions in offers and answers, and there are certain details to this process that must be handled in the JSEP APIs.

Whether a session description was sent or received affects the meaning of that description. For example, the list of codecs sent to a remote party indicates what the local side is willing to decode,
and what the remote party should send. Not all parameters follow this rule; the SRTP parameters [RFC4568] sent to a remote party indicate what the local side will use to encrypt, and thereby how the remote party should expect to receive.

In addition, various RFCs put different conditions on the format of offers versus answers. For example, a offer may propose multiple SRTP configurations, but an answer may only contain a single SRTP configuration.

Lastly, while the exact media parameters are only known only after a offer and an answer have been exchanged, it is possible for the offerer to receive media after they have sent an offer and before they have received an answer. To properly process incoming media in this case, the offerer’s media handler must be aware of the details of the offerer before the answer arrives.

Therefore, in order to handle session descriptions properly, PeerConnection needs:

1. To know if a session description pertains to the local or remote side.

2. To know if a session description is an offer or an answer.

3. To allow the offer to be specified independently of the answer.

JSEP addresses this by adding both a setLocalDescription and a setRemoteDescription method, and both these methods take as a first parameter either the value SDP_OFFER, SDP_PRANSWER (for a non-final answer) or SDP_ANSWER (for a final answer). This satisfies the requirements listed above for both the offerer, who first calls setLocalDescription(SDP_OFFER, sdp) and then later setRemoteDescription(SDP_ANSWER, sdp), as well as for the answerer, who first calls setRemoteDescription(SDP_OFFER, sdp) and then later setLocalDescription(SDP_ANSWER, sdp).

While it could be possible to implicitly determine the value of the offer/answer argument inside of PeerConnection, requiring it to be specified explicitly seems substantially more robust, allowing invalid combinations (i.e. an answer before an offer) to generate an appropriate error.

4.3. Session Description Format

In the current WebRTC specification, session descriptions are formatted as SDP messages. While this format is not optimal for manipulation from Javascript, it is widely accepted, and frequently
updated with new features. Any alternate encoding of session descriptions would have to keep pace with the changes to SDP, at least until the time that this new encoding eclipsed SDP in popularity. As a result, JSEP continues to use SDP as the internal representation for its session descriptions.

However, to simplify Javascript processing, and provide for future flexibility, the SDP syntax is encapsulated within a SessionDescription object, which can be constructed from SDP, and be serialized out to SDP. If we were able to agree on a JSON format for session descriptions, we could easily enable this object to generate/expect JSON.

Other methods may be added to SessionDescription in the future to simplify handling of SessionDescriptions from Javascript.

4.4. Separation of Signaling and ICE State Machines

Previously, PeerConnection operated two state machines, referred to in the spec as an "ICE Agent", which handles the establishment of peer-to-peer connectivity, and an "SDP Agent", which handles the state of the offer-answer signaling. The states of these state machines were exposed through the iceState and sdpState attributes on PeerConnection, with an additional readyState attribute that reflected the high-level state of the PeerConnection.

JSEP does away with the SDP Agent within the browser; this functionality is now controlled directly by the application, which uses the setLocalDescription and setRemoteDescription APIs to tell PeerConnection what SDP has been negotiated. The ICE Agent remains in the browser, as it still needs to perform gathering of candidates, connectivity checking, and related ICE functionality.

The net effect of this is that sdpState goes away, and processSignalingMessage becomes processIceMessage, which now specifically handles incoming ICE candidates. To allow the application to control exactly when it wants to start ICE negotiation (e.g. either on receipt of the call, or only after accepting the call), a startIce method has been added.

4.5. ICE Candidate Trickling

Candidate trickling is a technique through which a caller may incrementally provide candidates to the callee after the initial offer has been dispatched. This allows the callee to begin acting upon the call and setting up the ICE (and perhaps DTLS) connections immediately, without having to wait for the caller to allocate all possible candidates, resulting in faster call startup in many cases.
JSEP supports optional candidate trickling by providing APIs that provide control and feedback on the ICE candidate gathering process. Applications that support candidate trickling can send the initial offer immediately and send individual candidates when they get a callback with a new candidate; applications that do not support this feature can simply wait for the callback that indicates gathering is complete, and simply create and send their offer, with all the candidates, at this time.

To be clear, applications that do not make use of candidate trickling can ignore processIceMessage entirely, and use IceCallback solely to indicate when candidate gathering is complete.

4.6. ICE Candidate Format

As with session descriptions, we choose to provide an IceCandidate object that provides some abstraction, but can be easily converted to/from SDP a=candidate lines.

The IceCandidate object has a field to indicate which m= line it should be associated with, and a method to convert to a SDP representation, ex:

```
a=candidate:1 1 UDP 1694498815 66.77.88.99 10000 typ host
```

Currently, a=candidate lines are the only thing that are contained within IceCandidate, as this is the only information that is needed that is not present in the initial offer (i.e. for trickle candidates).

5. Media Setup Overview

The example here shows a typical call setup using the JSEP model. We assume the following architecture in this example, where UA is synonymous with "browser", and JS is synonymous with "web application":

```
OffererUA <-> OffererJS <-> WebServer <-> AnswererJS <-> AnswererUA
```

5.1. Initiating the Session

The initiator creates a PeerConnection, installs its IceCallback, and adds the desired MediaStreams (presumably obtained via getUserMedia). The PeerConnection is in the NEW state.

```
OffererJS->OffererUA: var pc = new PeerConnection(config, iceCb);
OffererJS->OffererUA: pc.addStream(stream);
```
5.1.1. Generating An Offer

The initiator then creates a session description to offer to the callee. This description includes the codecs and other necessary session parameters, as well as information about each of the streams that has been added (e.g. SSRC, CNAME, etc.) The created description includes all parameters that the offerer’s UA supports; if the initiator wants to influence the created offer, they can pass in a MediaHints object to createOffer that allows for customization (e.g. if the initiator wants to receive but not send video). The initiator can also directly manipulate the created session description as well, perhaps if it wants to change the priority of the offered codecs.

OffererJS->OffererUA: var offer = pc.createOffer(null);

5.1.2. Applying the Offer

The initiator then instructs the PeerConnection to use this offer as the local description for this session, i.e. what codecs it will use for received media, what SRTP keys it will use for sending media (if using SDES), etc. In order that the UA handle the description properly, the initiator marks it as an offer when calling setLocalDescription; this indicates to the UA that multiple capabilities have been offered, but this set may be pared back later, when the answer arrives.

Since the local user agent must be prepared to receive media upon applying the offer, this operation will cause local decoder resources to be allocated, based on the codecs indicated in the offer.

OffererJS->OffererUA: pc.setLocalDescription(SDP_OFFER, offer);

5.1.3. Initiating ICE

The initiator can now start the ICE process of candidate generation and connectivity checking. This results in callbacks to the application’s IceCallback. Candidates are provided to the IceCallback as they are allocated, with the |moreToFollow| argument set to true if there are still allocations pending; when the last allocation completes or times out, this callback will be invoked with |moreToFollow| set to false.

OffererJS->OffererUA: pc.startIce();
OffererUA->OffererJS: iceCallback(candidate, ...);

5.1.4. Serializing the Offer and Candidates

At this point, the offerer is ready to send its offer to the callee
using its preferred signaling protocol. Depending on the protocol, it can either send the initial session description first, and then "trickle" the ICE candidates as they are given to the application, or it can wait for all the ICE candidates to be collected, and then send the offer and list of candidates all at once.

5.2. Receiving the Session

Through the chosen signaling protocol, the recipient is notified of an incoming session request. It creates a PeerConnection, and installs its own IceCallback.

AnswererJS->AnswererUA: var pc = new PeerConnection(config, iceCb);

5.2.1. Receiving the Offer

The recipient converts the received offer from its signaling protocol into SDP format, and supplies it to its PeerConnection, again marking it as an offer. As a remote description, the offer indicates what codecs the remote side wants to use for receiving, as well as what SRTP keys it will use for sending. The setting of the remote description causes callbacks to be issued, informing the application of what kinds of streams are present in the offer.

This step will also cause encoder resources to be allocated, based on the codecs specified in |offer|.

AnswererJS->AnswererUA: pc.setRemoteDescription(SDP_OFFER, offer);
AnswererUA->AnswererJS: onAddStream(stream);

5.2.2. Initiating ICE

The recipient then starts its own ICE state machine, to allow connectivity to be established as quickly as possible.

AnswererJS->AnswererUA: pc.startIce();
AnswererUA->AnswererJS: iceCallback(candidate, ...);

5.2.3. Handling ICE Messages

If ICE candidates from the remote site were included in the offer, the ICE Agent will automatically start trying to use them. Otherwise, if ICE candidates are sent separately, they are passed into the PeerConnection when they arrive.

AnswererJS->AnswererUA: pc.processIceMessage(candidate);

5.2.4. Generating the Answer
Once the recipient has decided to accept the session, it generates an answer session description. This process performs the appropriate intersection of codecs and other parameters to generate the correct answer. As with the offer, MediaHints can be provided to influence the answer that is generated, and/or the application can post-process the answer manually.

AnswererJS->AnswererUA: pc.createAnswer(offer, null);

5.2.5. Applying the Answer

The recipient then instructs the PeerConnection to use the answer as its local description for this session, i.e. what codecs it will use to receive media, etc. It also marks the description as an answer, which tells the UA that these parameters are final. This causes the PeerConnection to move to the ACTIVE state, and transmission of media by the answerer to start.

AnswererJS->AnswererUA: pc.setLocalDescription(SDP_ANSWER, answer);
AnswererUA->OffererUA: <media>

5.2.6. Serializing the Answer

As with the offer, the answer (with or without candidates) is now converted to the desired signaling format and sent to the initiator.

5.3. Completing the Session

5.3.1. Receiving the Answer

The initiator converts the answer from the signaling protocol and applies it as the remote description, marking it as an answer. This causes the PeerConnection to move to the ACTIVE state, and transmission of media by the offerer to start.

OffererJS->OffererUA: pc.setRemoteDescription(SDP_ANSWER, answer);
OffererUA->AnswererUA: <media>

5.4. Updates to the Session

Updates to the session are handled with a new offer/answer exchange. However, since media will already be flowing at this point, the new offerer needs to support both its old session description as well as the new one it has offered, until the change is accepted by the remote side.

Note also that in an update scenario, the roles may be reversed, i.e. the update offerer can be different than the original offerer.
6. Proposed WebRTC API changes

6.1. PeerConnection API

The text below indicates the recommended changes to the PeerConnection API to implement the JSEP functionality. Methods marked with a [+] are new/proposed; methods marked with a [-] have been removed in this proposal.

Constructor (in DOMString configuration, in IceCallback iceCb)

```javascript
interface PeerConnection {
    // creates a blob of SDP to be provided as an offer.
    [+]
    SessionDescription createOffer (MediaHints hints);
    // creates a blob of SDP to be provided as an answer.
    [+]
    SessionDescription createAnswer (DOMString offer, MediaHints hints);
    // actions, for setLocalDescription/setRemoteDescription
    [+]
    const unsigned short SDP_OFFER = 0x100;
    [+]
    const unsigned short SDP_PRANSWER = 0x200;
    [+]
    const unsigned short SDP_ANSWER = 0x300;
    // sets the local session description
    [+]
    void setLocalDescription (unsigned short action, SessionDescription desc);
    // sets the remote session description
    [+]
    void setRemoteDescription (unsigned short action, SessionDescription desc);
    // returns the current local session description
    [+]
    readonly SessionDescription localDescription;
    // returns the current remote session description
    [+]
    readonly SessionDescription remoteDescription;
    [-]
    void processSignalingMessage (DOMString message);
    const unsigned short NEW = 0; // initial state
    [+]
    const unsigned short OPENING = 1; // local or remote desc set
    const unsigned short ACTIVE = 2; // local and remote desc set
    const unsigned short CLOSED = 3; // ended state
    readonly attribute unsigned short readyState;
    // starts ICE connection/handshaking
    [+]
    void startIce (optional IceOptions options);
    // processes received ICE information
    [+]
    void processIceMessage (IceCandidate candidate);
    const unsigned short ICE_GATHERING = 0x100;
    const unsigned short ICE_WAITING = 0x200;
    const unsigned short ICE_CHECKING = 0x300;
    const unsigned short ICE_CONNECTED = 0x400;
    const unsigned short ICE_COMPLETED = 0x500;
    const unsigned short ICE_FAILED = 0x600;
    const unsigned short ICE_CLOSED = 0x700;
    readonly attribute unsigned short iceState;
}
```
[-] const unsigned short SDP_IDLE = 0x1000;
[-] const unsigned short SDP_WAITING = 0x2000;
[-] const unsigned short SDP_GLARE = 0x3000;
[ ] readonly attribute unsigned short sdpState;
  void addStream (MediaStream stream, MediaStreamHints hints);
  void removeStream (MediaStream stream);
  readonly attribute MediaStream[] localStreams;
  readonly attribute MediaStream[] remoteStreams;
  void close ();
  [ rest of interface omitted ]
);

[Constructor (in DOMString sdp)]
interface SessionDescription {
  // adds the specified candidate to the description
  void addCandidate (IceCandidate candidate);
  // serializes the description to SDP
  DOMString toSdp();
};

[Constructor (in DOMString label, in DOMString candidateLine)]
interface IceCandidate {
  // the m= line this candidate is associated with
  readonly DOMString label;
  // creates a SDP-ized form of this candidate
  DOMString toSdp();
};

6.1.1 MediaHints

MediaHints is an object that can be passed into createOffer or createAnswer to affect the type of offer/answer that is generated.

The following properties can be set on MediaHints:

  has_audio: boolean

  Indicates whether we want to receive audio; defaults to true if we have audio streams, else false

  has_video: boolean

  Indicates whether we want to receive video; defaults to true if we have video streams, else false

As an example, MediaHints could be used to create a session that transmits only audio, but is able to receive video from the remote side, by forcing the inclusion of a m=video line even when no video
6.1.2 createOffer

The createOffer method generates a blob of SDP that contains a RFC 3264 offer with the supported configurations for the session, including descriptions of the local MediaStreams attached to this PeerConnection, the codec/RTP/RTCP options supported by this implementation, and any candidates that have been gathered by the ICE Agent. The |hints| parameter may be supplied to provide additional control over the generated offer.

As an offer, the generated SDP will contain the full set of capabilities supported by the session (as opposed to an answer, which will include only a specific negotiated subset to use); for each SDP line, the generation of the SDP must follow the appropriate process for generating an offer. In the event createOffer is called after the session is established, createOffer will generate an offer that is compatible with the current session, incorporating any changes that have been made to the session since the last complete offer-answer exchange, such as addition or removal of streams. If no changes have been made, the offer will be identical to the current local description.

Session descriptions generated by createOffer must be immediately usable by setLocalDescription; if a system has limited resources (e.g. a finite number of decoders), createOffer should return an offer that reflects the current state of the system, so that setLocalDescription will succeed when it attempts to acquire those resources.

Calling this method does not change the state of the PeerConnection; its use is not required.

A TBD exception is thrown if the |hints| parameter is malformed.

6.1.3 createAnswer

The createAnswer method generates a blob of SDP that contains a RFC 3264 SDP answer with the supported configuration for the session that is compatible with the parameters supplied in |offer|. Like createOffer, the returned blob contains descriptions of the local MediaStreams attached to this PeerConnection, the codec/RTP/RTCP options negotiated for this session, and any candidates that have been gathered by the ICE Agent. The |hints| parameter may be supplied to provide additional control over the generated answer.

As an answer, the generated SDP will contain a specific configuration
that specifies how the media plane should be established. For each SDP line, the generation of the SDP must follow the appropriate process for generating an answer.

Session descriptions generated by createAnswer must be immediately usable by setLocalDescription; like createOffer, the returned description should reflect the current state of the system.

Calling this method does not change the state of the PeerConnection; its use is not required.

A TBD exception is thrown if the |hints| parameter is malformed, or the |offer| parameter is missing or malformed.

6.1.4 SDP_OFFER, SDP_PRANSWER, and SDP_ANSWER

The SDP_XXXX enums serve as arguments to setLocalDescription and setRemoteDescription. They provide information as to how the |description| parameter should be parsed, and how the media state should be changed.

SDP_OFFER indicates that a description should be parsed as an offer; said description may include many possible media configurations. A description used as a SDP_OFFER may be applied anytime the PeerConnection is in a stable state, or as an update to a previously sent but unanswered SDP_OFFER.

SDP_PRANSWER indicates that a description should be parsed as an answer, but not a final answer, and so should not result in the starting of media transmission. A description used as a SDP_PRANSWER may be applied as a response to a SDP_OFFER, or an update to a previously sent SDP_PRANSWER.

SDP_ANSWER indicates that a description should be parsed as an answer, and the offer-answer exchange should be considered complete. A description used as a SDP_ANSWER may be applied as a response to a SDP_OFFER, or an update to a previously send SDP_PRANSWER.

6.1.5 setLocalDescription

The setLocalDescription method instructs the PeerConnection to apply the supplied SDP blob as its local configuration. The |type| parameter indicates whether the blob should be processed as an offer (SDP_OFFER), provisional answer (SDP_PRANSWER), or final answer (SDP_ANSWER); offers and answers are checked differently, using the various rules that exist for each SDP line.

This API changes the local media state; among other things, it sets
up local resources for receiving and decoding media. In order to successfully handle scenarios where the application wants to offer to change from one media format to a different, incompatible format, the PeerConnection must be able to simultaneously support use of both the old and new local descriptions (e.g. support codecs that exist in both descriptions) until a final answer is received, at which point the PeerConnection can fully adopt the new local description, or roll back to the old description if the remote side denied the change.

Changes to the state of media transmission will only occur when a final answer is successfully applied.

A TBD exception is thrown if |description| is invalid. A TBD exception is thrown if there are insufficient local resources to apply |description|.

6.1.6 setRemoteDescription

The setRemoteDescription method instructs the PeerConnection to apply the supplied SDP blob as the desired remote configuration. As in setLocalDescription, the |type| parameter indicates how the blob should be processed.

This API changes the local media state; among other things, it sets up local resources for sending and encoding media.

Changes to the state of media transmission will only occur when a final answer is successfully applied.

A TBD exception is thrown if |description| is invalid. A TBD exception is thrown if there are insufficient local resources to apply |description|.

6.1.7 localDescription

The localDescription method returns a copy of the current local configuration, i.e. what was most recently passed to setLocalDescription, plus any local candidates that have been generated by the ICE Agent.

A null object will be returned if the local description has not yet been established.

6.1.8 remoteDescription

The remoteDescription method returns a copy of the current remote configuration, i.e. what was most recently passed to setRemoteDescription, plus any remote candidates that have been
supplied via processIceMessage.

A null object will be returned if the remote description has not yet been established.

6.1.9 IceOptions

IceOptions is an object that can be passed into startIce to restrict the candidates that are provided to the application and used for connectivity checks. This can be useful if the application wants to only use TURN candidates for privacy reasons, or only local + STUN candidates for cost reasons.

The following properties can be set on IceOptions:

use_candidates: "all", "no_relay", "only_relay"

Indicates what types of local candidates should be used; defaults to "all"

6.1.10 startIce

The startIce method starts or updates the ICE Agent process of gathering local candidates and pinging remote candidates. The |options| argument can be used to restrict which types of local candidates are provided to the application and used for pinging; this can be used to limit the use of TURN candidates by a callee to avoid leaking location information prior to the call being accepted.

This call may result in a change to the state of the ICE Agent, and may result in a change to media state if it results in connectivity being established.

A TBD exception will be thrown if |options| is malformed.

6.1.11 processIceMessage

The processIceMessage method provides a remote candidate to the ICE Agent, which will be added to the remote description. If startIce has been called, connectivity checks will be sent to the new candidates.

This call will result in a change to the state of the ICE Agent, and may result in a change to media state if it results in connectivity being established.

A TBD exception will be thrown if |candidate| is missing or malformed.
7. Example API Flows

Below are several sample flows for the new PeerConnection and library APIs, demonstrating when the various APIs are called in different situations and with various transport protocols.

7.1. Call using ROAP

This example demonstrates a ROAP call, without the use of trickle candidates.

// Call is initiated toward Answerer
OffererJS->OffererUA: pc = new PeerConnection();
OffererJS->OffererUA: pc.addStream(localStream, null);
OffererJS->OffererUA: pc.startIce();
OffererUA->OffererJS: iceCallback(candidate, false);
OffererJS->OffererUA: offer = pc.createOffer(null);
OffererJS->OffererUA: pc.setLocalDescription(SDP_OFFER, offer.toSdp());
OffererJS->AnswererJS: {"type":"OFFER", "sdp":"<offer>"}

// OFFER arrives at Answerer
AnswererJS->AnswererUA: pc = new PeerConnection();
AnswererJS->AnswererUA: pc.setRemoteDescription(SDP_OFFER, msg.sdp);
AnswererUA->AnswererJS: onaddstream(remoteStream);
AnswererJS->AnswererUA: pc.startIce();
AnswererUA->OffererUA: iceCallback(candidate, false);

// Answerer accepts call
AnswererJS->AnswererUA: peer.addStream(localStream, null);
AnswererJS->AnswererUA: answer = peer.createAnswer(msg.offer, null);
AnswererJS->AnswererUA: peer.setLocalDescription(SDP_ANSWER, answer);
AnswererJS->OffererJS: {"type":"ANSWER","sdp":"<answer>"}

// ANSWER arrives at Offerer
OffererJS->OffererUA: peer.setRemoteDescription(ANSWER, answer);
OffererUA->OffererUA: onaddstream(remoteStream);

// ICE Completes (at Answerer)
AnswererUA->AnswererJS: onopen();
AnswererUA->OffererUA: Media

// ICE Completes (at Offerer)
OffererUA->OffererJS: onopen();
OffererJS->AnswererJS: {"type":"OK"}
OffererUA->AnswererUA: Media

7.2. Call using XMPP
This example demonstrates an XMPP call, making use of trickle candidates.

// Call is initiated toward Answerer
OffererJS->OffererUA: pc = new PeerConnection();
OffererJS->OffererUA: pc.addStream(localStream, null);
OffererJS->OffererUA: offer = pc.createOffer(null);
OffererJS->OffererUA: pc.setLocalDescription(SDP_OFFER, offer);
OffererJS: xmpp = createSessionInitiate(offer);
OffererJS->AnswererJS: <jingle action="session-initiate"/>

OffererJS->OffererUA: pc.startIce();
OffererUA->OffererJS: iceCallback(cand);
OffererJS: createTransportInfo(cand, ...);
OffererJS->AnswererJS: <jingle action="transport-info"/>

// session-initiate arrives at Answerer
AnswererJS->AnswererUA: pc = new PeerConnection();
AnswererJS: offer = parseSessionInitiate(xmpp);
AnswererJS->AnswererUA: pc.setRemoteDescription(SDP_OFFER, offer);
AnswererUA->AnswererJS: onaddstream(remoteStream);

// transport-infos arrive at Answerer
AnswererJS->AnswererUA: candidates = parseTransportInfo(xmpp);
AnswererJS->AnswererUA: pc.processIceMessage(candidates);
AnswererJS->AnswererUA: pc.startIce();
AnswererUA->AnswererJS: iceCallback(cand, ...)
AnswererJS: createTransportInfo(cand);
AnswererJS->OffererJS: <jingle action="transport-info"/>

// transport-infos arrive at Offerer
OffererJS->OffererUA: candidates = parseTransportInfo(xmpp);
OffererJS->OffererUA: pc.processIceMessage(candidates);

// Answerer accepts call
AnswererJS->AnswererUA: peer.addStream(localStream, null);
AnswererJS->AnswererUA: answer = peer.createAnswer(offer, null);
AnswererJS: xmpp = createSessionAccept(answer);
AnswererJS->AnswererUA: pc.setLocalDescription(SDP_ANSWER, answer);
AnswererJS->OffererJS: <jingle action="session-accept"/>

// session-accept arrives at Offerer
OffererJS: answer = parseSessionAccept(xmpp);
OffererJS->OffererUA: peer.setRemoteDescription(ANSWER, answer);
OffererUA->OffererJS: onaddstream(remoteStream);

// ICE Completes (at Answerer)
AnswererUA->AnswererJS: onopen();
7.3. Adding video to a call, using XMPP

This example demonstrates an XMPP call, where the XMPP content-add mechanism is used to add video media to an existing session. For simplicity, candidate exchange is not shown.

Note that the offerer for the change to the session may be different than the original call offerer.

```
// Offerer adds video stream
OffererJS->OffererUA: pc.addStream(videoStream)
OffererJS->OffererUA: offer = pc.createOffer(null);
OffererJS: xmpp = createContentAdd(offer);
OffererJS->OffererUA: pc.setLocalDescription(SDP_OFFER, offer);
```

```
OffererJS->AnswererJS: <jingle action="content-add"/>
```

```
// content-add arrives at Answerer
AnswererJS: offer = parseContentAdd(xmpp);
AnswererJS->AnswererUA: pc.setRemoteDescription(SDP_OFFER, offer);
AnswererJS->AnswererUA: answer = pc.createAnswer(offer, null);
AnswererJS->AnswererUA: pc.setLocalDescription(SDP_ANSWER, answer);
AnswererJS: xmpp = createContentAccept(answer);
AnswererJS->OffererJS: <jingle action="content-accept"/>
```

```
// content-accept arrives at Offerer
OffererJS: answer = parseContentAccept(xmpp);
OffererJS->OffererUA: pc.setRemoteDescription(SDP_ANSWER, answer);
```

7.4. Simultaneous add of video streams, using XMPP

This example demonstrates an XMPP call, where new video sources are added at the same time to a call that already has video; since adding these sources only affects one side of the call, there is no conflict. The XMPP description-info mechanism is used to indicate the new sources to the remote side.

```
// Offerer and "Answerer" add video streams at the same time
OffererJS->OffererUA: pc.addStream(offererVideoStream2)
OffererJS->OffererUA: offer = pc.createOffer(null);
OffererJS: xmpp = createDescriptionInfo(offer);
OffererJS->OffererUA: pc.setLocalDescription(SDP_OFFER, offer);
```
OffererJS->AnswererJS: <jingle action="description-info"/>

AnswererJS->AnswererUA: pc.addStream(answererVideoStream2)
AnswererJS->AnswererUA: offer = pc.createOffer(null);
AnswererJS: xmpp = createDescriptionInfo(offer);
AnswererJS->AnswererUA: pc.setLocalDescription(SDP_OFFER, offer);
AnswererJS->OffererJS: <jingle action="description-info"/>

// description-info arrives at "Answerer", and is acked
AnswererJS: offer = parseDescriptionInfo(xmpp);
AnswererJS->OffererJS: <iq type="result"/> // ack

// description-info arrives at Offerer, and is acked
OffererJS: offer = parseDescriptionInfo(xmpp);
OffererJS->AnswererJS: <iq type="result"/> // ack

// ack arrives at Offerer; remote offer is used as an answer
OffererJS->OffererUA: pc.setRemoteDescription(SDP_ANSWER, offer);

// ack arrives at "Answerer"; remote offer is used as an answer
AnswererJS->AnswererUA: pc.setRemoteDescription(SDP_ANSWER, offer);

7.5. Call using SIP

This example demonstrates a simple SIP call (e.g. where the client
talks to a SIP proxy over WebSockets).

// Call is initiated toward Answerer
OffererJS->OffererUA: pc = new PeerConnection();
OffererJS->OffererUA: pc.addStream(localStream, null);
OffererJS->OffererUA: pc.startIce();
OffererUA->OffererJS: iceCallback(candidate, false);
OffererJS->OffererUA: offer = pc.createOffer(null);
OffererJS->OffererUA: pc.setLocalDescription(SDP_OFFER, offer);
OffererJS: sip = createInvite(offer);
OffererJS->AnswererJS: SIP INVITE w/ SDP

// INVITE arrives at Answerer
AnswererJS->AnswererUA: pc = new PeerConnection();
AnswererJS: offer = parseInvite(sip);
AnswererJS->AnswererUA: pc.setRemoteDescription(SDP_OFFER, offer);
AnswererUA->AnswererJS: onaddstream(remoteStream);
AnswererJS->AnswererUA: pc.startIce();
AnswererUA->OffererUA: iceCallback(candidate, false);

// Answerer accepts call
AnswererJS->AnswererUA: peer.addStream(localStream, null);
AnswererJS->AnswererUA: answer = peer.createAnswer(offer, null);
AnswererJS:    sip = createResponse(200, answer);
AnswererJS->AnswererUA: peer.setLocalDescription(SDP_ANSWER, answer);
AnswererJS->OffererJS:  200 OK w/ SDP

// 200 OK arrives at Offerer
OffererJS:    answer = parseResponse(sip);
OffererJS->OffererUA: peer.setRemoteDescription(ANSWER, answer);
OffererUA->OffererJS:  onaddstream(remoteStream);
OffererJS->AnswererJS:  ACK

// ICE Completes (at Answerer)
AnswererUA->AnswererJS: onopen();
AnswererUA->OffererUA:  Media

// ICE Completes (at Offerer)
OffererUA->OffererJS:  onopen();
OffererUA->AnswererUA:  Media

7.6. Handling early media (e.g. 1-800-FEDEX), using SIP

This example demonstrates how early media could be handled; for simplicity, only the offerer side of the call is shown.

// Call is initiated toward Answerer
OffererJS->OffererUA:  pc = new PeerConnection();
OffererJS->OffererUA:  pc.addStream(localStream, null);
OffererJS->OffererUA:  pc.startIce();
OffererUA->OffererJS:  iceCallback(candidate, false);
OffererJS->OffererUA:  offer = pc.createOffer(null);
OffererJS->OffererUA:  pc.setLocalDescription(SDP_OFFER, offer);
OffererJS:              sip = createInvite(offer);
OffererJS->AnswererJS:  SIP INVITE w/ SDP

// 180 Ringing is received by offerer, w/ SDP
OffererJS:    answer = parseResponse(sip);
OffererJS->OffererUA:  pc.setRemoteDescription(SDP_PRANSWER, answer);
OffererUA->OffererJS:  onaddstream(remoteStream);

// ICE Completes (at Offerer)
OffererUA->OffererJS:  onopen();
OffererUA->AnswererUA:  Media

// 200 OK arrives at Offerer
OffererJS:    answer = parseResponse(sip);
OffererJS->OffererUA:  pc.setRemoteDescription(SDP_ANSWER, answer);
OffererJS->AnswererJS:  ACK

8. Example Application
The following example demonstrates a simple video calling application, roughly corresponding to the flow in Example 7.1.

```javascript
var signalingChannel = createSignalingChannel();
var pc = null;
var hasCandidates = false;

function start(isCaller) {
  // create a PeerConnection and hook up the IceCallback
  pc = new webkitPeerConnection('', function (candidate, moreToFollow) {
    if (!moreToFollow) {
      hasCandidates = true;
      maybeSignal(isCaller);
    }
  });

  // get the local stream and show it in the local video element
  navigator.webkitGetUserMedia({
    "audio": true, "video": true}, function (localStream) {
    selfView.src = webkitURL.createObjectURL(localStream);
    pc.addStream(localStream);
    maybeSignal(isCaller);
  });

  // once remote stream arrives, show it in the remote video element
  pc.onaddstream = function(evt) {
    remoteView.src = webkitURL.createObjectURL(evt.stream);
  };

  // if we're the caller, create and install our offer,
  // and start candidate generation
  if (isCaller) {
    offer = pc.createOffer(null);
    pc.setLocalDescription(SDP_OFFER, offer);
    pc.startIce();
  }
}

function maybeSignal(isCaller) {
  // only signal once we have a local stream and local candidates
  if (localStreams.size() == 0 || !hasCandidates) return;
  if (isCaller) {
    offer = pc.localDescription;
    signalingChannel.send(JSON.stringify({ "type": "offer", "sdp": offer }));
  } else {
    // if we're the callee, generate, apply, and send the answer
  }
}
```
answer = pc.createAnswer(pc.remoteDescription, null);
pc.setLocalDescription(SDP_ANSWER, answer);
signalingChannel.send(
    JSON.stringify({ "type": "answer", "sdp": answer }));
}
}
signalingChannel.onmessage = function(evt) {
    var msg = JSON.parse(evt.data);
    if (msg.type == "offer") {
        // create the PeerConnection
        start(false);
        // feed the received offer into the PeerConnection and
        // start candidate generation
        pc.setRemoteDescription(PeerConnection.SDP_OFFER, msg.sdp);
        pc.startIce();
    } else if (msg.type == "answer") {
        // feed the answer into the PeerConnection to complete setup
        pc.setRemoteDescription(PeerConnection.SDP_ANSWER, msg.sdp);
    }
}

9. Security Considerations

TODO

10. IANA Considerations

This document requires no actions from IANA.

11. Acknowledgements

Harald Alvestrand, Dan Burnett, Neil Stratford, Eric Rescorla, and
Anant Narayanan all provided valuable feedback on this proposal.
Matthew Kaufman provided the observation that keeping state out of
the browser allows a call to continue even if the page is reloaded.
Adam Bergvist provided a code example that served as the basis for
the example in Section 8.

12. References

12.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate

with Session Description Protocol (SDP)", RFC 3264, June 2002.

12.2. Informative References


Available at http://dev.w3.org/2011/webrtc/editor/webrtc.html

Appendix A. Open Issues

- Determine list of exceptions that can be thrown by each method. Leaning toward something like a PCEXception, a la https://developer.mozilla.org/en/IndexedDB/IDBDatabaseException

- Need callback to indicate that the transport is down, e.g. ICE_DISCONNECTED or ondisconnected().

Appendix B. Change log

00: Migrated from draft-uberti-rtcweb-jsep-02.

Authors’ Addresses

Justin Uberti
Google
5 Cambridge Center
Cambridge, MA 02142

Email: justin@uberti.name

Cullen Jennings
Cisco
170 West Tasman Drive
San Jose, CA 95134
USA

Email: fluffy@cisco.com

Uberti Expires September 4, 2012
Security Considerations for RTC-Web
draft-ietf-rtcweb-security-02

Abstract

The Real-Time Communications on the Web (RTC-Web) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for RTC-Web technology are real-time audio and/or video calls, Web conferencing, and direct data transfer. Unlike most conventional real-time systems (e.g., SIP-based soft phones) RTC-Web communications are directly controlled by some Web server, which poses new security challenges. For instance, a Web browser might expose a JavaScript API which allows a server to place a video call. Unrestricted access to such an API would allow any site which a user visited to "bug" a user’s computer, capturing any activity which passed in front of their camera. This document defines the RTC-Web threat model and defines an architecture which provides security within that threat model.

Legal

THIS DOCUMENT AND THE INFORMATION CONTAINED THEREIN ARE PROVIDED ON AN "AS IS" BASIS AND THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST, AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any

Rescorla                     Expires September 13, 2012
[Page 1]
# Table of Contents

1. Introduction .................................................. 4  
2. Terminology .................................................... 5  
3. The Browser Threat Model ....................................... 5  
   3.1. Access to Local Resources ................................. 6  
   3.2. Same Origin Policy ....................................... 6  
   3.3. Bypassing SOP: CORS, WebSockets, and consent to communicate ............................................. 7  
4. Security for RTC-Web Applications ............................ 7  
   4.1. Access to Local Devices ................................. 8  
      4.1.1. Calling Scenarios and User Expectations .......... 8  
      4.1.1.1. Dedicated Calling Services ...................... 8  
      4.1.1.2. Calling the Site You’re On ....................... 9  
      4.1.1.3. Calling to an Ad Target .......................... 9  
   4.1.2. Origin-Based Security ............................... 10  
   4.2. Communications Consent Verification ................... 12  
      4.2.1. ICE .................................................. 12  
      4.2.2. Masking ............................................. 13  
      4.2.3. Backward Compatibility ............................ 13  
      4.2.4. IP Location Privacy ................................ 14  
   4.3. Communications Security ................................ 14  
      4.3.1. Protecting Against Retrospective Compromise .... 15  
      4.3.2. Protecting Against During-Call Attack ........... 16  
      4.3.2.1. Key Continuity .................................. 16  
      4.3.2.2. Short Authentication Strings .................... 17  
      4.3.2.3. Third Party Identity ............................. 18  
5. Security Considerations ........................................ 18  
6. Acknowledgements .............................................. 18  
7. References .......................................................... 19  
    7.1. Normative References ................................... 19  
    7.2. Informative References .................................. 19  
Author’s Address .................................................. 21
1. Introduction

The Real-Time Communications on the Web (RTC-Web) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for RTC-Web technology are real-time audio and/or video calls, Web conferencing, and direct data transfer. Unlike most conventional real-time systems, (e.g., SIP-based [RFC3261] soft phones) RTC-Web communications are directly controlled by some Web server. A simple case is shown below.

![Diagram of a simple RTC-Web system]

In the system shown in Figure 1, Alice and Bob both have RTC-Web enabled browsers and they visit some Web server which operates a calling service. Each of their browsers exposes standardized JavaScript calling APIs which are used by the Web server to set up a call between Alice and Bob. While this system is topologically similar to a conventional SIP-based system (with the Web server acting as the signaling service and browsers acting as softphones), control has moved to the central Web server; the browser simply provides API points that are used by the calling service. As with any Web application, the Web server can move logic between the server and JavaScript in the browser, but regardless of where the code is executing, it is ultimately under control of the server.

It should be immediately apparent that this type of system poses new security challenges beyond those of a conventional VoIP system. In particular, it needs to contend with malicious calling services. For example, if the calling service can cause the browser to make a call at any time to any callee of its choice, then this facility can be
used to bug a user’s computer without their knowledge, simply by placing a call to some recording service. More subtly, if the exposed APIs allow the server to instruct the browser to send arbitrary content, then they can be used to bypass firewalls or mount denial of service attacks. Any successful system will need to be resistant to this and other attacks.

A companion document [I-D.ietf-rtcweb-security-arch] describes a security architecture intended to address the issues raised in this document.

2. 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].

3. The Browser Threat Model

The security requirements for RTC-Web follow directly from the requirement that the browser’s job is to protect the user. Huang et al. [huang-w2sp] summarize the core browser security guarantee as:

Users can safely visit arbitrary web sites and execute scripts provided by those sites.

It is important to realize that this includes sites hosting arbitrary malicious scripts. The motivation for this requirement is simple: it is trivial for attackers to divert users to sites of their choice. For instance, an attacker can purchase display advertisements which direct the user (either automatically or via user clicking) to their site, at which point the browser will execute the attacker’s scripts. Thus, it is important that it be safe to view arbitrarily malicious pages. Of course, browsers inevitably have bugs which cause them to fall short of this goal, but any new RTC-Web functionality must be designed with the intent to meet this standard. The remainder of this section provides more background on the existing Web security model.

In this model, then, the browser acts as a TRUSTED COMPUTING BASE (TCB) both from the user’s perspective and to some extent from the server’s. While HTML and JS provided by the server can cause the browser to execute a variety of actions, those scripts operate in a sandbox that isolates them both from the user’s computer and from each other, as detailed below.
Conventionally, we refer to either WEB ATTACKERS, who are able to induce you to visit their sites but do not control the network, and NETWORK ATTACKERS, who are able to control your network. Network attackers correspond to the [RFC3552] "Internet Threat Model". In general, it is desirable to build a system which is secure against both kinds of attackers, but realistically many sites do not run HTTPS [RFC2818] and so our ability to defend against network attackers is necessarily somewhat limited. Most of the rest of this section is devoted to web attackers, with the assumption that protection against network attackers is provided by running HTTPS.

3.1. Access to Local Resources

While the browser has access to local resources such as keying material, files, the camera and the microphone, it strictly limits or forbids web servers from accessing those same resources. For instance, while it is possible to produce an HTML form which will allow file upload, a script cannot do so without user consent and in fact cannot even suggest a specific file (e.g., /etc/passwd); the user must explicitly select the file and consent to its upload. [Note: in many cases browsers are explicitly designed to avoid dialogs with the semantics of "click here to screw yourself", as extensive research shows that users are prone to consent under such circumstances.]

Similarly, while Flash SWFs can access the camera and microphone, they explicitly require that the user consent to that access. In addition, some resources simply cannot be accessed from the browser at all. For instance, there is no real way to run specific executables directly from a script (though the user can of course be induced to download executable files and run them).

3.2. Same Origin Policy

Many other resources are accessible but isolated. For instance, while scripts are allowed to make HTTP requests via the XMLHttpRequest() API those requests are not allowed to be made to any server, but rather solely to the same ORIGIN from whence the script came.[RFC6454] (although CORS [CORS] and WebSockets [RFC6455] provides a escape hatch from this restriction, as described below.) This SAME ORIGIN POLICY (SOP) prevents server A from mounting attacks on server B via the user’s browser, which protects both the user (e.g., from misuse of his credentials) and the server (e.g., from DoS attack).

More generally, SOP forces scripts from each site to run in their own, isolated, sandboxes. While there are techniques to allow them to interact, those interactions generally must be mutually consensual
(by each site) and are limited to certain channels. For instance, multiple pages/browser panes from the same origin can read each other's JS variables, but pages from the different origins--or even iframes from different origins on the same page--cannot.

3.3. Bypassing SOP: CORS, WebSockets, and consent to communicate

While SOP serves an important security function, it also makes it inconvenient to write certain classes of applications. In particular, mash-ups, in which a script from origin A uses resources from origin B, can only be achieved via a certain amount of hackery. The W3C Cross-Origin Resource Sharing (CORS) spec [CORS] is a response to this demand. In CORS, when a script from origin A executes what would otherwise be a forbidden cross-origin request, the browser instead contacts the target server to determine whether it is willing to allow cross-origin requests from A. If it is so willing, the browser then allows the request. This consent verification process is designed to safely allow cross-origin requests.

While CORS is designed to allow cross-origin HTTP requests, WebSockets [RFC6455] allows cross-origin establishment of transparent channels. Once a WebSockets connection has been established from a script to a site, the script can exchange any traffic it likes without being required to frame it as a series of HTTP request/response transactions. As with CORS, a WebSockets transaction starts with a consent verification stage to avoid allowing scripts to simply send arbitrary data to another origin.

While consent verification is conceptually simple--just do a handshake before you start exchanging the real data--experience has shown that designing a correct consent verification system is difficult. In particular, Huang et al. [huang-w2sp] have shown vulnerabilities in the existing Java and Flash consent verification techniques and in a simplified version of the WebSockets handshake. In particular, it is important to be wary of CROSS-PROTOCOL attacks in which the attacking script generates traffic which is acceptable to some non-Web protocol state machine. In order to resist this form of attack, WebSockets incorporates a masking technique intended to randomize the bits on the wire, thus making it more difficult to generate traffic which resembles a given protocol.

4. Security for RTC-Web Applications
4.1. Access to Local Devices

As discussed in Section 1, allowing arbitrary sites to initiate calls violates the core Web security guarantee; without some access restrictions on local devices, any malicious site could simply bug a user. At minimum, then, it MUST NOT be possible for arbitrary sites to initiate calls to arbitrary locations without user consent. This immediately raises the question, however, of what should be the scope of user consent.

For the rest of this discussion we assume that the user is somehow going to grant consent to some entity (e.g., a social networking site) to initiate a call on his behalf. This consent may be limited to a single call or may be a general consent. In order for the user to make an intelligent decision about whether to allow a call (and hence his camera and microphone input to be routed somewhere), he must understand either who is requesting access, where the media is going, or both. So, for instance, one might imagine that at the time access to camera and microphone is requested, the user is shown a dialog that says "site X has requested access to camera and microphone, yes or no" (though note that this type of in-flow interface violates one of the guidelines in Section 3). The user's decision will of course be based on his opinion of Site X. However, as discussed below, this is a complicated concept.

4.1.1. Calling Scenarios and User Expectations

While a large number of possible calling scenarios are possible, the scenarios discussed in this section illustrate many of the difficulties of identifying the relevant scope of consent.

4.1.1.1. Dedicated Calling Services

The first scenario we consider is a dedicated calling service. In this case, the user has a relationship with a calling site and repeatedly makes calls on it. It is likely that rather than having to give permission for each call that the user will want to give the calling service long-term access to the camera and microphone. This is a natural fit for a long-term consent mechanism (e.g., installing an app store "application" to indicate permission for the calling service.) A variant of the dedicated calling service is a gaming site (e.g., a poker site) which hosts a dedicated calling service to allow players to call each other.

With any kind of service where the user may use the same service to talk to many different people, there is a question about whether the user can know who they are talking to. In general, this is difficult as most of the user interface is presented by the calling site.
However, communications security mechanisms can be used to give some assurance, as described in Section 4.3.2.

4.1.1.2. Calling the Site You’re On

Another simple scenario is calling the site you’re actually visiting. The paradigmatic case here is the "click here to talk to a representative" windows that appear on many shopping sites. In this case, the user’s expectation is that they are calling the site they’re actually visiting. However, it is unlikely that they want to provide a general consent to such a site; just because I want some information on a car doesn’t mean that I want the car manufacturer to be able to activate my microphone whenever they please. Thus, this suggests the need for a second consent mechanism where I only grant consent for the duration of a given call. As described in Section 3.1, great care must be taken in the design of this interface to avoid the users just clicking through. Note also that the user interface chrome must clearly display elements showing that the call is continuing in order to avoid attacks where the calling site just leaves it up indefinitely but shows a Web UI that implies otherwise.

4.1.1.3. Calling to an Ad Target

In both of the previous cases, the user has a direct relationship (though perhaps a transient one) with the target of the call. Moreover, in both cases he is actually visiting the site of the person he is being asked to trust. However, this is not always so. Consider the case where a user is a visiting a content site which hosts an advertisement with an invitation to call for more information. When the user clicks the ad, they are connected with the advertiser or their agent.

The relationships here are far more complicated: the site the user is actually visiting has no direct relationship with the advertiser; they are just hosting ads from an ad network. The user has no relationship with the ad network, but desires one with the advertiser, at least for long enough to learn about their products. At minimum, then, whatever consent dialog is shown needs to allow the user to have some idea of the organization that they are actually calling.

However, because the user also has some relationship with the hosting site, it is also arguable that the hosting site should be allowed to express an opinion (e.g., to be able to allow or forbid a call) since a bad experience with an advertiser reflect negatively on the hosting site [this idea was suggested by Adam Barth]. However, this obviously presents a privacy challenge, as sites which host advertisements often learn very little about whether individual users
clicked through to the ads, or even which ads were presented.

4.1.2. Origin-Based Security

As discussed in Section 3.2, the basic unit of Web sandboxing is the origin, and so it is natural to scope consent to origin. Specifically, a script from origin A MUST only be allowed to initiate communications (and hence to access camera and microphone) if the user has specifically authorized access for that origin. It is of course technically possible to have coarser-scoped permissions, but because the Web model is scoped to origin, this creates a difficult mismatch.

Arguably, origin is not fine-grained enough. Consider the situation where Alice visits a site and authorizes it to make a single call. If consent is expressed solely in terms of origin, then at any future visit to that site (including one induced via mash-up or ad network), the site can bug Alice’s computer, use the computer to place bogus calls, etc. While in principle Alice could grant and then revoke the privilege, in practice privileges accumulate; if we are concerned about this attack, something else is needed. There are a number of potential countermeasures to this sort of issue.

Individual Consent
   Ask the user for permission for each call.

Callee-oriented Consent
   Only allow calls to a given user.

Cryptographic Consent
   Only allow calls to a given set of peer keying material or to a cryptographically established identity.

Unfortunately, none of these approaches is satisfactory for all cases. As discussed above, individual consent puts the user’s approval in the UI flow for every call. Not only does this quickly become annoying but it can train the user to simply click "OK", at which point the consent becomes useless. Thus, while it may be necessary to have individual consent in some case, this is not a suitable solution for (for instance) the calling service case. Where necessary, in-flow user interfaces must be carefully designed to avoid the risk of the user blindly clicking through.

The other two options are designed to restrict calls to a given target. Callee-oriented consent provided by the calling site not work well because a malicious site can claim that the user is calling any user of his choice. One fix for this is to tie calls to a cryptographically established identity. While not suitable for all
cases, this approach may be useful for some. If we consider the advertising case described in Section 4.1.1.3, it’s not particularly convenient to require the advertiser to instantiate an iframe on the hosting site just to get permission; a more convenient approach is to cryptographically tie the advertiser’s certificate to the communication directly. We’re still tying permissions to origin here, but to the media origin (and-or destination) rather than to the Web origin. [I-D.ietf-rtcweb-security-arch] and [I-D.rescorla-rtcweb-generic-idp] describe mechanisms which facilitate this sort of consent.

Another case where media-level cryptographic identity makes sense is when a user really does not trust the calling site. For instance, I might be worried that the calling service will attempt to bug my computer, but I also want to be able to conveniently call my friends. If consent is tied to particular communications endpoints, then my risk is limited. Naturally, it is somewhat challenging to design UI primitives which express this sort of policy.

4.1.3. Security Properties of the Calling Page

Origin-based security is intended to secure against web attackers. However, we must also consider the case of network attackers. Consider the case where I have granted permission to a calling service by an origin that has the HTTP scheme, e.g., http://calling-service.example.com. If I ever use my computer on an unsecured network (e.g., a hotspot or if my own home wireless network is insecure), and browse any HTTP site, then an attacker can bug my computer. The attack proceeds like this:

1. I connect to http://anything.example.org/. Note that this site is unaffiliated with the calling service.
2. The attacker modifies my HTTP connection to inject an IFRAME (or a redirect) to http://calling-service.example.com
3. The attacker forges the response apparently http://calling-service.example.com/ to inject JS to initiate a call to himself.

Note that this attack does not depend on the media being insecure. Because the call is to the attacker, it is also encrypted to him. Moreover, it need not be executed immediately; the attacker can "infect" the origin semi-permanently (e.g., with a web worker or a popunder) and thus be able to bug me long after I have left the infected network. This risk is created by allowing calls at all from a page fetched over HTTP.

Even if calls are only possible from HTTPS sites, if the site embeds active content (e.g., JavaScript) that is fetched over HTTP or from...
an untrusted site, because that JavaScript is executed in the
security context of the page [finer-grained]. Thus, it is also
dangerous to allow RTC-Web functionality from HTTPS origins that
embed mixed content. Note: this issue is not restricted to PAGES
which contain mixed content. If a page from a given origin ever
loads mixed content then it is possible for a network attacker to
infect the browser's notion of that origin semi-permanently.

4.2. Communications Consent Verification

As discussed in Section 3.3, allowing web applications unrestricted
network access via the browser introduces the risk of using the
browser as an attack platform against machines which would not
otherwise be accessible to the malicious site, for instance because
they are topologically restricted (e.g., behind a firewall or NAT).
In order to prevent this form of attack as well as cross-protocol
attacks it is important to require that the target of traffic
explicitly consent to receiving the traffic in question. Until that
consent has been verified for a given endpoint, traffic other than
the consent handshake MUST NOT be sent to that endpoint.

4.2.1. ICE

Verifying receiver consent requires some sort of explicit handshake,
but conveniently we already need one in order to do NAT hole-
punching. ICE [RFC5245] includes a handshake designed to verify that
the receiving element wishes to receive traffic from the sender. It
is important to remember here that the site initiating ICE is
presumed malicious; in order for the handshake to be secure the
receiving element MUST demonstrate receipt/knowledge of some value
not available to the site (thus preventing the site from forging
responses). In order to achieve this objective with ICE, the STUN
transaction IDs must be generated by the browser and MUST NOT be made
available to the initiating script, even via a diagnostic interface.
Verifying receiver consent also requires verifying the receiver wants
to receive traffic from a particular sender, and at this time; for
example a malicious site may simply attempt ICE to known servers that
are using ICE for other sessions. ICE provides this verification as
well, by using the STUN credentials as a form of per-session shared
secret. Those credentials are known to the Web application, but
would need to also be known and used by the STUN-receiving element to
be useful.

There also needs to be some mechanism for the browser to verify that
the target of the traffic continues to wish to receive it.
Obviously, some ICE-based mechanism will work here, but it has been
observed that because ICE keepalives are indications, they will not
work here, so some other mechanism is needed.
4.2.2. Masking

Once consent is verified, there still is some concern about misinterpretation attacks as described by Huang et al.[huang-w2sp]. As long as communication is limited to UDP, then this risk is probably limited, thus masking is not required for UDP. I.e., once communications consent has been verified, it is most likely safe to allow the implementation to send arbitrary UDP traffic to the chosen destination, provided that the STUN keepalives continue to succeed. In particular, this is true for the data channel if DTLS is used because DTLS (with the anti-chosen plaintext mechanisms required by TLS 1.1) does not allow the attacker to generate predictable ciphertext. However, with TCP the risk of transparent proxies becomes much more severe. If TCP is to be used, then WebSockets style masking MUST be employed. [Note: current thinking in the RTCWEB WG is not to support TCP and to support SCTP over DTLS, thus removing the need for masking.]

4.2.3. Backward Compatibility

A requirement to use ICE limits compatibility with legacy non-ICE clients. It seems unsafe to completely remove the requirement for some check. All proposed checks have the common feature that the browser sends some message to the candidate traffic recipient and refuses to send other traffic until that message has been replied to. The message/reply pair must be generated in such a way that an attacker who controls the Web application cannot forge them, generally by having the message contain some secret value that must be incorporated (e.g., echoed, hashed into, etc.). Non-ICE candidates for this role (in cases where the legacy endpoint has a public address) include:

- STUN checks without using ICE (i.e., the non-RTC-web endpoint sets up a STUN responder.)
- Use or RTCP as an implicit reachability check.

In the RTCP approach, the RTC-Web endpoint is allowed to send a limited number of RTP packets prior to receiving consent. This allows a short window of attack. In addition, some legacy endpoints do not support RTCP, so this is a much more expensive solution for such endpoints, for which it would likely be easier to implement ICE. For these two reasons, an RTCP-based approach does not seem to address the security issue satisfactorily.

In the STUN approach, the RTC-Web endpoint is able to verify that the recipient is running some kind of STUN endpoint but unless the STUN responder is integrated with the ICE username/password establishment system, the RTC-Web endpoint cannot verify that the recipient
consents to this particular call. This may be an issue if existing
STUN servers are operated at addresses that are not able to handle
bandwidth-based attacks. Thus, this approach does not seem
satisfactory either.

If the systems are tightly integrated (i.e., the STUN endpoint
responds with responses authenticated with ICE credentials) then this
issue does not exist. However, such a design is very close to an
ICE-Lite implementation (indeed, arguably is one). An intermediate
approach would be to have a STUN extension that indicated that one
was responding to RTC-Web checks but not computing integrity checks
based on the ICE credentials. This would allow the use of standalone
STUN servers without the risk of confusing them with legacy STUN
servers. If a non-ICE legacy solution is needed, then this is
probably the best choice.

Once initial consent is verified, we also need to verify continuing
consent, in order to avoid attacks where two people briefly share an
IP (e.g., behind a NAT in an Internet cafe) and the attacker arranges
for a large, unstoppable, traffic flow to the network and then
leaves. The appropriate technologies here are fairly similar to
those for initial consent, though are perhaps weaker since the
threats is less severe.

4.2.4. IP Location Privacy

Note that as soon as the callee sends their ICE candidates, the
callee learns the callee’s IP addresses. The callee’s server
reflexive address reveals a lot of information about the callee’s
location. In order to avoid tracking, implementations may wish to
suppress the start of ICE negotiation until the callee has answered.
In addition, either side may wish to hide their location entirely by
forcing all traffic through a TURN server.

4.3. Communications Security

Finally, we consider a problem familiar from the SIP world:
communications security. For obvious reasons, it MUST be possible
for the communicating parties to establish a channel which is secure
against both message recovery and message modification. (See
[RFC5479] for more details.) This service must be provided for both
data and voice/video. Ideally the same security mechanisms would be
used for both types of content. Technology for providing this
service (for instance, DTLS [RFC4347] and DTLS-SRTP [RFC5763]) is
well understood. However, we must examine this technology to the
RTC-Web context, where the threat model is somewhat different.

In general, it is important to understand that unlike a conventional
SIP proxy, the calling service (i.e., the Web server) controls not only the channel between the communicating endpoints but also the application running on the user’s browser. While in principle it is possible for the browser to cut the calling service out of the loop and directly present trusted information (and perhaps get consent), practice in modern browsers is to avoid this whenever possible. "In-flow" modal dialogs which require the user to consent to specific actions are particularly disfavored as human factors research indicates that unless they are made extremely invasive, users simply agree to them without actually consciously giving consent. [abarth-rtcweb]. Thus, nearly all the UI will necessarily be rendered by the browser but under control of the calling service. This likely includes the peer’s identity information, which, after all, is only meaningful in the context of some calling service.

This limitation does not mean that preventing attack by the calling service is completely hopeless. However, we need to distinguish between two classes of attack:

Retrospective compromise of calling service.
   The calling service is is non-malicious during a call but subsequently is compromised and wishes to attack an older call.

During-call attack by calling service.
   The calling service is compromised during the call it wishes to attack.

Providing security against the former type of attack is practical using the techniques discussed in Section 4.3.1. However, it is extremely difficult to prevent a trusted but malicious calling service from actively attacking a user’s calls, either by mounting a MITM attack or by diverting them entirely. (Note that this attack applies equally to a network attacker if communications to the calling service are not secured.) We discuss some potential approaches and why they are likely to be impractical in Section 4.3.2.

4.3.1. Protecting Against Retrospective Compromise

In a retrospective attack, the calling service was uncompromised during the call, but that an attacker subsequently wants to recover the content of the call. We assume that the attacker has access to the protected media stream as well as having full control of the calling service.

If the calling service has access to the traffic keying material (as in SDES [RFC4568]), then retrospective attack is trivial. This form of attack is particularly serious in the Web context because it is
standard practice in Web services to run extensive logging and monitoring. Thus, it is highly likely that if the traffic key is part of any HTTP request it will be logged somewhere and thus subject to subsequent compromise. It is this consideration that makes an automatic, public key-based key exchange mechanism imperative for RTC-Web (this is a good idea for any communications security system) and this mechanism SHOULD provide perfect forward secrecy (PFS). The signaling channel/calling service can be used to authenticate this mechanism.

In addition, the system MUST NOT provide any APIs to extract either long-term keying material or to directly access any stored traffic keys. Otherwise, an attacker who subsequently compromised the calling service might be able to use those APIs to recover the traffic keys and thus compromise the traffic.

4.3.2. Protecting Against During-Call Attack

Protecting against attacks during a call is a more difficult proposition. Even if the calling service cannot directly access keying material (as recommended in the previous section), it can simply mount a man-in-the-middle attack on the connection, telling Alice that she is calling Bob and Bob that he is calling Alice, while in fact the calling service is acting as a calling bridge and capturing all the traffic. While in theory it is possible to construct techniques which protect against this form of attack, in practice these techniques all require far too much user intervention to be practical, given the user interface constraints described in [abarth-rtcweb].

4.3.2.1. Key Continuity

One natural approach is to use "key continuity". While a malicious calling service can present any identity it chooses to the user, it cannot produce a private key that maps to a given public key. Thus, it is possible for the browser to note a given user’s public key and generate an alarm whenever that user’s key changes. SSH [RFC4251] uses a similar technique. (Note that the need to avoid explicit user consent on every call precludes the browser requiring an immediate manual check of the peer’s key).

Unfortunately, this sort of key continuity mechanism is far less useful in the RTC-Web context. First, much of the virtue of RTC-Web (and any Web application) is that it is not bound to particular piece of client software. Thus, it will be not only possible but routine for a user to use multiple browsers on different computers which will of course have different keying material (SACRED [RFC3760] notwithstanding.) Thus, users will frequently be alerted to key
mismatches which are in fact completely legitimate, with the result that they are trained to simply click through them. As it is known that users routinely will click through far more dire warnings [cranor-wolf], it seems extremely unlikely that any key continuity mechanism will be effective rather than simply annoying.

Moreover, it is trivial to bypass even this kind of mechanism. Recall that unlike the case of SSH, the browser never directly gets the peer's identity from the user. Rather, it is provided by the calling service. Even enabling a mechanism of this type would require an API to allow the calling service to tell the browser "this is a call to user X". All the calling service needs to do to avoid triggering a key continuity warning is to tell the browser that "this is a call to user Y" where Y is close to X. Even if the user actually checks the other side's name (which all available evidence indicates is unlikely), this would require (a) the browser to trusted UI to provide the name and (b) the user to not be fooled by similar appearing names.

4.3.2.2. Short Authentication Strings

ZRTP [RFC6189] uses a "short authentication string" (SAS) which is derived from the key agreement protocol. This SAS is designed to be read over the voice channel and if confirmed by both sides precludes MITM attack. The intention is that the SAS is used once and then key continuity (though a different mechanism from that discussed above) is used thereafter.

Unfortunately, the SAS does not offer a practical solution to the problem of a compromised calling service. "Voice conversion" systems, which modify voice from one speaker to make it sound like another, are an active area of research. These systems are already good enough to fool both automatic recognition systems [farus-conversion] and humans [kain-conversion] in many cases, and are of course likely to improve in future, especially in an environment where the user just wants to get on with the phone call. Thus, even if SAS is effective today, it is likely not to be so for much longer. Moreover, it is possible for an attacker who controls the browser to allow the SAS to succeed and then simulate call failure and reconnect, trusting that the user will not notice that the "no SAS" indicator has been set (which seems likely).

Even were SAS secure if used, it seems exceedingly unlikely that users will actually use it. As discussed above, the browser UI constraints preclude requiring the SAS exchange prior to completing the call and so it must be voluntary; at most the browser will provide some UI indicator that the SAS has not yet been checked. However, it is well-known that when faced with optional mechanisms
such as fingerprints, users simply do not check them [whitten-johnny].

Thus, it is highly unlikely that users will ever perform the SAS exchange.

Once users have checked the SAS once, key continuity is required to
avoid them needing to check it on every call. However, this is
problematic for reasons indicated in Section 4.3.2.1. In principle
it is of course possible to render a different UI element to indicate
that calls are using an unauthenticated set of keying material
(recall that the attacker can just present a slightly different name
so that the attack shows the same UI as a call to a new device or to
someone you haven’t called before) but as a practical matter, users
simply ignore such indicators even in the rather more dire case of
mixed content warnings.

4.3.2.3. Third Party Identity

The conventional approach to providing communications identity has of
course been to have some third party identity system (e.g., PKI) to
authenticate the endpoints. Such mechanisms have proven to be too
cumbersome for use by typical users (and nearly too cumbersome for
administrators). However, a new generation of Web-based identity
providers (BrowserID, Federated Google Login, Facebook Connect,
OAuth, OpenID, WebFinger), has recently been developed and use Web
technologies to provide lightweight (from the user’s perspective)
third-party authenticated transactions. It is possible (see
[I-D.rescorla-rtcweb-generic-idp]) to use systems of this type to
authenticate RTCWEB calls, linking them to existing user notions of
identity (e.g., Facebook adjacencies). Calls which are authenticated
in this fashion are naturally resistant even to active MITM attack by
the calling site.

5. Security Considerations

This entire document is about security.

6. Acknowledgements

Bernard Aboba, Harald Alvestrand, Cullen Jennings, Hadriel Kaplan (S
4.2.1), Matthew Kaufman, Magnus Westerland.

7. References
7.1. Normative References


7.2. Informative References


[I-D.ietf-rtcweb-security-arch]

[I-D.kaufman-rtcweb-security-ui]

[I-D.rescorla-rtcweb-generic-idp]
Rescorla, E., "RTCWEB Generic Identity Provider Interface", draft-rescorla-rtcweb-generic-idp-00 (work in progress), January 2012.


[abarth-rtcweb] Barth, A., "Prompting the user is security failure", RTC-Web Workshop.


[kain-conversion]

[whitten-johnny]

Author’s Address

Eric Rescorla
RTFM, Inc.
2064 Edgewood Drive
Palo Alto, CA  94303
USA

Phone:  +1 650 678 2350
Email:  ekr@rtfm.com
RTCWEB Security Architecture

draft-ietf-rtcweb-security-arch-01

Abstract

The Real-Time Communications on the Web (RTCWEB) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for RTCWEB technology are real-time audio and/or video calls, Web conferencing, and direct data transfer. Unlike most conventional real-time systems (e.g., SIP-based soft phones) RTCWEB communications are directly controlled by some Web server, which poses new security challenges. For instance, a Web browser might expose a JavaScript API which allows a server to place a video call. Unrestricted access to such an API would allow any site which a user visited to "bug" a user's computer, capturing any activity which passed in front of their camera. [I-D.ietf-rtcweb-security] defines the RTCWEB threat model. This document defines an architecture which provides security within that threat model.

Legal

THIS DOCUMENT AND THE INFORMATION CONTAINED THEREIN ARE PROVIDED ON AN "AS IS" BASIS AND THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST, AND THE INTERNET ENGINEERING TASK FORCE, DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months

Rescorla Expires September 13, 2012 [Page 1]
and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 13, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.
Table of Contents

1. Introduction .................................................. 4
2. Terminology .................................................. 4
3. Trust Model .................................................. 4
   3.1. Authenticated Entities .................................. 5
   3.2. Unauthenticated Entities ................................. 5
4. Overview ..................................................... 6
   4.1. Initial Signaling ........................................ 7
   4.2. Media Consent Verification ............................... 9
   4.3. DTLS Handshake .......................................... 10
   4.4. Communications and Consent Freshness .................. 10
5. Detailed Technical Description ............................... 10
   5.1. Origin and Web Security Issues ......................... 10
   5.2. Device Permissions Model ............................... 11
   5.3. Communications Consent ................................ 12
   5.4. IP Location Privacy .................................... 13
   5.5. Communications Security ............................... 13
   5.6. Web-Based Peer Authentication ......................... 15
6. Security Considerations ...................................... 16
   6.1. Communications Security ............................... 16
   6.2. Privacy ................................................ 16
   6.3. Denial of Service ...................................... 17
7. Acknowledgements ............................................ 18
8. References .................................................. 18
   8.1. Normative References ................................. 18
   8.2. Informative References ............................... 19
Author’s Address ................................................. 19
1. Introduction

The Real-Time Communications on the Web (RTCWEB) working group is tasked with standardizing protocols for real-time communications between Web browsers. The major use cases for RTCWEB technology are real-time audio and/or video calls, Web conferencing, and direct data transfer. Unlike most conventional real-time systems, (e.g., SIP-based RFC3261) soft phones) RTCWEB communications are directly controlled by some Web server, as shown in Figure 1.

![Simple RTCWEB System Diagram]

Figure 1: A simple RTCWEB system

This system presents a number of new security challenges, which are analyzed in [I-D.ietf-rtcweb-security]. This document describes a security architecture for RTCWEB which addresses the threats and requirements described in that document.

2. 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].

3. Trust Model

The basic assumption of this architecture is that network resources exist in a hierarchy of trust, rooted in the browser, which serves as the user’s TRUSTED COMPUTING BASE (TCB). Any security property which
the user wishes to have enforced must be ultimately guaranteed by the browser (or transitively by some property the browser verifies). Conversely, if the browser is compromised, then no security guarantees are possible. Note that there are cases (e.g., Internet kiosks) where the user can’t really trust the browser that much. In these cases, the level of security provided is limited by how much they trust the browser.

Optimally, we would not rely on trust in any entities other than the browser. However, this is unfortunately not possible if we wish to have a functional system. Other network elements fall into two categories: those which can be authenticated by the browser and thus are partly trusted—though to the minimum extent necessary—and those which cannot be authenticated and thus are untrusted. This is a natural extension of the end-to-end principle.

3.1. Authenticated Entities

There are two major classes of authenticated entities in the system:

- **Calling services:** Web sites whose origin we can verify (optimally via HTTPS).
- **Other users:** RTCWEB peers whose origin we can verify cryptographically (optimally via DTLS-SRTP).

Note that merely being authenticated does not make these entities trusted. For instance, just because we can verify that https://www.evil.org/ is owned by Dr. Evil does not mean that we can trust Dr. Evil to access our camera and microphone. However, it gives the user an opportunity to determine whether he wishes to trust Dr. Evil or not; after all, if he desires to contact Dr. Evil (perhaps to arrange for ransom payment), it’s safe to temporarily give him access to the camera and microphone for the purpose of the call, but he doesn’t want Dr. Evil to be able to access his camera and microphone other than during the call. The point here is that we must first identify other elements before we can determine whether and how much to trust them.

It’s also worth noting that there are settings where authentication is non-cryptographic, such as other machines behind a firewall. Naturally, the level of trust one can have in identities verified in this way depends on how strong the topology enforcement is.

3.2. Unauthenticated Entities

Other than the above entities, we are not generally able to identify other network elements, thus we cannot trust them. This does not mean that it is not possible to have any interaction with them, but
it means that we must assume that they will behave maliciously and
design a system which is secure even if they do so.

4. Overview

This section describes a typical RTCWeb session and shows how the
various security elements interact and what guarantees are provided
to the user. The example in this section is a "best case" scenario
in which we provide the maximal amount of user authentication and
media privacy with the minimal level of trust in the calling service.
Simpler versions with lower levels of security are also possible and
are noted in the text where applicable. It’s also important to
recognize the tension between security (or performance) and privacy.
The example shown here is aimed towards settings where we are more
cconcerned about secure calling than about privacy, but as we shall
see, there are settings where one might wish to make different
tradeoffs--this architecture is still compatible with those settings.

For the purposes of this example, we assume the topology shown in the
figure below. This topology is derived from the topology shown in
Figure 1, but separates Alice and Bob’s identities from the process
of signaling. Specifically, Alice and Bob have relationships with
some Identity Provider (IdP) that supports a protocol such OpenID or
BrowserID) that can be used to attest to their identity. This
separation isn’t particularly important in "closed world" cases where
Alice and Bob are users on the same social network and have
identities based on that network. However, there are important
settings where that is not the case, such as federation (calls from
one network to another) and calling on untrusted sites, such as where
two users who have a relationship via a given social network want to
call each other on another, untrusted, site, such as a poker site.
4.1. Initial Signaling

Alice and Bob are both users of a common calling service; they both have approved the calling service to make calls (we defer the discussion of device access permissions till later). They are both connected to the calling service via HTTPS and so know the origin with some level of confidence. They also have accounts with some identity provider. This sort of identity service is becoming increasingly common in the Web environment in technologies such (BrowserID, Federated Google Login, Facebook Connect, OAuth, OpenID, WebFinger), and is often provided as a side effect service of your ordinary accounts with some service. In this example, we show Alice and Bob using a separate identity service, though they may actually be using the same identity service as calling service or have no identity service at all.

Alice is logged onto the calling service and decides to call Bob. She can see from the calling service that he is online and the calling service presents a JS UI in the form of a button next to Bob’s name.
which says "Call". Alice clicks the button, which initiates a JS
callback that instantiates a PeerConnection object. This does not
require a security check: JS from any origin is allowed to get this
far.

Once the PeerConnection is created, the calling service JS needs to
set up some media. Because this is an audio/video call, it creates
two MediaStreams, one connected to an audio input and one connected
to a video input. At this point the first security check is
required: untrusted origins are not allowed to access the camera and
microphone. In this case, because Alice is a long-term user of the
calling service, she has made a permissions grant (i.e., a setting in
the browser) to allow the calling service to access her camera and
microphone any time it wants. The browser checks this setting when
the camera and microphone requests are made and thus allows them.

In the current W3C API, once some streams have been added, Alice’s
browser + JS generates a signaling message. The format of this data is
currently undefined. It may be a complete message as defined by ROAP
[I-D.jennings-rtcweb-signaling] or separate media description and
transport messages as defined in [I-D.ietf-rtcweb-jsep] or may be
assembled piecemeal by the JS. In either case, it will contain:

- Media channel information
- ICE candidates
- A fingerprint attribute binding the communication to Alice’s
  public key [RFC5763]

[Note that it is currently unclear where JSEP will eventually put
this information, in the SDP or in the transport info.] Prior to
sending out the signaling message, the PeerConnection code contacts
the identity service and obtains an assertion binding Alice’s
identity to her fingerprint. The exact details depend on the
identity service (though as discussed in
[I-D.rescorla-rtcweb-generic-idp] PeerConnection can be agnostic to
them), but for now it’s easiest to think of as a BrowserID assertion.
The assertion may bind other information to the identity besides the
fingerprint, but at minimum it needs to bind the fingerprint.

This message is sent to the signaling server, e.g., by XMLHttpRequest
[XmLHttpReques] or by WebSockets [RFC6455] The signaling server
processes the message from Alice’s browser, determines that this is a
call to Bob and sends a signaling message to Bob’s browser (again,
the format is currently undefined). The JS on Bob’s browser
processes it, and alerts Bob to the incoming call and to Alice’s
identity. In this case, Alice has provided an identity assertion and
so Bob’s browser contacts Alice’s identity provider (again, this is
done in a generic way so the browser has no specific knowledge of the
IdP) to verify the assertion. This allows the browser to display a trusted element indicating that a call is coming in from Alice. If Alice is in Bob’s address book, then this interface might also include her real name, a picture, etc. The calling site will also provide some user interface element (e.g., a button) to allow Bob to answer the call, though this is most likely not part of the trusted UI.

If Bob agrees [I am ignoring early media for now], a PeerConnection is instantiated with the message from Alice’s side. Then, a similar process occurs as on Alice’s browser: Bob’s browser verifies that the calling service is approved, the media streams are created, and a return signaling message containing media information, ICE candidates, and a fingerprint is sent back to Alice via the signaling service. If Bob has a relationship with an IdP, the message will also come with an identity assertion.

At this point, Alice and Bob each know that the other party wants to have a secure call with them. Based purely on the interface provided by the signaling server, they know that the signaling server claims that the call is from Alice to Bob. Because the far end sent an identity assertion along with their message, they know that this is verifiable from the IdP as well. Of course, the call works perfectly well if either Alice or Bob doesn’t have a relationship with an IdP; they just get a lower level of assurance. Moreover, Alice might wish to make an anonymous call through an anonymous calling site, in which case she would of course just not provide any identity assertion and the calling site would mask her identity from Bob.

4.2. Media Consent Verification

As described in ([I-D.ietf-rtcweb-security]; Section 4.2) This proposal specifies that media consent verification be performed via ICE. Thus, Alice and Bob perform ICE checks with each other. At the completion of these checks, they are ready to send non-ICE data.

At this point, Alice knows that (a) Bob (assuming he is verified via his IdP) or someone else who the signaling service is claiming is Bob is willing to exchange traffic with her and (b) that either Bob is at the IP address which she has verified via ICE or there is an attacker who is on-path to that IP address detouring the traffic. Note that it is not possible for an attacker who is on-path but not attached to the signaling service to spoof these checks because they do not have the ICE credentials. Bob’s security guarantees with respect to Alice are the converse of this.
4.3. DTLS Handshake

Once the ICE checks have completed [more specifically, once some ICE checks have completed], Alice and Bob can set up a secure channel. This is performed via DTLS [RFC4347] (for the data channel) and DTLS-SRTP [RFC5763] for the media channel. Specifically, Alice and Bob perform a DTLS handshake on every channel which has been established by ICE. The total number of channels depends on the amount of muxing; in the most likely case we are using both RTP/RTCP mux and muxing multiple media streams on the same channel, in which case there is only one DTLS handshake. Once the DTLS handshake has completed, the keys are exported [RFC5705] and used to key SRTP for the media channels.

At this point, Alice and Bob know that they share a set of secure data and/or media channels with keys which are not known to any third-party attacker. If Alice and Bob authenticated via their IdPs, then they also know that the signaling service is not attacking them. Even if they do not use an IdP, as long as they have minimal trust in the signaling service not to perform a man-in-the-middle attack, they know that their communications are secure against the signaling service as well.

4.4. Communications and Consent Freshness

From a security perspective, everything from here on in is a little anticlimactic: Alice and Bob exchange data protected by the keys negotiated by DTLS. Because of the security guarantees discussed in the previous sections, they know that the communications are encrypted and authenticated.

The one remaining security property we need to establish is "consent freshness", i.e., allowing Alice to verify that Bob is still prepared to receive her communications. ICE specifies periodic STUN keepalives but only if media is not flowing. Because the consent issue is more difficult here, we require RTCWeb implementations to periodically send keepalives. If a keepalive fails and no new ICE channels can be established, then the session is terminated.

5. Detailed Technical Description

5.1. Origin and Web Security Issues

The basic unit of permissions for RTCWEB is the origin [RFC6454]. Because the security of the origin depends on being able to authenticate content from that origin, the origin can only be securely established if data is transferred over HTTPS [RFC2818].
Thus, clients MUST treat HTTP and HTTPS origins as different permissions domains. [Note: this follows directly from the origin security model and is stated here merely for clarity.]

Many web browsers currently forbid by default any active mixed content on HTTPS pages. I.e., when JS is loaded from an HTTP origin onto an HTTPS page, an error is displayed and the content is not executed unless the user overrides the error. Any browser which enforces such a policy will also not permit access to RTCWEB functionality from mixed content pages. It is RECOMMENDED that browsers which allow active mixed content nevertheless disable RTCWEB functionality in mixed content settings. [[ OPEN ISSUE: Should this be a 2119 MUST? It’s not clear what set of conditions would make this OK, other than that browser manufacturers have traditionally been permissive here here here.]] Note that it is possible for a page which was not mixed content to become mixed content during the duration of the call. Implementations MAY choose to terminate the call or display a warning at that point, but it is also permissible to ignore this condition. This is a deliberate implementation complexity versus security tradeoff.

5.2. Device Permissions Model

Implementations MUST obtain explicit user consent prior to providing access to the camera and/or microphone. Implementations MUST at minimum support the following two permissions models:

- Requests for one-time camera/microphone access.
- Requests for permanent access.

In addition, they SHOULD support requests for access to a single communicating peer. E.g., “Call customerservice@ford.com”. Browsers servicing such requests SHOULD clearly indicate that identity to the user when asking for permission.

API Requirement: The API MUST provide a mechanism for the requesting JS to indicate which of these forms of permissions it is requesting. This allows the client to know what sort of user interface experience to provide. In particular, browsers might display a non-invasive door hanger ("some features of this site may not work..." when asking for long-term permissions) but a more invasive UI ("here is your own video") for single-call permissions. The API MAY grant weaker permissions than the JS asked for if the user chooses to authorize only those permissions, but if it intends to grant stronger ones it SHOULD display the appropriate UI for those permissions and MUST clearly indicate what permissions are being requested.
API Requirement: The API MUST provide a mechanism for the requesting JS to relinquish the ability to see or modify the media (e.g., via MediaStream.record()). Combined with secure authentication of the communicating peer, this allows a user to be sure that the calling site is not accessing or modifying their conversion.

UI Requirement: The UI MUST clearly indicate when the user’s camera and microphone are in use. This indication MUST NOT be suppressible by the JS and MUST clearly indicate how to terminate a call, and provide a UI means to immediately stop camera/microphone input without the JS being able to prevent it.

UI Requirement: If the UI indication of camera/microphone use are displayed in the browser such that minimizing the browser window would hide the indication, or the JS creating an overlapping window would hide the indication, then the browser SHOULD stop camera and microphone input. [Note: this may not be necessary in systems that are non-windows-based but that have good notifications support, such as phones.]

Clients MAY permit the formation of data channels without any direct user approval. Because sites can always tunnel data through the server, further restrictions on the data channel do not provide any additional security. (though see Section 5.3 for a related issue).

Implementations which support some form of direct user authentication SHOULD also provide a policy by which a user can authorize calls only to specific counterparties. Specifically, the implementation SHOULD provide the following interfaces/controls:

o Allow future calls to this verified user.
o Allow future calls to any verified user who is in my system address book (this only works with address book integration, of course).

Implementations SHOULD also provide a different user interface indication when calls are in progress to users whose identities are directly verifiable. Section 5.5 provides more on this.

5.3. Communications Consent

Browser client implementations of RTCWEB MUST implement ICE. Server gateway implementations which operate only at public IP addresses may implement ICE-Lite.

Browser implementations MUST verify reachability via ICE prior to sending any non-ICE packets to a given destination. Implementations MUST NOT provide the ICE transaction ID to JavaScript. [Note: this
document takes no position on the split between ICE in JS and ICE in the browser. The above text is written the way it is for editorial convenience and will be modified appropriately if the WG decides on ICE in the JS.

Implementations MUST send keepalives no less frequently than every 30 seconds regardless of whether traffic is flowing or not. If a keepalive fails then the implementation MUST either attempt to find a new valid path via ICE or terminate media for that ICE component. Note that ICE [RFC5245]; Section 10 keepalives use STUN Binding Indications which are one-way and therefore not sufficient. Instead, the consent freshness mechanism [I-D.muthu-behave-consent-freshness] MUST be used.

5.4. IP Location Privacy

A side effect of the default ICE behavior is that the peer learns one’s IP address, which leaks large amounts of location information, especially for mobile devices. This has negative privacy consequences in some circumstances. The following two API requirements are intended to mitigate this issue:

API Requirement: The API MUST provide a mechanism to suppress ICE negotiation (though perhaps to allow candidate gathering) until the user has decided to answer the call [note: determining when the call has been answered is a question for the JS.] This enables a user to prevent a peer from learning their IP address if they elect not to answer a call and also from learning whether the user is online.

API Requirement: The API MUST provide a mechanism for the calling application to indicate that only TURN candidates are to be used. This prevents the peer from learning one’s IP address at all. The API MUST provide a mechanism for the calling application to reconfigure an existing call to add non-TURN candidates. Taken together, these requirements allow ICE negotiation to start immediately on incoming call notification, thus reducing post-dial delay, but also to avoid disclosing the user’s IP address until they have decided to answer.

5.5. Communications Security

Implementations MUST implement DTLS [RFC4347] and DTLS-SRTP [RFC5763][RFC5764]. All data channels MUST be secured via DTLS. DTLS-SRTP MUST be offered for every media channel and MUST be the default; i.e., if an implementation receives an offer for DTLS-SRTP and SDES and/or plain RTP, DTLS-SRTP MUST be selected.
[OPEN ISSUE: What should the settings be here? MUST?]
Implementations MAY support SDES and RTP for media traffic for backward compatibility purposes.

API Requirement: The API MUST provide a mechanism to indicate that a fresh DTLS key pair is to be generated for a specific call. This is intended to allow for unlinkability. Note that there are also settings where it is attractive to use the same keying material repeatedly, especially those with key continuity-based authentication.

API Requirement: The API MUST provide a mechanism to indicate that a fresh DTLS key pair is to be generated for a specific call. This is intended to allow for unlinkability.

API Requirement: When DTLS-SRTP is used, the API MUST NOT permit the JS to obtain the negotiated keying material. This requirement preserves the end-to-end security of the media.

UI Requirements: A user-oriented client MUST provide an "inspector" interface which allows the user to determine the security characteristics of the media. [largely derived from I-D.kaufman-rtcweb-security-ui]
The following properties SHOULD be displayed "up-front" in the browser chrome, i.e., without requiring the user to ask for them:

* A client MUST provide a user interface through which a user may determine the security characteristics for currently-displayed audio and video stream(s)
* A client MUST provide a user interface through which a user may determine the security characteristics for transmissions of their microphone audio and camera video.
* The "security characteristics" MUST include an indication as to whether or not the transmission is cryptographically protected and whether that protection is based on a key that was delivered out-of-band (from a server) or was generated as a result of a pairwise negotiation.
* If the far endpoint was directly verified (see Section 5.6) the "security characteristics" MUST include the verified information.
The following properties are more likely to require some "drill-down" from the user:

* If the transmission is cryptographically protected, the algorithms in use (For example: "AES-CBC" or "Null Cipher".)
* If the transmission is cryptographically protected, the "security characteristics" MUST indicate whether PFS is provided.
If the transmission is cryptographically protected via an end-to-end mechanism the "security characteristics" MUST include some mechanism to allow an out-of-band verification of the peer, such as a certificate fingerprint or an SAS.

5.6. Web-Based Peer Authentication

In a number of cases, it is desirable for the endpoint (i.e., the browser) to be able to directly identity the endpoint on the other side without trusting only the signaling service to which they are connected. For instance, users may be making a call via a federated system where they wish to get direct authentication of the other side. Alternately, they may be making a call on a site which they minimally trust (such as a poker site) but to someone who has an identity on a site they do trust (such as a social network.)

Recently, a number of Web-based identity technologies (OAuth, BrowserID, Facebook Connect), etc. have been developed. While the details vary, what these technologies share is that they have a Web-based (i.e., HTTP/HTTPS identity provider) which attests to your identity. For instance, if I have an account at example.org, I could use the example.org identity provider to prove to others that I was alice@example.org. The development of these technologies allows us to separate calling from identity provision: I could call you on Poker Galaxy but identify myself as alice@example.org.

Whatever the underlying technology, the general principle is that the party which is being authenticated is NOT the signaling site but rather the user (and their browser). Similarly, the relying party is the browser and not the signaling site. Thus, the browser MUST securely generate the input to the IdP assertion process and MUST securely display the results of the verification process to the user in a way which cannot be imitated by the calling site.

In order to make this work, we must standardize the following items:

- The precise information from the signaling message that must be cryptographically bound to the user’s identity. At minimum this MUST be the fingerprint, but we may choose to add other information as the signaling protocol firms up. This will be defined in a future version of this document.
- The interface to the IdP. [I-D.rescorla-rtcweb-generic-idp] specifies a specific protocol mechanism which allows the use of any identity protocol without requiring specific further protocol support in the browser.
- The JavaScript interfaces which the calling application can use to specify the IdP to use to generate assertions and to discover what assertions were received. These interfaces should be defined in
6. Security Considerations

Much of the security analysis of this problem is contained in [I-D.ietf-rtcweb-security] or in the discussion of the particular issues above. In order to avoid repetition, this section focuses on (a) residual threats that are not addressed by this document and (b) threats produced by failure/misbehavior of one of the components in the system.

6.1. Communications Security

While this document favors DTLS-SRTP, it permits a variety of communications security mechanisms and thus the level of communications security actually provided varies considerably. Any pair of implementations which have multiple security mechanisms in common are subject to being downgraded to the weakest of those common mechanisms by any attacker who can modify the signaling traffic. If communications are over HTTP, this means any on-path attacker. If communications are over HTTPS, this means the signaling server. Implementations which wish to avoid downgrade attack should only offer the strongest available mechanism, which is DTLS/DTLS-SRTP. Note that the implication of this choice will be that interop to non-DTLS-SRTP devices will need to happen through gateways.

Even if only DTLS/DTLS-SRTP are used, the signaling server can potentially mount a man-in-the-middle attack unless implementations have some mechanism for independently verifying keys. The UI requirements in Section 5.5 are designed to provide such a mechanism for motivated/security conscious users, but are not suitable for general use. The identity service mechanisms in Section 5.6 are more suitable for general use. Note, however, that a malicious signaling service can strip off any such identity assertions, though it cannot forge new ones.

6.2. Privacy

The requirements in this document are intended to allow:

- Users to participate in calls without revealing their location.
- Potential callees to avoid revealing their location and even presence status prior to agreeing to answer a call.

However, these privacy protections come at a performance cost in terms of using TURN relays and, in the latter case, delaying ICE. Sites SHOULD make users aware of these tradeoffs.
Note that the protections provided here assume a non-malicious calling service. As the calling service always knows the users status and (absent the use of a technology like Tor) their IP address, they can violate the users privacy at will. Users who wish privacy against the calling sites they are using must use separate privacy enhancing technologies such as Tor. Combined RTCWEB/Tor implementations SHOULD arrange to route the media as well as the signaling through Tor. [Currently this will produce very suboptimal performance.]

6.3. Denial of Service

The consent mechanisms described in this document are intended to mitigate denial of service attacks in which an attacker uses clients to send large amounts of traffic to a victim without the consent of the victim. While these mechanisms are sufficient to protect victims who have not implemented RTCWEB at all, RTCWEB implementations need to be more careful.

Consider the case of a call center which accepts calls via RTCWeb. An attacker proxies the call center’s front-end and arranges for multiple clients to initiate calls to the call center. Note that this requires user consent in many cases but because the data channel does not need consent, he can use that directly. Since ICE will complete, browsers can then be induced to send large amounts of data to the victim call center if it supports the data channel at all. Preventing this attack requires that automated RTCWEB implementations implement sensible flow control and have the ability to triage out (i.e., stop responding to ICE probes on) calls which are behaving badly, and especially to be prepared to remotely throttle the data channel in the absence of plausible audio and video (which the attacker cannot control).

Another related attack is for the signaling service to swap the ICE candidates for the audio and video streams, thus forcing a browser to send video to the sink that the other victim expects will contain audio (perhaps it is only expecting audio!) potentially causing overload. Muxing multiple media flows over a single transport makes it harder to individually suppress a single flow by denying ICE keepalives. Media-level (RTCP) mechanisms must be used in this case.

[TODO: Write up Magnus’s ICE forking attack when we get some clarity on it.]

Note that attacks based on confusing one end or the other about consent are possible primarily even in the face of the third-party identity mechanism as long as major parts of the signaling messages are not signed. On the other hand, signing the entire message
severely restricts the capabilities of the calling application, so there are difficult tradeoffs here.

7. Acknowledgements

Bernard Aboba, Harald Alvestrand, Cullen Jennings, Hadriel Kaplan, Matthew Kaufman, Magnus Westerland.

8. References

8.1. Normative References


[RFC6454] Barth, A., "The Web Origin Concept", RFC 6454,
8.2. Informative References

[I-D.ietf-rtcweb-jsep]

[I-D.jennings-rtcweb-signaling]

[I-D.kaufman-rtcweb-security-ui]

[I-D.rescorla-rtcweb-generic-idp]
Rescorla, E., "RTCWeb Generic Identity Provider Interface", draft-rescorla-rtcweb-generic-idp-00 (work in progress), January 2012.


[XmlHttpRequest]
van Kesteren, A., "XMLHttpRequest Level 2".
Author’s Address

Eric Rescorla
RTFM, Inc.
2064 Edgewood Drive
Palo Alto, CA 94303
USA

Phone: +1 650 678 2350
Email: ekr@rtfm.com
Abstract

This document is the specification of the syntax and semantics of the Uniform Resource Identifier (URI) scheme for the Session Traversal Utilities for NAT (STUN) protocol.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 13, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ................................................. 3
2. Terminology .................................................. 3
3. Syntax of a STUN or STUNS URI ............................... 4
   3.1. URI Scheme Syntax ....................................... 4
   3.2. URI Scheme Semantics ................................... 4
4. Security Considerations ...................................... 5
5. IANA Considerations ......................................... 5
   5.1. STUN URI Registration ................................... 5
   5.2. STUNS URI Registration ................................. 6
6. Acknowledgements ............................................ 6
7. References .................................................... 7
   7.1. Normative References ................................... 7
   7.2. Informative References ................................ 7
Appendix A. Examples ........................................... 7
Authors’ Addresses ............................................. 8
1. Introduction

This document specifies the syntax and semantics of the Uniform Resource Identifier (URI) scheme for the Session Traversal Utilities for NAT (STUN) protocol.

STUN is a protocol that serves as a tool for other protocols in dealing with Network Address Translator (NAT) traversal. It can be used by an endpoint to determine the IP address and port allocated to it by a NAT, to perform connectivity checks between two endpoints, and used as a keepalive protocol to maintain NAT bindings. RFC 5389 [RFC5389] defines the specifics of the STUN protocol.

The 'stun/stuns' URI scheme is used to designate a standalone STUN server or any Internet host performing the operations of a STUN server in the context of STUN usages (Section 14 RFC 5389 [RFC5389]). With the advent of standards such as WEBRTC [WEBRTC], we anticipate a plethora of endpoints and web applications to be able to identify and communicate with such a STUN server to carry out the STUN protocol. This also implies those endpoints and/or applications to be provisioned with appropriate configuration required to identify the STUN server. Having an inconsistent syntax has its drawbacks and can result in non-interoperable solutions. It can result in solutions that are ambiguous and have implementation limitations on the different aspects of the syntax and alike. The 'stun/stuns' URI scheme helps alleviate most of these issues by providing a consistent way to describe, configure and exchange the information identifying a STUN server. This would also prevent the shortcomings inherent with encoding similar information in non-uniform syntaxes such as the ones proposed in the WEBRTC Standards [WEBRTC], for example.

A reference implementation [REF-IMPL] is available.

2. 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 [RFC2119].

"SHOULD", "SHOULD NOT", "RECOMMENDED", and "NOT RECOMMENDED" are appropriate when valid exceptions to a general requirement are known to exist or appear to exist, and it is infeasible or impractical to enumerate all of them. However, they should not be interpreted as permitting implementors to fail to implement the general requirement when such failure would result in interoperability failure.
3. Syntax of a STUN or STUNS URI

3.1. URI Scheme Syntax

The "stun" URI takes the following form (the example below is non-normative):

stun:<stun-host>:<stun-port>
stuns:<stun-host>:<stun-port>

Note that the <port> part and the preceding ":" (colon) character, is OPTIONAL.

A STUN/STUNS URI has the following formal ABNF syntax [RFC5234]:

```
stunURI       = scheme "":" stun-host [ ":":" stun-port ]
scheme        = "stun" / "stuns"
stun-host     = IP-literal / IPv4address / reg-name
stun-port     = *DIGIT
IP-literal    = "[" ( IPv6address / IPvFuture ) "]"
IPvFuture     = "v" 1*HEXDIG "." 1*( unreserved / sub-delims / ":" )
IPv6address   = 6( h16 ":" ) ls32
                / "::" 5( h16 ":" ) ls32
                / [ h16 ] ":" 4( h16 ":" ) ls32
                / [ *1( h16 ":" ) h16 ] ":" 3( h16 ":" ) ls32
                / [ *2( h16 ":" ) h16 ] ":" 2( h16 ":" ) ls32
                / [ *3( h16 ":" ) h16 ] ":" h16 ": " ls32
                / [ *4( h16 ":" ) h16 ] ":" h16 ls32
                / [ *5( h16 ":" ) h16 ] ":" h16
                / [ *6( h16 ":" ) h16 ] ":" h16
h16           = 1*4HEXDIG
ls32          = ( h16 "":" h16 ) / IPv4address
IPv4address   = dec-octet "." dec-octet "." dec-octet "." dec-octet
dec-octet     = DIGIT ; 0-9
                / %x31-39 DIGIT ; 10-99
                / "1" 2DIGIT ; 100-199
                / "2" %x30-34 DIGIT ; 200-249
                / "25" %x30-35 ; 250-255
reg-name      = *( unreserved / pct-encoded / sub-delims )
```

<unreserved>, <sub-delims>, and <pct-encoded> are specified in [RFC3986]. The core rules <DIGIT> and <HEXDIGIT> are used as described in Appendix B of RFC 5234 [RFC5234].

3.2. URI Scheme Semantics

The STUN protocol supports sending messages over UDP, TCP or TLS-over-TCP. The "stuns" URI scheme SHALL be used when STUN is run over
TLS-over-TCP (or in the future DTLS-over-UDP) and the "stun" scheme SHALL be used otherwise.

The required <stun-host> part of the "stun" URI denotes the STUN server host.

For the optional DNS Discovery procedure mentioned in the Section 9 of RFC5389, "stun" URI scheme implies UDP as the transport protocol for SRV lookup and "stuns" URI scheme indicates TCP as the transport protocol.

The <stun-port> part, if present, denotes the port on which the STUN server is awaiting connection requests. If it is absent, the default port is 3478 for both UDP and TCP and 5349 for STUN over TLS as per Section 9 of RFC 5389 [RFC5389].

4. Security Considerations

The "stun" and "stuns" URI schemes do not introduce any specific security issues beyond the security considerations discussed in [RFC3986].

5. IANA Considerations

This section contains the registration information for the "stun" and "stuns" URI Schemes (in accordance with [RFC4395]).

5.1. STUN URI Registration

URI scheme name: stun

Status: permanent

URI scheme syntax: See Section 3.1.

URI scheme semantics: See Section 3.2.

Encoding considerations: There are no encoding considerations beyond those in [RFC3986].

Applications/protocols that use this URI scheme name:

The "stun" URI scheme is intended to be used by applications that might need access to a STUN server.

Interoperability considerations: N/A
5.2. STUNS URI Registration

URI scheme name: stuns

Status: permanent

URI scheme syntax: See Section 3.1.

URI scheme semantics: See Section 3.2.

Encoding considerations: There are no encoding considerations beyond those in [RFC3986].

Applications/protocols that use this URI scheme name:

The "stuns" URI scheme is intended to be used by applications that might need access to a STUN server over a secure connection.

Interoperability considerations: N/A


Contact: Suhas Nandakumar <snandaku@cisco.com>

Author/Change controller: The IESG

References: RFCXXXX

[[NOTE TO RFC EDITOR: Please change XXXX to the number assigned to this specification, and remove this paragraph on publication.]]

6. Acknowledgements

Many thanks to Cullen Jennings for his detailed review and thoughtful comments on this document.
This document was written with the xml2rfc tool described in [RFC2629].

7. References

7.1. Normative References


7.2. Informative References


[WEBRTC] W3C, "WebRTC 1.0: Real-time Communication Between Browsers".


[REF-IMPL] Petit-Huguenin, MPH., "Reference Implementation of STUN URI parser".


Appendix A. Examples

Table 1 shows examples for 'stun/stuns'uri scheme. For all these examples, the <host> component is populated with "example.org".
Authors’ Addresses

Suhas Nandakumar
Cisco Systems
170 West Tasman Drive
San Jose, CA  95134
US
Email: snandaku@cisco.com

Gonzalo Salgueiro
Cisco Systems
7200-12 Kit Creek Road
Research Triangle Park, NC  27709
US
Email: gsalguei@cisco.com

Paul E. Jones
Cisco Systems
7025 Kit Creek Road
Research Triangle Park, NC  27709
US
Email: paulej@packetizer.com

Marc Petit-Huguenin
Unaffiliated
Email: petithug@acm.org
Traversal Using Relays around NAT (TURN) Uniform Resource Identifiers
draft-petithuguenin-behave-turn-uris-01

Abstract

This document specifies the syntax of Uniform Resource Identifier (URI) schemes for the Traversal Using Relays around NAT (TURN) protocol. It defines two URI schemes that can be used to provision the configuration values needed by the resolution mechanism defined in [RFC5928].

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79. This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 13, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of
publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction ...................................................... 3
2. Terminology ....................................................... 3
3. Syntax of a TURN or TURNS URI ................................. 4
   3.1. URI Scheme Syntax ........................................... 4
   3.2. URI Scheme Semantics .................................... 5
4. Security Considerations ........................................... 5
5. IANA Considerations ............................................... 5
   5.1. TURN URI Registration ................................... 5
   5.2. TURNS URI Registration .................................... 6
6. Acknowledgements ................................................ 7
7. References ........................................................ 7
   7.1. Normative References ....................................... 7
   7.2. Informative References ..................................... 8
Appendix A. Examples ................................................ 8
Appendix B. Release notes ............................................ 8
   B.1. Modifications between
        petithuguenin-behave-turn-uris-01 and
        petithuguenin-behave-turn-uris-00  ...................... 9
   B.2. Merge of draft-nandakumar-rtcweb-turn-uri-00 and
        draft-petithuguenin-behave-turn-uri-bis-05 .............. 9
   B.3. Modifications between petithuguenin-05 and
        petithuguenin-04 ...................................... 9
   B.4. Modifications between petithuguenin-04 and
        petithuguenin-03 ...................................... 9
   B.5. Modifications between petithuguenin-03 and
        petithuguenin-02 ...................................... 9
   B.6. Modifications between petithuguenin-02 and
        petithuguenin-01 ...................................... 9
   B.7. Modifications between petithuguenin-01 and
        petithuguenin-00 ...................................... 9
   B.8. Design Notes .............................................. 9
Authors’ Addresses .................................................. 10
1. Introduction

This document specifies the syntax and semantics of the Uniform Resource Identifier (URI) scheme for the Traversal Using Relays around NAT (TURN) protocol.

The TURN protocol is a specification allowing hosts behind NAT to control the operation of a relay server. The relay server allows hosts to exchange packets with its peers. The peers themselves may also be behind NATs. RFC 5766 [RFC5766] defines the specifics of the TURN protocol.

The "turn/turns" URI scheme is used to designate a TURN server (also known as a relay) on Internet hosts accessible using the TURN protocol. With the advent of standards such as [WEBRTC], we anticipate a plethora of endpoints and web applications to be able to identify and communicate with such a TURN server to carry out the TURN protocol. This also implies those endpoints and/or applications to be provisioned with appropriate configuration required to identify the TURN server. Having an inconsistent syntax has its drawbacks and can result in non-interoperable solutions. It can result in solutions that are ambiguous and have implementation limitations on the different aspects of the syntax and alike. The "turn/turns" URI scheme helps alleviate most of these issues by providing a consistent way to describe, configure and exchange the information identifying a TURN server. This would also prevent the shortcomings inherent with encoding similar information in non-uniform syntaxes such as the ones proposed in [WEBRTC], for example.

[RFC5928] defines a resolution mechanism to convert a secure flag, a host name or IP address, an eventually empty port, and an eventually empty transport to a list of IP address, port, and TURN transport tuples.

To simplify the provisioning of TURN clients, this document defines a TURN and a TURNS URI scheme that can carry the four components needed for the resolution mechanism.

A reference implementation [REF-IMPL] is available.

2. 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 [RFC2119].

"SHOULD", "SHOULD NOT", "RECOMMENDED", and "NOT RECOMMENDED" are
appropriate when valid exceptions to a general requirement are known
to exist or appear to exist, and it is infeasible or impractical to
enumerate all of them. However, they should not be interpreted as
permitting implementors to fail to implement the general requirement
when such failure would result in interoperability failure.

3. Syntax of a TURN or TURNS URI

3.1. URI Scheme Syntax

The "turn" URI takes the following form (the syntax below is non-
normative):

$$\text{turn}:<\text{host}>:<\text{port}>$$
$$\text{turns}:<\text{host}>:<\text{port}>$$

Note that the <port> part and the preceding ":" (colon) character, is
OPTIONAL.

A TURN/TURNS URI has the following formal ABNF syntax [RFC5234]:

$$\text{turnURI} \ = \ \text{scheme} \ "\text{turn-host} \ [ \ "\text{transport} \ = \ \text{transport} \ ] \ [ \ ?transport=\text{transport} \ ]$$
$$\text{scheme} \ = \ "\text{turn}" / "\text{turns}"$$
$$\text{transport} \ = \ "\text{udp}" / "\text{tcp}" / transport-ext$$
$$\text{transport-ext} \ = \ 1*unreserved$$
$$\text{turn-host} \ = \ \text{IP-literal} / \ \text{IPv4address} / \ \text{reg-name}$$
$$\text{turn-port} \ = \ *\text{DIGIT}$$
$$\text{IP-literal} \ = \ \[ ( \ \text{IPv6address} \ / \ \text{IPvFuture} \ ) \ ]"$$
$$\text{IPvFuture} \ = \ \"v\" \ 1*\text{HEXDIG} \ \." \ 1*\ ( \ \text{unreserved} \ / \ \text{sub-delims} \ / \ \"\" \ )$$
$$\text{IPv6address} \ = \ 6( \ \text{h16} \:"\" \ ) \ \text{ls32}$$
$$/ \ "\:" \ 5( \ \text{h16} \:"\" \ ) \ \text{ls32}$$
$$/ \ [ \ \text{h16} \ ] \ "\:" \ 4( \ \text{h16} \:"\" \ ) \ \text{ls32}$$
$$/ \ [ *1( \ \text{h16} \:"\" \ ) \ \text{h16} \ ] \ "\:" \ 3( \ \text{h16} \:"\" \ ) \ \text{ls32}$$
$$/ \ [ *2( \ \text{h16} \:"\" \ ) \ \text{h16} \ ] \ "\:" \ 2( \ \text{h16} \:"\" \ ) \ \text{ls32}$$
$$/ \ [ *3( \ \text{h16} \:"\" \ ) \ \text{h16} \ ] \ "\:" \ \text{h16} \:"\" \ \text{ls32}$$
$$/ \ [ *4( \ \text{h16} \:"\" \ ) \ \text{h16} \ ] \ "\:" \ \text{ls32}$$
$$/ \ [ *5( \ \text{h16} \:"\" \ ) \ \text{h16} \ ] \ "\:" \ \text{h16} \:"\"$$
$$/ \ [ *6( \ \text{h16} \:"\" \ ) \ \text{h16} \ ] \ "\:"$$

$$\text{h16} \ = \ 1*\text{HEXDIG}$$
$$\text{ls32} \ = \ ( \ \text{h16} \:"\" \ \text{h16} \ ) / \ \text{IPv4address}$$
$$\text{IPv4address} \ = \ \text{dec-octet} \ "." \ \text{dec-octet} \ "." \ \text{dec-octet} \ "." \ \text{dec-octet}$$
$$\text{dec-octet} \ = \ \text{DIGIT} \ ; \ 0-9$$
$$/ \ %x31-39 \ \text{DIGIT} \ ; \ 10-99$$
$$/ \ "1" \ 2\text{DIGIT} \ ; \ 100-199$$
$$/ \ "2" \ %x30-34 \ \text{DIGIT} \ ; \ 200-249$$
$$/ \ "25" \ %x30-35 \ ; \ 250-255$$
reg-name = *( unreserved / pct-encoded / sub-delims )

<unreserved>, <sub-delims>, and <pct-encoded> are specified in [RFC3986]. The core rules <DIGIT> and <HEXDIGIT> are used as described in Appendix B of RFC 5234 [RFC5234].

The <host>, <port> and <transport> components are passed without modification to the [RFC5928] algorithm. <secure> is set to false if <scheme> is equal to "turn" and set to true if <scheme> is equal to "turns" and passed to the [RFC5928] algorithm with the other components.

3.2. URI Scheme Semantics

The TURN protocol supports sending messages over UDP, TCP or TLS-over-TCP. The "turns" URI scheme SHALL be used when TURN is run over TLS-over-TCP (or in the future DTLS-over-UDP) and the "turn" scheme SHALL be used otherwise.

The required <host> part of the "turn" URI denotes the TURN server host.

The <port> part, if present, denotes the port on which the TURN server is awaiting connection requests. If it is absent, the default port SHALL be 3478 for both UDP and TCP. The default port for TURN over TLS SHALL be 5349.

4. Security Considerations

Security considerations for the resolution mechanism are discussed in [RFC5928].

The "turn" and "turns" URI schemes do not introduce any specific security issues beyond the security considerations discussed in [RFC3986].

5. IANA Considerations

This section contains the registration information for the "turn" and "turns" URI Schemes (in accordance with [RFC4395]).

5.1. TURN URI Registration

URI scheme name: turn

Status: permanent
URI scheme syntax: See Section 3.

URI scheme semantics: See [RFC5928].

Encoding considerations: There are no encoding considerations beyond those in [RFC3986].

Applications/protocols that use this URI scheme name:

The "turn" URI scheme is intended to be used by applications that might need access to a TURN server.

Interoperability considerations: N/A


Contact: Marc Petit-Huguenin <petithug@acm.org>

Author/Change controller: The IESG

References: RFCXXXX

[[NOTE TO RFC EDITOR: Please change XXXX to the number assigned to this specification, and remove this paragraph on publication.]]

5.2. TURNs URI Registration

URI scheme name: turns

Status: permanent

URI scheme syntax: See Section 3.

URI scheme semantics: See [RFC5928].

Encoding considerations: There are no encoding considerations beyond those in [RFC3986].

Applications/protocols that use this URI scheme name:

The "turns" URI scheme is intended to be used by applications that might need access to a TURN server over a secure connection.

Interoperability considerations: N/A


Contact: Marc Petit-Huguenin <petithug@acm.org>
6. Acknowledgements

Thanks to Margaret Wasserman, Magnus Westerlund, Juergen Schoenwaelder, Sean Turner, Ted Hardie, Dave Thaler, Alfred E. Heggestad, Eilon Yardeni, Dan Wing, Alfred Hoenes, and Jim Kleck for their comments, suggestions and questions that helped to improve the draft-petithuguenin-behave-turn-uri-bis document.

Many thanks to Cullen Jennings for his detailed review and thoughtful comments on the draft-nandakumar-rtcweb-turn-uri document.

The <turn-port> and <turn-host> ABNF productions have been copied from the <port> and <host> ABNF productions from [RFC3986].

This document was written with the xml2rfc tool described in [RFC2629].

7. References

7.1. Normative References


7.2. Informative References


[WEBRTC] W3C, "WebRTC 1.0: Real-time Communication Between Browsers".

[REF-IMPL] Petit-Huguenin, MPH., "Reference Implementation of TURN resolver and TURN URI parser".

Appendix A. Examples

Table 1 shows how the <secure>, <port> and <transport> components are populated from various URIs. For all these examples, the <host> component is populated with "example.org".

<table>
<thead>
<tr>
<th>URI</th>
<th>&lt;secure&gt;</th>
<th>&lt;port&gt;</th>
<th>&lt;transport&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>turn:example.org</td>
<td>false</td>
<td></td>
<td></td>
</tr>
<tr>
<td>turns:example.org</td>
<td>true</td>
<td></td>
<td></td>
</tr>
<tr>
<td>turn:example.org:8000</td>
<td>false</td>
<td>8000</td>
<td></td>
</tr>
<tr>
<td>turn:example.org?transport=udp</td>
<td>false</td>
<td></td>
<td>UDP</td>
</tr>
<tr>
<td>turn:example.org?transport=tcp</td>
<td>false</td>
<td></td>
<td>TCP</td>
</tr>
<tr>
<td>turns:example.org?transport=tcp</td>
<td>true</td>
<td></td>
<td>TLS</td>
</tr>
</tbody>
</table>

+---------------------------------+----------+--------+-------------+
| Table 1                          |          |        |             |

Appendix B. Release notes

This section must be removed before publication as an RFC.
B.1. Modifications between petithuguenin-behave-turn-uris-01 and petithuguenin-behave-turn-uris-00
   o Removed userinfo.

B.2. Merge of draft-nandakumar-rtcweb-turn-uri-00 and draft-petithuguenin-behave-turn-uri-bis-05
   o Changed author list.
   o Draft is now standard track.
   o Merged abstract, introduction, acknowledgement and security sections.
   o Added two introductory paragraphs to the beginning of the introduction.
   o Took Section 3 and divided it into Section 3.1 URI Scheme Syntax and Section 3.2 URI Scheme Semantics.
   o Explained that most components are passed as is to RFC 5928.
   o Added username and password in ABNF.
   o Added RFC 5389 as reference.
   o Added examples.
   o Updated design notes.
   o Various minor nits and grammatical issues fixed.

B.3. Modifications between petithuguenin-05 and petithuguenin-04
   o Nits.
   o Fixed schemes registration.

B.4. Modifications between petithuguenin-04 and petithuguenin-03
   o Fixed references code link.

B.5. Modifications between petithuguenin-03 and petithuguenin-02
   o Updated RFC references.

B.6. Modifications between petithuguenin-02 and petithuguenin-01
   o Nits.

B.7. Modifications between petithuguenin-01 and petithuguenin-00
   o Shorten I-D references.

B.8. Design Notes
<password> is not used in the URIs because it is deprecated. <username> is not used in the URIs because it is not used to guide the resolution mechanism.

- As discussed in Dublin, there is no generic parameters in the URI to prevent compatibility issues.

Authors’ Addresses

Marc Petit-Huguenin
Unaffiliated

Email: petithug@acm.org

Suhas Nandakumar
Cisco Systems
170 West Tasman Drive
San Jose, CA  95134
US

Email: snandaku@cisco.com

Gonzalo Salgueiro
Cisco Systems
7200-12 Kit Creek Road
Research Triangle Park, NC  27709
US

Email: gsalguei@cisco.com

Paul E. Jones
Cisco Systems
7025 Kit Creek Road
Research Triangle Park, NC  27709
US

Email: paulej@packetizer.com
The Case for Layered Codecs
draft-wenger-rtcweb-layered-codec-00

Abstract

RTCWEB is in the process of developing a protocol infrastructure and a browser API to support browser-to-browser real-time communication over IP. Real-time communication necessarily requires the use of encoders and decoders (codecs) for media data. The document advocates mandating support for a class of codecs known as scalable or layered codecs, for their superior network adaptivity, error resilience, and application adaptivity. Examples are provided for use cases currently under discussion, focusing on video coding as the, perhaps, most challenging media type currently under consideration.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 5, 2012.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents.
carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction .............................................. 3
2. Terminology ............................................... 3
3. Detour: Suggested additional Requirements .................. 3
4. Scalable Codecs in the defined Use Cases ................... 4
   4.1. Use case: Browser to browser use-cases ................ 5
   4.2. Telephony use cases ................................... 7
   4.3. Video conferencing use cases ........................... 8
       4.3.1. Use-case: Multiparty Video Communication .......... 8
       4.3.2. Use-case: Video Conferencing w/ Central Server ... 10
   4.4. Embedded voice communication use cases ............... 10
   4.5. Bandwidth/QoS/mobility use cases ..................... 11
5. Concluding Remarks ......................................... 11
6. Security Considerations .................................... 12
7. References ................................................ 12
   7.1. Normative References .................................. 12
   7.2. Informative References ............................... 12
Authors’ Addresses ........................................... 13
1. Introduction

In this document we advocate the mandatory support of scalable coding techniques for RTCWEB. If consensus towards such as position is not achievable for whatever technical, commercial, or legal reasons, we suggest that not having mandatory codecs may be better than mandating the use of an inferior, non-scalable codec.

The current status of RTCWEB’s use case and requirements discussion is captured in [I-D.ietf-rtcweb-use-cases-and-requirements]. (Note: Version 00 of this I-D was used for the comparison in this document.) We use this document as a guide and attempt to compare the broad categories of scalable and non-scalable codecs in terms of how well they can satisfy the requirements specified therein. We use video codecs as an example, and conclude that scalable codecs are considerably better-suited for RTCWEB than non-scalable codecs.

We first address some additional - and, we believe, self-evident - requirements.

2. 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] and indicate requirement levels for compliant implementations.

3. Detour: Suggested additional Requirements

There are certain requirements that, we believe, appear so obvious to the authors of [I-D.ietf-rtcweb-use-cases-and-requirements] that they have not been explicitly captured. One of these requirements is:

Fxx: The browser MUST enable communication with a peer (other browser(s), MCUs, etc.) with a latency adequate for "real-time" communication.

This is a crucial requirement when selecting a (video) codec for reasons that will become apparent shortly. We believe that RTCWEB should strive for solutions allowing latency (not counting long-distance network delays) below 200 msec. Furthermore, any solution that cannot guarantee a latency below 500 msec (before dropping video altogether) should not be considered. At the time of writing there are mailing list discussions ongoing regarding a requirement that may be differently formulated but addresses the same operational aspect. We don’t specifically care about the formulation, but highly
recommend to go through the pain of agreeing to fixed millisecond numbers.

An additional requirement that should be considered relates to heterogeneity:

Fx_y: The browser must be able to send fully decodable bitstreams, ideally without wasting resources, for a broad range of receivers (handheld to desktop) and, in multi-party cases, for a heterogeneous receiver population.

Seems obvious? Yes, to us. Have the implications, however, been thoroughly considered? Not that we are aware of. We discuss this further when addressing the multiparty use case.

Finally, an additional requirement that relates to the heterogeneity of peers has to do with the heterogeneity of connection characteristics among peers:

Fx_z: The browser MUST be able to perform error control on multiple streams, with potentially different error characteristics, simultaneously.

In a full-mesh, multipoint scenario, for example, handling of individual peer connection characteristics can be very challenging. Assume, by way of example, that in a five-way conference one of the streams gets damaged or that the corresponding connection is subject to congestion. If the browser chooses to produce a more robust stream, then the other three receiving peers will be penalized through the overhead spent for error protection and resulting inferior picture quality, even if they have perfect connections to the sender. If, on the other hand, the browser chooses to produce separate streams for each of its peers, then it must be able to handle an increasing computational load. Even if the computational load is not a concern, battery power consumption certainly is. In fact, there is only one thing that grows faster than CPU power (according to Moore’s law): per-pixel cycle demands of video codecs. Also, we should bear in mind that even when staying with moderately complex (and efficient) video codecs, video resolution constantly goes up. Worse, resolution grows in both horizontal and vertical dimensions, whereas CPU power grows only in one dimension.

4. Scalable Codecs in the defined Use Cases

The concept of scalable coding (for video, audio, or any media) consists of providing a representation of the source at multiple levels of fidelity using a bitstream comprised of a corresponding
number of layers. Typically these layers are formed in a pyramidal fashion, such that a given level of fidelity requires the availability of some or all of the lower layers (i.e., those that correspond to lower levels of fidelity). The dependency exists because the encoding process uses prediction between layers, thus providing increased compression efficiency. If no such prediction is used, then the layers are independent of one another, and we have what is called simulcasting. Indeed, from a media coding point of view, simulcasting is scalable coding without inter-layer prediction. In simulcast, each layer provides a different level of fidelity and is independently decodable.

One example of a standardized codec that uses such pyramidal layering to implement scalable coding is H.264/SVC. MPEG-2 and H.263+ are two other similar examples. The scalable layers in H.264/SVC enhance fidelity in any of three dimensions: temporal rate, spatial resolution, or spatial quality (or signal-to-noise ratio, SNR). More than one such enhancement can exist in a given video at the same time, i.e., scalability enhancements can be combined.

Scalable coding has been a well-known design architecture in the media coding community since at least 1993, but it has only recently been used in products. As a relatively recent addition to the commercial audio-video communication arsenal, it is not yet widely known among people who are not video coding experts. An overview of the H.264/SVC standard can be read in [wiegand2007]. A detailed discussion of how it can be used in videoconferencing systems is provided in [elefth2006].

In the following we revisit each of the use cases and discuss how scalable coding, and particularly pyramidal scalable video coding, can be used to provide a significantly improved user experience.

4.1. Use case: Browser to browser use-cases

This category has four sub-cases. The sub-cases "simple video communication" and "simple video communication with inter-operator calling" are, from a codec viewpoint, the same, with the exception of the mentioned interoperability argument resulting in a need for a baseline codec. From a (video) codec viewpoint the "Hockey" sub-case appears to consist of two independent video streams being sent by the mobile phone. (We are actually not sure whether this, rather exotic, multi-camera scenario warrants its own use case, but we are not opposed to it, either.) The sub-case "video size change" appears, to us, to introduce a feature that would be relevant to all video-capable use cases. As a result, from a codec viewpoint, all four use cases appear to have similar requirements and can be discussed jointly.
On the surface, these use cases appear to be trivially supported by any video codec as long as basic rate control and error resilience features are utilized (probably with the help of protocol support such as RTCP receiver reports and feedback messages therein, retransmission, and so on).

Let's see, however, how scalable codecs can offer significant advantages over non-scalable codecs.

First, temporal scalability (where pictures coded in an enhancement layer can be decoded in combination with a lower frame rate base layer) can greatly enhance error resilience, through techniques such as the one described in [wenger1998]. Especially in conjunction with feedback messages, virtually latency-neutral repair mechanisms can be devised without relying on latency- and bandwidth-unfriendly intra (I) pictures. It is important to note that such "reactive" techniques do not penalize a system when no errors happen to occur. Also, by being reactive, they are amenable to heterogeneous error control support (thus addressing requirement Fxz).

(Note: sending intra pictures is typically not a good idea. First, when transmitting video over a bandwidth-limited link that is close to capacity (e.g., a 3G link), an I frame can bring the latency up to the seconds range. Second, I frames are much larger than P or B frames and, therefore, statistically much more likely to be hit by a packet loss. Sending I frames for error control is, in almost all cases, a bad idea. It has been done for many years, but only because few, if any, better error control mechanisms were available.)

A second scalability dimension is spatial scalability. Here the enhancement layer enables decoding of the video in a higher resolution than a base layer, but can predict information from the base layer. Spatial scalability can also offer advantages for error control, something that we would gladly elaborate on in the future. In this use case, however, its key advantage lies in the graceful and, if properly implemented, latency-neutral handling of both changes in available network bandwidth (addressing the bandwidth and error characteristics aspects of F23, among others) and receiver-side rendering requirements (addressing F22).

Most, if not all, video coding experts will agree that there is nothing better than shedding pixels when running into a bandwidth issue, and this can be trivially done (without sending an I frame!) when using spatially scalable coding techniques. Similarly, recovering from bandwidth shortages can be achieved without sending an I frame, again in those cases where the sender properly implements spatial scalability. These techniques are generally more flexible and often lead to better-quality pictures compared to the rate-
control techniques used in single-layer codecs (QP adjustment).

Another point that speaks for spatial scalability is the option to gracefully and quickly react to rendering size requirements, i.e., users enlarging or "full-screening" the rendering window (F22). Again adding a high resolution scalable layer can be done without sending an I frame (which, as already pointed out, are evil), and, if implemented properly, can be done without interrupting the smoothness of the video playback experience. (Note that even today's most advanced spatial scalability techniques still encode only a small set of discrete picture sizes in their finite, and small, number of layers. As a result, some form of resampling in the rendering interface will probably still be required.)

It has been remarked that simulcasting low and high resolution representations of the same picture can have very similar positive effects. This is correct. However, simulcast comes at a price, which is bandwidth and compute cycle requirements. The quite extensive subjective evaluation tests performed by MPEG in conjunction with the H.264/SVC verification tests have shown that a 2006-generation scalable video encoder offers between 17% and 40% bitrate savings over a same generation encoder using simulcast [mpeg2007].

(Note that "subjective" evaluation is the Mercedes of media quality testing, whereas "objective" evaluation compares to a pre-"Volt" Chevrolet :-). Subjective tests, when performed properly, are by no means unscientific and they are, in fact, considered a much better way to assess the quality of a codec compared with objective tests (e.g., the PSNR, or less crude metrics devised by groups such as the Video Quality Experts Group). Unfortunately, subjective evaluations are also orders of magnitude more expensive and time-consuming than objective tests, as it takes many person-hours to evaluate a single test case, whereas objective tests take only CPU cycles.)

Further, while a modern scalable encoder creating two spatial layers may require roughly the same number of cycles as two encoders coding the same pixel count, there are savings in the decoder due to what SVC calls "single loop decoding" (i.e., the fact that the receiver only needs to maintain a prediction loop just for the layer that is being decoded, and not for the layers that the to-be-decoded layer may depend on).

4.2. Telephony use cases

As the telephony use cases appear to address interaction with legacy telephone equipment only, there is probably little use for the quality scalability that modern scalable speech/audio codecs offer.
As formulated, from a codec viewpoint, any speech/audio codec used in telco environments (and many that have been specified outside this environment) ought to be acceptable choices.

4.3. Video conferencing use cases

There are two sub-cases in this broad category: full-mesh video conferencing, and centralized conferencing with resolution switching. The use cases also make particular assumptions regarding audio; the first case involves mono audio with panning at the web application, whereas the second involves mono or stereo audio with mixing at the server.

We first point out that these two use cases offer only a small subset of the functionality offered in today’s multiparty videoconferencing solutions. In particular, the centralized server sub-case appears to deal with one, not frequently used, aspect of MCU-based communication that is implemented in a crude and unoptimized fashion (see below). We believe that video conferencing use cases in a standard drafted in 2011 should encompass at least as much functionality as is routinely offered in 2005 generation MCUs.

We wonder, specifically, why the group is not (yet?) considering other use cases in the same broad category of centralized conferencing that require more flexible server architectures. For example, sometimes active speaker size-up is not wanted, i.e., in scenarios involving presentations with questions. Or, one wants to be able to up-size more than one (but less than the whole population) of speakers. Or a full continuous presence conference where it is up to the user of each receiving browser to decide how he/she wishes to render the received signals. All of these capabilities are available with today’s MCUs. Many similar scenarios could be described. We plan to contribute more detailed descriptions in the future unless we receive pushback.

We now examine each of the two use-case categories.

4.3.1. Use-case: Multiparty Video Communication

On the surface, this use case is not very different from the Simple Video use case, at least from a codec viewpoint. Indeed, it appears that a browser just needs to be able to simultaneously process N incoming independent video bitstreams (as spelled out in F12 and F14). A slightly deeper examination will reveal that this use case appears to be a picture-perfect example of why scalable codecs offer superior performance.

First off, all the arguments that were made for scalable codecs in
the Simple Video use case still apply. Let’s consider, however, what happens when more than one stream is received.

A first consideration is the CPU requirements for decoding all the pictures in full resolution. Suppose a browser is receiving, for example, four video pictures in standard TV resolution and in a non-scalable format. The computational load for decoding these pictures in addition to the load of encoding one’s outgoing picture would pretty much max out today’s desktop CPUs. In typical multipoint layouts, the reconstructed pictures of most of the peers would be shown in thumbnail format, thus throwing away anywhere between 75-95% of the reconstructed pixel count! What a waste.

But waste what? Battery power? Yes, for handheld devices, which do not have the screen real-estate to show all the pictures in full resolution. We come to that in a moment. Electricity? Yes. This is not a joke. A desktop motherboard can easily double its power intake based on CPU load and memory access. Say, a motherboard’s power consumption goes up from 100W to 200W, just because video is decoded in unnecessarily high resolution. One kWh of electricity costs at one of the author’s home (Bay Area, California, USA, PG&E as the electricity provider) about 26 cents at his consumption level (single family house). This means that 8 hours of unnecessarily decoding of full-resolution video that is not rendered at full resolution will cost $0.20. This, by coincidence, is the exact same maximum amount that MPEG-LA charges as licensing fees for using H.264/SVC :-) Never mind the numbers of trees that can be saved...

A second consideration is when multiple streams are received, or transmitted, by the browser in a heterogeneous receiver population. Browsers are today ubiquitous and can be found anywhere from handheld devices with QVGA screens to gaming racks with multiple 2K pixel resolution screens. That, incidentally, is a big part of the appeal of the RTCWEB activity. If you are sitting behind a 23-inch 1080p monitor, your display is approximately 120 dpi. A QVGA image, perfectly appropriate for being rendered on the screens of many smartphones, would be just 2-in by 1.5-in on the PC screen. On one of the authors’ laptop (a 1080p, 13.3-in model, better than 200 dpi) it would literally be the size of a thumbnail. Not exactly what one expects to see, judging form daily uses of Skype and our own (Vidyo’s) desktop video conferencing services. Further, when upsampling to, say, a quarter HD resolution, a QVGA signal simply doesn’t look good no matter how well your upsample filter is designed.

This means that a transmitting browser that creates bitstreams for a receiver population including a smartphone and a PC, would have to, ideally, generate at least three resolutions: thumbnail, something
useful for smartphones (e.g., QVGA), and something useful for PC
users (VGA or better). More option would be better. There are
commercial products available today that ship, using a software codec
on a high-end PC, 1080p60 video conferencing. Two years down the
road, and a mainstream PC will be able to handle that type of load.

It has already been pointed out that the compute cycle requirements
for simulcast and scalable encoding are roughly the same, assuming a
common set of coding tools (i.e., H.264 Baseline profile vs. H.264
Scalable Baseline profile). The sending bandwidth requirements,
however, are not the same: scalable beats simulcast by several tens
of percentage points.

There are additional considerations for simulcast vs. scalable coding
that have to do with RTP packetization and access unit alignment,
resolution switching, and error resilience, that require much more
extensive analysis than what’s intended in this document.

4.3.2. Use-case: Video Conferencing w/ Central Server

This particular use case is interesting because, as formulated, it
appears to assume that multiple video resolutions are produced by
each sending browser and simulcast to the server. As formulated, the
receiving browser receives the active speaker in full resolution and
the other participants in low resolution. The server selects what to
forward based on speech activity. Apparently, what is emulated here
is one operation point implemented in many continuous presence MCUs
(namely voice activation), without requiring the video transcoding
features of a continuous presence MCU.

This use case almost begs for use of scalable video coding. The
simulcasting alternative would have all the drawbacks discussed in
the Simple Video Communication Service. For example, consider what
happens when the active speaker changes. How is the receiving
participant to switch resolutions between the two simulcast streams?
It appears to require the request (from server to sending browser) of
an I frame, with all the drawbacks that entails. Actually, as
described, there appears to be no value in simulcasting all
resolutions to the server, because of the need of such interaction...

4.4. Embedded voice communication use cases

The benefits discussed in multiparty video communication can apply
here for scalable audio codecs. Scalable audio codecs haven’t been
mentioned prominently before, as in audio-visual systems the cost of
video processing (in terms of compute cycles and bit rate, among
others) typically outweighs that of audio. The requirements for
today’s games are so high that the CPU and bandwidth requirements for
a couple of audio channels may fall into the category of background noise. However, note that this ceases to be the case as the number of audio channels increases and the quality and (therefore) complexity of the audio codecs grows. We understand that the audio quality here would probably not be restricted to toll quality, and require something like Opus or MP3 for coding. While one or two of these codecs probably would still qualify as "background noise" on a gaming rack, 20 certainly do not. What appears to be useful here would be scalability both in terms of bitrate scalability and complexity scalability.

4.5. Bandwidth/QoS/mobility use cases

The use cases as formulated appear to have limited impact on the codec choices beyond aspects already raised above. NIC changes (or other changes that materially affect connectivity) are best addressed by codecs that can flexibly change their operation points without essentially restarting the codec (such as sending I frames in video, or user-generated codebooks in audio).

If QoS mechanisms that take advantage of QoS marking are supported in the underlying network - something that is at least questionable in today’s Internet, but may be very relevant in the future and in special application fields such as the military - the pyramid structure of scalable codecs makes the selection of bits that are best to be transported at higher QoS trivial: the lower the layer, the higher the desired QoS. We note that scalable systems today emulate such a desirable network behavior by protecting base layers better than enhancement layers, through techniques such as retransmission or FEC.

5. Concluding Remarks

Scalability, while known as a concept for decades, is a relatively new technique in the commercial sphere of video and audio communication products. As a result, a lot of people are not familiar with how it works, and how it can be beneficial to real-time communication systems.

Most significantly, scalable coding has been successfully used to solve several fundamental, decades-old challenges in packet video communication, and it therefore behooves any standardization activity that considers codec design or adoption to take it into serious consideration.
6. Security Considerations

None

7. References

7.1. Normative References


7.2. Informative References


Authors’ Addresses

Stephan Wenger
Vidyo, Inc.
2400 Skyfarm Dr.
Hillsborough, CA  94010
US
Email: stewe@stewe.org

Alex Eleftheriadis
Vidyo, Inc.
433 Hackensack Avenue
Seventh Floor
Hackensack, NJ  07601
US
Email: alex@vidyo.com