Zoom RTMS Overall Architecture Design for Third-Parties

Rene Ke Arun Janakiraman and 2 editors Updated at 18:48 01/22/2025 17 mins

Overview

This design documentation outlines the architecture, protocols, and message formats for integrating the second-party and third-party apps with RTMS and covers the following key aspects:

- Architecture Diagram
- Protocols between RTMS server and Second-Party or Third-Party
- Detailed message formats for various interactions
- Connection establishment process
- Security measures, including token verification
- Handling of media streams and participant events
- Error reporting and connection management

The subsequent chapters will explore the comprehensive architecture design and interaction details of the first phase release. This section aims to offer a thorough understanding of the system's structure and functionality. The workflow is meticulously defined based on the server mode, ensuring optimal performance and scalability. This approach facilitates efficient resource allocation and seamless integration of various components within the system.

Terminology

RTMS: Real-Time Media Streams

2p: Second-Party - apps built by customer developers for internal use within their organizations

3p: Third-Party - apps built by developers for publishing on Zoom marketplace where anyone can use

RTP: Real-time Transport Protocol

MMR: Multi-Media Router

AAN: Active App Notifier

Service Model

The RTMS will support two types of service models: **Server Mode** and **Client Mode**. Server Mode will serve as the default approach for 2p and 3p to communicate with Zoom by adhering to protocols defined by Zoom.

- Server Mode: In this mode, the RTMS server operates as a server and provides listening ports for various protocols, enabling 2p/3p to connect as clients and transmit data.
- Client Mode: In this mode, the RTMS server functions as a client, establishing connections to 2p or 3p socket servers or integrating with external SDKs to facilitate data transmission.

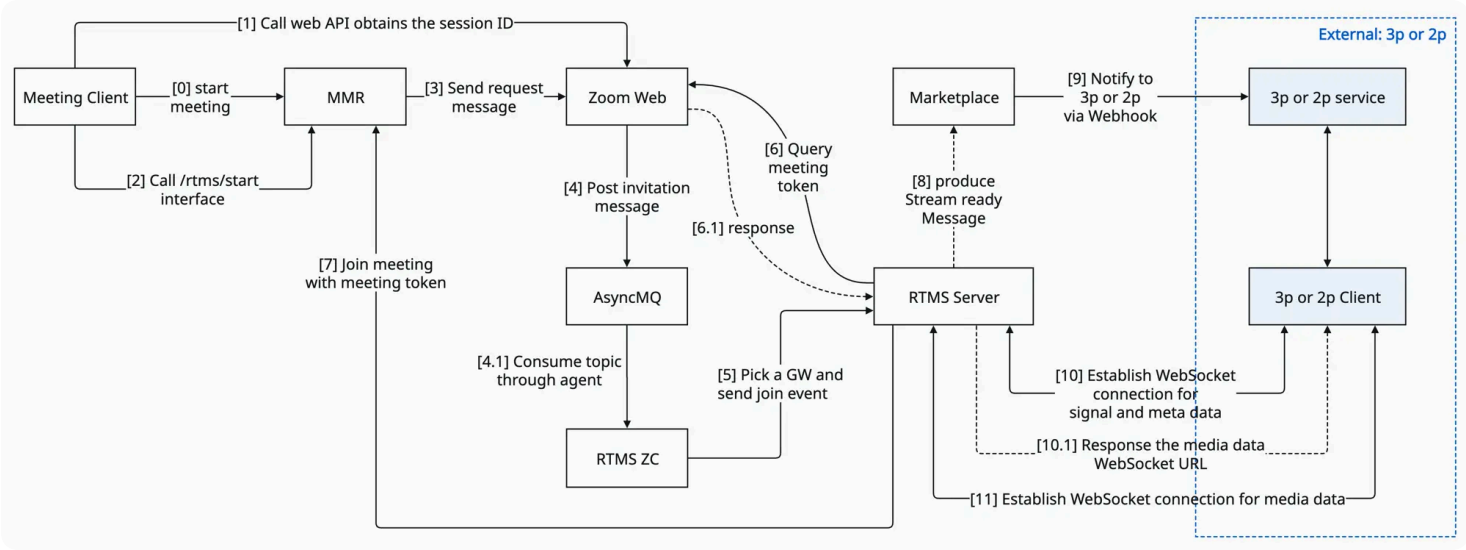
Transmission Protocol

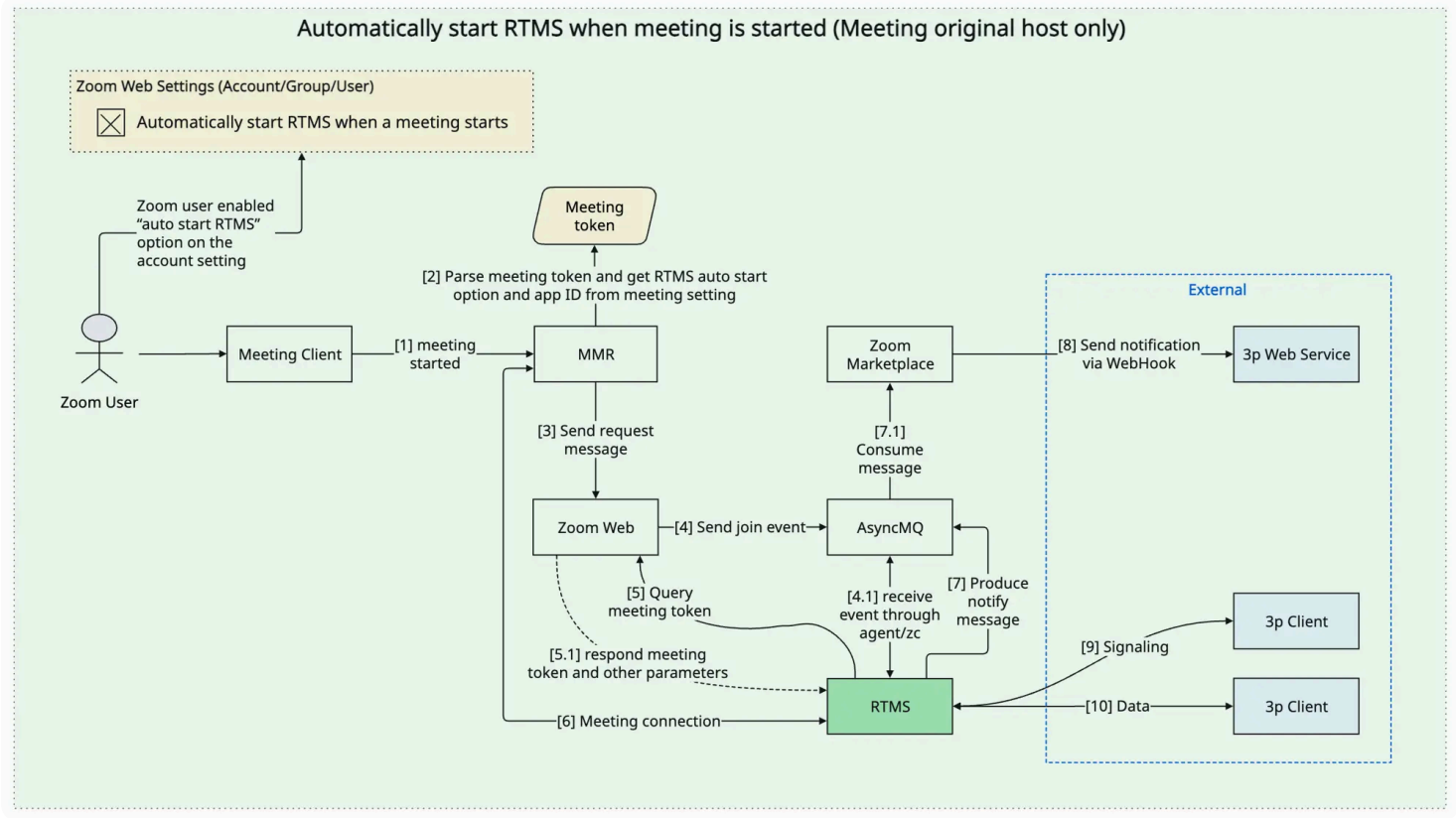
The RTMS will incorporate various protocols for data transmission, including WebSocket, RTMP, UDP for media connection, SRT, and WebRTC. In the first version, **WebSocket will be prioritized for implementation on both signal and media data connection, along with UDP for media data connection**. Additional protocols will be integrated in future releases.

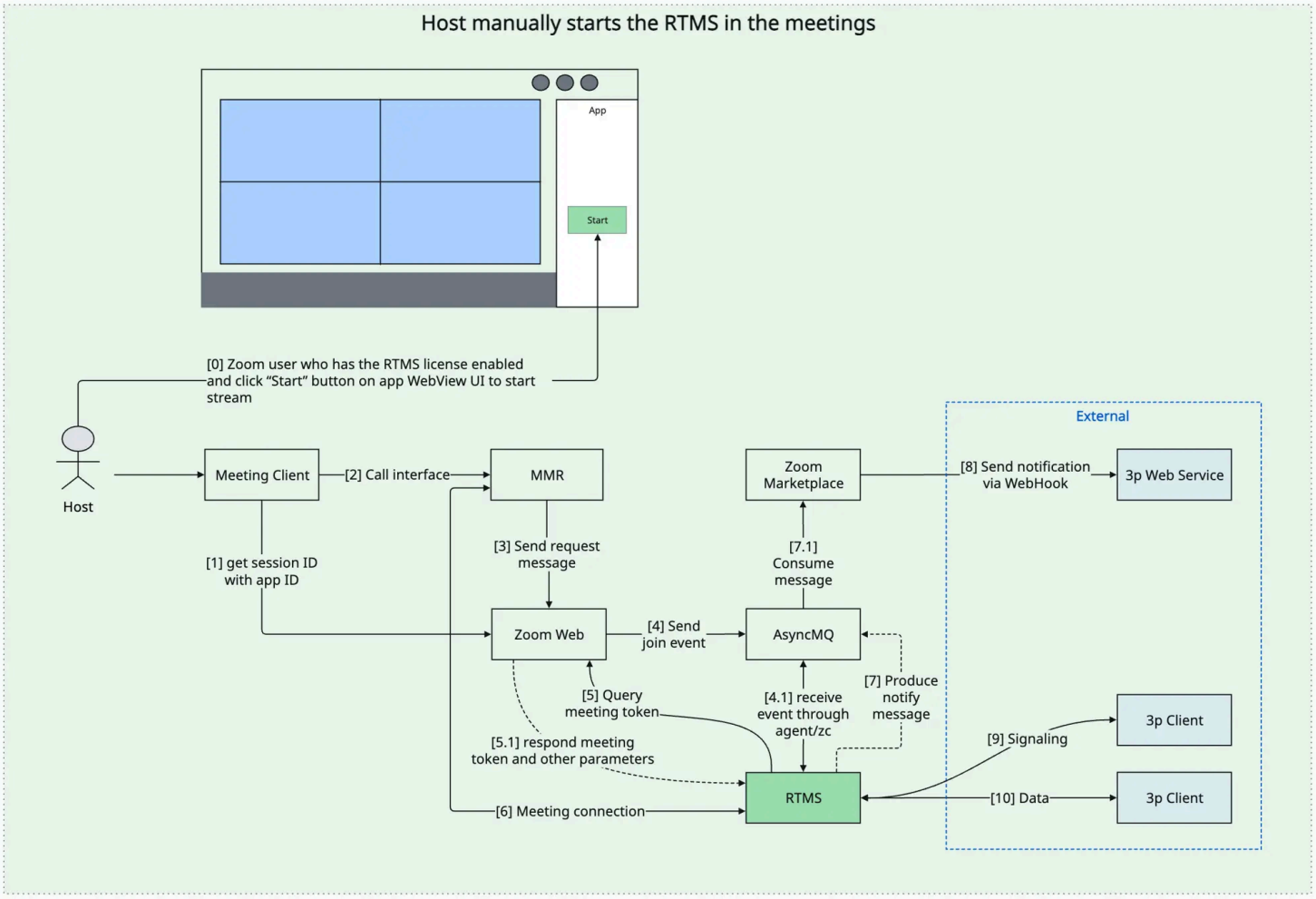
Architecture Diagram

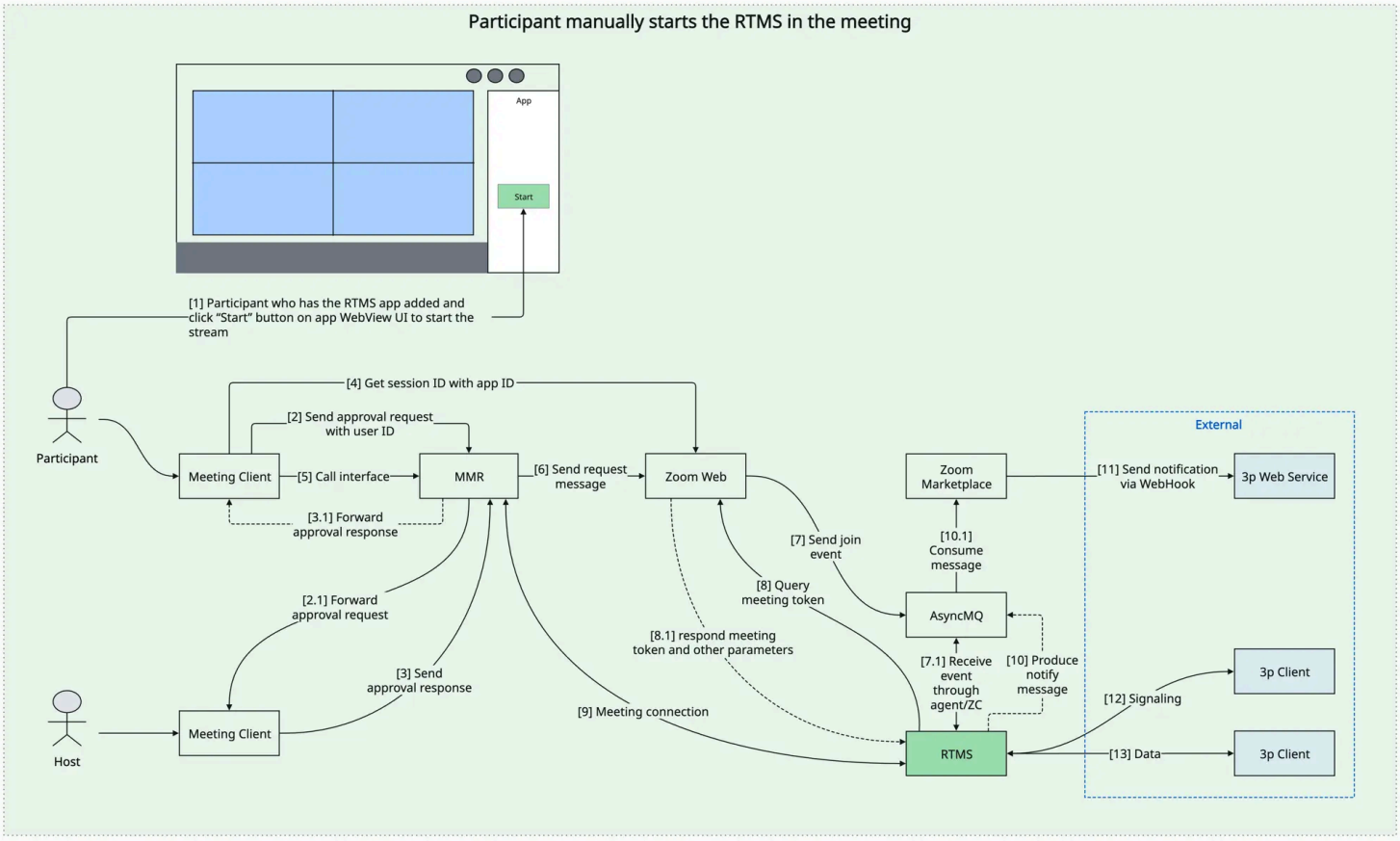
In the first version, the RTMS server will only support being invited to join a meeting as a virtual client and subscribing to real-time events and media data within the meeting.

Below are various interaction workflows. For more detailed information, please refer to the RTMS workflow whiteboard.

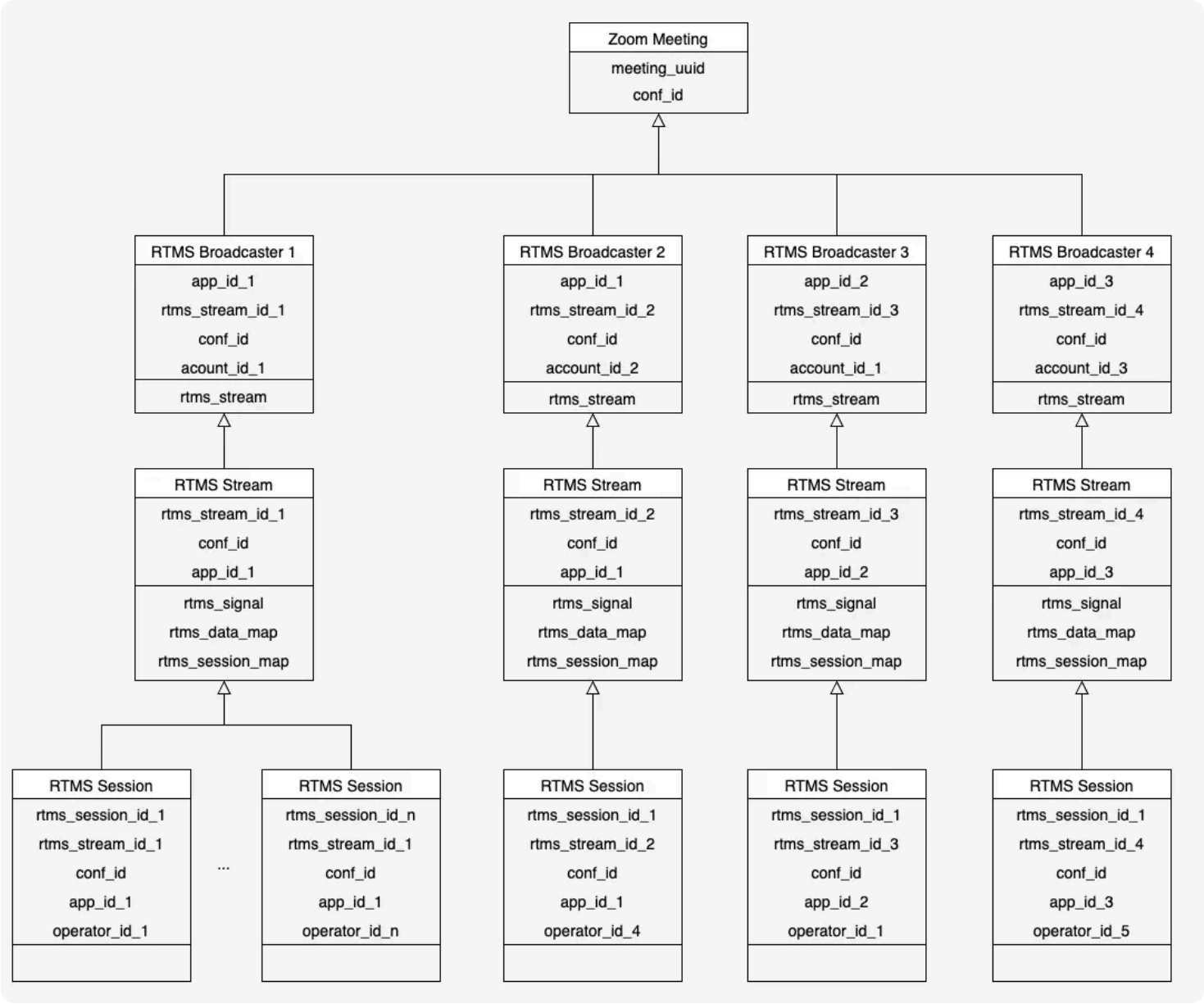








RTMS Object Structure



User Stories

Automatically Start RTMS

Note: automatically start case only supports the meeting original host.

- Recently, the administrator of Alice's organization added a third-party application for all users on the Zoom Marketplace, which will use powerful AI models to analyze media data of meetings to provide richer and more effective real-time analysis results. On the web settings, the administrator enabled auto start RTMS option on the application for all users.
- Alice started a meeting on the Zoom desktop client, and the MMR server will check the meeting configurations from the meeting token, and found automatic

start option of the RTMS feature is enabled. Then, the MMR server calls the Zoom Web API with the gateway ID and meeting ID to start the RTMS, and the Zoom web will get application ID from the meeting original host settings and generate a session ID respond back to the MMR.

- Next, the Zoom web then composes a join event (includes session ID and 3p service type, etc.) and posts a message to the AsyncMQ.
- Then see General Flows.

Manually Start RTMS

- Recently, the administrator of Alice's organization added a third-party application for all users on the Zoom Marketplace, which will use powerful AI models to analyze media data of meetings to provide richer and more effective real-time analysis results.
- Alice started a meeting on the Zoom desktop client, and then she clicked the "Apps" button on the bottom bar. In the list of apps on the right panel, Alice finds the RTMS app and clicks it to open. Then, clicks the "Start" button on the app WebView UI.
- When the meeting client receives this signal of the button click, as the design flow, the meeting client will obtain the application ID (aka client ID) from Alice's user information obtained from the Zoom web.
- The meeting client then takes the client ID as a parameter and calls the Zoom Web API to get a gateway session ID, which will be sent to the meeting server (MMR) to invite the RTMS server to join. Meanwhile, when the Zoom web received the client ID, the Zoom web will verify it whether belongs to the user who initiated this request.
- After finishing the verification, the Zoom web generates a session ID associated with this request and responds to the meeting client. The meeting client then calls the MMR function via the client SDK interface.
- After MMR receives this message, the MMR will send a RTMS server invitation request to the Zoom web, the Zoom web then composes a join event (includes

session ID and 3p service type, etc.) and post a message to the AsyncMQ.

- Then see General Flows.

General Flows

- This event will be sent to a RTMS server through the AsyncMQ agent (consume topic) and the RTMS ZC. Once the join event is received, the RTMS server calls the Zoom web "conf/gw/j" API to obtain a meeting token which is used to join MMR. When the Zoom web received the API call from the RTMS server, it will compose a response and send back.
    - Besides the original parameters, the following parameters will also be added into response. The `signatures` is a list containing hashes signed with all client secrets (for covering secret rotate case).

```
userId == who initiated this request
clientID == the ID of the application on the Zoom marketplace
ownerId == the user ID associated with the application
accountId == the user's organization account ID
signatures == the hash list of the application info
```

- The RTMS server starts to join the meeting on MMR. After joining successfully, on the meeting clients, participants will receive a disclaimer informing them that a third-party application will acquire and transmit the meeting streams. Participants have the option to opt out or remain in the meeting. On the backend, The RTMS server produces a message and posts it to the AsyncMQ topic.
    - The payload of the notification message includes the following parameters. The server URL follows the standard pattern including protocol header, server domain or IP, and port.

```
{
    "eventType": "meeting.rtms.started",
    "eventTime": 1732313171881,
    "clientId": "xxxxxxxxxx",
    "userId": "xxxxxxxxxx",
    "accountId": "xxxxxxxxxx",
    "payload": {
        "event": "meeting.rtms.started",
        "event_ts": 1732313171881,
        "payload": {
            "operator_id": "xxxxxxxxxx",
            "object": {
                "meeting_uuid": "4444AAAiAAAAAiAiAiiAii==",
                "rtms_stream_id":
"609340fb2a7946909659956c8aa9250c",
                "server_urls": "wss://127.0.0.1:443"
            }
        }
    }
}
```

- The marketplace event service consumes this topic and subsequently looks up the 2p or 3p service URL based on the client ID, user ID, and account ID. Upon obtaining the service URL, the event service then sends a notification to the 2p or 3p via Webhook.
    - The notification event sent to 2p or 3p is as follows.

```
{
    "event": "meeting.rtms.started",
    "event_ts": 1732313171881,
    "payload": {
        "operator_id": "xxxxxxxxxx",
        "object": {
            "meeting_uuid": "4444AAAiAAAAAiAiAiiAii==",
            "rtms_stream_id":
"609340fb2a7946909659956c8aa9250c",
            "server_urls": "wss://127.0.0.1:443"
        }
    }
}
```

```
{
    "event": "meeting.rtms.stopped",
    "event_ts": 1732313171881,
    "payload": {
        "operator_id": "xxxxxxxxxx",
        "object": {
            "meeting_uuid": "4444AAAiAAAAAiAiAiiAii==",
            "rtms_stream_id":
"609340fb2a7946909659956c8aa9250c"
        }
    }
}
```

- When the 2p or 3p service receives the RTMS started event, their clients can establish a connection to the RTMS server using the server URL obtained from the Zoom notification. After completing the TCP connection, the 2p or 3p client must send a `SIGNALING_HAND_SHAKE_REQ` message through the socket. The payload of the message must include the `meeting_uuid`, `rtms_stream_id`, and `signature`.

- The 2p or 3p will need to generate the signature by adhering to the following structure then sign it with the client secret.

```
HMACSHA256(client_id + "," + meeting_uuid + "," +
rtms_stream_id, secret);
```

- The RTMS server receives the signaling handshake message and will
  - Firstly, verify the `meeting_uuid` and `rtms_stream_id` whether they exist on the current RTMS server. If not, terminate the connection immediately.
  - Secondly, use the signature list sent by the Zoom web to compare and verify the signature sent by the client. If the verification fails, respond with a `SIGNALING_HAND_SHAKE_RESP` message containing a status code and reason, terminate the connection to the 2p or 3p, send an error response to the upper layer, and terminate the session and leave the meeting.
  - Once the above verifications are passed, respond with a `SIGNALING_HAND_SHAKE_RESP` message containing a `media_server` information. In the first phase, the media server section only includes a `server_urls` field, which lists all protocols supported by RTMS. The 2p and 3p can then select any one of these protocols to establish a connection.
  - Meanwhile, RTMS server will send a `SESSION_STATE_UPDATE` message containing the `session_id` and state of `STARTED` through the signal connection where informing the 2p or 3p that a new RTMS session is added and started.
- The 2p or 3p must establish another connection for the media data transmission to the RTMS server using the server URL obtained from the signaling handshake response message. After completing the connection, the 2p or 3p client must send a `DATA_HAND_SHAKE_REQ` message to the RTMS server.
  - The payload of the message must include the `meeting_uuid`, `rtms_stream_id`, and `signature` which used in the signaling connection.
  - The `media_type` is also must be included and it used to identify which type of media data that you want to receive through this data connection. If the value set to `ALL`, meaning all type of media data will be transmitting

through a single connection, however, it's not recommended. Zoom suggests each type of media data through a separate connection.

- If developers want to encrypt the payload, the `payload_encryption` must be set to true on each connection.
  - **Note:** Even though the `payload_encryption` value is false but if the transmission protocol is UDP, the payload still will be encrypted enforced.
- The `media_params` is optional and used to define the parameters for corresponding media type that the app wants to transmit in that connection.
  - For audio data transmission
    - The RTMS server supports transmitting mixed audio stream and separate (active speakers) audio streams.
    - The RTMS server will utilize the following media parameters as default value. However, if the 2p or 3p requires a different one, they must specify it in the request.

```
{
    "content_type": RAW_AUDIO,      // Refer to MEDIA_CONT
    "sample_rate": SR_16K,          // Refer to AUDIO_SAMP
    "channel": MONO,                // Refer to AUDIO_CHAN
    "codec": L16,                   // Refer to MEDIA_PAYL
    "data_opt": AUDIO_MIXED_STREAM, // Refer to MEDIA_DATA
    "send_interval": 20
}
```

  - For video data transmission
    - In the V1, the RTMS server only supports transmitting active speaker video stream. The `data_opt` will not be taken effect even if set to other values.
    - The RTMS server will utilize the following media parameters as default value. However, if the 2p or 3p requires a different one, they must specify it in the request.

```
{
    "content_type": RAW_VIDEO, // Refer to MEDIA_CONTENT_T
    "codec": JPG,              // Refer to MEDIA_PAYLOAD_T
    "resolution": HD,          // Refer to MEDIA_RESOLUTIC
    "fps": 5
}
```

- The RTMS server receives the data handshake message and will
  a. Firstly, verify the `meeting_uuid`, `rtms_stream_id`, and `signature` in the same manner as the signaling flow.
  b. Besides, if the media parameters are specified, the RTMS server will verify each value. If any of the values are not allowed from the list, the RTMS server will respond with a `DATA_HAND_SHAKE_RESP` message, indicating a `STATUS_INVALID_MEDIA_PARAM` status code and the reason for the 2p or 3p. Subsequently, the RTMS server will wait for a new request for 5s, if no new message is received within this timeframe, terminate the media connection and corresponding signaling connection, respond with an error to the upper layer, and then terminate the session and leave the meeting.
  c. Once the above verifications are passed, respond with a `DATA_HAND_SHAKE_RESP` message containing a `STATUS_OK` status code, the `sequence` and the `media_params`. **Each value within the media parameter represents the negotiated format that will be applied to the audio data.**
- At this point, once signaling and media connections established, the session becomes fully active and operational. On the meeting clients, participants will receive a toast notification indicating that the stream has started, while an indicator (AAN) appears in the top right corner displaying the third-party application and user who is acquiring the streams through the indicator panel. Simultaneously, on the backend, a `STREAM_STATE_UPDATE` message will be transmitted, containing `rtms_stream_id`, the state of `STARTED` and the `timestamp` of the first packet through the signaling connection. Meanwhile, the data connection should be started to receive the media data.

- Once the whole connection establishment process is completed, 2p or 3p can send a `EVENT_SUBSCRIPTION` message to subscribe to or unsubscribe from desired events.
    - Currently, three event messages are supported for subscription or un-subscription in the RTMS: `ACTIVE_SPEAKER_CHANGE`, `PARTICIPANT_JOIN`, and `PARTICIPANT_LEAVE`.
    - By default, if 2p or 3p chooses to receive the `AUDIO_MIXED_STREAM`, the corresponding `ACTIVE_SPEAKER_CHANGE` event will be sent out to 2p or 3p. If 2p or 3p chooses to receive the `AUDIO_MIXED_STREAM`, then the `PARTICIPANT_JOIN` and `PARTICIPANT_LEAVE` events will be sent out.
- To maintain the stability of the connections and prevent timeouts and disconnections, the RTMS server sends a `KEEP_ALIVE_REQ` message and the 2p or 3p must respond with a `KEEP_ALIVE_RESP` message.
    - In the signaling connection, the keep-alive request is sent every 5 seconds if the last message sends time exceeds 5 seconds. As a result, the 2p or 3p client must respond before the next keep-alive message arrives. If three consecutive keep-alive messages remain unanswered, the RTMS server will terminate both signaling and media connections, report an error to the upper layer, and subsequently leave the meeting.
    - In the media connection, if the last data packet transmission time exceeds 5 seconds or the stream status is PAUSED, the RTMS server actively sends a keep-alive request. Subsequently, if three consecutive keep-alive messages remain unanswered, the RTMS server will terminate both signaling and media connections, report an error to the upper layer, and then leave the meeting.
- When users pause/resume/stop the RTMS stream, RTMS server will send a `SESSION_STATE_UPDATE` message containing the `session_id` and state of `PAUSED` or `RESUMED` or `STOPPED` through the signal connection where informing the 2p or 3p that a RTMS session state has changed.
- Once all sessions have been terminated, RTMS server will send a `STREAM_STATE_UPDATE` message that includes the `rtms_stream_id`, the state

of `STOPPED` with `stop_reason`, and the `timestamp` that indicating a particular stream has completely ended.

Data Type Definitions

```
enum RTMS_MESSAGE_TYPE
{
    UNKNOWN,
    SIGNALING_HAND_SHAKE_REQ,  // Initializes the signaling connectior
    SIGNALING_HAND_SHAKE_RESP, // Indicates the response of initialed
    DATA_HAND_SHAKE_REQ,       // Initializes the media data connectic
    DATA_HAND_SHAKE_RESP,      // Indicates the response of initialed
    EVENT_SUBSCRIPTION,        // Indicates which events want to subsc
    EVENT_UPDATE,              // Indicates a specific event happens,
    STREAM_STATE_UPDATE,       // Indicates the stream state changed,
    SESSION_STATE_UPDATE,      // Indicates the session state updated,
    SESSION_STATE_REQ,         // Indicates querying the session state
    SESSION_STATE_RESP,        // Indicates the response of session st
    KEEP_ALIVE_REQ,            // Indicates it is a keep-alive request
    KEEP_ALIVE_RESP,           // Indicates it is a keep-alive respons
    MEDIA_DATA_AUDIO,          // Indicates audio data is being transr
    MEDIA_DATA_VIDEO,          // Indicates video data is being transr
    MEDIA_DATA_SHARE,          // Indicates sharing data is being trar
    MEDIA_DATA_CHAT,           // Indicates the meeting chat messages
    MEDIA_DATA_TRANSCRIPT      // Indicates the transcripts of meeting
}
```

```
enum RTMS_EVENT_TYPE
{
    ACTIVE_SPEAKER_CHANGE, // Indicates who the most recent active spe
    PARTICIPANT_JOIN,      // Indicates a new participant joined this
    PARTICIPANT_LEAVE      // Indicates a participant is leaving this
}
```

```
enum RTMS_STATUS_CODE
{
    STATUS_OK,
    STATUS_CONNECTION_TIMEOUT,
    STATUS_INVALID_JSON_MSG,
    STATUS_INVALID_MESSAGE_TYPE,
    STATUS_MSG_TYPE_NOT_EXIST,
    STATUS_MSG_TYPE_NOT_UINT,
    STATUS_MEETING_UUID_NOT_EXIST,
    STATUS_MEETING_UUID_IS_EMPTY,
    STATUS_RTMS_STREAM_ID_NOT_EXIST,
    STATUS_RTMS_STREAM_ID_IS_EMPTY,
    STATUS_SESSION_NOT_FOUND,
    STATUS_SIGNATURE_NOT_EXIST,
    STATUS_INVALID_SIGNATURE,
    STATUS_INVALID_MEETING_OR_STREAM_ID,
    STATUS_DUPLICATE_SIGNAL_REQUEST,
    STATUS_MEDIA_TYPE_NOT_EXIST,
    STATUS_MEDIA_TYPE_NOT_UINT,
    STATUS_MEDIA_DATA_ALL_CONNECTION_EXIST,
    STATUS_DUPLICATE_MEDIA_DATA_CONNECTION,
    STATUS_MEDIA_PARAMS_NOT_EXIST,
    STATUS_INVALID_MEDIA_PARAMS,
    STATUS_NO_MEDIA_TYPE_SPECIFIED,
    STATUS_INVALID_MEDIA_AUDIO_PARAMS,
    STATUS_MEDIA_AUDIO_CONTENT_TYPE_NOT_UINT,
    STATUS_INVALID_MEDIA_AUDIO_CONTENT_TYPE,
    STATUS_MEDIA_AUDIO_SAMPLE_RATE_NOT_UINT,
    STATUS_INVALID_MEDIA_AUDIO_SAMPLE_RATE,
    STATUS_MEDIA_AUDIO_CHANNEL_NOT_UINT,
    STATUS_INVALID_MEDIA_AUDIO_CHANNEL,
    STATUS_MEDIA_AUDIO_CODEC_NOT_UINT,
    STATUS_INVALID_MEDIA_AUDIO_CODEC,
    STATUS_MEDIA_AUDIO_DATA_OPT_NOT_UINT,
```

```
    STATUS_INVALID_MEDIA_AUDIO_DATA_OPT,
    STATUS_MEDIA_AUDIO_SEND_INTERVAL_NOT_UINT,
    STATUS_MEDIA_AUDIO_COMBINE_FRAME_NOT_BOOL,
    STATUS_INVALID_MEDIA_VIDEO_PARAMS,
    STATUS_INVALID_MEDIA_SHARE_PARAMS,
    STATUS_INVALID_AUDIO_DATA_BUFFER,
    STATUS_POST_FIRST_PACKET_FAILURE
}
```

```
enum RTMS_SESSION_STATE
{
    INACTIVE,   // Default state
    INITIALIZE, // A new session is initializing
    STARTED,    // A new Session is started
    PAUSED,     // A session is paused
    RESUMED,    // A session is resumed
    STOPPED     // A session is stopped
}
```

```
enum RTMS_STREAM_STATE
{
    INACTIVE,   // Default state
    ACTIVE,     // Media data has started to transmit
    TERMINATED, // Stream is terminated, e.g. all sessions are stopped or meeting is ended
    INTERRUPTED // Signal or any data connections encountered a problem
}
```

```
enum RTMS_STOP_REASON
{
    UNKNOWN,
    // Stopped when triggered by host, returned on session update mess
    STOP_BC_HOST_TRIGGERED,
    // Stopped when triggered by userself, returned on session update
    STOP_BC_USER_TRIGGERED,
    // Stopped when app user left meeting, returned on session update
    STOP_BC_USER_LEFT,
    // Stopped when app user ejected by meeting host, returned on sess
    STOP_BC_USER_EJECTED,
    // Stopped when host disabled app user or entire app in the meetir
    STOP_BC_APP_DISABLED_BY_HOST,
    // Stopped when meeting is ended, returned on stream update messag
    STOP_BC_MEETING_ENDED,
    // Stopped when stream canceled by participant request, returned c
    STOP_BC_STREAM_CANCELED,
    // Stopped when host disabled all apps in the meeting, returned or
    STOP_BC_ALL_APPS_DISABLED,
    // Stopped if any internal exceptions, e.g. post asyncmq message t
    STOP_BC_INTERNAL_EXCEPTION,
    // Stopped when the connection timed out, returned on stream updat
    STOP_BC_CONNECTION_TIMEOUT,
    // Stopped when a connection interrupted, returned on stream updat
    STOP_BC_CONNECTION_INTERRUPTED,
    // Stopped when a connection closded by app, returned on stream up
    STOP_BC_CONNECTION_CLOSED_BY_CLIENT,
    // Stopped when received exit signal, returned on stream update me
    STOP_BC_EXIT_SIGNAL
}
```

```
enum MEDIA_CONTENT_TYPE
{
    RTP = 1,     // Real-time audio and video
    RAW_AUDIO,   // Real-time audio
    RAW_VIDEO,   // Real-time video
    FILE_STREAM, // File stream
    TEXT         // Media data is text based, such as Chat messages and Transcripts
}
```

```
enum MEDIA_PAYLOAD_TYPE
{
    L16 = 1, // Audio, uncompressed raw data
    PCMA,    // Audio
    PCMU,    // Audio
    G722,    // Audio
    OPUS,    // Audio
    JPG,     // Video and Sharing, when fps <= 5
    H264     // Video and Sharing, when fps > 5
}
```

```
enum MEDIA_DATA_TYPE
{
    AUDIO = 1,
    VIDEO,
    DESKSHARE,
    TRANSCRIPT,
    CHAT,
    ALL
}
```

```
enum MEDIA_DATA_OPTION
{
    AUDIO_MIXED_STREAM = 1,     // Data is mixed audio stream
    AUDIO_MULTI_STREAMS,        // Data is audio stream(s) of active speaker(s)
    VIDEO_SINGLE_ACTIVE_STREAM, // Data is single video stream of active speaker
    VIDEO_MIXED_SPEAKER_VIEW,   // Data is mixed video stream using speaker view
    VIDEO_MIXED_GALLERY_VIEW    // Data is mixed video stream using gallery view
}
```

```
enum MEDIA_RESOLUTION
{
    SD = 1, // 480p or 360p, 854x480 or 640x360
    HD,     // 720p, 1280 x 720
    FHD,    // 1080p, 1920 x 1080
    QHD     // 2K, 2560 x 1440
}
```

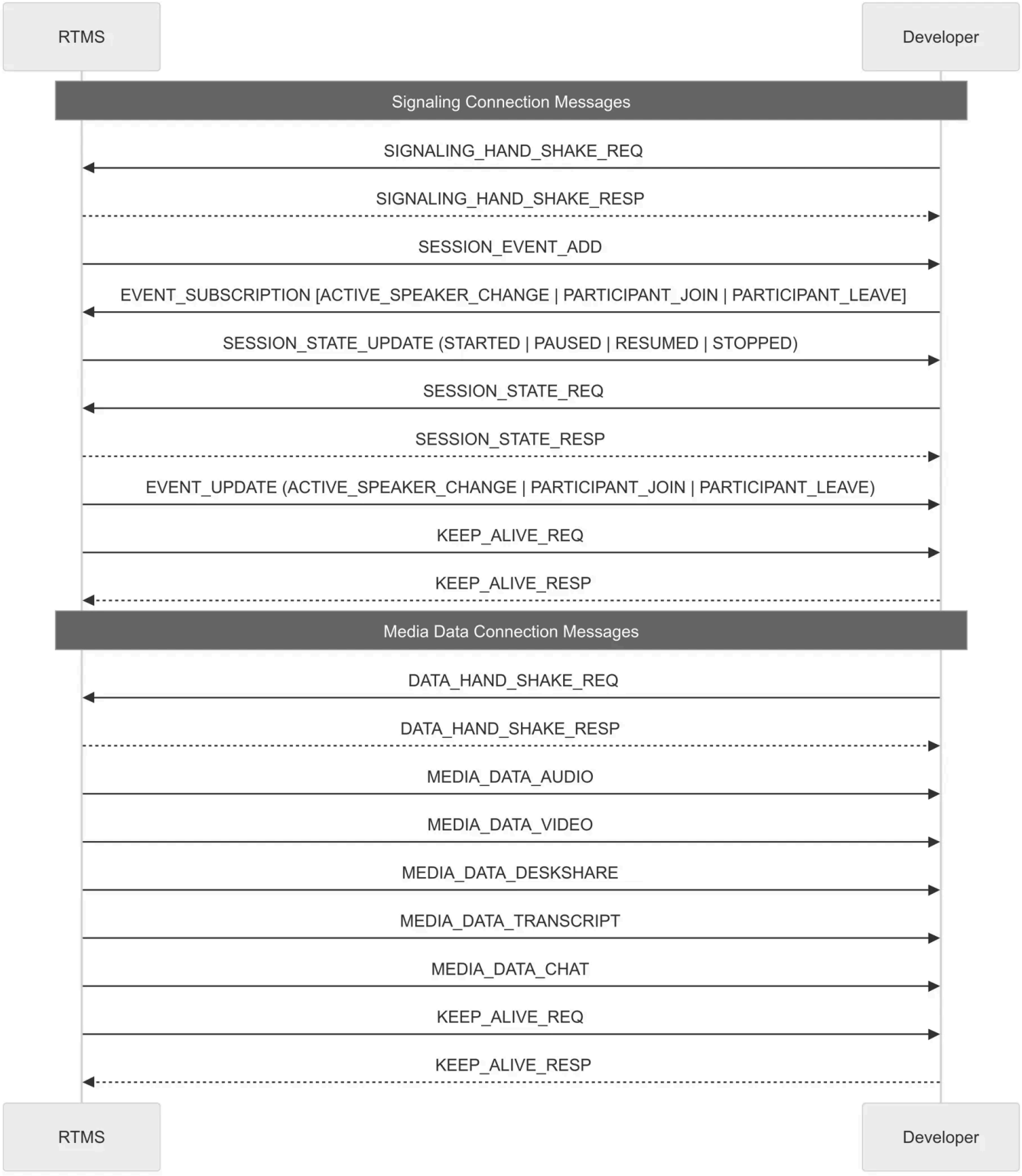
```
enum AUDIO_SAMPLE_RATE
{
    SR_16K = 1,
    SR_32K,
    SR_48K
}
```

```
enum AUDIO_CHANNEL
{
    MONO = 1,
    STEREO
}
```

```
enum TRANSMISSION_PROTOCOL
{
    WEBSOCKET = 1,
    RTMP,
    UDP,
    WEBRTC
}
```

Messages

This section defines interactive messages including signaling and media data.



Signaling and Meta-Data

The request message for the signaling handshake

```
    {
        "msg_type": SIGNALING_HAND_SHAKE_REQ,
        "protocol_version": 1,
        "meeting_uuid": "xxxxxxxxxx",
        "rtms_stream_id": "xxxxxxxxxx",
        "signature": "xxxxxxxxxx"
    }
```

The response message for the signaling handshake

```
{
    "msg_type": SIGNALING_HAND_SHAKE_RESP,
    "protocol_version": 1,
    "status_code": STATUS_OK, // If the status is not STATUS_OK (0) me
    "reason": "", // If the status is failed, the reason will be fille
    "media_server": {
        "server_urls": {
            "audio": "wss://0.0.0.0:443,udp://0.0.0.0:8801",
            "video": "wss://0.0.0.0:443,udp://0.0.0.0:8801",
            "transcript": "wss://0.0.0.0:443",
            "all": "wss://0.0.0.0:443"
        },
        "srtp_keys": {
            "audio": "xxxxxxxxxx",
            "video": "xxxxxxxxxx",
            "share": "xxxxxxxxxx"
        }
    }
}
```

The message for subscribing or unsubscribing stream events

```
{
    "msg_type": EVENT_SUBSCRIPTION,
    "events": [
        {
            "event_type": ACTIVE_SPEAKER_CHANGE,
            "subscribe": true|false // true means subscribe, false mea
        },
        {
            "event_type": PARTICIPANT_JOIN,
            "subscribe": true
        },
        {
            "event_type": PARTICIPANT_LEAVE,
            "subscribe": false
        }
    ]
}
```

- This message does not require a response.
- Currently, only `ACTIVE_SPEAKER_CHANGE` , `PARTICIPANT_JOIN` , and `PARTICIPANT_LEAVE` three events are supported.

The message for updating stream state

```
{
    "msg_type": STREAM_STATE_UPDATE,
    "rtms_stream_id": "xxxxxxxxxx",
    "state": ACTIVE|TERMINATED,      // Refer to RTMS_STREAM_STATE de
    "reason": STOP_BC_MEETING_ENDED, // Refer to RTMS_STOP_REASON def
    "timestamp": 1727384349123
}
```

- The `STARTED` state update message will be sent when the first media packet has been transmitted.
- The `STOPPED` state update message will be sent when the stream needs to be stopped, such as, the session(s) associated with this stream are ended, or meeting is closed, or other exceptional cases.
- This message does not require a response.

The message for updating session state

```
{
    "msg_type": SESSION_STATE_UPDATE,
    "session_id": "xxxxxxxxxx",
    "state": STARTED|PAUSED|RESUMED|STOPPED,
    "stop_reason": ACTION_BY_HOST|ACTION_BY_USER, // Refer to RTMS_ST(
    "timestamp": 1727384349123
}
```

- This message does not require a response.

The request message for querying the current session state

```
{
    "msg_type": SESSION_STATE_REQ,
    "session_id": "xxxxxxxxxx"
}
```

- The current session state can be queried by `session_id` .
- The RTMS server will reply with a message of type " `SESSION_STATE_RESP` " containing the current session state.

The response message for the current session state request

```
{
    "msg_type": SESSION_STATE_RESP,
    "session_id": "xxxxxxxxxx",
    "session_state": STARTED|PAUSED|RESUMED
}
```

The message of the meta-data for the active speaker change

```
{
    "msg_type": EVENT_UPDATE,
    "event": {
        "event_type": ACTIVE_SPEAKER_CHANGE,
        "current_id": 0|11223344, // If current_id is 0 means it's fi
        "new_id": 22334455,
        "name": "John Smith",
        "timestamp": 11111111111
    }
}
```

- This message does not require a response.

The message of the meta-data for the participant events

```
{
    "msg_type": EVENT_UPDATE,
    "event": {
        "event_type": PARTICIPANT_JOIN,
        "participants": [
            {
                "user_id": 16778240,   // The unique participant id ir
                "name": "John Smith"   // User display name in the mee
            },
            {
                "user_id": 33556610,
                "name": "Alice"
            }
        ]
    }
}
```

```
{
    "msg_type": EVENT_UPDATE,
    "event": {
        "event_type": PARTICIPANT_LEAVE,
        "participants": [16778240, 33556610] // The user_id list of wh
    }
}
```

- By default, `PARTICIPANT_JOIN` and `PARTICIPANT_LEAVE` events will be sent to third parties if `AUDIO_MULTI_STREAMS` is being selected.
- These messages do not require a response.

The message of the keep-alive request

```
{
    "msg_type": KEEP_ALIVE_REQ,
    "sequence": 0,
    "timestamp": 1727384349123
}
```

- The sequence number has to be presented in the response.

The response message of the keep-alive request

```
{
    "msg_type": KEEP_ALIVE_RESP,
    "sequence": 0,
    "timestamp": 1727384349123 // this timestamp is the request timest
}
```

Media Data

The request message for the data handshake

```
{
    "msg_type": DATA_HAND_SHAKE_REQ,
    "protocol_version": 1,
    "sequence": 0,
    "meeting_uuid": "xxxxxxxxxx",
    "rtms_stream_id": "xxxxxxxxxx",
    "signature": "xxxxxxxxxx",
    /*
     * Refer to MEDIA_DATA_TYPE definition. If the value set to "ALL",
     * meaning all media category data where the app supported will be
     * transmitted through one socket connection.
     */
    "media_type": AUDIO|VIDEO|TRANSCRIPT|ALL,
    /*
     * The default value is false (0) for any TLS connection, and the
     * encryption keys are provided in the signal hand shake resp.
     *  - If parameter value is true (1) means the payload will be enc
     *  - If parameter value is false (0) but the transmission protoco
     *    the payload still will be encrypted enforced
     */
    "payload_encryption": true|false,
    /*
     * Optional, the params will be set to default value if not speci
     * and return in the DATA_HAND_SHAKE_RESP message
     */
    "media_params": {
        "audio": {
            "content_type": RAW_AUDIO,      // Refer to MEDIA_CONTENT_
            "sample_rate": SR_16K,          // Refer to AUDIO_SAMPLE_R
            "channel": MONO,                // Refer to AUDIO_CHANNEL
            "codec": L16,                   // Refer to MEDIA_PAYLOAD_
            "data_opt": AUDIO_MIXED_STREAM, // Refer to MEDIA_DATA_OPT
            "send_rate": 100  // Sending rate, must be multiple of 20
        },
```

```
        "video": {
            "codec": JPG,     // Refer to MEDIA_PAYLOAD_TYPE definitic
            "resolution": HD, // Refer to MEDIA_RESOLUTION definitions
            "fps": 5
        }
    }
}
```

The response message for the data handshake

```
{
    "msg_type": DATA_HAND_SHAKE_RESP,
    "protocol_version": 1,
    "status_code": STATUS_OK, // If the status is not STATUS_OK meanir
    "reason": "", // If the status is failed, the reason will be fille
    "sequence": 0,
    "payload_encrypted": true|false, // For UDP, the payload will be e
    "media_params": {
        "audio": {
            "content_type": RAW_AUDIO,      // Refer to MEDIA_CONTENT_
            "sample_rate": SR_16K,          // refer to AUDIO_SAMPLE_F
            "channel": MONO,                // refer to AUDIO_CHANNEL
            "codec": L16,                   // refer to MEDIA_PAYLOAD_
            "data_opt": AUDIO_MIXED_STREAM, // Refer to MEDIA_DATA_OPT
            "send_rate": 100   // Sending rate, must be multiple of 20
        },
        "video": {
            "content_type": RAW_VIDEO,
            "codec": JPG,     // Refer to MEDIA_PAYLOAD_TYPE definitic
            "resolution": HD, // Refer to MEDIA_RESOLUTION definition
            "fps": 5
        },
        "transcript": {
            "content_type": TEXT
        }
    }
}
```

The message of sending audio data

Note: If developers select `AUDIO_MULTI_STREAMS`, then within a fixed interval, each user's audio data will be transmitted through different messages.

```
{
    "msg_type": MEDIA_DATA_AUDIO, // Media p is raw
    "content": {
        "user_id": 16778240|0, // if user_id is 0, means the packet is
        "data": (media packet),
        "timestamp": 1111111111,
    }
}
```

```
{
    "msg_type": MEDIA_DATA_AUDIO, // Media data packs by RTP
    "content": {
        "data": (media packet)
    }
}
```

The message of video data

```
{
    "msg_type": MEDIA_DATA_VIDEO,
    "content": {
        "user_id": 16778240,
        "data": (media packet)
    }
}
```

The message of chat and transcript data

Note: Each user's transcript data will be transmitted through different messages.

```
{
    "msg_type": MEDIA_DATA_CHAT|MEDIA_DATA_TRANSCRIPT,
    "content": {
        "user_id": 19778240,
        "timestamp": 1727384349000,
        "data": "xxxxxxxxxx"
    }
}
```

Failover

Connection Broken between RTMS and MMR

When the connection between the RTMS server and MMR server is broken, the RTMS server will try to rejoin with old meeting parameters. Meanwhile, the MMR will also re-invite the RTMS server to join the meeting.

- If the rejoin request reaches the MMR server first, the MMR will reject the re-invite request. After the rejoin successfully, nothing impact on the connections between the RTMS and 2p and 3p. The RTMS server will continue to send the media packets.
- If the re-invite request reaches the MMR server first, MMR will reject the rejoin request. The old RTMS server will then terminate the connection and delete all information about the conference and session from the server.

Connection Broken between RTMS and Receiver

When the connection between the RTMS server and receiver is broken, the RTMS server will send a new invitation message with old parameters. In the meantime, the receiver may establish a new connection. No matter which one reaches the RTMS server first, the RTMS server will only proceed with the first connection and reject the second one.

RTMS Server Selection Mechanism

To simplify the regional RTMS server selection mechanism, the following strategies are currently used.

- If the RTMS has a zone configured in the Data Center where the meeting top MMR is located, prioritize the RTMS zone corresponding to the meeting top MMR zone, Zoom web chooses up to 3 zones and posts the invitation messages.
- If the RTMS does not have a zone configured in the Data Center where the meeting top MMR is located (not US), prioritize and return the RTMS zone in the Data Center of US first, Zoom web chooses up to 3 zones and posts the invitation messages.
- If the RTMS does not have a zone configured in the Data Center where the meeting top MMR is located (non-US), and the US is not in the list of countries allowed by the meeting original host, then up to 3 zones are randomly selected from the datacenters where RTMS is deployed.