AVDECC clarifications

Part 1 – ACMP

Revision 3 [WIP]
8/5/2016

1. Introduction

The goal of this document is to clarify some parts of the AVDECC specification (IEEE Std. 1722.1™-2013).

This document tries to describe precisely all the fields of the ACMPDU for all the possible message types. It also describes a few scenarios that raise some questions not precisely answered in the specification.

This document is open for comments and proposals. Please do not hesitate to add/discuss/correct any point if needed. The goal is to finally have everybody agree so that a manufacturer releasing an AVDECC device now can be confident that it will interoperate with any other AVDECC device that will be released in the future.

2. Revision History

Version	Description
1	Initial release of the document February 19th, 2016
2	Updated the “ACMP scenarios” section according to comments from AVnu members. Updated description of the “stream_dest_mac” field of the GET_TX_STATE_RESPONSE and GET_TX_CONNECTION_RESPONSE messages. February 25th, 2016
3	Changed description of “connection_count”, “FAST_CONNECT” and “SAVED_STATE” fields Added “discussion” in “ACMP PDU format” section Changed Talker behaviour when not enough bandwidth on the upstream link Changed acquirement/lock scenario: accept connection even if acquired/locked Changed Listener behaviour when SRP parameters are different from AVDECC parameters August 5th, 2016

Contents

1. Introduction 2
2. Revision History 2
3. ACMP PDU format 5
   - 3.1. CONNECT_RX_COMMAND and CONNECT_RX_RESPONSE 5
   - 3.2. CONNECT_TX_COMMAND and CONNECT_TX_RESPONSE 7
   - 3.3. DISCONNECT_RX_COMMAND and DISCONNECT_RX_RESPONSE 10
   - 3.4. DISCONNECT_TX_COMMAND and DISCONNECT_TX_RESPONSE 12
   - 3.5. GET_RX_STATE_COMMAND and GET_RX_STATE_RESPONSE 15
   - 3.6. GET_TX_STATE_COMMAND and GET_TX_STATE_RESPONSE 18
   - 3.7. GET_TX_CONNECTION_COMMAND and GET_TX_CONNECTION_RESPONSE 21
   - 3.8. Discussion 24
     - 3.8.1. Operations of ACMP towards SRP 24
     - 3.8.2. “connection_count” field 24
     - 3.8.3. “CLASS_B”, “SUPPORTS_ENCRYPTED” and “ENCRYPTED_PDU” fields 25
     - 3.8.4. Saved state and Fast Connect mode 25
4. ACMP scenarios 26
   - 4.1. Acquirement/lock and connections – Listener acceptance 26
   - 4.2. Acquirement/lock and connections – Talker acceptance 26
   - 4.3. State of the Listener during Fast Connect attempt 27
   - 4.4. Request to stop Fast Connect attempts 28
   - 4.5. Connection succeeded on Talker and failed on Listener 29
   - 4.6. Disconnection succeeded on Listener and failed on Talker 30
   - 4.7. Talker connected to a ghost Listener 31
   - 4.8. ACMP connection fails if not enough bandwidth on the upstream link 32
   - 4.9. ACMP connection fails because MAAP fails 33
   - 4.10. MAAP fails after a successful ACMP connection 33
   - 4.11. SRP stream parameters different from AVDECC stream parameters 34
   - 4.12. ACMP connection succeeds even if SRP has not reserved bandwidth yet 35

3. ACMP PDU format

3.1. CONNECT_RX_COMMAND and CONNECT_RX_RESPONSE

	CONNECT_RX_COMMAND	**CONNECT_RX_RESPONSE (status=SUCCESS)**	**CONNECT_RX_RESPONSE (status<>SUCCESS)**
controller_entity_id	Entity ID of the controller sending the command	Copy “controller_entity_id” of the CONNECT_RX_COMMAND message	Copy “controller_entity_id” of the CONNECT_RX_COMMAND message
talker_entity_id	Entity ID of the talker which is target of the command	Copy “talker_entity_id” of the CONNECT_RX_COMMAND message	Copy “talker_entity_id” of the CONNECT_RX_COMMAND message
listener_entity_id	Entity ID of the listener which is target of the command	Copy “listener_entity_id” of the CONNECT_RX_COMMAND message	Copy “listener_entity_id” of the CONNECT_RX_COMMAND message
talker_unique_id	Unique ID of the Stream source which is target of the command	Copy “talker_unique_id” of the CONNECT_RX_COMMAND message	Copy “talker_unique_id” of the CONNECT_RX_COMMAND message
listener_unique_id	Unique ID of the Stream sink which is target of the command	Copy “listener_unique_id” of the CONNECT_RX_COMMAND message	Copy “listener_unique_id” of the CONNECT_RX_COMMAND message
stream_id	00:00:00:00:00:00:00:00	Copy “stream_id” of the CONNECT_TX_RESPONSE message	Copy “stream_id” of the CONNECT_TX_RESPONSE message (00:00:00:00:00:00:00:00 if there was no message from the talker)
stream_dest_mac	00:00:00:00:00:00	Copy “stream_dest_mac” of the CONNECT_TX_RESPONSE message	Copy “stream_dest_mac” of the CONNECT_TX_RESPONSE message (00:00:00:00:00:00 if there was no message from the talker)
stream_vlan_id	0	Copy “stream_vlan_id” of the CONNECT_TX_RESPONSE message	Copy “stream_vlan_id” of the CONNECT_TX_RESPONSE message (0 if there was no message from the talker)

connection_count	0	Copy “connection_count” of the CONNECT_TX_RESPONSE message	Copy “connection_count” of the CONNECT_TX_RESPONSE message (0 if there was no message from the talker)
sequence_id	changed by the sender at each new command sent	Copy “sequence_id” of the CONNECT_RX_COMMAND message	Copy “sequence_id” of the CONNECT_RX_COMMAND message
flags.CLASS_B	0	Copy “flags.CLASS_B” of the CONNECT_TX_RESPONSE message	Copy “flags.CLASS_B” of the CONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.FAST_CONNECT	0	0	0
flags.SAVED_STATE	0	1 if the listener implements Fast Connect mode, 0 otherwise	0
flags.STREAMING_WAIT	0 if the controller wants that the talker starts streaming data and the listener starts playing these data immediately, 1 otherwise	Copy “flags.STREAMING_WAIT” of the CONNECT_TX_RESPONSE message	Copy “flags.STREAMING_WAIT” of the CONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.SUPPORTS_ENCRYPTED	0	Copy “flags.SUPPORTS_ENCRYPTED” of the CONNECT_TX_RESPONSE message	Copy “flags.SUPPORTS_ENCRYPTED” of the CONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.ENCRYPTED_PDU	0	Copy “flags.ENCRYPTED_PDU” of the CONNECT_TX_RESPONSE message	Copy “flags.ENCRYPTED_PDU” of the CONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.TALKER_FAILED	0	1 if the listener is already receiving a Talker Failed attribute for this stream, 0 otherwise	0
Other bits of the flags	0	Copy other bits of the flags of the CONNECT_TX_RESPONSE message	Copy other bits of the flags of the CONNECT_TX_RESPONSE message
reserved	0	0	0

Marc ILLOUZ 8/5/16 5:49 PM

Comment [1]: Can the controller set the field to 1? If yes, does it mean “force Class B”, or “prefer Class B”? What does 0 mean? “force Class A”, or “prefer Class A”, or “choose by yourself”?
Is it possible for the controller to force other parameters of the stream (stream_id, stream_dest_mac, stream_vlan_id, ENCRYPTED_PDU) in the CONNECT_RX_COMMAND?

3.2. CONNECT_TX_COMMAND and CONNECT_TX_RESPONSE

	CONNECT_TX_COMMAND	**CONNECT_TX_RESPONSE (status<>TALKER_UNKNOWN_ID)**	**CONNECT_TX_RESPONSE (status=TALKER_UNKNOWN_ID)**
controller_entity_id	Copy "controller_entity_id" of the CONNECT_RX_COMMAND message (saved value from the initial message in case of Fast Connect mode)	Copy "controller_entity_id" of the CONNECT_TX_COMMAND message	Copy "controller_entity_id" of the CONNECT_TX_COMMAND message
talker_entity_id	Copy "talker_entity_id" of the CONNECT_RX_COMMAND message (saved value from the initial message in case of Fast Connect mode)	Copy "talker_entity_id" of the CONNECT_TX_COMMAND message	Copy "talker_entity_id" of the CONNECT_TX_COMMAND message
listener_entity_id	Copy "listener_entity_id" of the CONNECT_RX_COMMAND message (saved value from the initial message in case of Fast Connect mode)	Copy "listener_entity_id" of the CONNECT_TX_COMMAND message	Copy "listener_entity_id" of the CONNECT_TX_COMMAND message
talker_unique_id	Copy "talker_unique_id" of the CONNECT_RX_COMMAND message (saved value from the initial message in case of Fast Connect mode)	Copy "talker_unique_id" of the CONNECT_TX_COMMAND message	Copy "talker_unique_id" of the CONNECT_TX_COMMAND message
listener_unique_id	Copy "listener_unique_id" of the CONNECT_RX_COMMAND message (saved value from the initial message in case of Fast Connect mode)	Copy "listener_unique_id" of the CONNECT_TX_COMMAND message	Copy "listener_unique_id" of the CONNECT_TX_COMMAND message
stream_id	Copy "stream_id" of the CONNECT_RX_COMMAND message (00:00:00:00:00:00:00:00 in case	– If the Talker source is connected to at least one Listener sink: ID of the stream – Otherwise:	Copy "stream_id" of the CONNECT_TX_COMMAND message

	of Fast Connect mode)	00:00:00:00:00:00:00:00	
stream_dest_mac	Copy “stream_dest_mac” of the CONNECT_RX_COMMAND message (00:00:00:00:00:00 in case of Fast Connect mode)	– If the Talker source is connected to at least one Listener sink: Destination MAC address of the stream – Otherwise: 00:00:00:00:00:00	Copy “stream_dest_mac” of the CONNECT_TX_COMMAND message
stream_vlan_id	Copy “stream_vlan_id” of the CONNECT_RX_COMMAND message (0 in case of Fast Connect mode)	– If the Talker source is connected to at least one Listener sink: VLAN ID of the stream (0 indicates default VLAN ID of the SRP domain) – Otherwise: 0	Copy “stream_vlan_id” of the CONNECT_TX_COMMAND message
connection_count	0	Total count of Listener sinks connected to this stream source after execution of the command	0
sequence_id	changed by the sender at each new command sent	Copy “sequence_id” of the CONNECT_TX_COMMAND message	Copy “sequence_id” of the CONNECT_TX_COMMAND message
flags.CLASS_B	Copy “flags.CLASS_B” of the CONNECT_RX_COMMAND message (0 in case of Fast Connect mode)	– If the Talker source is connected to at least one Listener sink: 0 if the stream is Class A, 1 if the stream is Class B – Otherwise: 0	Copy “flags.CLASS_B” of the CONNECT_TX_COMMAND message
flags.FAST_CONNECT	0 if Controller Connect mode 1 if Fast Connect mode	Copy “flags.FAST_CONNECT” of the CONNECT_TX_COMMAND message	Copy “flags.FAST_CONNECT” of the CONNECT_TX_COMMAND message
flags.SAVED_STATE	0 if Controller Connect mode 1 if Fast Connect mode	Copy “flags.SAVED_STATE” of the CONNECT_TX_COMMAND message	Copy “flags.SAVED_STATE” of the CONNECT_TX_COMMAND message
flags.STREAMING_WAIT	Copy “flags.STREAMING_WAIT” of the CONNECT_RX_COMMAND message (saved value from last state in case of Fast Connect mode)	Copy “flags.STREAMING_WAIT” of the CONNECT_TX_COMMAND message	Copy “flags.STREAMING_WAIT” of the CONNECT_TX_COMMAND message
flags.SUPPORTS_ENCRYPTED	Copy “flags.SUPPORTS_ENCRYPTED” of	1 if the talker supports encryption of the PDUs, 0 otherwise	Copy “flags.SUPPORTS_ENCRYPTED” of

	the CONNECT_RX_COMMAND message (0 in case of Fast Connect mode)		the CONNECT_TX_COMMAND message
flags.ENCRYPTED_PDU	Copy “flags.ENCRYPTED_PDU” of the CONNECT_RX_COMMAND message (0 in case of Fast Connect mode)	– If the Talker source is connected to at least one Listener sink: 1 if the talker is configured to use encrypted PDUs for this stream, 0 otherwise – Otherwise: 0	Copy “flags.ENCRYPTED_PDU” of the CONNECT_TX_COMMAND message
flags.TALKER_FAILED	Copy “flags.TALKER_FAILED” of the CONNECT_RX_COMMAND message (0 in case of Fast Connect mode)	Copy “flags.TALKER_FAILED” of the CONNECT_TX_COMMAND message	Copy “flags.TALKER_FAILED” of the CONNECT_TX_COMMAND message
Other bits of the flags	Copy other bits of the flags of the CONNECT_RX_COMMAND message (0 in case of Fast Connect mode)	Copy other bits of the flags of the CONNECT_TX_COMMAND message	Copy other bits of the flags of the CONNECT_TX_COMMAND message
reserved	0	0	0

3.3. DISCONNECT_RX_COMMAND and DISCONNECT_RX_RESPONSE

	DISCONNECT_RX_COMMAND	**DISCONNECT_RX_RESPONSE (status=SUCCESS)**	**DISCONNECT_RX_RESPONSE (status<>SUCCESS)**
controller_entity_id	Entity ID of the controller sending the command	Copy “controller_entity_id” of the DISCONNECT_RX_COMMAND message	Copy “controller_entity_id” of the DISCONNECT_RX_COMMAND message
talker_entity_id	Entity ID of the talker which is target of the command	Copy “talker_entity_id” of the DISCONNECT_RX_COMMAND message	Copy “talker_entity_id” of the DISCONNECT_RX_COMMAND message
listener_entity_id	Entity ID of the listener which is target of the command	Copy “listener_entity_id” of the DISCONNECT_RX_COMMAND message	Copy “listener_entity_id” of the DISCONNECT_RX_COMMAND message
talker_unique_id	Unique ID of the Stream source which is target of the command	Copy “talker_unique_id” of the DISCONNECT_RX_COMMAND message	Copy “talker_unique_id” of the DISCONNECT_RX_COMMAND message
listener_unique_id	Unique ID of the Stream sink which is target of the command	Copy “listener_unique_id” of the DISCONNECT_RX_COMMAND message	Copy “listener_unique_id” of the DISCONNECT_RX_COMMAND message
stream_id	00:00:00:00:00:00:00:00	Copy “stream_id” of the DISCONNECT_TX_RESPONSE message	Copy “stream_id” of the DISCONNECT_TX_RESPONSE message (00:00:00:00:00:00:00:00 if there was no message from the talker)
stream_dest_mac	00:00:00:00:00:00	Copy “stream_dest_mac” of the DISCONNECT_TX_RESPONSE message	Copy “stream_dest_mac” of the DISCONNECT_TX_RESPONSE message (00:00:00:00:00:00 if there was no message from the talker)
stream_vlan_id	0	Copy “stream_vlan_id” of the DISCONNECT_TX_RESPONSE message	Copy “stream_vlan_id” of the DISCONNECT_TX_RESPONSE message (0 if there was no message from the talker)

connection_count	0	Copy “connection_count” of the DISCONNECT_TX_RESPONSE message	Copy “connection_count” of the DISCONNECT_TX_RESPONSE message (0 if there was no message from the talker)
sequence_id	changed by the sender at each new command sent	Copy “sequence_id” of the DISCONNECT_RX_COMMAND message	Copy “sequence_id” of the DISCONNECT_RX_COMMAND message
flags.CLASS_B	0	Copy “flags.CLASS_B” of the DISCONNECT_TX_RESPONSE message	Copy “flags.CLASS_B” of the DISCONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.FAST_CONNECT	0	0	0
flags.SAVED_STATE	0	0	1 if the listener implements Fast Connect mode, 0 otherwise
flags.STREAMING_WAIT	0	Copy “flags.STREAMING_WAIT” of the DISCONNECT_TX_RESPONSE message	Copy “flags.STREAMING_WAIT” of the DISCONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.SUPPORTS_ENCRYPTED	0	Copy “flags.SUPPORTS_ENCRYPTED” of the DISCONNECT_TX_RESPONSE message	Copy “flags.SUPPORTS_ENCRYPTED” of the DISCONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.ENCRYPTED_PDU	0	Copy “flags.ENCRYPTED_PDU” of the DISCONNECT_TX_RESPONSE message	Copy “flags.ENCRYPTED_PDU” of the DISCONNECT_TX_RESPONSE message (0 if there was no message from the talker)
flags.TALKER_FAILED	0	0	0
Other bits of the flags	0	Copy other bits of the flags of the DISCONNECT_TX_RESPONSE message	Copy other bits of the flags of the DISCONNECT_TX_RESPONSE message
reserved	0	0	0

3.4. DISCONNECT_TX_COMMAND and DISCONNECT_TX_RESPONSE

	DISCONNECT_TX_COMMAND	**DISCONNECT_TX_RESPONSE (status<>TALKER_UNKNOWN_ID)**	**DISCONNECT_TX_RESPONSE (status=TALKER_UNKNOWN_ID)**
controller_entity_id	Copy "controller_entity_id" of the DISCONNECT_RX_COMMAND message (saved value from the current connection in case of Fast Disconnect mode)	Copy "controller_entity_id" of the DISCONNECT_TX_COMMAND message	Copy "controller_entity_id" of the DISCONNECT_TX_COMMAND message
talker_entity_id	Copy "talker_entity_id" of the DISCONNECT_RX_COMMAND message (saved value from the current connection in case of Fast Disconnect mode)	Copy "talker_entity_id" of the DISCONNECT_TX_COMMAND message	Copy "talker_entity_id" of the DISCONNECT_TX_COMMAND message
listener_entity_id	Copy "listener_entity_id" of the DISCONNECT_RX_COMMAND message (saved value from the current connection in case of Fast Disconnect mode)	Copy "listener_entity_id" of the DISCONNECT_TX_COMMAND message	Copy "listener_entity_id" of the DISCONNECT_TX_COMMAND message
talker_unique_id	Copy "talker_unique_id" of the DISCONNECT_RX_COMMAND message (saved value from the current connection in case of Fast Disconnect mode)	Copy "talker_unique_id" of the DISCONNECT_TX_COMMAND message	Copy "talker_unique_id" of the DISCONNECT_TX_COMMAND message
listener_unique_id	Copy "listener_unique_id" of the DISCONNECT_RX_COMMAND message (saved value from the current connection in case of Fast Disconnect mode)	Copy "listener_unique_id" of the DISCONNECT_TX_COMMAND message	Copy "listener_unique_id" of the DISCONNECT_TX_COMMAND message
stream_id	Copy "stream_id" of the DISCONNECT_RX_COMMAND message (00:00:00:00:00:00:00:00 in case	– If the Talker source is still connected to at least one Listener sink: ID of the stream – Otherwise:	Copy "stream_id" of the DISCONNECT_TX_COMMAND message

	of Fast Disconnect mode)	00:00:00:00:00:00:00:00	
stream_dest_mac	Copy “stream_dest_mac” of the DISCONNECT_RX_COMMAND message (00:00:00:00:00:00 in case of Fast Disconnect mode)	– If the Talker source is still connected to at least one Listener sink: Destination MAC address of the stream – Otherwise: 00:00:00:00:00:00	Copy “stream_dest_mac” of the DISCONNECT_TX_COMMAND message
stream_vlan_id	Copy “stream_vlan_id” of the DISCONNECT_RX_COMMAND message (0 in case of Fast Disconnect mode)	– If the Talker source is still connected to at least one Listener sink: VLAN ID of the stream (0 indicates default VLAN ID of the SRP domain) – Otherwise: 0	Copy “stream_vlan_id” of the DISCONNECT_TX_COMMAND message
connection_count	0	Total count of Listener sinks connected to this stream source after execution of the command	0
sequence_id	changed by the sender at each new command sent	Copy “sequence_id” of the DISCONNECT_TX_COMMAND message	Copy “sequence_id” of the DISCONNECT_TX_COMMAND message
flags.CLASS_B	Copy “flags.CLASS_B” of the DISCONNECT_RX_COMMAND message (0 in case of Fast Disconnect mode)	– If the Talker source is still connected to at least one Listener sink: 0 if the stream is Class A, 1 if the stream is Class B – Otherwise: 0	Copy “flags.CLASS_B” of the DISCONNECT_TX_COMMAND message
flags.FAST_CONNECT	0	Copy “flags.FAST_CONNECT” of the DISCONNECT_TX_COMMAND message	Copy “flags.FAST_CONNECT” of the DISCONNECT_TX_COMMAND message
flags.SAVED_STATE	0	Copy “flags.SAVED_STATE” of the DISCONNECT_TX_COMMAND message	Copy “flags.SAVED_STATE” of the DISCONNECT_TX_COMMAND message
flags.STREAMING_WAIT	Copy “flags.STREAMING_WAIT” of the DISCONNECT_RX_COMMAND message (saved value from last state in case of Fast Disconnect	Copy “flags.STREAMING_WAIT” of the DISCONNECT_TX_COMMAND message	Copy “flags.STREAMING_WAIT” of the DISCONNECT_TX_COMMAND message

	mode)		
flags.SUPPORTS_ENCRYPTED	Copy "flags.SUPPORTS_ENCRYPTED" of the DISCONNECT_RX_COMMAND message (0 in case of Fast Disconnect mode)	1 if the talker supports encryption of the PDUs, 0 otherwise	Copy "flags.SUPPORTS_ENCRYPTED" of the DISCONNECT_TX_COMMAND message
flags.ENCRYPTED_PDU	Copy "flags.ENCRYPTED_PDU" of the DISCONNECT_RX_COMMAND message (0 in case of Fast Disconnect mode)	– If the Talker source is still connected to at least one Listener sink: 1 if the talker is configured to use encrypted PDUs for this stream, 0 otherwise – Otherwise: 0	Copy "flags.ENCRYPTED_PDU" of the DISCONNECT_TX_COMMAND message
flags.TALKER_FAILED	Copy "flags.TALKER_FAILED" of the DISCONNECT_RX_COMMAND message (0 in case of Fast Disconnect mode)	Copy "flags.TALKER_FAILED" of the DISCONNECT_TX_COMMAND message	Copy "flags.TALKER_FAILED" of the DISCONNECT_TX_COMMAND message
Other bits of the flags	Copy other bits of the flags of the DISCONNECT_RX_COMMAND message (0 in case of Fast Disconnect mode)	Copy other bits of the flags of the DISCONNECT_TX_COMMAND message	Copy other bits of the flags of the DISCONNECT_TX_COMMAND message
reserved	0	0	0

3.5. GET_RX_STATE_COMMAND and GET_RX_STATE_RESPONSE

	GET_RX_STATE_COMMAND	**GET_RX_STATE_RESPONSE (status=SUCCESS)**	**GET_RX_STATE_RESPONSE (status<>SUCCESS)**
controller_entity_id	Entity ID of the controller sending the command	Copy "controller_entity_id" of the GET_RX_STATE_COMMAND message	Copy "controller_entity_id" of the GET_RX_STATE_COMMAND message
talker_entity_id	00:00:00:00:00:00:00:00	– If the Stream sink is connected to a Talker source, or if it has saved state (see flags.SAVED_STATE): Entity ID of the talker – If not connected and doesn't have saved state: 00:00:00:00:00:00:00:00	00:00:00:00:00:00:00:00
listener_entity_id	Entity ID of the listener which is target of the command	Copy "listener_entity_id" of the GET_RX_STATE_COMMAND message	Copy "listener_entity_id" of the GET_RX_STATE_COMMAND message
talker_unique_id	0	– If the Stream sink is connected to a Talker source, or if it has saved state (see flags.SAVED_STATE): Unique ID of the talker source – If not connected and doesn't have saved state: 0	0
listener_unique_id	Unique ID of the Stream sink which is target of the command	Copy "listener_unique_id" of the GET_RX_STATE _COMMAND message	Copy "listener_unique_id" of the GET_RX_STATE _COMMAND message
stream_id	00:00:00:00:00:00:00:00	– If the Stream sink is connected to a Talker	00:00:00:00:00:00:00:00

Marc ILLOUZ 8/3/16 4:25 PM
Comment [2]: To be checked

Marc ILLOUZ 8/3/16 4:25 PM
Comment [3]: To be checked

		source: ID of the stream – If not connected: 00:00:00:00:00:00:00:00	
stream_dest_mac	00:00:00:00:00:00	– If the Stream sink is connected to a Talker source: Destination MAC address of the stream – If not connected: 00:00:00:00:00:00	00:00:00:00:00:00
stream_vlan_id	0	– If the Stream sink is connected to a Talker source: VLAN ID of the stream (0 indicates default VLAN ID of the SRP domain) – If not connected: 0	0
connection_count	0	1 if the Stream sink is connected to a Talker source, 0 otherwise	0
sequence_id	changed by the sender at each new command sent	Copy "sequence_id" of the GET_RX_STATE_COMMAND message	Copy "sequence_id" of the GET_RX_STATE_COMMAND message
flags.CLASS_B	0	– If the Stream sink is connected to a Talker source : 0 if the stream is Class A, 1 if the stream is Class B – If not connected: 0	0
flags.FAST_CONNECT	0	1 if the listener implements Fast Connect mode AND either is connected in Fast Connect mode, or is trying to connect in Fast Connect mode	0
flags.SAVED_STATE	0	1 if the listener implements Fast Connect mode AND either is	0

		connected (in Controller Connect or Fast Connect mode) or is trying to connect in Fast Connect mode, 0 otherwise	
flags.STREAMING_WAIT	0	0	0
flags.SUPPORTS_ENCRYPTED	0	0	0
flags.ENCRYPTED_PDU	0	– If the Stream sink is connected to a Talker source : 1 if the stream is encrypted, 0 otherwise – If not connected: 0	0
flags.TALKER_FAILED	0	– If the Stream sink is connected to a Talker source : 1 if the listener is receiving a Talker Failed attribute for this stream, 0 otherwise – If not connected: 0	0
Other bits of the flags	0	0	0
reserved	0	0	0

3.6. GET_TX_STATE_COMMAND and GET_TX_STATE_RESPONSE

	GET_TX_STATE_COMMAND	**GET_TX_STATE_RESPONSE (status<>TALKER_UNKNOWN_ID)**	**GET_TX_STATE_RESPONSE (status=TALKER_UNKNOWN_ID)**
controller_entity_id	Entity ID of the controller sending the command	Copy "controller_entity_id" of the GET_TX_STATE_COMMAND message	Copy "controller_entity_id" of the GET_TX_STATE_COMMAND message
talker_entity_id	Entity ID of the talker which is target of the command	Copy "talker_entity_id" of the GET_TX_STATE_COMMAND message	Copy "talker_entity_id" of the GET_TX_STATE_COMMAND message
listener_entity_id	00:00:00:00:00:00:00:00	Copy "listener_entity_id" of the GET_TX_STATE _COMMAND message	Copy "listener_entity_id" of the GET_TX_STATE _COMMAND message
talker_unique_id	Unique ID of the Stream source which is target of the command	Copy "talker_unique_id" of the GET_TX_STATE _COMMAND message	Copy "talker_unique_id" of the GET_TX_STATE_COMMAND message
listener_unique_id	0	Copy "listener_unique_id" of the GET_TX_STATE _COMMAND message	Copy "listener_unique_id" of the GET_TX_STATE _COMMAND message
stream_id	00:00:00:00:00:00:00:00	– If the Stream source is connected to at least one Listener sink: ID of the stream – If not connected: 00:00:00:00:00:00:00:00	Copy "stream_id" of the GET_TX_STATE _COMMAND message
stream_dest_mac	00:00:00:00:00:00	– If the Stream source is connected to at least one Listener sink: Destination MAC address of the stream (00:00:00:00:00:00 if the MAAP range previously allocated by the Talker has been lost and a new range has	Copy "stream_dest_mac" of the GET_TX_STATE _COMMAND message

		not been allocated yet) – If not connected: 00:00:00:00:00:00	
stream_vlan_id	0	– If the Stream source is connected to at least one Listener sink: VLAN ID of the stream (0 indicates default VLAN ID of the SRP domain) – If not connected: 0	Copy “stream_vlan_id” of the GET_TX_STATE _COMMAND message
connection_count	0	Number of Listener sinks connected to this Talker source	0
sequence_id	changed by the sender at each new command sent	Copy “sequence_id” of the GET_TX_STATE_COMMAND message	Copy “sequence_id” of the GET_TX_STATE_COMMAND message
flags.CLASS_B	0	– If the Talker source is connected to at least one Listener sink: 0 if the stream is Class A, 1 if the stream is Class B – Otherwise: 0	Copy “flags.CLASS_B” of the GET_TX_STATE_COMMAND message
flags.FAST_CONNECT	0	Copy “flags.FAST_CONNECT” of the GET_TX_STATE_COMMAND message	Copy “flags.FAST_CONNECT” of the GET_TX_STATE_COMMAND message
flags.SAVED_STATE	0	Copy “flags.SAVED_STATE” of the GET_TX_STATE_COMMAND message	Copy “flags.SAVED_STATE” of the GET_TX_STATE_COMMAND message
flags.STREAMING_WAIT	0	Copy “flags.STREAMING_WAIT” of the GET_TX_STATE_COMMAND message	Copy “flags.STREAMING_WAIT” of the GET_TX_STATE_COMMAND message
flags.SUPPORTS_ENCRYPTED	0	1 if the talker supports encryption of the PDUs, 0 otherwise	Copy “flags.SUPPORTS_ENCRYPTED” of the GET_TX_STATE_COMMAND message

flags.ENCRYPTED_PDU	0	– If the Talker source is connected to at least one Listener sink: 1 if the talker is configured to use encrypted PDUs for this stream, 0 otherwise – Otherwise: 0	Copy “flags.ENCRYPTED_PDU” of the GET_TX_STATE_COMMAND message
flags.TALKER_FAILED	0	Copy “flags.TALKER_FAILED” of the GET_TX_STATE_COMMAND message	Copy “flags.TALKER_FAILED” of the GET_TX_STATE_COMMAND message
Other bits of the flags	0	Copy other bits of the flags of the GET_TX_STATE_COMMAND message	Copy other bits of the flags of the GET_TX_STATE_COMMAND message
reserved	0	0	0

3.7. GET_TX_CONNECTION_COMMAND and GET_TX_CONNECTION_RESPONSE

	GET_TX_CONNECTION_COMMAND	**GET_TX_CONNECTION_RESPONSE (status<>TALKER_UNKNOWN_ID)**	**GET_TX_ CONNECTION _RESPONSE (status=TALKER_UNKNOWN_ID)**
controller_entity_id	Entity ID of the controller sending the command	Copy "controller_entity_id" of the GET_TX_CONNECTION_COMMAND message	Copy "controller_entity_id" of the GET_TX_ CONNECTION _COMMAND message
talker_entity_id	Entity ID of the talker which is target of the command	Copy "talker_entity_id" of the GET_TX_ CONNECTION _COMMAND message	Copy "talker_entity_id" of the GET_TX_ CONNECTION _COMMAND message
listener_entity_id	00:00:00:00:00:00:00:00	Entity ID of the connected Listener (00:00:00:00:00:00:00:00 if status=NO_SUCH_CONNECTION)	Copy "listener_entity_id" of the GET_TX_ CONNECTION _COMMAND message
talker_unique_id	Unique ID of the Stream source which is target of the command	Copy "talker_unique_id" of the GET_TX_ CONNECTION _COMMAND message	Copy "talker_unique_id" of the GET_TX_ CONNECTION _COMMAND message
listener_unique_id	0	Unique ID of the connected Listener sink (0 if status=NO_SUCH_CONNECTION)	Copy "listener_unique_id" of the GET_TX_ CONNECTION _COMMAND message
stream_id	00:00:00:00:00:00:00:00	– If the Stream source is connected to at least one Listener sink: ID of the stream – If not connected: 00:00:00:00:00:00:00:00	Copy "stream_id" of the GET_TX_ CONNECTION _COMMAND message
stream_dest_mac	00:00:00:00:00:00	– If the Stream source is connected to at least one Listener sink: Destination MAC address of the stream (00:00:00:00:00:00 if the MAAP range previously allocated by the Talker has been lost and a new range has	Copy "stream_dest_mac" of the GET_TX_ CONNECTION _COMMAND message

		not been allocated yet) – If not connected: 00:00:00:00:00:00	
stream_vlan_id	0	– If the Stream source is connected to at least one Listener sink: VLAN ID of the stream (0 indicates default VLAN ID of the SRP domain) – If not connected: 0	Copy “stream_vlan_id” of the GET_TX_ CONNECTION _COMMAND message
connection_count	Index of the connection which is target of the command (the first connection of the list has index 0)	Number of Listener sinks connected to this Talker source	0
sequence_id	changed by the sender at each new command sent	Copy “sequence_id” of the GET_TX_ CONNECTION _COMMAND message	Copy “sequence_id” of the GET_TX_ CONNECTION _COMMAND message
flags.CLASS_B	0	– If the Talker source is connected to at least one Listener sink: 0 if the stream is Class A, 1 if the stream is Class B – Otherwise: 0	Copy “flags.CLASS_B” of the GET_TX_ CONNECTION _COMMAND message
flags.FAST_CONNECT	0	Copy “flags.FAST_CONNECT” of the GET_TX_ CONNECTION _COMMAND message	Copy “flags.FAST_CONNECT” of the GET_TX_ CONNECTION _COMMAND message
flags.SAVED_STATE	0	Copy “flags.SAVED_STATE” of the GET_TX_ CONNECTION _COMMAND message	Copy “flags.SAVED_STATE” of the GET_TX_ CONNECTION _COMMAND message
flags.STREAMING_WAIT	0	Copy “flags.STREAMING_WAIT” of the GET_TX_ CONNECTION _COMMAND message	Copy “flags.STREAMING_WAIT” of the GET_TX_ CONNECTION _COMMAND message
flags.SUPPORTS_ENCRYPTED	0	1 if the talker supports encryption of the PDUs, 0 otherwise	Copy “flags.SUPPORTS_ENCRYPTED” of the GET_TX_ CONNECTION _COMMAND message
flags.ENCRYPTED_PDU	0	If the Talker source is connected to	Copy “flags.ENCRYPTED_PDU” of the

		at least one Listener sink: 1 if the talker is configured to use encrypted PDUs for this stream, 0 otherwise – Otherwise: 0	GET_TX_ CONNECTION _COMMAND message
flags.TALKER_FAILED	0	Copy “flags.TALKER_FAILED” of the GET_TX_ CONNECTION _COMMAND message	Copy “flags.TALKER_FAILED” of the GET_TX_ CONNECTION _COMMAND message
Other bits of the flags	0	Copy other bits of the flags of the GET_TX_ CONNECTION _COMMAND message	Copy other bits of the flags of the GET_TX_ CONNECTION _COMMAND message
reserved	0	0	0

3.8. Discussion

3.8.1. Operations of ACMP towards SRP

Operations of ACMP are completely asynchronous towards SRP. This means that ACMP can be connected while stream reservation is not completed yet. SRP acts as a slave of ACMP: when ACMP is ready to connect, it gives all the necessary information to the SRP layer and requests to reserve bandwidth for the stream. When SRP status changes, SRP reports this status information to the ACMP layer. For example, ACMP makes use of the TALKER_FAILED information, which indicates that the listener is receiving an MSRP Talker Failed attribute for this stream. Please note that ACMP must NOT use the stream parameters which may be reported by SRP (destination MAC address, VLAN ID, class). If SRP parameters are different from ACMP parameters, ACMP takes precedence (such conflicts must be solved by the controller).

3.8.2. "connection_count" field

Contrary to what is specified in IEEE 1722.1-2013, "connection_count" is not always the number of connections the talker thinks it has. Indeed, in **GET_RX_STATE_RESPONSE** messages, "connection_count" is 1 if the Listener is connected, 0 otherwise.

To summarize:

- In command messages:
  - **GET_TX_CONNECTION_COMMAND**: set to the index of the connection which is target of the command
  - All other commands: clear to 0 on send, ignored on receive
- In response messages:
  - **GET_RX_STATE_RESPONSE**: set to the value of the "connected" field of the ListenerStreamInfo structure
  - All other responses: set to the value of the "connection_count" field of the TalkerStreamInfo structure. Note: for **CONNECT_RX_RESPONSE/DISCONNECT_RX_RESPONSE**, the value is copied from the **CONNECT_TX_RESPONSE/DISCONNECT_TX_RESPONSE**

3.8.3. “CLASS_B”, “SUPPORTS_ENCRYPTED” and “ENCRYPTED_PDU” fields

Contrary to what is specified in IEEE 1722.1-2013, the talker doesn’t have to set these fields in the response message to the value they had in the command message. In **CONNECT_TX_RESPONSE**, **DISCONNECT_TX_RESPONSE**, **GET_TX_STATE_RESPONSE** and **GET_TX_CONNECTION_RESPONSE**, these fields contain the actual parameter of the stream (CLASS_B=1 if the stream is Class B and not Class A, SUPPORTS_ENCRYPTED=1 if the talker supports encryption for this stream and ENCRYPTED_PDU=1 if the talker currently uses encryption for this stream).

Marc ILLOUZ 8/3/16 4:28 PM

Comment [4]: Can CLASS_B be set by the Controller to force the Talker to use Class B instead of Class A ?
Are there other fields like this (for example dest mac adr) ?
If yes, how to distinguish between the intention to force a parameter and the intention to let the talker chose ?

3.8.4. Saved state and Fast Connect mode

A listener supporting Fast Connect mode saves the following pieces of information from the **CONNECT_TX_RESPONSE** message on a non-volatile memory each time a new connection is established (the connectListener() function returns SUCCESS) on one of its sinks:

- controller_entity_id
- talker_entity_id
- talker_unique_id
- flags.STREAMING_WAIT

Marc ILLOUZ 8/3/16 5:28 PM

Comment [5]: If, after connection, the controller has modified STREAMING_WAIT of the Talker, the listener will put an obsolete value in the new CONNECT_TX_COMMAND ...

This set of information is called the “saved state” of the sink. It is used to build the **CONNECT_TX_COMMAND** message sent by the listener after a power cycle.

The saved state is erased only when the connection is closed (the listener receives a **DISCONNECT_RX_COMMAND** message and the disconnectListener() function returns SUCCESS). The saved state is NOT erased by a power cycle or a reboot of the listener.

To summarize: the listener is said to have a “saved state” when it supports Fast Connect mode AND either it is connected (Controller Connect or Fast Connect mode) or is trying to connect in Fast Connect mode.

The SAVED_STATE field of the state message is set to 1 when the listener has a saved state. Otherwise, it is cleared to 0.

The FAST_CONNECT field is similar to SAVED_STATE, but is set to 1 only if the current mode is Fast Connect (if the listener is connected in Controller Connect mode then SAVED_STATE=1 and FAST_CONNECT=1).

4. ACMP scenarios

4.1. Acquirement/lock and connections – Listener acceptance

In this scenario, there are 4 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- Two controllers: *Controller1* and *Controller2*

The goal of this scenario is to show that a listener shall accept a connection/disconnection request from a controller even if it has been acquired/locked by another controller.

1) *Controller1* acquires or locks successfully one or several STREAM_INPUT descriptor(s) of *Listener* through AECP
2) *Controller2* sends **CONNECT_RX_COMMAND** with **listener_entity_id** equal to the Entity ID of *Listener* and **listener_unique_id** equal to the index of one of the STREAM_INPUT descriptors acquired/locked by *Controller1*
3) *Listener* doesn't check the acquired/locked state of the STREAM_INPUT descriptor. It accepts the request and processes it normally (it sends **CONNECT_TX_COMMAND** to *Talker*)

Notes:

- In order to acquire/lock a STREAM_INPUT descriptor, *Controller1* may either acquire/lock the STREAM_INPUT descriptor only, or the AUDIO_UNIT descriptor associated to this STREAM_INPUT descriptor, or the ENTITY descriptor.
- Next revision of 1722.1 will allow a Listener to refuse a connection from a controller if locked/acquired by another controller. For this, the Listener will have to set a specific bit in its advertised "entity_capabilities"

4.2. Acquirement/lock and connections – Talker acceptance

In this scenario, there are 4 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- Two controllers: *Controller1* and *Controller2*

The goal of this scenario is to show that a talker shall accept a connection/disconnection request from a controller even if it has been acquired/locked by another controller.

1) *Controller1* acquires/locks successfully one or several STREAM_OUTPUT descriptor(s) of *Talker* through AECP
2) *Controller2* sends **CONNECT_RX_COMMAND** to *Listener* with **talker_entity_id** equal to the Entity ID of *Talker* and **talker_unique_id** equal to the index of one of the STREAM_OUTPUT descriptors acquired/locked by *Controller1*
3) *Listener* sends **CONNECT_TX_COMMAND** to *Talker*
4) *Talker* doesn't check the acquired/locked stated of the STREAM_OUTPUT descriptor. It accepts the request, processes it and replies **CONNECT_TX_RESPONSE** with **status=SUCCESS**
5) *Listener* receives **CONNECT_TX_RESPONSE** with **status=SUCCESS**
6) *Listener* connects and sends **CONNECT_RX_RESPONSE** with **status=SUCCESS**

Notes:

- In order to acquire/lock a STREAM_OUTPUT descriptor, *Controller1* may either acquire/lock the STREAM_OUTPUT descriptor only, or the AUDIO_UNIT descriptor associated to this STREAM_OUTPUT descriptor, or the ENTITY descriptor.
- Next revision of 1722.1 will allow a Talker to refuse a connection from a controller if locked/acquired by another controller. For this, the Talker will have to set a specific bit in its advertised "entity_capabilities"

4.3. State of the Listener during Fast Connect attempt

In this scenario, there are 3 AVDECC entities:

- A listener which implements the Fast Connect mode: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show what a Listener performing a Fast Connect shall advertise to the controller.

1) *Controller* successfully establishes a connection between *Talker* and *Listener*
2) Power is switched off (every device shuts down)
3) *Talker* is removed from the network
4) Later, power is switched on again
5) *Listener* reboots and sends some **CONNECT_TX_COMMAND** messages to *Talker* with **connection_count=0**, **flags.FAST_CONNECT=1** and **flags.SAVED_STATE=1**

Marc ILLOUZ 8/3/16 4:23 PM
Comment [6]: To be checked

6) During this time, *Controller* sends **GET_RX_STATE_COMMAND** to *Listener*
7) *Listener* replies **GET_RX_STATE_RESPONSE** with **connection_count=0**, **flags.FAST_CONNECT=1** and **flags.SAVED_STATE=1**
8) After some time, *Talker* is added back to the network and replies **CONNECT_TX_COMMAND** with **status=SUCCESS**
9) Then *Controller* sends **GET_RX_STATE_COMMAND** again to *Listener*
10) *Listener* replies **GET_RX_STATE_RESPONSE** with **connection_count=1**, **flags.FAST_CONNECT=1** and **flags.SAVED_STATE=1**

Note: there are three ways for the controller to know that the listener is currently attempting to connect in Fast Connect mode:

- Either sniff the **CONNECT_TX_COMMAND** messages on the network and see that **flags.FAST_CONNECT** is set. The **CONNECT_TX_COMMAND** message contains the Entity IDs of both the listener and the talker, and it also contains the Unique IDs of the sink and the source. This is a necessary information if the controller wants to stop the Fast Connect attempts.
- Or send a **GET_RX_STATE_COMMAN D** to the listener. The listener will set **flags.CONNECTED=0**, **flags.FAST_CONNECT=1** and **flags.SAVED_STATE=1** in the **GET_RX_STATE_RESPONSE**. The **GET_RX_STATE_RESPONSE** message contains the Entity ID of the talker and also the Unique ID of the source. This is a necessary information if the controller wants to stop the Fast Connect attempts.
- Or send a **GET_STREAM_INFO** command to the listener. The listener will set **flags.CONNECTED=0**, **flags.FAST_CONNECT=1** and **flags.SAVED_STATE=1** in the response. The drawback of this method is that the controller doesn't have any information about the talker source to which the listener is trying to connect to.

4.4. Request to stop Fast Connect attempts

In this scenario, there are 3 AVDECC entities:

- A listener which implements the Fast Connect mode: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show how a Controller can ask a Listener to stop attempting to connect in Fast Connect mode.

1) *Controller* successfully establishes a connection between *Talker* and *Listener*
2) Power is switched off (every device shuts down)
3) *Talker* is removed from the network
4) Later, power is switched on again
5) *Listener* reboots and sends some **CONNECT_TX_COMMAND** messages to *Talker* with **connection_count=0**, **flags.FAST_CONNECT=1** and **flags.SAVED_STATE=1**
6) During this time, *Controller* sends **DISCONNECT_RX_COMMAND** to *Listener*

7) *Listener* receives **DISCONNECT_RX_COMMAND**
8) *Listener* stops attempting to connect in Fast Connect mode
9) *Listener* replies **DISCONNECT_RX_RESPONSE** with **status=NOT_CONNECTED**
10) *Controller* sends **GET_RX_STATE_COMMAND** to *Listener*
11) *Listener* replies **GET_RX_STATE_RESPONSE** with **connection_count=0**, **flags.FAST_CONNECT=0** and **flags.SAVED_STATE=0**

Notes:
1) There is no way for the controller to globally disable the Fast Connect feature of a listener. A listener implementing Fast Connect and which is rebooted without being cleanly disconnected will always try to connect in Fast Connect mode (until the controller sends a **DISCONNECT_RX_COMMAND** message).
2) When a listener is not connected nor trying to connect in Fast Connect mode, there is no way to know whether it implements Fast Connect mode or not

4.5. Connection succeeded on Talker and failed on Listener

In this scenario, there are 3 AVDECC entities:
- A listener: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show that the controller shall always request the state of the entities after a connection failure.

1) *Controller* sends **CONNECT_RX_COMMAND** to *Listener*
2) *Listener* receives **CONNECT_RX_COMMAND**
3) *Listener* sends **CONNECT_TX_COMMAND** to *Talker*
4) *Talker* receives **CONNECT_TX_COMMAND**
5) *Talker* executes successfully the “connectTalker” function
6) *Talker* sends **CONNECT_TX_RESPONSE** with **status=SUCCESS**
7) *Listener* receives **CONNECT_TX_RESPONSE** with **status=SUCCESS**
8) *Listener* executes the “connectListener” function and it fails for any reason (it returns status<>SUCCESS)
9) *Listener* sends **CONNECT_RX_RESPONSE** with **status<>SUCCESS**
10) *Controller* receives **CONNECT_RX_RESPONSE** with **status<>SUCCESS**
11) *Controller* gets the state of *Listener* thanks to a **GET_RX_STATE_COMMAND/GET_RX_STATE_RESPONSE** exchange

12) *Controller* gets the state of *Talker* thanks to a **GET_TX_STATE_COMMAND/GET_TX_STATE_RESPONSE** + some **GET_TX_CONNECTION_COMMAND/GET_TX_CONNECTION_RESPONSE** exchanges
13) *Controller* notices that the connection failed on *Listener* and tries again sending a **CONNECT_RX_COMMAND** to *Listener*

Note: if the listener continuously fails to establish the connection, it will be impossible to disconnect the talker. Indeed, the controller cannot send directly a **DISCONNECT_TX_COMMAND** to the talker, and it's no use to send a **DISCONNECT_RX_COMMAND** to the listener because it will always reply with **status=NOT_CONNECTED** without even trying to send a **DISCONNECT_TX_COMMAND** to the talker.

Note2: it would be nice if the listener could handle this case by itself, this means automatically send a **DISCONNECT_TX_COMMAND** to the talker when the "connectListener" function fails (in addition to sending the **CONNECT_RX_RESPONSE** with **status<>SUCCESS** to the controller). Unfortunately a listener behaving this way is not conform to the ACMP Listener state machine specified in IEEE 1722.1-2013.

4.6. Disconnection succeeded on Listener and failed on Talker

In this scenario, there are 3 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show that the controller shall always request the state of the entities after a disconnection failure.

1) *Controller* sends **DISCONNECT_RX_COMMAND** to *Listener*
2) *Listener* receives **DISCONNECT_RX_COMMAND**
3) *Listener* executes successfully the "disconnectListener" function
4) *Listener* sends **DISCONNECT_TX_COMMAND** to *Talker*
5) *Talker* receives **DISCONNECT_TX_COMMAND**
6) *Talker* executes the "disconnectTalker" function and it fails for any reason (it returns status<>SUCCESS)
7) *Talker* sends **DISCONNECT_TX_RESPONSE** with **status<>SUCCESS**
8) *Listener* receives **DISCONNECT_TX_RESPONSE** with **status<>SUCCESS**
9) *Listener* sends **DISCONNECT_RX_RESPONSE** with **status<>SUCCESS**
10) *Controller* receives **DISCONNECT_RX_RESPONSE** with **status<>SUCCESS**
11) *Controller* gets the state of *Listener* thanks to a **GET_RX_STATE_COMMAND/GET_RX_STATE_RESPONSE** exchange

12) *Controller* gets the state of *Talker* thanks to a **GET_TX_STATE_COMMAND/GET_TX_STATE_RESPONSE** + some **GET_TX_CONNECTION_COMMAND/GET_TX_CONNECTION_RESPONSE** exchanges
13) *Controller* notices that the disconnection failed on *Talker* and tries to reconnect *Listener* with a **CONNECT_RX_COMMAND.** Once *Listener* will be connected again, *Controller* will be able to try again a clean disconnection

Note: the controller always has to reconnect the listener before trying again the disconnection. Indeed, the controller cannot send directly a **DISCONNECT_TX_COMMAND** to the talker, and it's no use to send a **DISCONNECT_RX_COMMAND** to the listener because it will always reply with **status=NOT_CONNECTED** without even trying to send a **DISCONNECT_TX_COMMAND** to the talker.

4.7. Talker connected to a ghost Listener

In this scenario, there are 3 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show that there are situations where the controller may be unable to disconnect a talker.

1) *Controller* successfully establishes a connection between *Talker* and *Listener*
2) *Listener* is removed from the network without clean disconnection
3) *Controller* cannot disconnect *Talker* because it is not allowed to send a **DISCONNECT_TX_COMMAND**

Note: the only way to exit from this locked situation is to add back the listener to the network and to send a **DISCONNECT_RX_COMMAND**, or to reboot the talker. While the talker source is connected to at least one listener sink, even if this listener has disappeared, it will continue to advertise its stream and consume some SRP resources in the network.

4.8. ACMP connection fails if not enough bandwidth on the upstream link

In this scenario, there are 3 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show in which situation the TALKER_NO_BANDWIDTH error is used.

1) *Controller* sends **CONNECT_RX_COMMAND** to *Listener*
2) *Listener* receives **CONNECT_RX_COMMAND**
3) *Listener* sends **CONNECT_TX_COMMAND** to *Talker*
4) *Talker* receives **CONNECT_TX_COMMAND**
5) *Talker* executes the "connectTalker" function. The "connectTalker" function checks the bandwidth currently used by the streams already requested by the Talker for its other sources towards the amount of available bandwidth on the upstream link. Let's suppose that the Talker has been designed to support a high channel count/number of streams on a 1Gbps link but the user has plugged it into a 100Mbps link. In this case only a fraction of the stream sources can actually be used. If there is not enough bandwidth for the new stream, the "connectTalker" function will not even request the SRP stack to advertise any MRP attribute. It will just return with the **TALKER_NO_BANDWIDTH** error.
6) *Talker* sends **CONNECT_TX_RESPONSE** with **status**=**TALKER_NO_BANDWIDTH**
7) *Listener* receives **CONNECT_TX_RESPONSE** with **status= TALKER_NO_BANDWIDTH**
8) *Listener* sends **CONNECT_RX_RESPONSE** with **status= TALKER_NO_BANDWIDTH**
9) *Controller* receives **CONNECT_RX_RESPONSE** with **status= TALKER_NO_BANDWIDTH**

Note: in this example, all the decisions have been taken without even requesting the SRP stack to reserve bandwidth. The talker requests the SRP stack to reserve bandwidth only when there is no error at the ACMP level.

Marc ILLOUZ 8/3/16 4:41 PM
Comment [7]: To be checked

4.9. ACMP connection fails because MAAP fails

In this scenario, there are 3 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show that a MAAP error will prevent an ACMP connection to be done.

1) *Controller* sends **CONNECT_RX_COMMAND** to *Listener*
2) *Listener* receives **CONNECT_RX_COMMAND**
3) *Listener* sends **CONNECT_TX_COMMAND** to *Talker*
4) *Talker* receives **CONNECT_TX_COMMAND**
5) Let's assume here that *Talker* is configured to dynamically allocate a destination MAC address through MAAP (this means that it didn't receive a SET_STREAM_INFO command with stream_dest_mac<>00:00:00:00:00:00 and flags.STREAM_DEST_MAC_VALID=1.). Let's assume also that *Talker* doesn't manage to allocate a MAC address through MAAP (even after a 1.5 second timeout). This may happen if there is an aggressive device on the network that always defends the addresses chosen by *Talker*.
6) *Talker* cannot establish the ACMP connection and sends **CONNECT_TX_RESPONSE** with **status=TALKER_DEST_MAC_FAIL**
7) *Listener* receives **CONNECT_TX_RESPONSE** with **status= TALKER_DEST_MAC_FAIL**
8) *Listener* doesn't connect because **status<>SUCCESS**
9) *Listener* sends **CONNECT_RX_RESPONSE** with **status= TALKER_DEST_MAC_FAIL**
10) *Controller* receives **CONNECT_RX_RESPONSE** with **status= TALKER_DEST_MAC_FAIL**

Marc ILLOUZ 8/3/16 4:57 PM
Comment [8]: To be checked. Other solution is to ignore CONNECT_TX_COMMAND messages until MAC addresses have been allocated

We see there that the ACMP connection is not established due to the fact that the talker has no MAC address for its stream. Please note that if the talker continuously fails in dynamically allocating a MAC address through MAAP, the controller may assign a predefined MAC address to it thanks to the SET_STREAM_INFO command (if the talker implements this command).

4.10. MAAP fails after a successful ACMP connection

In this scenario, there are 3 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*

- A controller: *Controller*

The goal of this scenario is to show that a MAAP error occurring after a successful ACMP connection will not break the ACMP connection.

1) *Controller* successfully establishes a connection between *Talker* and *Listener*
2) *Talker* loses the MAAP address range it had allocated for this connection
3) *Talker* immediately tries to allocate a new MAAP address range. During this time, *Talker* doesn't send any packet to the old MAC address and asks the SRP stack to stop advertising any Talker attribute.
4) As soon as *Talker* has managed to allocate a new MAAP address range, it asks the SRP stack to advertise a Talker attribute with the new MAC address. Please note that the SRP stack may delay this declaration (up to 30 seconds) due to the inherent constraints of the SRP protocol.

Notes:

- If the talker never manages to allocate a new MAAP address range, it stays connected but doesn't declare any Talker attribute.
- If the talker manages to allocate a new MAAP address range for its stream, all subsequent **CONNECT_TX_RESPONSE**, **GET_TX_STATE_RESPONSE** and **GET_TX_CONNECTION_RESPONSE** messages will carry the new MAC address
- A talker changing its stream destination MAC address will not inform anybody asynchronously through ACMP messages, but it can send an unsolicited SET_STREAM_INFO message to the registered controllers
- The listener will receive SRP stream parameters different from the parameters advertised in the **CONNECT_TX_RESPONSE** message, but it must NOT take them into account. It must wait for the controller to inform it of the new parameters with a SET_STREAM_INFO command.

Marc ILLOUZ 8/3/16 5:04 PM

Comment [9]: Is there a way for the talker to inform the controller that the destination MAC address is no longer valid ?
If the talker receives a GET_TX_STATE_COMMAND message or a GET_STREAM_INFO message while in this state, which value should he put in the stream_dest_mac field ?

4.11. SRP stream parameters different from AVDECC stream parameters

In this scenario, there are 3 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show that a Talker shall give precedence to AVDECC stream parameters over SRP stream parameters.

1) *Controller* successfully establishes a connection between *Talker* and *Listener*
2) *Listener* sets the parameters of the stream it is going to receive with the fields of the **CONNECT_TX_RESPONSE** message (**stream_dest_mac**, **stream_vlan_id** and **flags.CLASS_B**). In particular, *Listener* subscribes to the right VLAN ID and starts listening to data packets with the correct destination MAC address and the priority code point associated to the correct SR class.

3) Later, the SRP stack of *Listener* receives a Talker attribute with different stream parameters (destination MAC address and/or VLAN ID and/or SR class). *Listener* shall continue using the parameters received by ACMP in the **CONNECT_TX_RESPONSE** and ignore the parameters reported by SRP.
4) Later, *Controller* sends a **SET_STREAM_INFO** command to *Listener* with new stream parameters (destination MAC address and/or VLAN ID and/or SR class). *Listener* now has to change its stream parameters and use those of the **SET_STREAM_INFO** command. In particular, it may have to subscribe to another VLAN ID, listen to another destination MAC address and another priority code point.

Notes:

- It's up to the *Controller* to ensure that the Listener uses the right stream parameters. If, after ACMP connection establishment, the *Talker* changes the stream parameters (for any reason), the *Controller* has to be informed (either by unsolicited notification, or by polling) and has to inform the *Listener* (by a **SET_STREAM_INFO** command)
- The Listener shall always fill the fields of the **GET_RX_STATE_RESPONSE** message with the stream parameters it is currently using. This means that if these parameters have changed since the ACMP connection has been established, the listener will return the latest received stream parameters.

4.12. ACMP connection succeeds even if SRP has not reserved bandwidth yet

In this scenario, there are 3 AVDECC entities:

- A listener: *Listener*
- A talker: *Talker*
- A controller: *Controller*

The goal of this scenario is to show that the current state of the SRP reservation process doesn't change the connection state of ACMP.

1) *Controller* sends **CONNECT_RX_COMMAND** to *Listener*
2) *Listener* receives **CONNECT_RX_COMMAND**
3) *Listener* sends **CONNECT_TX_COMMAND** to *Talker*
4) *Talker* receives **CONNECT_TX_COMMAND**
5) *Talker* executes successfully the "connectTalker" function. Let's suppose that, contrary to the example of chapter "4.8 ACMP connection fails if not enough bandwidth on the upstream link", there is still bandwidth available on the upstream link. The "connectTalker" function requests the SRP stack of *Talker* to advertise and register the right MRP attributes as soon as possible in order to advertise its stream and reserve

bandwidth for it on the path from *Talker* to *Listener*. Let's suppose that there is no more bandwidth available on the link. Thus it immediately advertises a Talker Failed attribute instead of a Talker Advertise.

6) *Talker* sends **CONNECT_TX_RESPONSE** with **status**= **SUCCESS** (and not TALKER_NO_BANDWIDTH!!!)
7) *Listener* receives **CONNECT_TX_RESPONSE** with **status= SUCCESS**
8) *Listener* executes successfully the "connectListener" function
9) *Listener* sends **CONNECT_RX_RESPONSE** with **status= SUCCESS**
10) *Controller* receives **CONNECT_RX_RESPONSE** with **status= SUCCESS**

We see there that the ACMP connection is established with no error between the talker and the listener, but no bandwidth has been reserved for the stream (the listener will be aware of that because it will receive a Talker Failed attribute).

Note: the fundamental difference between the example of this chapter and the example of chapter chapter "4.8 ACMP connection fails if not enough bandwidth on the upstream link" is that here, the ACMP layer hasn't detected any error and has delegated the remaining work to the asynchronous SRP layer.