Web Real-Time Communications Developer's Guide

Web Real-Time Communications 8.5.2

12/30/2021

Table of Contents

Developer's Guide

The Developer's Guide contains information about how to use the Genesys WebRTC JavaScript API to allow your agents and customers to place voice or video calls from their web browser without downloading special plug-ins or apps.

The first topic describes the product architecture. It is followed by a topic that shows how to use the API and then by an API Reference.

Architecture

Genesys WebRTC Service provides web browsers with Real-Time Communications (RTC) capabilities via simple JavaScript APIs. This means you can write rich, real-time multimedia applications—such as video chat—on the web, without requiring plug-ins, downloads, or installs. Genesys WebRTC Service is able to do this by using the Web Real-Time Communications technologies that have been developed by the World Wide Web Consortium and the Internet Engineering Task Force.

The key component of Genesys WebRTC Service is the Genesys WebRTC Gateway. The Gateway converts WebRTC communications to SIP, allowing for the seamless integration of WebRTC-capable devices into a SIP-based infrastructure—in particular, an architecture that uses Genesys SIP Server. From that perspective, the WebRTC Gateway plays the role of a Media Gateway / Session Border Controller.

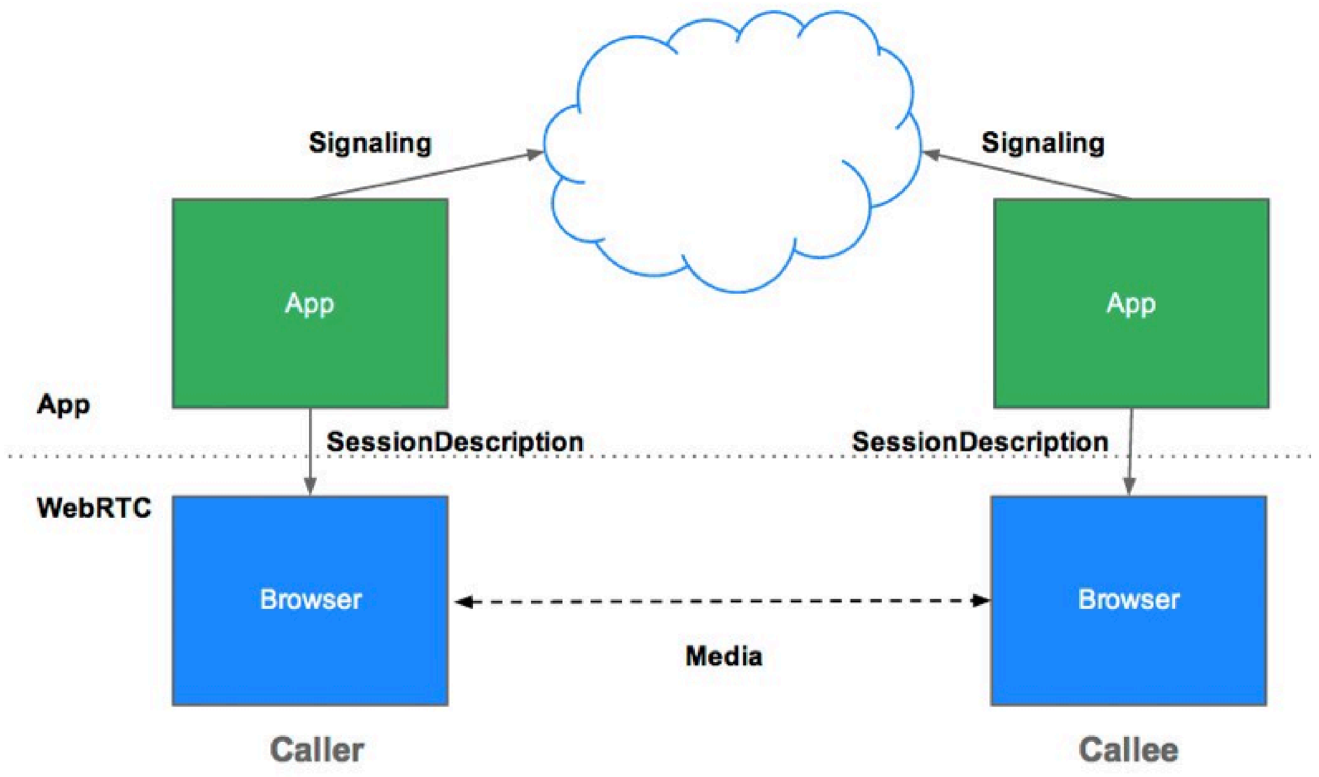
All calls that have been initiated or received by a browser, including browser-to-browser calls, are bridged through the Gateway. This is done to ensure that every call is handled by SIP Server as a SIP call, so that all of the core Genesys features—including routing, treatments, and IVR—can be provided by SIP Server.

Overview

Genesys WebRTC Service uses peers to communicate streams of data. But it also needs a signaling mechanism to send control messages between peers. Signaling proceeds like this:

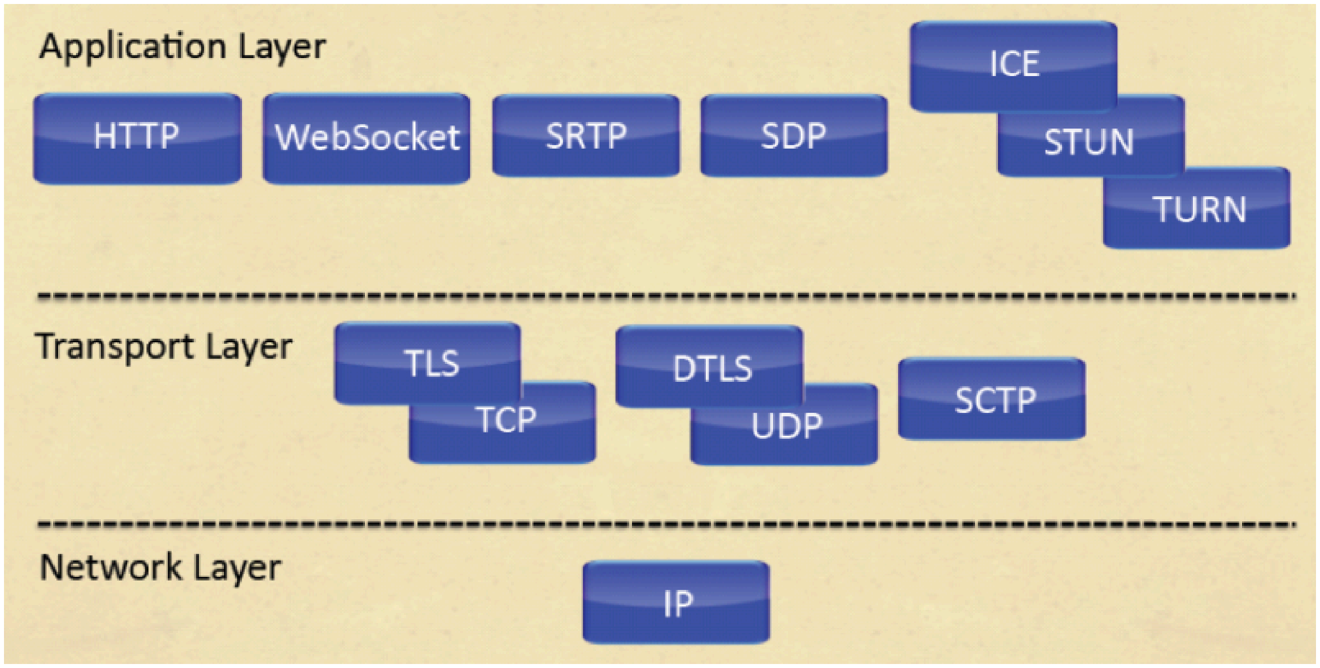
1. Caller sends offer.
2. Callee receives offer.
3. Callee sends answer.
4. Caller receives answer.

Once the signaling process has completed successfully, data can be streamed directly, peer to peer, between the caller and callee, as shown in the following diagram, or via an intermediary server.



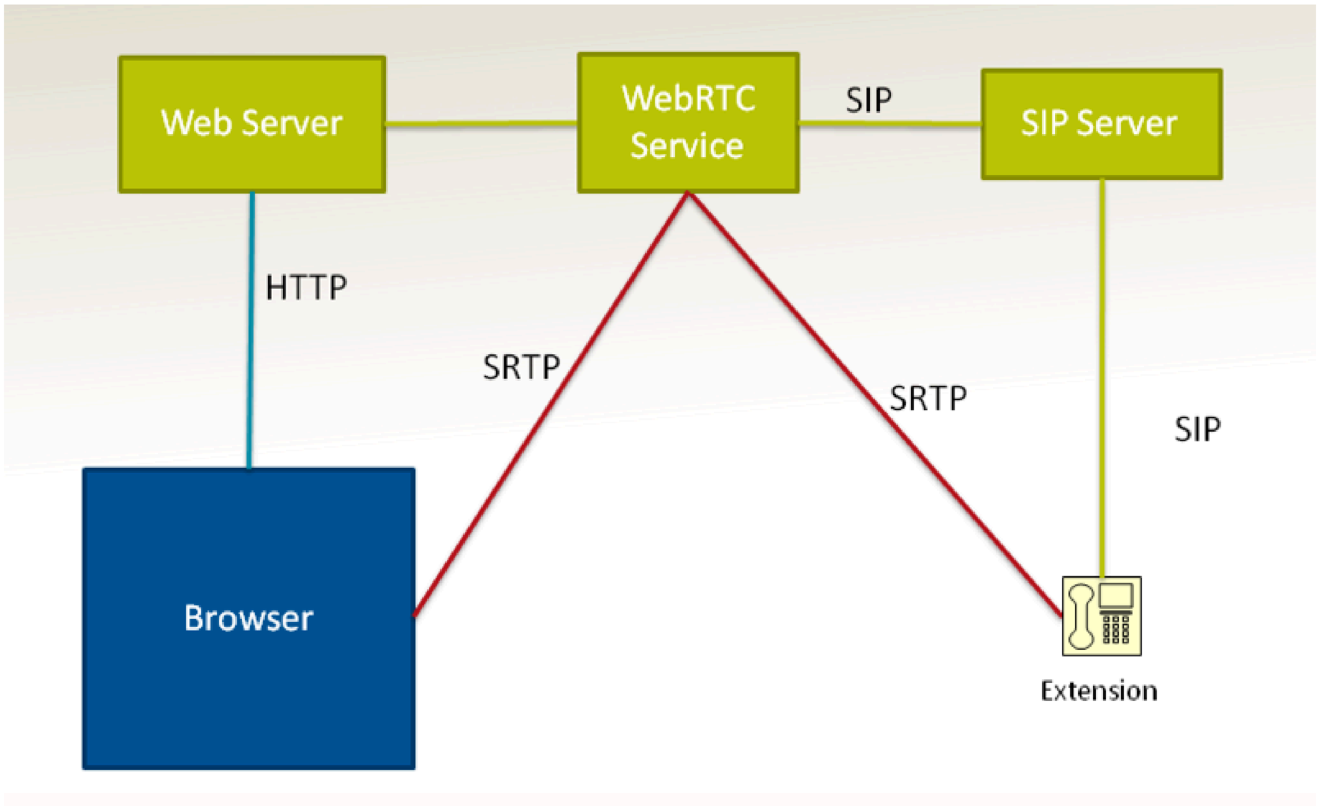
WebRTC Protocols

The following protocols are used in the Genesys WebRTC Service environment:



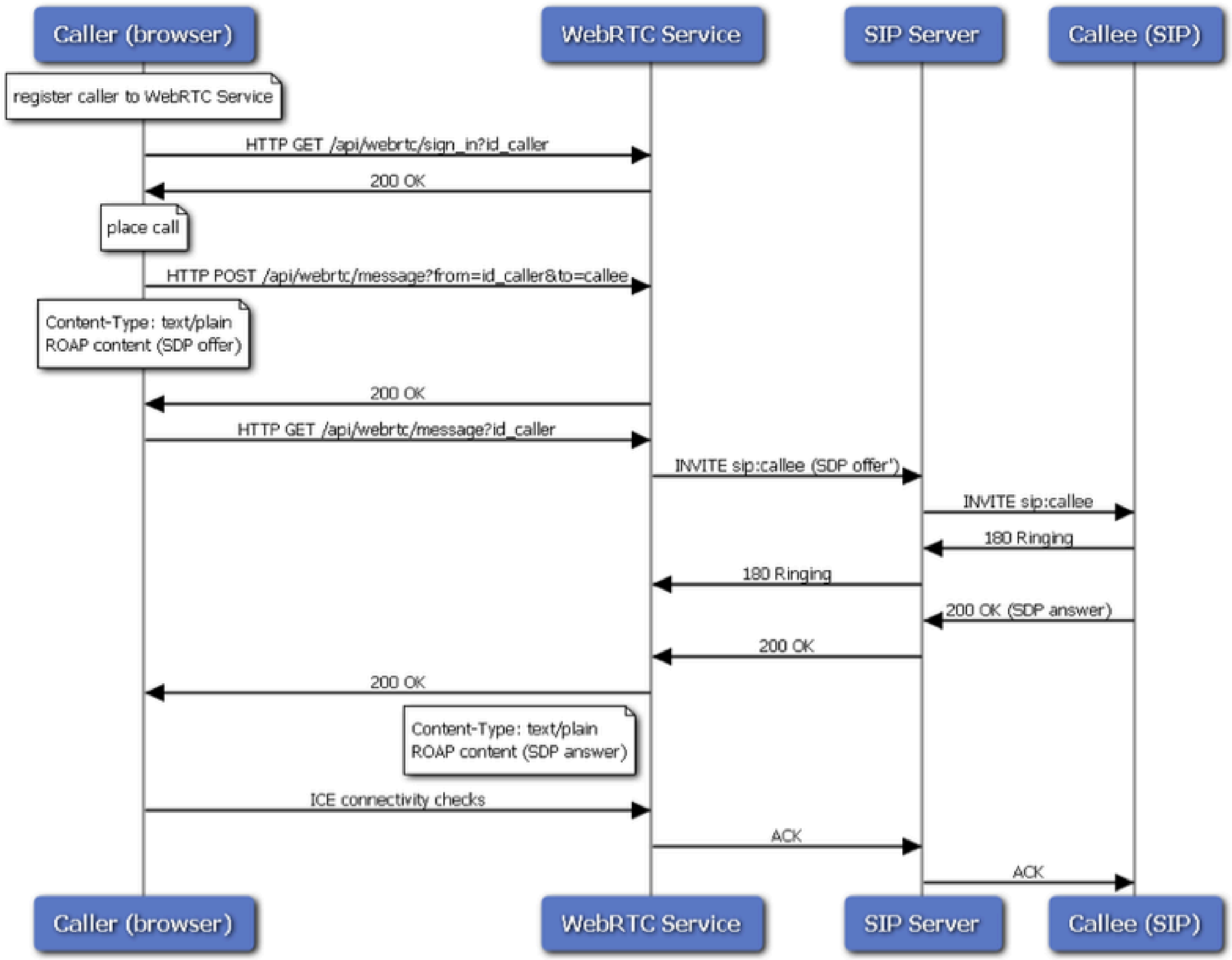
Component Perspective

The following simplified diagram shows the media gateway architecture by which Genesys WebRTC Service interoperates with SIP Server and the Browser. There is an HTTP connection between the Browser and the WebRTC Gateway, and this is used for signaling purposes, where the gateway acts as an HTTP server. The Web application, however, needs to be installed on a regular Web Server.



Basic Call Flow

Here is the basic call flow for a WebRTC call:


Caller (browser)
WebRTC Service
SIP Server
Callee (SIP)
register caller to WebRTC Service
HTTP GET /api/webrtc/sign_in?id_caller
200 OK
place call
HTTP POST /api/webrtc/message?from=id_caller&to=callee
Content-Type: text/plain
ROAP content (SDP offer)
200 OK
HTTP GET /api/webrtc/message?id_caller
INVITE sip:callee (SDP offer')
INVITE sip:callee
180 Ringing
180 Ringing
200 OK (SDP answer)
200 OK
200 OK
Content-Type: text/plain
ROAP content (SDP answer)
ICE connectivity checks
ACK
ACK
Caller (browser)
WebRTC Service
SIP Server
Callee (SIP)

Using the API

The Genesys WebRTC JavaScript API is easy to use, requiring only a few simple steps to register with the WebRTC Gateway and establish media sessions.

Configuration

For information on how to configure your application, see Grtc.Client Instance Attributes.

Support Scripts

To use WebRTC on a browser page, you must include the following JavaScript in your HTML source:

```
<script type="text/javascript" src="<path>/grtc.js"></script>
```

Call Handling

This section describes the things you must do to conduct a call.

Connect to and Register with the Gateway

Follow these steps in order to connect to and register with the WebRTC Gateway:

1. Create an instance of `Grtc.Client`. For example:

   ```
   var configuration = {
      'webrtc_gateway': 'http://WebRTC.genesyslab.com:8086',
      'stun_server': 'stun.genesyslab.com:3478',
      'dtls_srtp' : true
    };
    var grtcClient = new Grtc.Client(configuration);
   ```

   Note: The HTTP port of the WebRTC Gateway is configurable using the parameter `rsmp.http-port`.

2. Invoke either the `connect()` method to connect anonymously or the `register(myDN)` method to establish a connection using a DN that is registered with the SIP Server. The following example demonstrates the `register` method:

   ```
   // set the callback to handle the event when registration is successful
    grtcClient.onRegister.add(onRegisterCallback);
    // set the call back to handle the event when registration fails
    grtcClient.onFailed.add(onFailedCallback);
    // now the client tries to register as DN 1020
    grtcClient.register(“1020”);
   ```

The result of the `register` call is indicated by two events: the `onRegister` event indicating the success of registration, and the `onFailed` event indicating an error. So, as the example shows, you need to add event handlers (callbacks) if you want to perform specific tasks when such events happen.

Enable Local Media Stream

You can enable user media either before or after connecting to the gateway, but it is normally done afterwards. Use these two methods:

- enableMediaSource()
- setViewFromStream()

Here is an example:

```
// set the callback to handle the success event
grtcClient.onMediaSuccess.add(function (obj) {
    // on success, port the local media in an HTML5 video element
    grtcClient.setViewFromStream(document.getElementById("localView"), obj.stream);
});
// set the callback to handle the failure event
grtcClient.onMediaFailure.add(function (obj) {
    window.alert(obj.message);
});
// enable local media source, with audio set to true, and video
// set by a constraint object where the width of video is specified
grtcClient.enableMediaSource(true, {mandatory:{minWidth:360}});
```

The result of the `enableMediaSource` call is indicated by two events: the `onMediaSuccess` event indicating the success of accessing the local media stream, and the `onMediaFailure` event indicating an error. This means you need to add event handlers (callbacks) to properly deal with these events. Typically, you should handle the `onMediaSuccess` event by retrieving the local media stream from the argument object, and then call the `setViewFromStream` method to attach the stream to an HTML5 <video> or <audio> element. The HTML code for the video element might look something like this:

```
<video width="160" height="120" id="localView" autoplay="autoplay">
```

Establish the Call

In order to establish a call, both peers must have carried out the two steps mentioned above. After that, the caller can invoke `makeCall(remoteDN)` and the callee can invoke `acceptCall()`, as described in the following sections.

Caller makeCall()

1. Create a `Grtc.MediaSession` object before making your call.
2. Set up a callback to handle the event that is triggered when the remote stream arrives. This is normally done by calling `setViewFromStream` to attach the remote stream to an HTML5 audio or video element.
3. You can also attach data to the call, such as a customer name and phone number to be sent to an agent.
4. Now you can make the call.

Here is an example:

```
// create a new session (passing the client object as argument)
var grtcSession = new Grtc.MediaSession(grtcClient);
// set callback to port remote media stream when it comes
grtcSession.onRemoteStream.add(function (data) {
    grtcClient.setViewFromStream(document.getElementById("remoteView"), data.stream);
});
// attach data if available: the data is an array of objects
// each object contains two properties named “key” and “value”
var dataToAttach = [
    {
        "key": "Name",
        "value": $('#login_name').val()
    },
    {
        "key": "Phone",
        "value": $('#phone_number').val()
    },
    {
        "key": "Email",
        "value": $('#email').val()
    }
];
grtcSession.setData(dataToAttach);
// preparation is ready, now make the call
grtcSession.makeCall(“1021”);
```

Callee acceptCall()

The JavaScript API fires an `onIncomingCall` event when an `OFFER` message is received. The callee web app is expected to handle this event by creating a `Grtc.MediaSession` instance, informing the user of the incoming call, and if the user authorizes, invoking `Grtc.MediaSession.acceptCall()` to establish a call session.

Here is an example:

```
grtcClient.onIncomingCall.add(function (data) {
    // create a session
    var grtcSession = new Grtc.MediaSession(grtcClient);
    // register a handler when remote stream is available
    grtcSession.onRemoteStream.add(function (data2) {
        grtcClient.setViewFromStream(document.getElementById("remoteView"), data2.stream);
    });
    // inform user of incoming call
    var userResponse = window.confirm(“Accept call from “ + data.peer + “?”);
    if (userResponse === true) {
        // accept the call
        grtcSession.acceptCall();

        // process attached data if available
        var dataAttached = grtcSession.getData();
        if (dataAttached) {
            // dataAttached is an array of objects, each having 2 properties "key" and "value"
            for (var i=0; i<dataAttached.length; ++i) {
                var obj = dataAttached[i];
                console.log("dataAttached[" + i + "]:" + obj.key + ", " + obj.value);
        }
    }
} else {
    grtcSession.rejectCall();
}
});
```

Terminate the Call

You can explicitly terminate a call by invoking `Grtc.MediaSession.terminateCall()`, followed by `Grtc.Client.disableMediaSource()`, as shown in this example:

```
grtcSession.terminateCall();
grtcClient.disableMediaSource();
// ... other processing, for example, call status update on the web page
```

If the remote peer terminates a call, the local peer needs to respond accordingly. You can prepare for this action by registering the `onPeerClosing` event on the client right after the client instance has been created, as shown in this example:

```
var grtcClient = new Grtc.Client(configuration);
// Set up other callbacks, as shown above...
grtcClient.onPeerClosing.add(function () {
    grtcClient.disableMediaSource();
    // ... other processing, for example, call status update on the web page
});
```

Backbone Example

This section provides two HTML files, one as the caller and the other as the callee, in order to give a full picture of how to use the WebRTC JavaScript API to establish a simple WebRTC call. What this example illustrates is very simple: the callee connects to the WebRTC Gateway using DN 1020, and the caller connects anonymously and calls the callee.

caller.html:

```
<html>
<head>
<style> video { border: 5px solid gray; } </style>
<script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
<script type="text/javascript" src="../../src/jsapi/classes/grtc.js"></script>
<script type="text/javascript">
    var conf = {
        "webrtc_gateway": "http://WebRTC.genesyslab.com:8086",
        "stun_server": "stun.genesyslab.com:3478",
        "dtls_srtp" : true
    };
    // construct a Grtc.Client instance
    var grtcClient = new Grtc.Client(conf);
    var grtcSession = null;

    // add a handler to do some work when the peer closes
    grtcClient.onPeerClosing.add(function () {
        $("#remoteStatus").empty();
        if (grtcSession) grtcSession = null;
    });

    // add a handler to disconnect when window is closed
    window.onbeforeunload = function() {
        grtcClient.disconnect();
    };

    grtcClient.onMediaSuccess.add(function (obj) {
        grtcClient.setViewFromStream(document.getElementById("localView"), obj.stream);
        grtcClient.onConnect.add(function () {
            $("#localStatus").empty();
            $("#localStatus").append("connected anonymously");
            // create a MediaSession instance and make a call on it
            grtcSession = new Grtc.MediaSession(grtcClient);
            grtcSession.onRemoteStream.add(function (data) {
                grtcClient.setViewFromStream(document.getElementById("remoteView"),
data.stream);
            });
            grtcSession.makeCall("1020");
        });
        grtcClient.onFailed.add(function (e) { window.alert(e.message); });
        grtcClient.connect();
    });
    grtcClient.onMediaFailure.add(function (obj) {
        window.alert(obj.message);
    });
    // enable microphone and camera
    grtcClient.enableMediaSource();

    function terminateCall() {
        grtcSession.terminateCall();
        grtcSession = null;
        $("#remoteStatus").empty();
    }

</script>
</head>

<body>
<div>
<input type="button" style="text-align:left;width:100px;" value="Terminate Call"
onClick="terminateCall();">
</div>

<div>
    <table>
        <tr> <td> local view </td> <td> remote view </td> </tr>
        <tr>
            <td> <video width="160" height="120" id="localView" autoplay="autoplay" controls>
</td>
            <td> <video width="160" height="120"id="remoteView" autoplay="autoplay" controls>
</td>
        </tr>
        <tr> <td> <span id="localStatus"></span> </td> <td> <span id="remoteStatus"></span>
</td> </tr>
    </table>
</div>

</body>
</html>
```

callee.html:

```
<html>
<head>
<style> video { border: 5px solid gray; } </style>
<script type="text/javascript" src="//code.jquery.com/jquery-latest.js"></script>
<script type="text/javascript" src="../../src/jsapi/classes/grtc.js"></script>
```

```
<script type="text/javascript">
    var conf = {
        "webrtc_gateway": "http://WebRTC.genesyslab.com:8086",
        "stun_server": "stun.genesyslab.com:3478",
        "dtls_srtp" : true
    };
    // construct a Grtc.Client instance
    var grtcClient = new Grtc.Client(conf);
    var grtcSession = null;

    // callee needs to register a handler to deal with incoming call
    grtcClient.onIncomingCall.add(function (data1) {
        // create a MediaSession to handle incoming call
        grtcSession = new Grtc.MediaSession(grtcClient);

        // register a handler when remote stream is available
        grtcSession.onRemoteStream.add(function (data2) {
            grtcClient.setViewFromStream(document.getElementById("remoteView"), data2.stream);
        });

        // ask user to confirm whether to accept or reject call
        var user_said = window.confirm("Do you want to accept the call from " + data1.peer +
"?");
        if (user_said === true) {
            $("#remoteStatus").empty();
            $("#remoteStatus").append("call from " + data1.peer);
            grtcSession.acceptCall();
        } else {
            grtcSession.rejectCall();
            grtcSession = null;
        }
    });

    // add a handler to do some work when the peer closes
    grtcClient.onPeerClosing.add(function () {
        $("#remoteStatus").empty();
        if (grtcSession) grtcSession = null;
    });

    // add a handler to disconnect when window is closed
    window.onbeforeunload = function() {
        grtcClient.disconnect();
    };

    grtcClient.onMediaSuccess.add(function (obj) {
        grtcClient.setViewFromStream(document.getElementById("localView"), obj.stream);
        // once microphone and camera are enabled, connect to gateway
        grtcClient.onRegister.add(function () {
            $("#localStatus").empty();
            $("#localStatus").append("connected as 1020");
        });
        grtcClient.onFailed.add(function (e) { window.alert(e.message); });
        grtcClient.register("1020");

    });
    grtcClient.onMediaFailure.add(function (obj) {
        window.alert(obj.message);
    });
    // enable microphone and camera
    grtcClient.enableMediaSource();

    function terminateCall() {
        grtcSession.terminateCall();
        grtcSession = null;
        $("#remoteStatus").empty();
    }
</script>
</head>

<body>
<div>
<input type="button" style="text-align:left;width:100px;" value="Terminate Call"
onClick="terminateCall();">
</div>

<div>
    <table>
        <tr> <td> local view </td> <td> remote view </td> </tr>
        <tr>
            <td> <video width="160" height="120" id="localView" autoplay="autoplay" controls>
</td>
            <td> <video width="160" height="120" id="remoteView" autoplay="autoplay"
controls> </td>
        </tr>
        <tr> <td> <span id="localStatus"></span> </td> <td> <span id="remoteStatus"></span>
</td> </tr>
    </table>
</div>

</body>
</html>
```

Browser Interoperability

Vendors tend to add their own prefixes to public API interfaces before the API standard has been finalized. For example, for the `getUserMedia` API specified by the W3C, Chrome provides this API as `webkitGetUserMedia`, while Firefox provides it as `mozGetUserMedia`. Other vendors may use other names. Refer to https://webrtc.org/web-apis/interop/ for a quick summary of the naming issues. The Genesys WebRTC JavaScript API has integrated the "polyfill" library suggested on that page in order to take care of some of the interoperability issues and allow developers to write to the unprefixed W3C standard names.

Known Issues

Firefox Renegotiation Issue

Firefox does not currently support renegotiation of an ongoing media session: once a media session has been set up, its parameters are fixed. For all practical purposes, this means that you cannot, for example, start an audio-only call and then add video to that same `PeerConnection` later in that session.

In order to add video mid-call, the recommended workaround is to destroy the audio-only `PeerConnection` and create a new `PeerConnection` that uses both audio and video. This is demonstrated in Demo Number 3, which comes with the WebRTC JSAPI IP. If you wish to track this issue, the current Firefox bug can be found at https://bugzilla.mozilla.org/show_bug.cgi?id=857115.

Note: This issue has now been resolved by Firefox. However, the default behavior in JSAPI has not changed, so it will still create a new `PeerConnection` on every renegotiation. This behavior in JSAPI can be overridden to reuse the same `PeerConnection` by calling the `Grtc.Client` method `setRenewSessionOnNeed(false)` with the value `false` during the initialization part in the client application. Also, the WebRTC Gateway option `rsmp.new-pc-support` must be set to `0` for this to work.

Mobile Browser Support

- Google Chrome for Android supports WebRTC in version 29 and higher.
- Mozilla Firefox for Android supports WebRTC in version 24 and higher.
- Opera for Android supports WebRTC in version 20 and higher.

Note that the following problems may be noticed when using WebRTC in Android browsers:

- Audio may sound choppy and warbled, especially on devices where the CPU is under a heavy load
- Acoustic echo cancellation on mobile platforms may not work well
- For Chrome, DTLS on Android can fail and may need to be disabled
- There may be general stability and complexity issues

Tips on making calls with WebRTC for Android:

- DTLS can be disabled for Chrome and Opera
- The default resolution used WebRTC is 640 x 480, which may be too complex for certain mobile devices. Therefore, if you notice a low frame rate or a loaded CPU, use a lower resolution, such as 320 x 240.

API Reference

The WebRTC JavaScript API contains the following classes:

- **Grtc**—This is the base class for the `Grtc` namespace.
- **Grtc.Client**—This class creates a WebRTC client and provides methods for connecting to and registering with the WebRTC Gateway.
- **Grtc.Error**—Some functions in the `Grtc` namespace can raise exceptions under certain circumstances. Each such exception is an instance of this class.
- **Grtc.MediaSession**—This class creates a peer connection and implements ROAP-based signaling that is built on JSEP as substrate.

Grtc

This is the base class for the `Grtc` namespace. It only contains static methods. There is no need to instantiate this class.

Static Variables

The following log levels are defined to be used with the `Grtc.Client` methond `setLogLevel()`. The numeric values of these levels increase in this order:

- LOG_LEVEL_NONE
- LOG_LEVEL_ERROR
- LOG_LEVEL_NOTICE
- LOG_LEVEL_INFO
- LOG_LEVEL_DEBUG

Static Methods

getWebrtcDetectedBrowser ( )

This method can be called to get the client browser type.

getWebrtcDetectedVersion ( )

This method can be called to get the client browser version.

Example

The following example demonstrates the use of both `getWebrtcDetectedBrowser()` and `getWebrtcDetectedVersion()`.

```
console.log("Browser Detected: " + Grtc.getWebrtcDetectedBrowser() + " " +
Grtc.getWebrtcDetectedVersion());
```

The output from this statement might look something like this:

```
Browser Detected: firefox 29
```

isWebrtcSupported ( )

This method can be called to check whether the client browser supports WebRTC. The return type is `boolean`.

Example

```
if (Grtc.isWebrtcSupported()) {
    // webrtc supported by browser
    // ... do whatever processing needed ...
} else {
    // webrtc not supported by browser; warn the user
    alert("Webrtc not supported by the browser in use.”);
}
```

Grtc.Client

This class creates a WebRTC client and provides methods for connecting to and registering with the WebRTC Gateway.

Instantiation

This class requires configuration parameters for its initialization. These parameters are provided through a configuration object as described in the section on Instance Attributes. Instantiation of this class generates a globally unique identifier for use by your application's callbacks. Once the instance has been created, event callbacks should be set before calling any instance methods.

Throws: CONFIGURATION_ERROR

Instantiation of this class raises an exception if any of the mandatory parameters have not been defined or if any of the parameter values are malformed.

Example

```
var configuration = {
  ‘webrtc_gateway’: ‘http://WebRTC.genesyslab.com:8080’,
  ‘stun_server’: ‘stun.genesyslab.com:3478’,
};
var grtcClient = new Grtc.Client(configuration);
// set the callback to handle the event when registration is successful
grtcClient.onRegister.add(onRegisterCallback);
// set the call back to handle the event when registration fails
grtcClient.onFailed.add(onFailedCallback);
// now the client tries to register as DN 1020
grtcClient.register(“1020”);
```

Instance Attributes

configuration

The configuration used by this instance. It is specified as a JSON object with the properties defined in the following tables.

Mandatory Parameters

Name	Description	Default
webrtc_gateway	HTTP URL (String) for the WebRTC Gateway(s). Multiple gateways may be specified as a comma-separated list. Here is an example of how to specify multiple gateways: `http://WebRTC1.genesyslab.com:8080, http://WebRTC2.genesyslab.com:8080`	None

Optional Parameters

Name	Description	Default
dtls_srtp	Specifies whether DLTS-SRTP should be used for keying	true
noanswer_timeout	When an OFFER has been sent from the web application and an answer has not been received within the timeout period specified by this parameter, the JavaScript API logs this. If a corresponding event handler exists in the web application, the JavaScript API also sends an event to the web application.	60000 (in milliseconds)
sip_password	SIP Server authentication password used for the user specified with `sip_username`.	None
sip_username	User name (String) to use when generating authentication credentials for SIP Server.	None
stun_server	STUN server (String) used for public IP address discovery. You can specify multiple STUN servers delimited by commas. Example values: `stun.genesyslab.com:3478`, or `stun1.genesyslab.com:3478,stun2.genesyslab.com:3478`	None
turn_password	TURN password (String) used for the given TURN username.	None
turn_server	TURN server URL (String) used for media relay. You can specify multiple TURN server URLs delimited by commas. When using this, you should also set `turn_username` and `turn_password`.	None

Name	Description	Default
	Example value: `turn.genesyslab.com:443,turn.genesyslab.com:443?transport=tcp`	
turn_username	TURN username (String) used for the given TURN server.	None
ice_timeout	Specifies the timeout value for ICE candidate gathering. This timeout is particularly helpful for hosts with multiple virtual interfaces. Minimum value is 1000 ms.	3000 ms
ice_optimization	This parameter is experimental; do not change the default value.	false
polling_timeout	Specifies the maximum time that the HTTP hanging-GET (long-polling) connection waits for a gateway response before timing out, and re-initiating a new hanging-GET. This timeout helps to detect a TCP connection problem faster.	30000 (in milliseconds)

Instance Methods

connect ( )

Connects the client to the WebRTC Gateway. The client is signed in to the gateway anonymously, using an automatically generated ID. Calling this method on an already signed in client will sign out the client, before signing it in again with a new anonymous ID.

If the operation is successful, the `onConnect` event is fired. You should set up a callback before calling this method.

If the operation is unsuccessful, the `onFailed` event is fired. You should set up a callback before calling this method.

Example

```
var grtcClient = new Grtc.Client(configuration);
// set the callback to handle the event when connection is successful
grtcClient.onConnect.add(onConnectCallback);
// set the call back to handle the event when connection fails
grtcClient.onFailed.add(onFailedCallback);
// now the client tries to connect to the gateway anonymously
grtcClient.connect();
```

disableMediaSource

Disables the local media source. For example, it can be used to stop the user's web camera.

Example

```
// in this example, we set the callback of “onDisconnect” to
// properly stop the media source (camera and/or microphone)
grtcClient.onDisconnect.add(function () {
    grtcClient.disableMediaSource();
});
grtcClient.disconnect();
```

disconnect ( )

Disconnects from the WebRTC Gateway. If registered to SIP Server, then the user is also unregistered from SIP Server.

If the operation is successful, the `onDisconnect` event handler is called. You should set up a callback before calling this method.

Throws: INVALID_STATE_ERROR

If the user is not connected yet, an exception is thrown.

Example

```
grtcClient.onDisconnect.add(handleOnDisconnect);
grtcClient.disconnect()
```

enableMediaSource ([audioConstraints] [,videoConstraints])

Asks the user to enable access to a local media source, such as a local camera or a microphone. Calling this method may cause the browser to invoke a user dialog for selecting local devices. If the client application only wants to create an audio call, then the client application should only ask for the microphone and not a webcam. This can be controlled by setting the optional arguments.

If the operation is successful, the `onMediaSuccess` event is fired. You should set up a callback before calling this method.

If the operation is unsuccessful, the `onMediaFailure` event is fired. You should set up a callback before calling this method.

Example

```
// set the callback to handle the success event
grtcClient.onMediaSuccess.add(function (obj) {
    // on success, attach the media to an HTML5 video element
    grtcClient.setViewFromStream(document.getElementById("localView"), obj.stream);
});
// set the callback to handle the failure event
grtcClient.onMediaFailure.add(function (obj) {
    window.alert(obj.message);
});
```

```
// enable local media source, with audio set to true, and video
// set by a constraint object where the width of video is specified
grtcClient.enableMediaSource(true, { width : {ideal : 1280 } });
```

Arguments

- **audioConstraints (optional)**—boolean value indicating whether audio is enabled; or an object that specifies more complex constraints. Default value is `true`.
- **videoConstraints (optional)**—boolean value indicating whether video is enabled; or an object that specifies more complex constraints. Default value is `true`.

The format of the constraint objects is specified by the W3C document on Media Capture and Streams. For more information, see https://www.w3.org/TR/mediacapture-streams/#idl-def-MediaStreamConstraints

Tip

The getUserMedia constraints are matched with the following list of resolutions that are independent of the resolutions supported by a particular camera. This list is fixed and is used on all platforms.

- 1280 x 720
- 960 x 720
- 640 x 360
- 640 x 480
- 320 x 240
- 320 x 180

Throws: WEBRTC_NOT_SUPPORTED_ERROR

This method throws an exception if the browser does not support WebRTC.

filterICECandidates

Filters out unneeded ICE candidates from the candidate list and returns the ones that should be sent to the remote peer in an offer or answer message.

Note that the default implementation will not filter any candidates (and will therefore return all candidates). However, this method can be overwritten by the user with their own implementation.

Example

```
var grtcClient = new Grtc.Client(conf);
...
// Redefine grtcClient.filterIceCandidates()
// to discard candidates that may delay ICE.
```

```
grtcClient.filterIceCandidates = function (Candidates) {
    outCandidates = [];
    var count = Candidates.length;
    for (var i = 0; i < count; i++) {
        var strCandidate = JSON.stringify(Candidates[i]);
        // Ignore private addresses, which are not necessary and
        // seem to add delay. Also ignore TCP candidates that
        // are not used.
        if (strCandidate.match(/ 192\.168\.\d{1,3}\.\d{1,3} \d+ typ host/i) === null &&
            strCandidate.match(/ tcp \d+/i) === null) {
            outCandidates.push(Candidates[i]);
        }
    }
    return outCandidates;
};
```

register (regDN)

Sends a registration request to the WebRTC Gateway on a particular DN, signing in the client using that DN. The gateway will also do the registration with the SIP Server by sending a `SIP REGISTER` request to it. Calling this method on an already signed in client will sign out the client first, before signing it in again.

Calling this method is optional for the user. The user may connect to the gateway anonymously using the connect() method instead.

If the operation is successful, the `onRegister` event is fired. You should set up a callback before calling this method.

If the operation is unsuccessful, the `onFailed` event is fired. You should set up a callback before calling this method.

Arguments

- **regDN (optional)**—DN for registering with SIP Server. If not specified, the configuration parameter `sip_username` will be used for the DN, optionally along with `sip_password` for authentication.

Example

```
var grtcClient = new Grtc.Client(configuration);
// set the callback to handle the event when registration is successful
grtcClient.onRegister.add(onRegisterCallback);
// set the call back to handle the event when registration fails
grtcClient.onFailed.add(onFailedCallback);
// now the client tries to register as DN 1020
grtcClient.register("1020");
```

setVideoBandwidth(bandwidth)

Sets the bandwidth for sending video data by adding a line for video, `b=AS:<bandwidth>`, in SDP. Setting the rate too low will cause connection attempts to fail. The default is `500 kbps`. A value of zero will remove bandwidth limits. Note that setting this may not work with Firefox.

Arguments

- **bandwidth**—value in kbps

Example

```
var grtcClient = new Grtc.Client(configuration);
...
grtcClient.setVideoBandwidth(300);
```

setViewFromStream (viewObject, stream)

Sets a DOM object as the container of a media stream. This method can be called after a `Grtc.Client.OnMediaSuccess` or `Grtc.MediaSession.onRemoteStream` event is handled for a local or remote stream, respectively.

Arguments

- **viewObject**—a DOM object (for example, an HTML5 `video` or `audio` object)
- **stream**—a local or remote stream to be attached

Throws: NOT_READY_ERROR

An error is thrown if the stream is not ready for rendering or if the wrong DOM object has been passed.

Example

The code sample in the `enableMediaSource()` section shows an example of `setViewFromStream` being called when the `onMediaSuccess` event is handled. That example shows how to port a local media stream. The following example shows how to port a remote media stream.

```
// register a handler when remote stream is available
grtcSession.onRemoteStream.add(function (data) {
    grtcClient.setViewFromStream(document.getElementById("remoteView"), data.stream);
})
```

setMediaConstraintsForOffer(audioConstraints [,videoConstraints])

Sets the constraint values used when making a call or a new offer. The default values for both of these constraints are `true`. Although these constraints can be specified when accepting a call (without an SDP offer), manually using the `acceptCall()` method or when making a call/offer using the `makeOffer()` method of `Grtc.MediaSession`, this method may be necessary when incoming calls without an SDP offer are to be auto-answered using the `talk` event, or when the JSAPI responds to incoming invite for offers that come in after a call is established.

Arguments

- **audioConstraints (optional)**—boolean value indicating whether audio is enabled; default value is `true`.

- **videoConstraints (optional)**—boolean value indicating whether video is enabled; default value is `true`.

setMediaConstraintsForAnswer(audioConstraints [,videoConstraints])

Sets the constraint values used when answering a call or a new offer. The default values for both of these constraints are `true`. Although these constraints can be specified when accepting a call manually using the `acceptCall()` method of `Grtc.MediaSession`, this method may be necessary when incoming calls with SDP offer are to be auto-answered using the `talk` event, or when incoming offers after a call is established are responded to by the JSAPI.

Arguments

- **audioConstraints (optional)**—boolean value indicating whether audio is enabled; default value is `true`.
- **videoConstraints (optional)**—boolean value indicating whether video is enabled; default value is `true`.

setLogLevel (level)

Set a log level using one defined in `Grtc`. The default level is `LOG_LEVEL_INFO`, and some logging will still take place during WebRTC JavaScript API initialization regardless of the log level.

Arguments

level—the log level to use, which should be one of the following:

- `LOG_LEVEL_NONE`—no logging will be done during a call.
- `LOG_LEVEL_ERROR`—only errors will be logged.
- `LOG_LEVEL_NOTICE`—basic operations will be logged, in addition to errors.
- `LOG_LEVEL_INFO`—more information will be logged, in addition to what is logged at level `NOTICE`.
- `LOG_LEVEL_DEBUG`—even more detailed information will be logged, in addition to what is logged at level `INFO`.

The following log methods are defined, which log the given message (and the exception, if given), but only if the current log level is equal or above the level corresponding to the method:

```
logError( message [, exception] ),
logNotice( message [, exception] ),
logInfo( message [, exception] ),
logDebug( message [, exception] )
```

Arguments

- **message**—a string to be logged.
- **exception** (optional)—an exception object, of which the message field will be logged following the "message" string.

Events

onConnect

This event is triggered when the client opens a connection to the WebRTC Gateway.

See the `connect()` API for an example of how to handle this event.

Event data parameters

- message—A message string describing the event

onDisconnect

This event is triggered when the client closes the connection to the WebRTC Gateway.

See the `connect()` API for an example of how to handle this event.

Event data parameters

- message—A message string describing the event

onFailed

This event is triggered when the WebRTC Gateway returns a failure response. This could be due to a connection or registration failure with the gateway. This event is also fired if multiple gateways have been configured and a failover is attempted.

See the `connect()` and `register()` APIs for examples of how to handle this event.

Event data parameters

- message—A message string describing the failure

onGatewayError

This event is triggered when an error message is received from the WebRTC Gateway.

Event data parameters

- error—The specific error type

onIceDisconnected

This event is triggered when it is detected that the ICE connection with the WebRTC gateway is disconnected, likely due to a temporary network issue. To retry establishing the ICE connection, the application should initiate SDP renegotiation in the handler for this event by calling the `makeOffer()`

method of `Grtc.MediaSession`, with no arguments. Note that the `onConnectionError` event handler should not do anything in this case, except perhaps notify the user that there is some connection issue. The hanging GET failure that triggered the `onConnectionError` event will not stop the JSAPI from retrying the hanging GET, which will take place after a three (3) second delay, and will be repeated.

Event data parameters

- status—a message string indicating the error

Example

```
    // Invoked on ICE failure after ICE is established, due to a network problem.
    // The app should try to re-establish ICE by initiating an offer to the gateway.
    // Note, if sending offer fails, JSAPI will retry until it succeeds or session closes.
    grtcClient.onIceDisconnected.add(function (obj) {
        if (grtcSession)
        {
            console.log("Trying to restore ICE connection...");
            grtcSession.makeOffer();
        }
        return false;
    });
```

onIncomingCall

This event is triggered when the an incoming call is received by the client. The client is expected to handle this event by informing the user of the incoming call and, if the user so authorizes, constructing a session and calling `Grtc.MediaSession.acceptCall()` to establish a call session.

If calls are to be automatically answered using the talk event (this is achieved by appropriately setting the `sip-cti-control` parameter in the `TServer` section of the DN), this event does not need to be handled by the client.

Event data parameters

- peer—The name of the remote peer

Example

```
grtcClient.onIncomingCall.add(function (data) {
    // create a session
    var grtcSession = new Grtc.MediaSession(grtcClient);
    // inform user of incoming call
    var userResponse = window.confirm(“Accept call from “ + data.peer + “?”);
    if (userResponse === true) {
        grtcSession.acceptCall();
    } else {
        grtcSession.rejectCall();
    }
});
```

onInfoFromPeer

This event is triggered when the client receives mid-call data from the remote peer.

Event data parameters

- data—a simple object containing the data received from the peer

Example

This example assumes that `client1` is sending mid-call data to `client2` using `sendInfo` and `client2` is handling the `onInfoFromPeer` event.

```
var grtcClient1 = new Grtc.Client(configuration);
var grtcSession1 = new Grtc.MediaSession(grtcClient1);
...
var grtcClient2 = new Grtc.Client(configuration);
var grtcSession2 = new Grtc.MediaSession(grtcClient2);
// ---- session established between client1 and client2 ----
...
// ---- client1 sends data to client2 ----
var data = {};
data.param1 = value1;
data.param2 = value2;
grtcSession1.sendInfo(data, false);
...
// ---- client2 processes data from client1 ----
grtcClient2.onInfoFromPeer.add(function (data) {
        alert("Got data from client1:\n" + JSON.stringify(data));
        return false;
});
```

onInvite

This event is obsolete and has been replaced by the `onIncomingCall` event.

This event is triggered when an INVITE message is received by the client. The client is expected to handle this event by informing the user of the incoming INVITE and, if the user so authorizes, constructing a session and calling `Grtc.MediaSession.makeCall()` to initiate a call session (sending an OFFER).

It is important to note the difference between this event and the `onIncoming` event. The `onIncoming` event occurs when there is an incoming OFFER. The handler of that event is expected to invoke `acceptCall`. For the `onInvite` event, no OFFER has been made yet, so the handler of this event is expected to initiate an OFFER by invoking `makeCall`.

Event data parameters

- peer—The name of the remote peer

Example

```
grtcClient.onInvite.add(function (data) {
    // create a MediaSession instance and make a call on it
    var grtcSession = new Grtc.MediaSession(grtcClient);
    grtcSession.onRemoteStream.add(function (s) {
        grtcClient.setViewFromStream(document.getElementById("remoteView"), s.stream);
    });
    grtcSession.makeCall(data.peer);
});
```

onMediaSuccess

This event is triggered when a call to `enableMediaSource()` has been successful.

See the `enableMediaSource()` API for an example of how to handle this event.

Event data parameters

- stream—the media stream

onMediaFailure

This event is triggered when a call to `enableMediaSource()` has failed.

See the `enableMediaSource()` API for an example of how to handle this event.

Event data parameters

- message—A message string describing the failure

onPeerNoanswer

This event is triggered to notify the client that no answer has been received from the peer after an offer was sent. Triggering of this event is controlled by the `noanswer_timeout` configuration parameter.

Example

```
var grtcClient = new Grtc.Client(configuration);
...
var grtcSession = new Grtc.MediaSession(grtcClient);
...
grtcClient.onPeerNoanswer.add(function ()
{ // The remote site did not send an answer within the specified
  // timeout interval. The call should be terminated from the
  // caller's end
grtcSession.terminateCall();
  // Other processing may be carried out, as well,
  // for example, a call status update on the web page }
);
```

onPeerClosing

This event is triggered when the WebRTC Gateway detects that the peer has closed. You should normally handle this event in order to clean up after the web application, for example, by resetting the call status, enabling or disabling buttons, informing the user of the situation, and so on.

Example

```
// when the peer closes, the onPeerClosing event will be fired;
// so add a handler to do some work if that happens
grtcClient.onPeerClosing.add(function () {
    document.getElementById("remoteView").style.opacity = 0;
    $("#ghcc-call-from").empty();
    if (grtcSession) {
        grtcSession = null;
    }
})
```

onRegister

This event is triggered when the client successfully registers with SIP Server.

See the `register()` API for an example of how to handle this event.

Event data parameters

- message—A message string describing the event

<references />

onNotifyEvent

This event is triggered when a valid `talk` or `hold` event is received. The client may not need to handle this, but can use it for informing the user of the call status. The first `talk` event indicates that the call has been answered. Any further `hold` or `talk` event should happen in pairs, in that order, and indicates that the call is held or resumed, respectively.

Event data parameters

- peer—peer ID
- event—`talk` or `hold`

Example

```
grtcClient.onNotifyEvent.add(function (data) {
    if (data.event === "talk") {
        console.log("The call is active");
    }
    else if (data.event === "hold") {
        console.log("The call is on-hold");
    }
    return false;
});
```

onWebrtcError

This event is triggered when an error happens with a WebRTC API call of the browser.

Event data parameters

- error—A message string describing the error

Example

```
grtcClient.onWebrtcError.add(function (obj)  {
    alert("Got WebRTC error: " + JSON.stringify(obj));
    return false;
});
```

onStatsFromServer

This event is triggered when call statistics are received from the gateway. It typically happens due to a request from the client. However, call statistics can be automatically sent by the gateway at certain points of a call, such as call end.

Event data parameters

One is passed in, and it is a multi-layered object containing the various call related statistics,. The following is an example, which describes the format of this object.

Example

```
{
  "timestamp":1417536926232, "duration":4727016,
  "audio":{
    "leg_id":101,"codec":"G.711/mu-law",
    "rx":{
      "ssrc":3439377433,"payload":0,
      "packets":236195,"bytes":40625540,"errors":0, "rtcp":947,"rtcp_errors":0,
      "lost":15,"jitter":20,"max_jitter":73
    },
    "tx":{
      "ssrc":1612950091,"payload":0,
      "packets":236194,"bytes":40625368,"errors":0,"rtcp":945,"rtcp_errors":0
    },
    "client":{"packets":235917,"lost":70,"jitter":30,"max_jitter":111},
    "rtt":67,"rtt_max":82,"rtt_average":64,"rtt_count":938
  },
  "video":{
    "leg_id":102,"codec":"VP8",
    "rx":{
      "ssrc":0,"payload":100,
      "packets":0,"bytes":0,"errors":0,"rtcp":0,"rtcp_errors":0,
      "lost":0,"jitter":0,"max_jitter":0
    },
    "tx":{
      "ssrc":1109142291,"payload":100,
      "packets":305858,"bytes":299216448,"errors":0,"rtcp":0,"rtcp_errors":0
    },
    "client":{"packets":0,"lost":0,"jitter":0,"max_jitter":0},
    "rtt":0,"rtt_max":0,"rtt_average":0,"rtt_count":0
  }
}
```

The timestamp is the current Epoch time in milliseconds (ms), and the duration is the call duration in ms. There will be an audio and/or video object, but only if there is a corresponding media stream (it can be uni-directional, however; for example, in the preceding example, video is `sendonly` for the gateway). Each media object has the following information:

- leg ID
- codec name
- receive statistics
- transmit statistics
- client statistics
- RTT (Round-Trip Time) values

The latter two sets are obtained using the RTCP messages received from the browser. RTT values contain the last value, maximum, average, and count of values that were used for average and maximum. In all cases, `packets` is the number of RTP/media packets that were successfully received or sent, `rtcp` is the number of RTCP packets that were received or sent, `lost` is the number of packets that were not received, and `jitter` is the RTP packet inter-arrival jitter in RTP timestamp units calculated according to `RFC 3550`.

Using the bytes and timestamp information in these stats, the audio/video bit-rate can be computed by getting two samples in sufficiently large time intervals, say from 1s to 5s. RTT values, jitter, and number of lost packets can be used to get an idea about the call quality and network condition.

onConnectionError

This event is triggered when an HTTP hanging GET request to the WebRTC Gateway fails (and in WebRTC JavaScript API versions 8.5.210.25 and earlier, no more hanging-GETs will be initiated automatically). This typically happens when the Gateway goes down. This error may be handled by alerting the user, and/or disconnecting, or signing in again. Note that, in a High Availability setup, a new sign-in, which also starts the hanging GET, is necessary for connecting to a working gateway instance.

Event data parameters

- status—A string indicating the error.

Example

```
    // Invoked on connection hanging get error, usually when gateway goes down.
    // The app may retry sign-in in the HA case, or at least, warn the user.
    grtcClient.onConnectionError.add(function (obj) {
        if (grtcClient) {
            alert("Got connection error: " + JSON.stringify(obj));
            grtcClient.disconnect();
            cleanupAfterCall();
        }
        return false;
    });
```

Grtc.Error

Some functions in the `Grtc` namespace can raise exceptions under certain circumstances. Each such exception is an instance of `Grtc.Error`.

Instance Attributes

name

A String representing the error code of the exception. The following error codes are used by `Grtc.Error`:

Code	Constant	Description
1	Grtc.CONFIGURATION_ERROR	Error while processing configuration information
2	Grtc.CONNECTION_ERROR	Error while connecting to the WebRTC Gateway
3	cGrtc.WEBRTC_NOT_SUPPORTED_ERROR	Browser does not support WebRTC functionality
4	Grtc.INVALID_STATE_ERROR	The client is in an invalid state for executing the given action
5	Grtc.NOT_READY_ERROR	The client is not ready to process this action
6	Grtc.GRTC_ERROR	Generic error not covered by other error codes
7	Grtc.GRTC_WARN	Generic warning
8	Grtc.WEBRTC_ERROR	Error on a WebRTC API call

message

Optional String describing the exception.

In the rest of this API Reference, when we say a method throws an error such as `NOT_READY_ERROR`, that is a shorter way of saying that the following code has been executed:

```
throw new Grtc.Error(Grtc.NOT_READY_ERROR, “some message”);
```

If there is a possibility that an API call might throw an exception, the web app is expected to catch it and do the relevant error processing, such as informing the user of the condition, presenting a status update on the web page, or logging the error message.

```
try {
    grtcClient.connect();
} catch (e) {
    // e may be an instance of Grtc.Error, where e.name is Grtc.CONNECTION_ERROR
    console.log(e.name);
}
```

Grtc.MediaSession

This class creates a peer connection. Once a user is signed in to the WebRTC Gateway, he or she can both initiate and receive calls by using the `Grtc.MediaSession` class.

Instantiation

To create a new `Grtc.MediaSession` object, pass the `Grtc.Client` object as an argument.

Example

```
var grtcClient = new Grtc.Client(configuration);
var grtcSession = new Grtc.MediaSession(grtcClient);
```

Instance Methods

acceptCall ([audioConstraints] [,videoConstraints])

Accepts the incoming call, with the possibility of accepting only audio or video mode by setting an optional parameter. By default, it is assumed that both audio and video should be enabled.

Arguments

- **audioConstraints (optional)**—boolean value indicating whether audio is enabled or not; default value is `true`.
- **videoConstraints (optional)**—boolean value indicating whether video is enabled or not; default value is `true`.

Throws: INVALID_STATE_ERROR

If there is no incoming call, an exception is thrown.

Example

```
grtcClient.onIncomingCall.add(function (data) {
    // create a session
    var grtcSession = new Grtc.MediaSession(grtcClient);
    // inform user of incoming call
    var userResponse = window.confirm("Accept call from " + data.peer + "?");
    if (userResponse === true) {
        // accept the call
        grtcSession.acceptCall();
        // process attached data if available
        var dataAttached = grtcSession.getData();
        if (dataAttached) {
            // dataAttached is an array of objects, each of which
            // has 2 properties "key" and "value"
            for (var i=0; i<dataAttached.length; ++i) {
                var obj = dataAttached[i];
                console.log("dataAttached[" + i + "]:" + obj.key + ", " + obj.value);
            }
        }
    } else {
        grtcSession.rejectCall();
    }
});
```

closeSession (sendBye)

Closes the ongoing media session on the client side, and also sends BYE to the remote peer if requested. The method terminateCall() is now obsolete, and instead developers should use closeSession (true) as a substitute for terminateCall().

Arguments

sendBye(optional)—boolean value which, if set to `true`, sends a BYE request to the remote peer, as well as closing the current media session on the client side. The default value is `false`.

Example

```
grtcSession.closeSession(true);
```

getData ( )

Get the JSON object attached to a `Grtc.MediaSession`.

The data retrieved is a JavaScript array with the same format as the data array passed to `setData(data)`. This means that the data in the array can be accessed using a loop.

Throws: GRTC_ERROR

If the data cannot be retrieved, an exception is thrown.

Example

```
var dataAttached = grtcSession.getData();
if (dataAttached) {
    // dataAttached is an array of objects, each of which
    // has 2 properties "key" and "value"
    for (var i=0; i<dataAttached.length; ++i) {
        var obj = dataAttached[i];
        console.log("dataAttached[" + i + "]:" + obj.key + ", " + obj.value);
    }
}
```

makeCall (remoteId, [audioConstraints] [,videoConstraints])

This method has been deprecated and is replaced with the `makeOffer` method.

Initiates an audio or video call to another user. The user can also make audio-only or video-only calls by setting an optional parameter.

Arguments

- **remoteId**—the ID of the other user (string).
- **audioConstraints (optional)**—boolean value indicating whether audio is enabled or not; default value is `true`.
- **videoConstraints (optional)**—boolean value indicating whether video is enabled or not; default value is `true`.

Example

```
// create a new session (passing the client object as argument)
var grtcSession = new Grtc.MediaSession(grtcClient);
// set callback to port remote media stream when it comes
grtcSession.onRemoteStream.add(function (data) {
    grtcClient.setViewFromStream(document.getElementById("remoteView"), data.stream;
});
// attach data if available: the data is an array of objects
// each object contains two properties named “key” and “value”
var dataToAttach = [
    {
        "key": "Name",
        "value": $('#login_name').val()
    },
    {
        "key": "Phone",
        "value": $('#phone_number').val()
    },
    {
        "key": "Email",
        "value": $('#email').val()
    }
];
grtcSession.setData(dataToAttach);
// preparation is ready, now make the call
grtcSession.makeCall(“1021”);
```

makeOffer(remoteId, [audioConstraints [,videoConstraints [,holdMedia]]])

Initiates an audio or video call to a peer, or update an existing call. The media types used and the directions of the media in the call depends on the local media stream(s) enabled, as well as the constraints used in this function call. Note that these optional constraints can be set ahead of time using the `Grtc.Client` method `setMediaConstraintsForOffer()`.

Note: This function replaces `makeCall`.

Arguments

- **remoteId**—the ID of the other user (string). If the value is set to `self` when a new call is made, the call becomes a loopback call with the WebRTC Gateway, which may be useful for testing purposes. In this case, no interaction takes place with the SIP-side, and the media sent by the browser is sent back to it. Note that the audio/video constraints specified here for the loopback call has to match the local media stream, since uni-directional media such as one with the `recvonly` attribute would not make sense.
- **audioConstraints (optional)**—boolean value indicating whether audio is enabled or not; default value is `true`.
- **videoConstraints (optional)**—boolean value indicating whether video is enabled or not; default value is `true`.
- **holdMedia** (optional)—if `true`, sets the media lines in the SDP to inactive in order to put the call on hold.

rejectCall ( )

Reject the incoming call.

Throws: INVALID_STATE_ERROR

If there is no incoming call, an exception is thrown.

Example

See the `acceptCall ( [audioConstraints] [,videoConstraints] )` example above.

sendDTMF ( )

Sends one or more DTMF tones from among the following possible values:

- [0-9]
- *
- #
- A, B, C, D

Important
For Firefox, supported in version 53 and higher only.

Arguments

- **tones**—String composed of one or multiple valid DTMF symbols

- **options**—Available options, including:
    - **duration**—The duration of the DTMF tones sent by this method. The default is 100 ms. The duration setting cannot be greater than 6000 ms or less than 70 ms.
    - **tonegap**—The gap between tones. It must be at least 50 ms. The default value is 50 ms.

Returns

- **0**—Success
- **-1**—Failure

Example

```
// This function uses our sendDTMF API to send DTMF tones
// for an already-created mediaSession object
function sendTone(tones){
  var options = {"duration": 500, "tonegap": 50};
  console.log("DTMF send result: [" + mediaSession.sendDTMF(tones, options) + "]");
}

// This function creates a simple DTMF Dial Pad for the HTML page
function createDialingPad() {
  var tones = '1234567890*#ABCD';
  var dialingPad = document.getElementById('dialingPad');
  for (var i = 0; i < tones.length; ++i) {
    var tone = tones.charAt(i);
    dialingPad.innerHTML += '<button id="' +
      tone + '" onclick="sendTone(\'' + tone +
      '\')" style="height:40px; width: 30px">' + tone + '</button>';
    if ((i + 1) % 4 == 0) {
      dialingPad.innerHTML += '<br>';
    }
  }
}
```

sendInfo (data)

Sends a mid-call ROAP INFO message to the WebRTC Gateway with the specified input data. The gateway, in turn, sends a SIP INFO message to the SIP Server with the data in the body of the message. The input data should be a simple object. A serialized representation of the data is created in URL query string format before the data is sent to the Gateway.

Arguments

- **data**—A simple object
- **mapData**—boolean value. When `mapData` is set to `true` or is undefined, the content-type of the SIP INFO message (from the Gateway to the SIP Server) is set to `application/x-www-form-urlencoded`. In this case, the SIP Server consumes the data and maps it to the corresponding T-Library events. When `mapData` is set to `false`, the content-type for the SIP INFO message is set to `application/octet-stream` and SIP Server simply passes the message to the remote end, which is another WebRTC or SIP EndPoint.

Throws: GRTC_ERROR

If the data is not well-formed, an exception is thrown.

Example

```
var data = {};
data.visitid = 'abc123';
data.token = '1234';
grtcSession.sendInfo(data, true);
```

setData (data)

Set a JSON object as the data to be attached to a `Grtc.MediaSession`.

Arguments

data—A well-formed JavaScript array of one or more objects, in which each object contains exactly two properties: key and `value`, where key is the identifier of the piece of data to be attached, and `value` is the value of the piece of data.

Throws: GRTC_ERROR

If the data is not well formed or cannot be attached, an exception is thrown.

Example

```
var dataToAttach = [
    {
        "key": "Name",
        "value": “My Name”
    },
    {
        "key": "Phone",
        "value": “800-555-1212”
    },
    {
        "key": "Email",
        "value": “my.address@somewhere.com”
    }
];
grtcSession.setData(dataToAttach);
```

terminateCall ( )

Terminates an existing media session.

Throws: INVALID_STATE_ERROR

If there is no active call, an exception is thrown.

Example

```
grtcSession.terminateCall();
```

updateCall (audioEnabled ,videoEnabled)

Updates the call in progress, muting or unmuting audio and/or video track, by setting the corresponding enabled flags.

Arguments

- **audioEnabled**—boolean value indicating whether audio is enabled; if not boolean or not specified, it will be ignored.
- **videoEnabled**—boolean value indicating whether video is enabled; if not boolean or not specified, it will be ignored.

Throws: INVALID_STATE_ERROR

If there is no active call, an exception is thrown.

holdCall()

Puts an active call on hold.

resumeCall()

Resumes a call from hold.

getServerStats()

Makes a request to the WebRTC Gateway for call statistics. When the statistics are received from the Gateway, JSAPI triggers the `Grtc.Client onStatsFromServer` event along with the data.

hasAudioEnabled()

Determines whether a call has audio enabled after an SDP negotiation with the peer.

Returns

Boolean (true/false)

hasVideoEnabled()

Determines whether a call has video enabled after an SDP negotiation with the peer.

Returns

Boolean (true/false)

isEstablished()

Returns true if a call session is established, and no renegotiation is occurring.

Events

onRemoteStream

This event is triggered when a remote stream is added to the peer connection. The user is expected to handle this event in order to deal with the received stream, for example, to embed the stream in an HTML5 <video> element.

Event data parameters

- stream—the remote stream

onSessionHold

This event is triggered when an active call goes into hold/inactive status. The application can use this event to update the call status. Note that, when the call becomes active again, the `onRemoteStream` event will be triggered.

Event data parameters

- isTrue—Set to true when the call is on hold. Currently, this event is not triggered with a value of "false" when the call becomes active again, although this can change in the future.