Web Real-Time Communications Developer's Guide

Grtc.Client

12/14/2025

Contents

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;
```

```
});
```