aiortc

Jeremy Lainé

Mar 11, 2024

CONTENTS

1 Why should I use `aiortc`? **3**
1.1 Examples . . . 3
1.2 API Reference . . . 3
1.3 Helpers . . . 17
1.4 Contributing . . . 19
1.5 Changelog . . . 20
1.6 License . . . 35

Python Module Index **37**

Index **39**

`aiortc` is a library for Web Real-Time Communication (WebRTC) and Object Real-Time Communication (ORTC) in Python. It is built on top of `asyncio`, Python's standard asynchronous I/O framework.

The API closely follows its Javascript counterpart while using pythonic constructs:

- promises are replaced by coroutines
- events are emitted using `pyee.EventEmitter`

CHAPTER

ONE

WHY SHOULD I USE AIORTC?

The main WebRTC and ORTC implementations are either built into web browsers, or come in the form of native code. While they are extensively battle tested, their internals are complex and they do not provide Python bindings. Furthermore they are tightly coupled to a media stack, making it hard to plug in audio or video processing algorithms.

In contrast, the `aiortc` implementation is fairly simple and readable. As such it is a good starting point for programmers wishing to understand how WebRTC works or tinker with its internals. It is also easy to create innovative products by leveraging the extensive modules available in the Python ecosystem. For instance you can build a full server handling both signaling and data channels or apply computer vision algorithms to video frames using OpenCV.

Furthermore, a lot of effort has gone into writing an extensive test suite for the `aiortc` code to ensure best-in-class code quality.

1.1 Examples

`aiortc` comes with a selection of examples, which are a great starting point for new users.

The examples can be browsed on GitHub:

https://github.com/aiortc/aiortc/tree/main/examples

1.2 API Reference

1.2.1 WebRTC

class aiortc.**RTCPeerConnection**(*configuration=None*)

The *RTCPeerConnection* interface represents a WebRTC connection between the local computer and a remote peer.

Parameters

configuration (Optional[*RTCConfiguration*]) – An optional *RTCConfiguration*.

property connectionState: str

The current connection state.

Possible values: *"connected"*, *"connecting"*, *"closed"*, *"failed"*, *"new"*.

When the state changes, the *"connectionstatechange"* event is fired.

property iceConnectionState: `str`

The current ICE connection state.

Possible values: *"checking"*, *"completed"*, *"closed"*, *"failed"*, *"new"*.

When the state changes, the *"iceconnectionstatechange"* event is fired.

property iceGatheringState: `str`

The current ICE gathering state.

Possible values: *"complete"*, *"gathering"*, *"new"*.

When the state changes, the *"icegatheringstatechange"* event is fired.

property localDescription: *RTCSessionDescription*

An *RTCSessionDescription* describing the session for the local end of the connection.

property remoteDescription: *RTCSessionDescription*

An *RTCSessionDescription* describing the session for the remote end of the connection.

property sctp: *RTCSctpTransport* | `None`

An *RTCSctpTransport* describing the SCTP transport being used for datachannels or *None*.

property signalingState

The current signaling state.

Possible values: *"closed"*, *"have-local-offer"*, *"have-remote-offer"*, *"stable"*.

When the state changes, the *"signalingstatechange"* event is fired.

await addIceCandidate(*candidate*)

Add a new *RTCIceCandidate* received from the remote peer.

The specified candidate must have a value for either *sdpMid* or *sdpMLineIndex*.

Parameters

candidate (*RTCIceCandidate*) – The new remote candidate.

Return type

`None`

addTrack(*track*)

Add a *MediaStreamTrack* to the set of media tracks which will be transmitted to the remote peer.

Return type

RTCRtpSender

addTransceiver(*trackOrKind*, *direction='sendrecv'*)

Add a new *RTCRtpTransceiver*.

Return type

RTCRtpTransceiver

await close()

Terminate the ICE agent, ending ICE processing and streams.

await createAnswer()

Create an SDP answer to an offer received from a remote peer during the offer/answer negotiation of a WebRTC connection.

Return type

RTCSessionDescription

createDataChannel(*label*, *maxPacketLifeTime=None*, *maxRetransmits=None*, *ordered=True*, *protocol=''*, *negotiated=False*, *id=None*)

Create a data channel with the given label.

Return type
RTCDataChannel

await createOffer()

Create an SDP offer for the purpose of starting a new WebRTC connection to a remote peer.

Return type
RTCSessionDescription

getReceivers()

Returns the list of *RTCRtpReceiver* objects that are currently attached to the connection.

Return type
List[*RTCRtpReceiver*]

getSenders()

Returns the list of *RTCRtpSender* objects that are currently attached to the connection.

Return type
List[*RTCRtpSender*]

await getStats()

Returns statistics for the connection.

Return type
RTCStatsReport

getTransceivers()

Returns the list of *RTCRtpTransceiver* objects that are currently attached to the connection.

Return type
List[*RTCRtpTransceiver*]

await setLocalDescription(*sessionDescription*)

Change the local description associated with the connection.

Parameters
sessionDescription (*RTCSessionDescription*) – An *RTCSessionDescription* generated by *createOffer()* or *createAnswer()*.

Return type
None

await setRemoteDescription(*sessionDescription*)

Changes the remote description associated with the connection.

Parameters
sessionDescription (*RTCSessionDescription*) – An *RTCSessionDescription* created from information received over the signaling channel.

Return type
None

class aiortc.**RTCSessionDescription**(*sdp*, *type*)

The *RTCSessionDescription* dictionary describes one end of a connection and how it's configured.

class aiortc.**RTCConfiguration**(*iceServers=None*)

The *RTCConfiguration* dictionary is used to provide configuration options for an *RTCPeerConnection*.

iceServers: `Optional[List[RTCIceServer]] = None`

A list of *RTCIceServer* objects to configure STUN / TURN servers.

1.2.2 Interactive Connectivity Establishment (ICE)

class aiortc.**RTCIceCandidate**(*component*, *foundation*, *ip*, *port*, *priority*, *protocol*, *type*, *relatedAddress=None*, *relatedPort=None*, *sdpMid=None*, *sdpMLineIndex=None*, *tcpType=None*)

The *RTCIceCandidate* interface represents a candidate Interactive Connectivity Establishment (ICE) configuration which may be used to establish an RTCPeerConnection.

class aiortc.**RTCIceGatherer**(*iceServers=None*)

The *RTCIceGatherer* interface gathers local host, server reflexive and relay candidates, as well as enabling the retrieval of local Interactive Connectivity Establishment (ICE) parameters which can be exchanged in signaling.

property state: `str`

The current state of the ICE gatherer.

await gather()

Gather ICE candidates.

Return type
`None`

classmethod getDefaultIceServers()

Return the list of default *RTCIceServer*.

Return type
`List[RTCIceServer]`

getLocalCandidates()

Retrieve the list of valid local candidates associated with the ICE gatherer.

Return type
`List[RTCIceCandidate]`

getLocalParameters()

Retrieve the ICE parameters of the ICE gatherer.

Return type
RTCIceParameters

class aiortc.**RTCIceTransport**(*gatherer*)

The *RTCIceTransport* interface allows an application access to information about the Interactive Connectivity Establishment (ICE) transport over which packets are sent and received.

Parameters
gatherer (*RTCIceGatherer*) – An *RTCIceGatherer*.

property iceGatherer: *RTCIceGatherer*

The ICE gatherer passed in the constructor.

property role: `str`

The current role of the ICE transport.

Either *'controlling'* or *'controlled'*.

property state: `str`

The current state of the ICE transport.

await addRemoteCandidate(*candidate*)

Add a remote candidate.

Parameters

candidate (Optional[*RTCIceCandidate*]) – The new candidate or *None* to signal end of candidates.

Return type

None

getRemoteCandidates()

Retrieve the list of candidates associated with the remote *RTCIceTransport*.

Return type

List[*RTCIceCandidate*]

await start(*remoteParameters*)

Initiate connectivity checks.

Parameters

remoteParameters (*RTCIceParameters*) – The *RTCIceParameters* associated with the remote *RTCIceTransport*.

Return type

None

await stop()

Irreversibly stop the *RTCIceTransport*.

Return type

None

class aiortc.**RTCIceParameters**(*usernameFragment=None*, *password=None*, *iceLite=False*)

The *RTCIceParameters* dictionary includes the ICE username fragment and password and other ICE-related parameters.

usernameFragment: Optional[str] = None

ICE username fragment.

password: Optional[str] = None

ICE password.

class aiortc.**RTCIceServer**(*urls*, *username=None*, *credential=None*, *credentialType='password'*)

The *RTCIceServer* dictionary defines how to connect to a single STUN or TURN server. It includes both the URL and the necessary credentials, if any, to connect to the server.

urls: str

This required property is either a single string or a list of strings, each specifying a URL which can be used to connect to the server.

username: Optional[str] = None

The username to use during authentication (for TURN only).

credential: Optional[str] = None

The credential to use during authentication (for TURN only).

1.2.3 Datagram Transport Layer Security (DTLS)

class aiortc.**RTCCertificate**(*key*, *cert*)

The *RTCCertificate* interface enables the certificates used by an *RTCDtlsTransport*.

To generate a certificate and the corresponding private key use *generateCertificate()*.

property expires: datetime

The date and time after which the certificate will be considered invalid.

getFingerprints()

Returns the list of certificate fingerprints, one of which is computed with the digest algorithm used in the certificate signature.

Return type
List[*RTCDtlsFingerprint*]

classmethod generateCertificate()

Create and return an X.509 certificate and corresponding private key.

Return type
RTCCertificate

class aiortc.**RTCDtlsTransport**(*transport*, *certificates*)

The *RTCDtlsTransport* object includes information relating to Datagram Transport Layer Security (DTLS) transport.

Parameters

- **transport** (*RTCIceTransport*) – An *RTCIceTransport*.
- **certificates** (List[*RTCCertificate*]) – A list of *RTCCertificate* (only one is allowed currently).

property state: str

The current state of the DTLS transport.

One of *'new'*, *'connecting'*, *'connected'*, *'closed'* or *'failed'*.

property transport

The associated *RTCIceTransport* instance.

getLocalParameters()

Get the local parameters of the DTLS transport.

Return type
RTCDtlsParameters

await start(*remoteParameters*)

Start DTLS transport negotiation with the parameters of the remote DTLS transport.

Parameters
remoteParameters (*RTCDtlsParameters*) – An *RTCDtlsParameters*.

Return type
None

await stop()

Stop and close the DTLS transport.

Return type
None

class aiortc.**RTCDtlsParameters**(*fingerprints=<factory>*, *role='auto'*)

The *RTCDtlsParameters* dictionary includes information relating to DTLS configuration.

fingerprints: List[*RTCDtlsFingerprint*]

List of *RTCDtlsFingerprint*, one fingerprint for each certificate.

role: str = 'auto'

The DTLS role, with a default of auto.

class aiortc.**RTCDtlsFingerprint**(*algorithm*, *value*)

The *RTCDtlsFingerprint* dictionary includes the hash function algorithm and certificate fingerprint.

algorithm: str

The hash function name, for instance *'sha-256'*.

value: str

The fingerprint value.

1.2.4 Real-time Transport Protocol (RTP)

class aiortc.**RTCRtpReceiver**(*kind*, *transport*)

The *RTCRtpReceiver* interface manages the reception and decoding of data for a *MediaStreamTrack*.

Parameters

- **kind** (str) – The kind of media (*'audio'* or *'video'*).
- **transport** (*RTCDtlsTransport*) – An *RTCDtlsTransport*.

property track: *MediaStreamTrack*

The *MediaStreamTrack* which is being handled by the receiver.

property transport: *RTCDtlsTransport*

The *RTCDtlsTransport* over which the media for the receiver's track is received.

classmethod getCapabilities(*kind*)

Returns the most optimistic view of the system's capabilities for receiving media of the given *kind*.

Return type
RTCRtpCapabilities

await getStats()

Returns statistics about the RTP receiver.

Return type
RTCStatsReport

getSynchronizationSources()

Returns a *RTCRtpSynchronizationSource* for each unique SSRC identifier received in the last 10 seconds.

Return type
List[*RTCRtpSynchronizationSource*]

await receive(*parameters*)

Attempt to set the parameters controlling the receiving of media.

Parameters

parameters (RTCRtpReceiveParameters) – The *RTCRtpParameters* for the receiver.

Return type

None

await stop()

Irreversibly stop the receiver.

Return type

None

class aiortc.**RTCRtpSender**(*trackOrKind*, *transport*)

The *RTCRtpSender* interface provides the ability to control and obtain details about how a particular *MediaStreamTrack* is encoded and sent to a remote peer.

Parameters

- **trackOrKind** (Union[*MediaStreamTrack*, str]) – Either a *MediaStreamTrack* instance or a media kind (*'audio'* or *'video'*).
- **transport** – An *RTCDtlsTransport*.

property track: *MediaStreamTrack*

The *MediaStreamTrack* which is being handled by the sender.

property transport

The *RTCDtlsTransport* over which media data for the track is transmitted.

classmethod getCapabilities(*kind*)

Returns the most optimistic view of the system's capabilities for sending media of the given *kind*.

Return type

RTCRtpCapabilities

await getStats()

Returns statistics about the RTP sender.

Return type

RTCStatsReport

await send(*parameters*)

Attempt to set the parameters controlling the sending of media.

Parameters

parameters (RTCRtpSendParameters) – The RTCRtpSendParameters for the sender.

Return type

None

await stop()

Irreversibly stop the sender.

class aiortc.**RTCRtpTransceiver**(*kind*, *receiver*, *sender*, *direction='sendrecv'*)

The RTCRtpTransceiver interface describes a permanent pairing of an *RTCRtpSender* and an *RTCRtpReceiver*, along with some shared state.

property currentDirection: str | None

The currently negotiated direction of the transceiver.

One of *'sendrecv'*, *'sendonly'*, *'recvonly'*, *'inactive'* or *None*.

property direction: `str`

The preferred direction of the transceiver, which will be used in `RTCPeerConnection.createOffer()` and `RTCPeerConnection.createAnswer()`.

One of *'sendrecv'*, *'sendonly'*, *'recvonly'* or *'inactive'*.

property receiver: `RTCRtpReceiver`

The `RTCRtpReceiver` that handles receiving and decoding incoming media.

property sender: `RTCRtpSender`

The `RTCRtpSender` responsible for encoding and sending data to the remote peer.

setCodecPreferences(*codecs*)

Override the default codec preferences.

See `RTCRtpSender.getCapabilities()` and `RTCRtpReceiver.getCapabilities()` for the supported codecs.

Parameters

codecs (`List[RTCRtpCodecCapability]`) – A list of `RTCRtpCodecCapability`, in decreasing order of preference. If empty, restores the default preferences.

Return type

`None`

await stop()

Permanently stops the `RTCRtpTransceiver`.

class aiortc.**RTCRtpSynchronizationSource**(*timestamp*, *source*)

The `RTCRtpSynchronizationSource` dictionary contains information about a synchronization source (SSRC).

timestamp: `datetime`

The timestamp associated with this source.

source: `int`

The SSRC identifier associated with this source.

class aiortc.**RTCRtpCapabilities**(*codecs=<factory>*, *headerExtensions=<factory>*)

The `RTCRtpCapabilities` dictionary provides information about support codecs and header extensions.

codecs: `List[RTCRtpCodecCapability]`

A list of `RTCRtpCodecCapability`.

headerExtensions: `List[RTCRtpHeaderExtensionCapability]`

A list of `RTCRtpHeaderExtensionCapability`.

class aiortc.**RTCRtpCodecCapability**(*mimeType*, *clockRate*, *channels=None*, *parameters=<factory>*)

The `RTCRtpCodecCapability` dictionary provides information on codec capabilities.

mimeType: `str`

The codec MIME media type/subtype, for instance *'audio/PCMU'*.

clockRate: `int`

The codec clock rate expressed in Hertz.

channels: `Optional[int] = None`

The number of channels supported (e.g. two for stereo).

parameters: Dict[str, Union[int, str, None]]

Codec-specific parameters available for signaling.

class aiortc.**RTCRtpHeaderExtensionCapability**(*uri*)

The *RTCRtpHeaderExtensionCapability* dictionary provides information on a supported header extension.

uri: str

The URI of the RTP header extension.

class aiortc.**RTCRtpParameters**(*codecs=<factory>*, *headerExtensions=<factory>*, *muxId=''*, *rtcp=<factory>*)

The *RTCRtpParameters* dictionary describes the configuration of an *RTCRtpReceiver* or an *RTCRtpSender*.

codecs: List[*RTCRtpCodecParameters*]

A list of *RTCRtpCodecParameters* to send or receive.

headerExtensions: List[RTCRtpHeaderExtensionParameters]

A list of RTCRtpHeaderExtensionParameters.

muxId: str = ''

The muxId assigned to the RTP stream, if any, empty string if unset.

rtcp: *RTCRtcpParameters*

Parameters to configure RTCP.

class aiortc.**RTCRtpCodecParameters**(*mimeType*, *clockRate*, *channels=None*, *payloadType=None*, *rtcpFeedback=<factory>*, *parameters=<factory>*)

The *RTCRtpCodecParameters* dictionary provides information on codec settings.

mimeType: str

The codec MIME media type/subtype, for instance *'audio/PCMU'*.

clockRate: int

The codec clock rate expressed in Hertz.

channels: Optional[int] = None

The number of channels supported (e.g. two for stereo).

payloadType: Optional[int] = None

The value that goes in the RTP Payload Type Field.

rtcpFeedback: List[RTCRtcpFeedback]

Transport layer and codec-specific feedback messages for this codec.

parameters: Dict[str, Union[int, str, None]]

Codec-specific parameters available for signaling.

class aiortc.**RTCRtcpParameters**(*cname=None*, *mux=False*, *ssrc=None*)

The *RTCRtcpParameters* dictionary provides information on RTCP settings.

cname: Optional[str] = None

The Canonical Name (CNAME) used by RTCP.

mux: bool = False

Whether RTP and RTCP are multiplexed.

ssrc: `Optional[int] = None`

The Synchronization Source identifier.

1.2.5 Stream Control Transmission Protocol (SCTP)

class aiortc.**RTCSctpTransport**(*transport*, *port=5000*)

The *RTCSctpTransport* interface includes information relating to Stream Control Transmission Protocol (SCTP) transport.

Parameters
transport (*RTCDtlsTransport*) – An *RTCDtlsTransport*.

property maxChannels: `int | None`

The maximum number of *RTCDataChannel* that can be used simultaneously.

property port: `int`

The local SCTP port number used for data channels.

property state: `str`

The current state of the SCTP transport.

property transport

The *RTCDtlsTransport* over which SCTP data is transmitted.

classmethod getCapabilities()

Retrieve the capabilities of the transport.
Return type
RTCSctpCapabilities

await start(*remoteCaps*, *remotePort*)

Start the transport.
Return type
None

await stop()

Stop the transport.
Return type
None

class State(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

class aiortc.**RTCSctpCapabilities**(*maxMessageSize*)

The *RTCSctpCapabilities* dictionary provides information about the capabilities of the *RTCSctpTransport*.

maxMessageSize: `int`

The maximum size of data that the implementation can send or 0 if the implementation can handle messages of any size.

1.2.6 Data channels

class aiortc.**RTCDataChannel**(*transport*, *parameters*, *send_open=True*)

The *RTCDataChannel* interface represents a network channel which can be used for bidirectional peer-to-peer transfers of arbitrary data.

Parameters

- **transport** – An *RTCSctpTransport*.
- **parameters** (*RTCDataChannelParameters*) – An *RTCDataChannelParameters*.

property bufferedAmount: int

The number of bytes of data currently queued to be sent over the data channel.

property bufferedAmountLowThreshold: int

The number of bytes of buffered outgoing data that is considered "low".

property negotiated: bool

Whether data channel was negotiated out-of-band.

property id: int | None

An ID number which uniquely identifies the data channel.

property label: str

A name describing the data channel.

These labels are not required to be unique.

property ordered: bool

Indicates whether or not the data channel guarantees in-order delivery of messages.

property maxPacketLifeTime: int | None

The maximum time in milliseconds during which transmissions are attempted.

property maxRetransmits: int | None

"The maximum number of retransmissions that are attempted.

property protocol: str

The name of the subprotocol in use.

property readyState: str

A string indicating the current state of the underlying data transport.

property transport

The *RTCSctpTransport* over which data is transmitted.

close()

Close the data channel.

Return type

None

send(*data*)

Send *data* across the data channel to the remote peer.

Return type

None

class aiortc.**RTCDataChannelParameters**(*label=''*, *maxPacketLifeTime=None*, *maxRetransmits=None*, *ordered=True*, *protocol=''*, *negotiated=False*, *id=None*)

The *RTCDataChannelParameters* dictionary describes the configuration of an *RTCDataChannel*.

label: str = ''

A name describing the data channel.

maxPacketLifeTime: Optional[int] = None

The maximum time in milliseconds during which transmissions are attempted.

maxRetransmits: Optional[int] = None

The maximum number of retransmissions that are attempted.

ordered: bool = True

Whether the data channel guarantees in-order delivery of messages.

protocol: str = ''

The name of the subprotocol in use.

negotiated: bool = False

Whether data channel will be negotiated out of-band, where both sides create data channel with an agreed-upon ID.

id: Optional[int] = None

An numeric ID for the channel; permitted values are 0-65534. If you don't include this option, the user agent will select an ID for you. Must be set when negotiating out-of-band.

1.2.7 Media

class aiortc.**MediaStreamTrack**

A single media track within a stream.

property id: str

An automatically generated globally unique ID.

abstractmethod await recv()

Receive the next AudioFrame, VideoFrame or Packet

Return type

Union[Frame, Packet]

1.2.8 Statistics

class aiortc.**RTCStatsReport**

Provides statistics data about WebRTC connections as returned by the *RTCPeerConnection.getStats()*, *RTCRtpReceiver.getStats()* and *RTCRtpSender.getStats()* coroutines.

This object consists of a mapping of string identifiers to objects which are instances of:

- *RTCInboundRtpStreamStats*
- *RTCOutboundRtpStreamStats*
- *RTCRemoteInboundRtpStreamStats*

- *RTCRemoteOutboundRtpStreamStats*
- *RTCTransportStats*

class aiortc.**RTCInboundRtpStreamStats**(*timestamp*, *type*, *id*, *ssrc*, *kind*, *transportId*, *packetsReceived*, *packetsLost*, *jitter*)

The *RTCInboundRtpStreamStats* dictionary represents the measurement metrics for the incoming RTP media stream.

class aiortc.**RTCOutboundRtpStreamStats**(*timestamp*, *type*, *id*, *ssrc*, *kind*, *transportId*, *packetsSent*, *bytesSent*, *trackId*)

The *RTCOutboundRtpStreamStats* dictionary represents the measurement metrics for the outgoing RTP stream.

class aiortc.**RTCRemoteInboundRtpStreamStats**(*timestamp*, *type*, *id*, *ssrc*, *kind*, *transportId*, *packetsReceived*, *packetsLost*, *jitter*, *roundTripTime*, *fractionLost*)

The *RTCRemoteInboundRtpStreamStats* dictionary represents the remote endpoint's measurement metrics for a particular incoming RTP stream.

class aiortc.**RTCRemoteOutboundRtpStreamStats**(*timestamp*, *type*, *id*, *ssrc*, *kind*, *transportId*, *packetsSent*, *bytesSent*, *remoteTimestamp=None*)

The *RTCRemoteOutboundRtpStreamStats* dictionary represents the remote endpoint's measurement metrics for its outgoing RTP stream.

class aiortc.**RTCTransportStats**(*timestamp*, *type*, *id*, *packetsSent*, *packetsReceived*, *bytesSent*, *bytesReceived*, *iceRole*, *dtlsState*)

RTCTransportStats(timestamp: datetime.datetime, type: str, id: str, packetsSent: int, packetsReceived: int, bytesSent: int, bytesReceived: int, iceRole: str, dtlsState: str)

packetsSent: int

Total number of packets sent over this transport.

packetsReceived: int

Total number of packets received over this transport.

bytesSent: int

Total number of bytes sent over this transport.

bytesReceived: int

Total number of bytes received over this transport.

iceRole: str

The current value of *RTCIceTransport.role*.

dtlsState: str

The current value of *RTCDtlsTransport.state*.

1.3 Helpers

These classes are not part of the WebRTC or ORTC API, but provide higher-level helpers for tasks like manipulating media streams.

1.3.1 Media sources

class aiortc.contrib.media.**MediaPlayer**(*file*, *format=None*, *options=None*, *timeout=None*, *loop=False*, *decode=True*)

A media source that reads audio and/or video from a file.

Examples:

```
# Open a video file.
player = MediaPlayer('/path/to/some.mp4')

# Open an HTTP stream.
player = MediaPlayer(
    'http://download.tsi.telecom-paristech.fr/'
    'gpac/dataset/dash/uhd/mux_sources/hevcds_720p30_2M.mp4')

# Open webcam on Linux.
player = MediaPlayer('/dev/video0', format='v4l2', options={
    'video_size': '640x480'
})

# Open webcam on OS X.
player = MediaPlayer('default:none', format='avfoundation', options={
    'video_size': '640x480'
})

# Open webcam on Windows.
player = MediaPlayer('video=Integrated Camera', format='dshow', options={
    'video_size': '640x480'
})
```

Parameters

- **file** – The path to a file, or a file-like object.
- **format** – The format to use, defaults to autodect.
- **options** – Additional options to pass to FFmpeg.
- **timeout** – Open/read timeout to pass to FFmpeg.
- **loop** – Whether to repeat playback indefinitely (requires a seekable file).

property audio: *MediaStreamTrack*

A *aiortc.MediaStreamTrack* instance if the file contains audio.

property video: *MediaStreamTrack*

A *aiortc.MediaStreamTrack* instance if the file contains video.

1.3.2 Media sinks

class aiortc.contrib.media.**MediaRecorder**(*file*, *format=None*, *options=None*)

A media sink that writes audio and/or video to a file.

Examples:

```
# Write to a video file.
player = MediaRecorder('/path/to/file.mp4')

# Write to a set of images.
player = MediaRecorder('/path/to/file-%3d.png')
```

Parameters

- **file** – The path to a file, or a file-like object.
- **format** – The format to use, defaults to autodect.
- **options** – Additional options to pass to FFmpeg.

addTrack(*track*)

Add a track to be recorded.

Parameters

track – A *aiortc.MediaStreamTrack*.

await start()

Start recording.

await stop()

Stop recording.

class aiortc.contrib.media.**MediaBlackhole**

A media sink that consumes and discards all media.

addTrack(*track*)

Add a track whose media should be discarded.

Parameters

track – A *aiortc.MediaStreamTrack*.

await start()

Start discarding media.

await stop()

Stop discarding media.

1.3.3 Media transforms

class aiortc.contrib.media.**MediaRelay**

A media source that relays one or more tracks to multiple consumers.

This is especially useful for live tracks such as webcams or media received over the network.

subscribe(*track*, *buffered=True*)

Create a proxy around the given *track* for a new consumer.

Parameters

- **track** (*MediaStreamTrack*) – Source `MediaStreamTrack` which is relayed.
- **buffered** (`bool`) – Whether there need a buffer between the source track and relayed track.

Return type

class

MediaStreamTrack

1.4 Contributing

Thanks for taking the time to contribute to `aiortc`!

1.4.1 Code of Conduct

This project and everyone participating in it is governed by the Code of Conduct. By participating, you are expected to uphold this code. Please report inappropriate behavior to jeremy DOT laine AT m4x DOT org.

1.4.2 Contributions

Bug reports, patches and suggestions are welcome!

Please open an issue or send a pull request.

Feedback about the examples or documentation are especially valuable as they make `aiortc` accessible to a wider audience.

Code contributions *must* come with full unit test coverage. WebRTC is a complex protocol stack and ensuring correct behaviour now and in the future requires a proper investment in automated testing.

1.4.3 Questions

GitHub issues aren't a good medium for handling questions. There are better places to ask questions, for example Stack Overflow.

If you want to ask a question anyway, please make sure that:

- it's a question about `aiortc` and not about `asyncio`;
- it isn't answered by the documentation;
- it wasn't asked already.

A good question can be written as a suggestion to improve the documentation.

1.5 Changelog

1.5.1 1.8.0

- Only send / receive RTP according to `RTCRtpTransceiver.currentDirection`.
- Close the `RTCPeerConnection` if all DTLS transports are closed.
- Free the encoder as soon as the `RTCRtpSender` stops to save memory.
- Modernise JavaScript in *server* and *webcam* examples.

1.5.2 1.7.0

- Add support for GCM based SRTP protection profiles.
- Reduce supported DTLS cipher list to avoid Client Hello fragmentation.
- Fix *utcnow()* deprecation warning on Python 3.12.

1.5.3 1.6.0

- Build wheels using *Py_LIMITED_ABI* to make them compatible with future Python versions.
- Build wheels using opus 1.4 and vpx 1.13.1.
- Use unique IDs for audio and video header extensions.
- Allow `MediaRecorder` to record audio from pulse.

1.5.4 1.5.0

- Make H.264 send a full picture when picture loss occurs.
- Fix TURN over TCP by updating *aioice* to 0.9.0.
- Make use of the *ifaddr* package instead of the unmaintained *netifaces* package.

1.5.5 1.4.0

- Build wheels for Python 3.11.
- Allow `MediaPlayer` to send media without transcoding.
- Allow `MediaPlayer` to specify a timeout when opening media.
- Make `RTCSctpTransport` transmit packets sooner to reduce datachannel latency.
- Refactor `RTCDtlsTransport` to use PyOpenSSL.
- Make `RTCPeerConnection` log sent and received SDP when using verbose logging.

1.5.6 1.3.2

- Limit size of NACK reports to avoid excessive packet size.
- Improve H.264 codec matching.
- Determine video size from first frame received by `MediaRecorder`.
- Fix a deprecation warning when using *av* >= 9.1.0.
- Tolerate STUN URLs containing a *protocol* querystring argument.

1.5.7 1.3.1

- Build wheels for aarch64 on Linux.
- Adapt `MediaPlayer` for PyAV 9.x.
- Ensure H.264 produces B-frames by resetting picture type.

1.5.8 1.3.0

- Build wheels for Python 3.10 and for arm64 on Mac.
- Build wheels against *libvpx* 1.10.
- Add support for looping in `MediaPlayer`.
- Add unbuffered option to `MediaRelay`.
- Calculate audio energy and send in RTP header extension.
- Fix a race condition in RTP sender/receiver shutdown.
- Improve performance of H.264 bitstream splitting code.
- Update imports for *pyee* version 9.x.
- Fully switch to *google-crc32c* instead of *crc32*.
- Drop support for Python 3.6.
- Remove *apprtc* code as the service is no longer publicly hosted.

1.5.9 1.2.1

- Add a clear error message when no common codec is found.
- Replace the *crc32* dependency with *google-crc32c* which offers a more liberal license.

1.5.10 1.2.0

- Fix jitter buffer to avoid severe picture corruption under packet loss and send Picture Loss Indication (PLI) when needed.
- Make H.264 encoder honour the bitrate from the bandwidth estimator.
- Add support for hardware-accelerated H.264 encoding on Raspberry Pi 4 using the *h264_omx* codec.
- Add `MediaRelay` class to allow sending media tracks to multiple consumers.

1.5.11 1.1.2

- Add `RTCPeerConnection.connectionState` property.
- Correctly detect RTCIceTransport *"failed"* state.
- Correctly route RTP packets when there are multiple tracks of the same kind.
- Use full module name to name loggers.

1.5.12 1.1.1

- Defer adding remote candidates until after transport bundling to avoid unnecessary mDNS lookups.

1.5.13 1.1.0

- Add support for resolving mDNS candidates.
- Improve support for TURN, especially long-lived connections.

1.5.14 1.0.0

Breaking

- Make `RTCPeerConnection.addIceCandidate()` a coroutine.
- Make `RTCIceTransport.addRemoteCandidate()` a coroutine.

Media

- Handle SSRC attributes in SDP containing a colon (#372).
- Limit number of H.264 NALU per packet (#394, #426).

Examples

- *server* make it possible to specify bind address (#347).

1.5.15 0.9.28

Provide binary wheels for Linux, Mac and Windows on PyPI.

1.5.16 0.9.27

Data channels

- Add `RTCSctpTransport.maxChannels` property.
- Recycle stream IDs (#256).
- Correctly close data channel when SCTP is not established (#300).

Media

- Add add `RTCRtpReceiver.track` property (#298).
- Fix a crash in *AimdRateControl* (#295).

1.5.17 0.9.26

DTLS

- Drop support for OpenSSL < 1.0.2.

Examples

- *apprtc* fix handling of empty "candidate" message.

Media

- Fix a MediaPlayer crash when stopping one track of a multi-track file (#237, #274).
- Fix a MediaPlayer error when stopping a track while waiting for the next frame.
- Make *RTCRtpSender* resilient to exceptions raised by media stream tracks (#283).

1.5.18 0.9.25

Media

- Do not repeatedly send key frames after receiving a PLI.

SDP

- Do not try to determine track ID if there is no Msid.
- Accept a star in rtcp-fb attributes.

1.5.19 0.9.24

Peer connection

- Assign DTLS role based on the SDP negotiation, not the resolved ICE role.
- When the peer is ICE lite, adopt the ICE controlling role, and do not use agressive nomination.
- Do not close transport on *setRemoteDescription* if media and data are bundled.
- Set RemoteStreamTrack.id based on the Msid.

Media

- Support alsa hardware output in MediaRecorder.

SDP

- Add support for the *ice-lite* attribute.
- Add support for receiving session-level *ice-ufrag*, *ice-pwd* and *setup* attributes.

Miscellaneous

- Switch from *attrs* to standard Python *dataclasses*.
- Use PEP-526 style variable annotations instead of comments.

1.5.20 0.9.23

- Drop support for Python 3.5.
- Drop dependency on PyOpenSSL.
- Use PyAV >= 7.0.0.
- Add partial type hints.

1.5.21 0.9.22

DTLS

- Display exception if data handler fails.

Examples

- *server* and *webcam* : add playsinline attribute for iOS compatibility.
- *webcam* : make it possible to play media from a file.

Miscellaneous

- Use aioice >= 0.6.15 to not fail on mDNS candidates.
- Use pyee version 6.x.

1.5.22 0.9.21

DTLS

- Call SSL_CTX_set_ecdh_auto for OpenSSL 1.0.2.

Media

- Correctly route REMB packets to the `RTCRtpSender`.

Examples

- `MediaPlayer` : release resources (e.g. webcam) when the player stops.
- `ApprtcSignaling` : make AppRTC signaling available for more examples.
- *datachannel-cli* : make uvloop optional.
- *videostream-cli* : animate the flag with a wave effect.
- *webcam* : explicitly set frame rate to 30 fps for webcams.

1.5.23 0.9.20

Data channels

- Support out-of-band negotiation and custom channel id.

Documentation

- Fix documentation build by installing *crc32c* instead of *crcmod*.

Examples

- `MediaPlayer` : skip frames with no presentation timestamp (pts).

1.5.24 0.9.19

Data channels

- Do not raise congestion window when it is not fully utilized.
- Fix Highest TSN Newly Acknowledged logic for striking lost chunks.
- Do not limit congestion window to 120kB, limit burst size instead.

Media

- Skip RTX packets with an empty payload.

Examples

- *apprtc* : make the initiator send messages using an HTTP POST instead of WebSocket.
- *janus* : new example to connect to the Janus WebRTC server.
- *server* : add cartoon effect to video transforms.

1.5.25 0.9.18

DTLS

- Do not use DTLSv1_get_timeout after DTLS handshake completes.

Data channels

- Add setter for `RTCDataChannel.bufferedAmountLowThreshold`.
- Use *crc32c* package instead of *crcmod*, it provides better performance.
- Improve parsing and serialization code performance.
- Disable logging code if it is not used to improve performance.

1.5.26 0.9.17

DTLS

- Do not bomb if SRTP is received before DTLS handshake completes.

Data channels

- Implement unordered delivery, so that the *ordered* option is honoured.
- Implement partial reliability, so that the *maxRetransmits* and *maxPacketLifeTime* options are honoured.

Media

- Put all tracks in the same stream for now, fixes breakage introduced in 0.9.14.
- Use case-insensitive comparison for codec names.
- Use a=msid attribute in SDP instead of SSRC-level attributes.

Examples

- *server* : make it possible to select unreliable mode for data channels.
- *server* : print the round-trip time for data channel messages.

1.5.27 0.9.16

DTLS

- Log OpenSSL errors if the DTLS handshake fails.
- Fix DTLS handshake in server mode with OpenSSL < 1.1.0.

Media

- Add `RTCRtpReceiver.getCapabilities()` and `RTCRtpSender.getCapabilities()`.
- Add `RTCRtpReceiver.getSynchronizationSources()`.
- Add `RTCRtpTransceiver.setCodecPreferences()`.

Examples

- *server* : make it possible to force audio codec.
- *server* : shutdown cleanly on Chrome which lacks `RTCRtpTransceiver.stop()`.

1.5.28 0.9.15

Data channels

- Emit a warning if the crcmod C extension is not present.

Media

- Support subsequent offer / answer exchanges.
- Route RTCP parameters to RTP receiver and sender independently.
- Fix a regression when the remote SSRC are not known.
- Fix VP8 descriptor parsing errors detected by fuzzing.
- Fix H264 descriptor parsing errors detected by fuzzing.

1.5.29 0.9.14

Media

- Add support for RTX retransmission packets.
- Fix RTP and RTCP parsing errors detected by fuzzing.
- Use case-insensitive comparison for hash algorithm in SDP, fixes interoperability with Asterisk.
- Offer NACK PLI and REMB feedback mechanisms for H.264.

1.5.30 0.9.13

Data channels

- Raise an exception if `RTCDataChannel.send()` is called when readyState is not *'open'*.
- Do not use stream sequence number for unordered data channels.

Media

- Set VP8 target bitrate according to Receiver Estimated Maximum Bandwidth.

Examples

- Correctly handle encoding in copy-and-paste signaling.
- *server* : add command line options to use HTTPS.
- *webcam* : add command line options to use HTTPS.
- *webcam* : add code to open webcam on OS X.

1.5.31 0.9.12

- Rework code in order to facilitate garbage collection and avoid memory leaks.

1.5.32 0.9.11

Media

- Make AudioStreamTrack and VideoStreamTrack produce empty frames more regularly.

Examples

- Fix a regession in copy-and-paste signaling which blocked the event loop.

1.5.33 0.9.10

Peer connection

- Send *raddr* and *rport* parameters for server reflexive and relayed candidates. This is required for Firefox to accept our STUN / TURN candidates.
- Do not raise an exception if ICE or DTLS connection fails, just change state.

Media

- Revert to using asyncio's *run_in_executor* to send data to the encoder, it greatly reduces the response time.
- Adjust package requirements to accept PyAV < 7.0.0.

Examples

- *webcam* : force Chrome to use "unified-plan" semantics to enabled *addTransceiver*.
- `MediaPlayer` : don't sleep at all when playing from webcam. This eliminates the constant one-second lag in the *webcam* demo.

1.5.34 0.9.9

Warning: *aiortc* now uses PyAV's `AudioFrame` and `VideoFrame` classes instead of defining its own.

Media

- Use a jitter buffer for incoming audio.
- Add `RTCPeerConnection.addTransceiver()` method.
- Add `RTCRtpTransceiver.direction` to manage transceiver direction.

Examples

- *apprtc* : demonstrate the use of `MediaPlayer` and `MediaRecorder`.
- *webcam* : new examples illustrating sending video from a webcam to a browser.
- `MediaPlayer` : don't sleep if a frame lacks timing information.
- `MediaPlayer` : remove *start()* and *stop()* methods.
- `MediaRecorder` : use *libx264* for encoding.
- `MediaRecorder` : make *start()* and *stop()* coroutines.

1.5.35 0.9.8

Media

- Add support for H.264 video, a big thank you to @dsvictor94!
- Add support for sending Receiver Estimate Maximum Bitrate (REMB) feedback.
- Add support for parsing / serializing more RTP header extensions.
- Move each media encoder / decoder its one thread instead of using a thread pool.

Statistics

- Add the `RTCPeerConnection.getStats()` coroutine to retrieve statistics.
- Add initial `RTCTransportStats` to report transport statistics.

Examples

- Add new `MediaPlayer` class to read audio / video from a file.
- Add new `MediaRecorder` class to write audio / video to a file.
- Add new `MediaBlackhole` class to discard audio / video.

1.5.36 0.9.7

Media

- Make RemoteStreamTrack emit an "ended" event, to simplify shutting down media consumers.
- Add RemoteStreamTrack.readyState property.
- Handle timestamp wraparound on sent RTP packets.

Packaging

- Add a versioned dependency on cffi>=1.0.0 to fix Raspberry Pi builds.

1.5.37 0.9.6

Data channels

- Optimize reception for improved latency and throughput.

Media

- Add initial `RTCRtpReceiver.getStats()` and `RTCRtpReceiver.getStats()` coroutines.

Examples

- *datachannel-cli*: display ping/pong roundtrip time.

1.5.38 0.9.5

Media

- Make it possible to add multiple audio or video streams.
- Implement basic RTP video packet loss detection / retransmission using RTCP NACK feedback.
- Respond to Picture Loss Indications (PLI) by sending a keyframe.
- Use shorter MID values to reduce RTP header extension overhead.
- Correctly shutdown and discard unused transports when using BUNDLE.

Examples

- *server* : make it possible to save received video to an AVI file.

1.5.39 0.9.4

Peer connection

- Add support for TURN over TCP.

Examples

- Add media and signaling helpers in *aiortc.contrib*.
- Fix colorspace OpenCV colorspace conversions.
- *apprtc* : send rotating image on video track.

1.5.40 0.9.3

Media

- Set PictureID attribute on outgoing VP8 frames.
- Negotiate and send SDES MID header extension for RTP packets.
- Fix negative packets_lost encoding for RTCP reports.

1.5.41 0.9.2

Data channels

- Numerous performance improvements in congestion control.

Examples

- *datachannel-filexfer*: use uvloop instead of default asyncio loop.

1.5.42 0.9.1

Data channels

- Revert making RTCDataChannel.send a coroutine.

1.5.43 0.9.0

Media

- Enable post-processing in VP8 decoder to remove (macro) blocks.
- Set target bitrate for VP8 encoder to 900kbps.
- Re-create VP8 encoder if frame size changes.
- Implement jitter estimation for RTCP reports.
- Avoid overflowing the DLSR field for RTCP reports.
- Raise video jitter buffer size.

Data channels

- BREAKING CHANGE: make RTCDataChannel.send a coroutine.
- Support spec-compliant SDP format for datachannels, as used in Firefox 63.
- Never send a negative advertised_cwnd.

Examples

- *datachannel-filexfer*: new example for file transfer over a data channel.
- *datachannel-vpn*: new example for a VPN over a data channel.
- *server*: make it possible to select video resolution.

1.5.44 0.8.0

Media

- Align VP8 settings with those used by WebRTC project, which greatly improves video quality.
- Send RTCP source description, sender report, receiver report and bye packets.

Examples

- *server*:
  - make it possible to not transform video at all.
  - allow video display to be up to 1280px wide.
- *videostream-cli*:
  - fix Python 3.5 compatibility

Miscellaneous

- Delay logging string interpolation to reduce cost of packet logging in non-verbose mode.

1.5.45 0.7.0

Peer connection

- Add `RTCPeerConnection.addIceCandidate()` method to handle trickled ICE candidates.

Media

- Make stop() methods of `RTCRtpReceiver`, `RTCRtpSender` and `RTCRtpTransceiver` coroutines to enable clean shutdown.

Data channels

- Clean up `RTCDataChannel` shutdown sequence.
- Support receiving an SCTP *RE-CONFIG* to raise number of inbound streams.

Examples

- *server*:
  - perform some image processing using OpenCV.
  - make it possible to disable data channels.
  - make demo web interface more mobile-friendly.
- *apprtc*:
  - automatically create a room if no room is specified on command line.
  - handle *bye* command.

1.5.46 0.6.0

Peer connection

- Make it possible to specify one STUN server and / or one TURN server.
- Add *BUNDLE* support to use a single ICE/DTLS transport for multiple media.
- Move media encoding / decoding off the main thread.

Data channels

- Use SCTP *ABORT* instead of *SHUTDOWN* when stopping `RTCSctpTransport`.
- Advertise support for SCTP *RE-CONFIG* extension.
- Make `RTCDataChannel` emit *open* and *close* events.

Examples

- Add an example of how to connect to appr.tc.
- Capture audio frames to a WAV file in server example.
- Show datachannel open / close events in server example.

1.6 License

PYTHON MODULE INDEX

a

aiortc, 3

INDEX

A

addIceCandidate() (*aiortc.RTCPeerConnection method*), 4
addRemoteCandidate() (*aiortc.RTCIceTransport method*), 6
addTrack() (*aiortc.contrib.media.MediaBlackhole method*), 18
addTrack() (*aiortc.contrib.media.MediaRecorder method*), 18
addTrack() (*aiortc.RTCPeerConnection method*), 4
addTransceiver() (*aiortc.RTCPeerConnection method*), 4
aiortc
    module, 3
algorithm (*aiortc.RTCDtlsFingerprint attribute*), 9
audio (*aiortc.contrib.media.MediaPlayer property*), 17

B

bufferedAmount (*aiortc.RTCDataChannel property*), 14
bufferedAmountLowThreshold (*aiortc.RTCDataChannel property*), 14
bytesReceived (*aiortc.RTCTransportStats attribute*), 16
bytesSent (*aiortc.RTCTransportStats attribute*), 16

C

channels (*aiortc.RTCRtpCodecCapability attribute*), 11
channels (*aiortc.RTCRtpCodecParameters attribute*), 12
clockRate (*aiortc.RTCRtpCodecCapability attribute*), 11
clockRate (*aiortc.RTCRtpCodecParameters attribute*), 12
close() (*aiortc.RTCDataChannel method*), 14
close() (*aiortc.RTCPeerConnection method*), 4
cname (*aiortc.RTCRtcpParameters attribute*), 12
codecs (*aiortc.RTCRtpCapabilities attribute*), 11
codecs (*aiortc.RTCRtpParameters attribute*), 12
connectionState (*aiortc.RTCPeerConnection property*), 3
createAnswer() (*aiortc.RTCPeerConnection method*), 4
createDataChannel() (*aiortc.RTCPeerConnection method*), 4
createOffer() (*aiortc.RTCPeerConnection method*), 5
credential (*aiortc.RTCIceServer attribute*), 7
currentDirection (*aiortc.RTCRtpTransceiver property*), 10

D

direction (*aiortc.RTCRtpTransceiver property*), 11
dtlsState (*aiortc.RTCTransportStats attribute*), 16

E

expires (*aiortc.RTCCertificate property*), 8

F

fingerprints (*aiortc.RTCDtlsParameters attribute*), 9

G

gather() (*aiortc.RTCIceGatherer method*), 6
generateCertificate() (*aiortc.RTCCertificate class method*), 8
getCapabilities() (*aiortc.RTCRtpReceiver class method*), 9
getCapabilities() (*aiortc.RTCRtpSender class method*), 10
getCapabilities() (*aiortc.RTCSctpTransport class method*), 13
getDefaultIceServers() (*aiortc.RTCIceGatherer class method*), 6
getFingerprints() (*aiortc.RTCCertificate method*), 8
getLocalCandidates() (*aiortc.RTCIceGatherer method*), 6
getLocalParameters() (*aiortc.RTCDtlsTransport method*), 8
getLocalParameters() (*aiortc.RTCIceGatherer method*), 6
getReceivers() (*aiortc.RTCPeerConnection method*), 5
getRemoteCandidates() (*aiortc.RTCIceTransport method*), 7
getSenders() (*aiortc.RTCPeerConnection method*), 5
getStats() (*aiortc.RTCPeerConnection method*), 5

getStats() (*aiortc.RTCRtpReceiver method*), 9
getStats() (*aiortc.RTCRtpSender method*), 10
getSynchronizationSources() (*aiortc.RTCRtpReceiver method*), 9
getTransceivers() (*aiortc.RTCPeerConnection method*), 5

H

headerExtensions (*aiortc.RTCRtpCapabilities attribute*), 11
headerExtensions (*aiortc.RTCRtpParameters attribute*), 12

I

iceConnectionState (*aiortc.RTCPeerConnection property*), 3
iceGatherer (*aiortc.RTCIceTransport property*), 6
iceGatheringState (*aiortc.RTCPeerConnection property*), 4
iceRole (*aiortc.RTCTransportStats attribute*), 16
iceServers (*aiortc.RTCConfiguration attribute*), 5
id (*aiortc.MediaStreamTrack property*), 15
id (*aiortc.RTCDataChannel property*), 14
id (*aiortc.RTCDataChannelParameters attribute*), 15

L

label (*aiortc.RTCDataChannel property*), 14
label (*aiortc.RTCDataChannelParameters attribute*), 15
localDescription (*aiortc.RTCPeerConnection property*), 4

M

maxChannels (*aiortc.RTCSctpTransport property*), 13
maxMessageSize (*aiortc.RTCSctpCapabilities attribute*), 13
maxPacketLifeTime (*aiortc.RTCDataChannel property*), 14
maxPacketLifeTime (*aiortc.RTCDataChannelParameters attribute*), 15
maxRetransmits (*aiortc.RTCDataChannel property*), 14
maxRetransmits (*aiortc.RTCDataChannelParameters attribute*), 15
MediaBlackhole (*class in aiortc.contrib.media*), 18
MediaPlayer (*class in aiortc.contrib.media*), 17
MediaRecorder (*class in aiortc.contrib.media*), 18
MediaRelay (*class in aiortc.contrib.media*), 18
MediaStreamTrack (*class in aiortc*), 15
mimeType (*aiortc.RTCRtpCodecCapability attribute*), 11
mimeType (*aiortc.RTCRtpCodecParameters attribute*), 12
module
    aiortc, 3
mux (*aiortc.RTCRtcpParameters attribute*), 12
muxId (*aiortc.RTCRtpParameters attribute*), 12

N

negotiated (*aiortc.RTCDataChannel property*), 14
negotiated (*aiortc.RTCDataChannelParameters attribute*), 15

O

ordered (*aiortc.RTCDataChannel property*), 14
ordered (*aiortc.RTCDataChannelParameters attribute*), 15

P

packetsReceived (*aiortc.RTCTransportStats attribute*), 16
packetsSent (*aiortc.RTCTransportStats attribute*), 16
parameters (*aiortc.RTCRtpCodecCapability attribute*), 11
parameters (*aiortc.RTCRtpCodecParameters attribute*), 12
password (*aiortc.RTCIceParameters attribute*), 7
payloadType (*aiortc.RTCRtpCodecParameters attribute*), 12
port (*aiortc.RTCSctpTransport property*), 13
protocol (*aiortc.RTCDataChannel property*), 14
protocol (*aiortc.RTCDataChannelParameters attribute*), 15

R

readyState (*aiortc.RTCDataChannel property*), 14
receive() (*aiortc.RTCRtpReceiver method*), 9
receiver (*aiortc.RTCRtpTransceiver property*), 11
recv() (*aiortc.MediaStreamTrack method*), 15
remoteDescription (*aiortc.RTCPeerConnection property*), 4
role (*aiortc.RTCDtlsParameters attribute*), 9
role (*aiortc.RTCIceTransport property*), 6
RTCCertificate (*class in aiortc*), 8
RTCConfiguration (*class in aiortc*), 5
RTCDataChannel (*class in aiortc*), 14
RTCDataChannelParameters (*class in aiortc*), 14
RTCDtlsFingerprint (*class in aiortc*), 9
RTCDtlsParameters (*class in aiortc*), 8
RTCDtlsTransport (*class in aiortc*), 8
RTCIceCandidate (*class in aiortc*), 6
RTCIceGatherer (*class in aiortc*), 6
RTCIceParameters (*class in aiortc*), 7
RTCIceServer (*class in aiortc*), 7
RTCIceTransport (*class in aiortc*), 6
RTCInboundRtpStreamStats (*class in aiortc*), 16
RTCOutboundRtpStreamStats (*class in aiortc*), 16
rtcp (*aiortc.RTCRtpParameters attribute*), 12
RTCPeerConnection (*class in aiortc*), 3

rtcpFeedback (*aiortc.RTCRtpCodecParameters attribute*), 12
RTCRemoteInboundRtpStreamStats (*class in aiortc*), 16
RTCRemoteOutboundRtpStreamStats (*class in aiortc*), 16
RTCRtcpParameters (*class in aiortc*), 12
RTCRtpCapabilities (*class in aiortc*), 11
RTCRtpCodecCapability (*class in aiortc*), 11
RTCRtpCodecParameters (*class in aiortc*), 12
RTCRtpHeaderExtensionCapability (*class in aiortc*), 12
RTCRtpParameters (*class in aiortc*), 12
RTCRtpReceiver (*class in aiortc*), 9
RTCRtpSender (*class in aiortc*), 10
RTCRtpSynchronizationSource (*class in aiortc*), 11
RTCRtpTransceiver (*class in aiortc*), 10
RTCSctpCapabilities (*class in aiortc*), 13
RTCSctpTransport (*class in aiortc*), 13
RTCSctpTransport.State (*class in aiortc*), 13
RTCSessionDescription (*class in aiortc*), 5
RTCStatsReport (*class in aiortc*), 15
RTCTransportStats (*class in aiortc*), 16

S

sctp (*aiortc.RTCPeerConnection property*), 4
send() (*aiortc.RTCDataChannel method*), 14
send() (*aiortc.RTCRtpSender method*), 10
sender (*aiortc.RTCRtpTransceiver property*), 11
setCodecPreferences() (*aiortc.RTCRtpTransceiver method*), 11
setLocalDescription() (*aiortc.RTCPeerConnection method*), 5
setRemoteDescription() (*aiortc.RTCPeerConnection method*), 5
signalingState (*aiortc.RTCPeerConnection property*), 4
source (*aiortc.RTCRtpSynchronizationSource attribute*), 11
ssrc (*aiortc.RTCRtcpParameters attribute*), 12
start() (*aiortc.contrib.media.MediaBlackhole method*), 18
start() (*aiortc.contrib.media.MediaRecorder method*), 18
start() (*aiortc.RTCDtlsTransport method*), 8
start() (*aiortc.RTCIceTransport method*), 7
start() (*aiortc.RTCSctpTransport method*), 13
state (*aiortc.RTCDtlsTransport property*), 8
state (*aiortc.RTCIceGatherer property*), 6
state (*aiortc.RTCIceTransport property*), 6
state (*aiortc.RTCSctpTransport property*), 13
stop() (*aiortc.contrib.media.MediaBlackhole method*), 18
stop() (*aiortc.contrib.media.MediaRecorder method*), 18
stop() (*aiortc.RTCDtlsTransport method*), 8
stop() (*aiortc.RTCIceTransport method*), 7
stop() (*aiortc.RTCRtpReceiver method*), 10
stop() (*aiortc.RTCRtpSender method*), 10
stop() (*aiortc.RTCRtpTransceiver method*), 11
stop() (*aiortc.RTCSctpTransport method*), 13
subscribe() (*aiortc.contrib.media.MediaRelay method*), 18

T

timestamp (*aiortc.RTCRtpSynchronizationSource attribute*), 11
track (*aiortc.RTCRtpReceiver property*), 9
track (*aiortc.RTCRtpSender property*), 10
transport (*aiortc.RTCDataChannel property*), 14
transport (*aiortc.RTCDtlsTransport property*), 8
transport (*aiortc.RTCRtpReceiver property*), 9
transport (*aiortc.RTCRtpSender property*), 10
transport (*aiortc.RTCSctpTransport property*), 13

U

uri (*aiortc.RTCRtpHeaderExtensionCapability attribute*), 12
urls (*aiortc.RTCIceServer attribute*), 7
username (*aiortc.RTCIceServer attribute*), 7
usernameFragment (*aiortc.RTCIceParameters attribute*), 7

V

value (*aiortc.RTCDtlsFingerprint attribute*), 9
video (*aiortc.contrib.media.MediaPlayer property*), 17