An Investigation of Protocol Command Translation as a means to enable Interoperability between Networked Audio Devices

Submitted in fulfilment
of the requirements of the degree
DOCTOR OF PHILOSOPHY
of Rhodes University

OSEDUM P. IGUMBOR

February 2013

Abstract

Digital audio networks allow multiple channels of audio to be streamed between devices. This eliminates the need for many different cables to route audio between devices. An added advantage of digital audio networks is the ability to configure and control the networked devices from a common control point. Common control of networked devices enables a sound engineer to establish and destroy audio stream connections between networked devices that are distances apart.

On a digital audio network, an audio transport technology enables the exchange of data streams. Typically, an audio transport technology is capable of transporting both control messages and audio data streams. There exist a number of audio transport technologies. Some of these technologies implement data transport by exchanging OSI/ISO layer 2 data frames, while others transport data within OSI/ISO layer 3 packets. There are some approaches to achieving interoperability between devices that utilize different audio transport technologies.

A digital audio device typically implements an audio control protocol, which enables it process configuration and control messages from a remote controller. An audio control protocol also defines the structure of the messages that are exchanged between compliant devices. There are currently a wide range of audio control protocols. Some audio control protocols utilize layer 3 audio transport technology, while others utilize layer 2 audio transport technology. An audio device can only communicate with other devices that implement the same control protocol, irrespective of a common transport technology that connects the devices.

The existence of different audio control protocols among devices on a network results in a situation where the devices are unable to communicate with each other. Furthermore, a single control application is unable to establish or destroy audio stream connections between the networked devices, since they implement different control protocols. When an audio engineer is designing an audio network installation, this interoperability challenge restricts the choice of devices that can be included. Even when audio transport interoperability has been achieved, common control of the devices remains a challenge.

This research investigates protocol command translation as a means to enable interoperability between networked audio devices that implement different audio control protocols. It proposes the use of a command translator that is capable of receiving messages conforming to one protocol from any of the networked devices, translating the

received message to conform to a different control protocol, then transmitting the translated message to the intended target which understands the translated protocol message. In so doing, the command translator enables common control of the networked devices, since a control application is able to configure and control devices that conform to different protocols by utilizing the command translator to perform appropriate protocol translation.

Acknowledgements

Many thanks to my supervisor Prof. Richard Foss for the encouragement and insights throughout the course of this research. In spite of his other commitments, he was always patient and willing to listen to my many ideas and suggestions, and more so guide me in my search for answers. To me, you have been a mentor, and a role model.

To my parents, Sam Nwa-Igumbor and Prof. Eunice Igumbor, I am grateful for the virtues of hard work and discipline that was instilled in me. These tools certainly played an important role during my academic journey. Mum and Dad, thanks for the dream, and for laying the path.

I am grateful to my brothers, Dr. Jude Igumbor, Ozenim Igumbor, and Prof. Ehi Igumbor, whose numerous calls to enquire about my well-being always left me even more motivated. Our many intellectual debates have immensely enriched me as a person.

I am most grateful to my fiancée, Kem Okecha, whose love, patience, and encouragement kept me balanced and focused. Kem, I am blessed to have you in my life.

I thank my colleagues in the Audio Research Group, and staff of the Department of Computer Science (Rhodes University), who in different ways have contributed to my knowledge. It is always great to have colleagues with whom one can exchange ideas.

I acknowledge the financial contribution of the *Distributed Multimedia Centre of Excellence* at Rhodes University with its sponsors, and the *Andrew Mellon foundation*. I am also grateful to *UMAN Technologies* for granting me access to their equipment and control application software that was used in the course of this research.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Audio devices have been connected into networks for deployment in various contexts. These include theme parks, airports, stadiums, casinos, shopping malls, audio production studios, places of worship and conference centers. The networks ensure that the sound produced by a source device is distributed to multiple destination devices, where the signal can be further processed and reproduced.

An earlier solution for distributing audio signals was the use of multiple analog cables with the appropriate connectors to connect the devices. This solution resulted in a lot of analog audio cables being used per device. The larger the venue and number of audio channels a device can receive and/or transmit, the larger the number of cables required to establish audio connections. The result is that it becomes difficult to trace cables that interconnect the devices.

Digital audio networking technology has provided an opportunity to transmit multiple channels of audio using a single cable.

In order to transport audio on a digital network, the transport medium and associated technology must cooperate to provide a number of qualities. These include:

- ensuring that audio data is transported with minimum delay
- ensuring that there is sufficient bandwidth on the path from source to destination device
- ensuring that the devices are synchronized so that there is no degradation in the sound quality.

Fulfilling these qualities requires careful design of the digital audio network. Some of the available audio networking technologies will be described in this thesis, with particular reference to how they ensure the above qualities.

When audio devices are interconnected within digital audio networks, it is desirable to be able to remotely monitor and control various features on the networked devices from a remote control center. Such monitoring and control of networked devices involves the exchange of messages between the controller and the networked audio devices. The messages that are exchanged are structured in a certain manner and processed by the target device(s). An audio control protocol is responsible for defining the structure and meaning of the exchanged messages.

Currently a number of audio control protocols exist. Some of them have been standardized, while others are proprietary. The manner in which the audio control protocols fulfill the above responsibilities will be discussed in this thesis, in order to provide an insight into the nature of audio control protocols. This discussion will reveal why protocol command interoperability remains a challenge.

1.1 Networked audio device

A digital audio device should be capable of transmitting audio data, as well as control commands and status information. A typical digital audio device consists of the following components:

- *audio transport* component - which is concerned with transmitting the actual audio streams. It should meet certain quality of service requirements in order to ensure reliable delivery of the audio data.
- *audio control* component - which is concerned with the exchange of messages that allow for remote monitoring, configuration, and control of the device. It utilizes the audio transport component for communication with other networked devices.
- *device model* component - which structurally represents the various features and controls within the device.
- *application* component - which utilizes the audio control protocol and interacts with the device model, in order to fulfill the overall functionality of the device.

Figure 1.1 depicts a digital audio device with the above mentioned components.

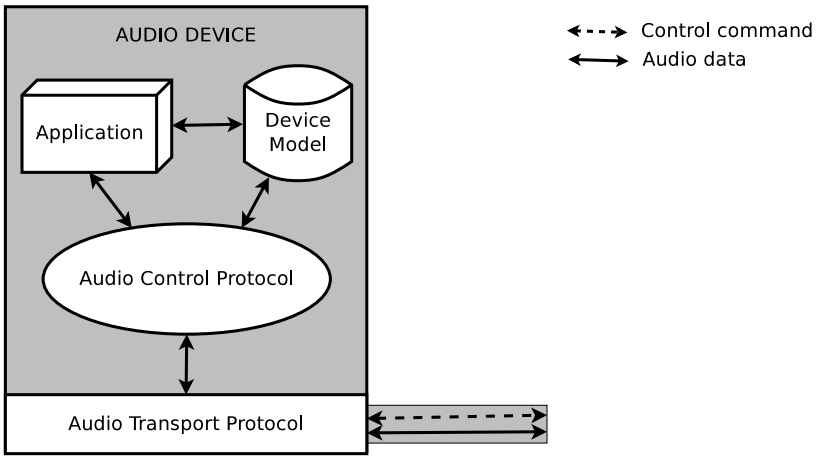


Figure 1.1: Digital audio device

The actual device functionality is implemented by the *application* component which is illustrated as *'Application'* in Figure 1.1. The *'Application'* interacts with the *'Device Model'* in order to modify and/or monitor the state of the device's features. The *'Device Model'* holds the state of the device. A change to a feature within the *'Device Model'* could cause the '*Audio Control Protocol'* to perform a task. For example if a remote controller has requested to be notified when the value of a control changes, then the audio control protocol would send a message containing the updated value to the remote controller. All protocol-specific implementations are implemented by the '*Audio Control Protocol'*. The *'Audio Transport Protocol'* is responsible for transmitting audio data and control commands.

The audio transport component forms the base on which audio device communication is built. It ensures communication and audio data exchange between networked digital audio devices.

1.2 Audio Transport Technology

Audio transport technology refers to the technology that allows audio data that is transmitted by a source device to be transported to any number of destination devices, without degradation in the quality of the audio and without a noticeable delay. This requires that the audio transport protocol be capable of ensuring that the necessary resources are available for the duration of the audio transmission. Given that the devices are connected on a network, they will need to share network resources such as the available bandwidth for data transmission on the network.

When audio is distributed on a network, it is necessary that the devices that receive the

audio are synchronized to the transmitting device. This will ensure that glitches and jitter are not present in the reproduced audio at the receiving device(s). Furthermore it ensures that there is control over the presentation time of the sound at the many destination devices.

At present, there are a number of audio transport protocols. They include:

- IEEE 1394 [2] - a serial bus networking architecture that allows for the transmission of digital audio streams and control data, as well as for the synchronization of networked IEEE 1394 nodes. IEEE 1394 nodes are daisy-chained on the bus, and each bus is capable of interconnecting a maximum of 63 nodes.
- Ethernet Audio Video Bridging (AVB) [3] - a suite of IEEE standards that allow for the deterministic and guaranteed delivery of time-sensitive data over Ethernet networks. One of the standards in this suite defines a mechanism for exchanging time information between participating nodes.
- CobraNet [4] - a proprietary technology for transmitting multiple channels of audio, as well as sample clocks on Ethernet networks.
- RockNet [5] - a proprietary technology for low latency audio data transmission using CAT5 cables. It operates on a ring networking technology with a maximum of 99 nodes on the network.
- EtherSound [6] - a proprietary technology that allows for the transmission of high-quality audio with low latency over standard Ethernet networks.
- Q-LAN [7] - a proprietary technology that will allow for the transmission of low-latency audio and control data over gigabit (or higher rate) Ethernet networks.
- RAVENNA [8] - an IP-based networking technology for low latency real-time audio data transport over a tightly synchronized network. It utilizes existing standards and networking infrastructure.
- Livewire [9] - a proprietary IP-based audio distribution technology over standard Ethernet infrastructure. It ensures low latency audio transmission over a Local Area Network (LAN).
- Dante [10] - a proprietary technology that utilizes an Ethernet infrastructure for the distribution of multiple channels of low latency media.

- AES-X192 [11] - an Audio Engineering Society (AES) project that is currently in progress, whose goal is the transmission of low latency audio over Internet Protocol (IP) based networks. It seeks to define a set of recommendations of existing standards and procedures so as to achieve interoperability between networked IP-based audio devices.

The above audio transport protocols are described in chapter 2. Some of them have been (or are in the process of being) published by standards organizations, thus access to documentation about the particular technology is readily available from the appropriate standards body. These include IEEE 1394 and Ethernet AVB, which have been published by the Institute of Electrical and Electronics Engineers (IEEE) [12], and AES-X192 which is in the process of standardization by the Audio Engineering Society (AES) [13].

Because the audio formats (encoding), supported sampling rates, and synchronization mechanisms that are used by the different audio networking technologies differ, devices that are networked using different technologies are unable to exchange audio data, even when they utilize the same transmission medium. For example, Ethernet AVB and EtherSound both utilize CAT5 cable and an Ethernet physical layer to network devices, but Ethernet AVB devices cannot receive audio streams that are transmitted by an EtherSound device, and vice versa.

While the audio transport protocols ensure that whatever is received for transmission is reliably and promptly delivered, the actual control of networked devices is determined by audio control protocols.

1.3 Audio Control Protocols

An audio control protocol creates defined instructions and responses as messages that are exchanged between devices on a network. It is also responsible for providing meaning to received instructions from a remote device. This ensures that only compliant devices can understand messages that conform to a particular protocol.

An audio control protocol can incorporate a scheme by which a device can be made aware of other devices on the network that implement the same audio control protocol. This is referred to as device discovery. Typically, a device discovery mechanism involves:

- a device that seeks to discover other devices on the network. This is typically called a controller. The controller requires knowledge of the other devices on the

network, and it may request to be informed whenever a device joins or leaves the network.

- a device that announces its presence and availability on the network. This device responds to a discovery query from the controller. The response will typically contain the unique identifier and address of the responding device.

Although any number of devices can announce their presence, the number of controllers permitted on a network depends on the particular audio control protocol. The discovery of networked devices by a controller is typically the first step before device enumeration, monitoring and control.

The *device model component* that was described in section 1.1 is defined by the audio control protocol. A device model provides a uniform way for compliant devices to present their features and controls. This makes it easy for a controller to enumerate the various controls and features that exist within a device. The particular messages that will be used to explore the device model are defined by the audio control protocol. Although the device model does not actually represent how the control data is stored internally by a device, it ensures that particular information can be consistently accessed (and modified) on compliant devices. For example, it might be possible to determine the maximum number of audio sources by accessing a particular location in the device model of all compliant devices.

The procedure used for establishing and destroying audio stream connections between devices is defined by the audio control protocol. This includes defining the types of messages and possible responses that can be obtained, as well as the order in which the messages should be transmitted. The process of 'setting up' (establishing) or 'tearing down' (destroying) audio stream connections is known as connection management. Each audio control protocol specifies its own technique for connection management.

Currently, there exist a number of audio control protocols. Some audio control protocols transport their messages within OSI/ISO layer 3 (mostly within IP) packets. Such audio control protocols will be referred to as *layer 3 audio control protocols*, in this document. Some other audio control protocols transport their messages as frames within OSI/ISO layer 2 frames. These will be referred to as *layer 2 audio control protocols*, in this document.

Some of the available audio control protocols are:

- OSC - the *Open Sound Control (OSC)* protocol is a transport layer independent media content format for real-time control messaging [14].

- ACN - the *Architecture for Control Networks (ACN)* is a suite of protocols and languages that can be combined to create a reliable control data distribution network [15].
- IEC 62379 - the *Common Control Interface for Networked Audio and Video Products (IEC 62379)* is a suite of standards that allow for control and audio data distribution over different network infrastructures [16].
- AES-64 - the *Audio Engineering Society standard for audio applications of networks - Command, control, and connect for integrated media (AES-64)* is an IP-based peer-to-peer command and control protocol that allows for device control and monitoring [17].
- OCA - the *Open Control Architecture (OCA)* is a protocol for controlling and monitoring networked devices [18].
- AV/C - the *Audio Video Control (AV/C)* protocol is an IEEE 1394-based device control protocol.
- IEEE 1722.1 - is a protocol for device discovery, connection management and control protocol for IEEE 1722-based devices on Ethernet AVB networks [19].
- mLAN - the *music Local Area Network* (mLAN) is an IEEE 1394-based audio control protocol that allows for the transmission of audio and music control data, as well as timing information between networked devices [20].

Further details about the above audio control protocols are provided in chapter 3.

Usually, an audio control protocol message is transported on the same audio transport technology that is used for transmitting audio stream data. For example, the IEEE 1394 serial bus provides two types of transactions for data transfers. One of the IEEE 1394 transactions is the *isochronous transaction* in which isochronous packets are transmitted at regular intervals. An isochronous transaction is used for time-sensitive data (such as audio) transmission. The other IEEE 1394 transaction is the *asynchronous transaction*, which entails the transfer of asynchronous packets and is typically used for control messaging. Audio control protocols such as mLAN and AV/C utilize asynchronous transactions for transporting protocol command messages.

An audio control protocol message could be transmitted as:

- *unicast* - from a controller to a particular target device.

- *multicast* - from a controller to any number of devices within a group. Each member of the group 'listens' for messages that are addressed to the entire group.
- *broadcast* - from a controller to every device on the network. This type of message is received by all devices on the network.

Each audio control protocol will define which of these message types it uses. Typically, an audio control protocol will transmit its messages using more than one of the above types of transmission, depending on what it is trying to achieve. For example, when attempting to discover all networked devices, the IEEE 1722.1 protocol utilizes multicast messaging. However, when an IEEE 1722.1 controller sends a message to control a particular feature on a target device, it transmits its control command as unicast to the target.

1.4 Problem Statement

Audio devices on a network will communicate via a defined audio control protocol. This entails the exchange of protocol messages between these devices. The messages may be received by any number of devices that are on the network. However, only devices that implement the same audio control protocol can understand the received messages. Thus, communication is only between devices that implement the same audio control protocol.

Figure 1.2 shows an example of OSC and AES-64 devices on an Ethernet AVB network.

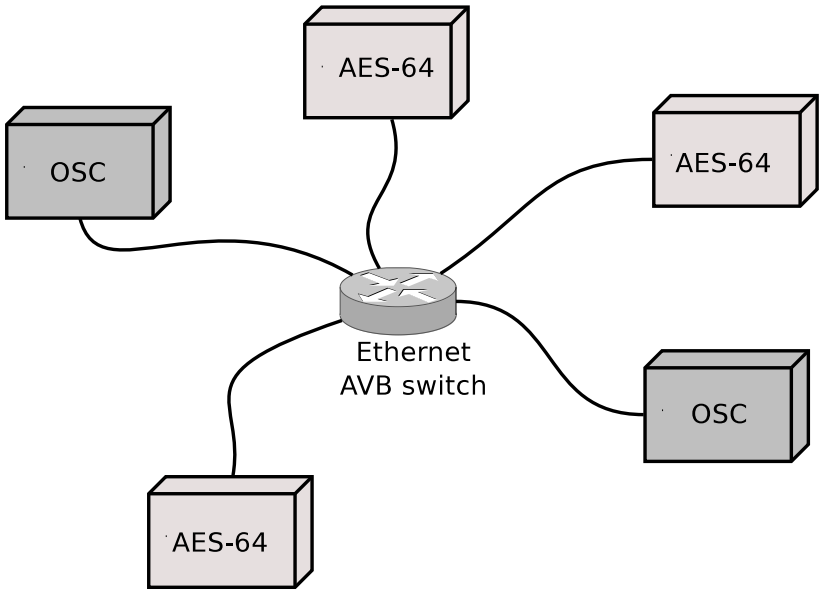


Figure 1.2: Ethernet AVB network with OSC and AES-64 devices

There are five devices on the network that is depicted in Figure 1.2. Two of these devices implement the OSC audio control protocol, and the other three implement AES-64. If an AES-64 network controller is seeking knowledge of all streaming Ethernet AVB devices on the network by broadcasting an AES-64 discovery message, only the AES-64 devices on the network will respond. The network controller will be unable to enumerate, monitor and control the state of the OSC devices on the network.

In this document, the above dilemma is referred to as the *interoperability challenge*. It results in a situation where it becomes challenging to have a single network manager (or control application) that is capable of configuring networked devices that utilize disparate audio control protocols. Further description of the interoperability challenge can be found in chapter 4.

This research project proposes a solution to the protocol command interoperability challenge. The proposed solution involves the use of a command translator that is capable of receiving messages that conform to one audio control protocol, then translating it to the equivalent message or messages in another protocol, in order to achieve the same desired result. The command translation approach is described in the next section.

1.5 Command Translation Approach

The command translation approach requires that a command translator receive commands that conform to one audio control protocol, translates them into commands for a second audio control protocol, and then transmits the translated messages to a target device. Figure 1.3 depicts an audio network that incorporates a command translator.

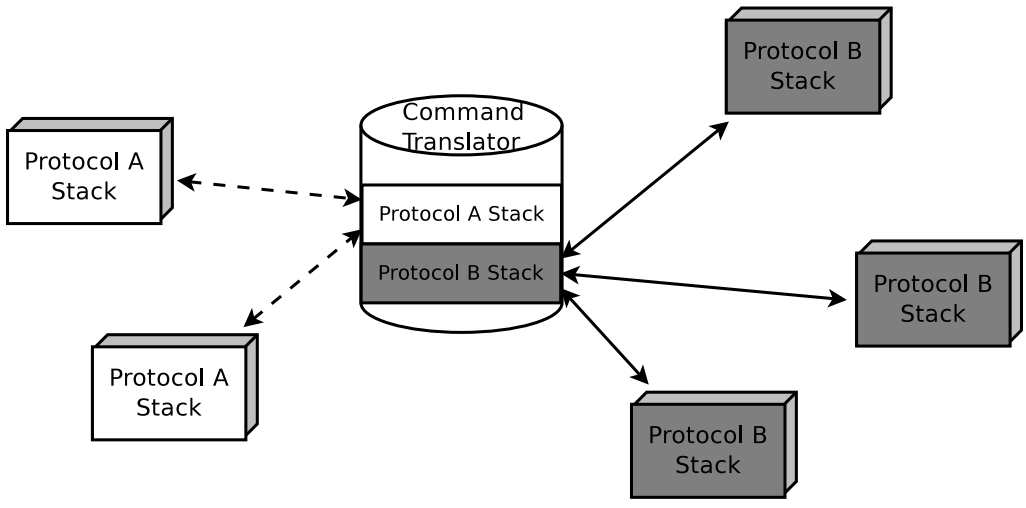


Figure 1.3: Audio network that incorporates a command translator

The *'Command Translator'* in Figure 1.3 is a proxy that enables devices that implement the *'Protocol A Stack'* to communicate with the devices that implement the *'Protocol B Stack'*, and vice versa. This enables interoperability between the networked devices, and allows a common controller to configure the devices on the network.

When implementing a command translator, the first consideration is to determine the protocol that will be used by the controller within the network. While making such a consideration, there are at least two possible scenarios. In one scenario a network of audio devices that implement a particular protocol has already been deployed, and an audio engineer will be seeking to incorporate devices that implement different control protocol(s) into the network. In another scenario an audio engineer is designing a new audio installation. Whichever is the case, it is desirable to have a single control protocol for the entire network.

This research document will describe the design and implementation of a command translator. This command translator has been implemented as a proxy that implements the audio control protocols of the devices that need to communicate on the network. The proxy is capable of enabling interoperability between audio devices that conform to different control protocols. Furthermore, the proxy enables a common controller to setup stream connections between the networked devices.

In order to investigate the effectiveness of the command translation approach, two proxy implementations will be described in this document. The first implementation, which is described in chapter 6, will enable connection management of networked devices that implement layer 3 audio control protocols. These are devices that transmit their control messages within OSI/ISO layer 3 packets, in particular OSC and AES-64 devices. The second implementation, which is described in chapter 8, will enable connection management between devices that implement layer 3 and layer 2 audio control protocol*s*. The *'*layer 2 audio control protocol*'* devices refer to audio devices that transport protocol messages within OSI/ISO layer 2 packets. The AES-64 and IEEE 1722.1 protocols will be used in the second proxy investigation. Motivations for choosing these protocols are provided in chapter 3.

Following the creation of the proxy, a number of tests will be conducted to determine the efficiency of the proxy. This will involve timing the connection management process to determine whether the proxy adds a significant overhead when observed by a user. If the overhead is large, it will discourage the adoption of the proxy in live audio distribution networks. Chapter 9 provides an analysis of the test results.

The layout of the rest of this document is described in the next section.

1.6 Chapter Layout

Chapter 2 describes the nature of networking technologies that are being used for the distribution of audio in different types of installation contexts, including *live sound reinforcement*, *commercial audio*, *audio recording*, and *audio post-production*. The chapter also describes some of the qualities that are required for an audio networking technology to distribute time-critical audio data across a network, thus fulfilling its role as an audio transport protocol. This is followed by an overview of some of the available audio networking technologies.

Chapter 3 describes some of the available audio control protocols that allow for remote device monitoring, configuration and control. To provide further insight into the operations of audio control protocols, three audio control protocols are described in detail. These are the OSC, AES-64 and IEEE 1722.1 protocols. A motivation for the choice of these three audio control protocols, is also provided.

Chapter 4 describes the *interoperability challenge* and how it affects current audio network design and deployment. It also provides further details about the command translation approach for interoperability between networked audio devices that implement different audio control protocols.

Chapter 5 describes the design and implementation of an Ethernet AVB end station that is capable of transmitting and receiving IEEE 1722 streams. The end station implements a layer 3 audio control protocol, which refers to an audio control protocol that communicates by transporting messages encapsulated within an OSI/ISO layer 3 packet. In particular, the Ethernet AVB end station that is described implements the OSC protocol. At the time of creating the OSC end station, OSC devices that implement Ethernet AVB were not commercially available. Hence the need to develop an OSC server that runs on a workstation.

In chapter 6 the design and implementation of a command translator, in particular a proxy, is described. The proxy that is described allows for interoperability between AES-64 and OSC devices. Furthermore, it will allow for an AES-64 controller to discover OSC devices, and configure audio stream connections between the OSC devices and AES-64 devices on an Ethernet AVB network.

Chapter 7 describes the design and implementation of Ethernet AVB end stations that implement the IEEE 1722.1 standard. The end station is referred to as an AVDECC entity. At the time of implementing the AVDECC entity, the IEEE 1722.1 standard was still being ratified by the IEEE standards association, thus there were no commercially available AVDECC entities. Before creating the AVDECC entity, the IEEE 1722.1 stan-

dard was implemented as a software library (called *libavdecc*) that can be utilized to create an AVDECC entity on a workstation. The libavdecc software library has been created for *Windows* and *(Ubuntu) Linux* platforms. The design, implementation and operation of *libavdecc* is also described in chapter 7.

Chapter 8 describes the design and implementation of a layer 2/layer 3 proxy that enables interoperability between devices that communicate by exchanging messages which are encapsulated within different OSI/ISO layer packets. In particular it demonstrates how a proxy can be used to ensure common control of networked AVDECC entities and AES-64 devices.

Chapter 9 is an analysis of the proxies created in chapter 6 and chapter 8 in order to determine their efficiency. This is carried out by determining the overhead that is added when connection management is performed with the aid of a proxy.

Chapter 2

Digital Audio Network Technologies and Interoperability

Digital audio networks are becoming widely deployed as the preferred technology for interconnecting audio devices. Typically on such networks audio is sampled, packetized, then transmitted on the network by a source device. A destination device receives the audio data packets, strips the audio data from the packet header (which is used for routing the packets on the network), then it reproduces the audio. It is the responsibility of the networking technology to ensure that the audio being transported arrives at its destination without degradation in quality. That is, that there are no glitches or jitter in the sound presented to the destination device.

When audio devices are networked such that the audio transmitted by a source device is reproduced at the destination device(s) without distortion in the sound, interoperability is said to exist between the devices. In order to achieve interoperability, the networked audio devices should be able to receive and/or transmit audio with similar formats, and the devices should have clocks that are synchronized.

This chapter describes some of the available audio networking technologies, and highlights some of the properties that enable them provide low-latency high quality audio transmission. Audio networking technology interoperability concerns are also discussed.

2.1 Audio Networking Technologies

In recent times, there has been a wide scale adoption of digital networking technologies as a means for interconnecting audio devices. This has been driven by a number of

factors, which include the ability to transmit multiple channels of audio on a single cable. Digital networking technologies make it possible to remotely control and monitor networked audio devices [21]. Also, digital audio networks are able to provide redundancy such that the network continues to operate after a single point of failure, thereby increasing the reliability of the audio transmission [22].

Audio networking technology refers to the technology that is utilized by a digital audio network to distribute multiple channels of audio over long distances. An audio networking technology incorporates a transport protocol which guarantees that the transmitted audio is delivered without distortions in sound quality. The technology is responsible for providing the transport mechanism for audio between networked devices, such that audio that is transmitted by a source device is routed and received by one or more destination device(s).

Audio networking technologies have been used in various applications to deliver high-quality audio among devices within a network. Some of the audio applications are [23]:

- *Live sound reinforcement* - where audio generated by a live performer is distributed across a venue, for instance a musical concert in a stadium.
- *Commercial audio* - where audio from a source is distributed along several routes to various parts of a venue, for instance background music playing in different rooms, areas, and lounges within a casino.
- *Audio recording* - where audio is captured in real-time, routed (in some instances, together with previously recorded tracks) through various signal processors, then monitored. This is typically the case in music recording studios, and broadcast studios.
- *Audio post-production* - where audio is edited and recorded onto a media storage device very often with video. This application typically consists of a network of digital audio workstations that are able to access digital audio from hard disks and edit it.

Each of the above applications emphasizes particular performance requirements for the networking technology. For instance in an auditorium with a live band performing, there can be no distortions or delays in the sound feed from speakers which are typically large distances apart. Furthermore the sound produced on such speakers must appear to be ‘in sync’, there cannot be perceived delays. When designing a particular audio network, care should be taken to ensure that the technology that is deployed does not interfere with the sound quality.

Whether an audio networking technology is used in the context of live sound reinforcement, commercial audio installations, audio recording, or audio post-production, there are certain requirements that the transport technology must fulfill. These include:

- network resource allocation,
- synchronization of networked devices, and
- network latency control

These requirements ensure that audio that is produced and transmitted by a source device, arrives at the intended destination device(s) and that the audio is replayed within a limited time interval. The above network requirements for audio data transmission are described in the following subsections.

2.1.1 Resource allocation

On a digital audio network, the network resources are shared between the interconnected nodes. For example, the network link has a fixed maximum bandwidth that will be shared by all connected nodes. These network resources are finite, and as more nodes are added on a network the resources become strained. It is the responsibility of the audio transport technology to ensure that there is an allocation scheme for the network resources. Such a scheme should ensure that a device wishing to transmit audio is guaranteed the necessary resources for the duration of the audio transmission. When the device no longer needs to transmit on the network, the resource allocation scheme should ensure that the resources become available to other devices wishing to transmit on the network. In essence, since there are limited resources on a digital audio network, the networking technology should ensure that the resources are adequately managed.

2.1.2 Device synchronization

Audio is sampled, packetized and transmitted within *streams* on most digital audio networks. Usually an audio stream is transmitted from a single source to any number of destination nodes. Consider a scenario where a source node is streaming audio on a network, and two destination nodes are receiving the audio stream. Without the right mechanism, if the two receiving nodes play back the audio, it is likely that the sound reproduced by each node is out of sync. In other words, the two audio streams reproduced at the destination nodes are not time aligned. It is the responsibility of the audio networking technology to implement a mechanism that will overcome this.

By implementing a synchronization mechanism, an audio networking technology ensures that the networked devices have a common sense of time. There are two aspects to synchronization on an audio network, and they are [24]:

- *Sample rate recovery* - which enables a common sample rate to be utilized by transmitting and receiving nodes. This ensures that buffer *overflow* and *underflow* are eliminated at the receiving node(s), and eliminates glitches and jitter in the sound quality. A buffer overflow occurs on the receiving node ddwhen the transmitting node is sampling audio at a higher sampling rate than the receiving node. A buffer underflow occurs on the receiving node when the transmitting node is sampling audio at a lower sampling rate than the receiving node.
- *Time alignment* - which ensures that the "presentation time" of audio at the receiving nodes are aligned. It compensates for delays that are introduced as audio is being routed from a transmitting node to any number of receiving nodes.

For example, on an IEEE 1394 network the sampling rate recovery on audio networks can be achieved by transmitting sampling rate information together with the audio data. The sampling rate information is added as part of the audio packet header. This is approach is not required if all devices on the network work on the same clock.

One approach to achieving time alignment is by distributing the clock information of one node to the other nodes on the network. The source of the time information is called the master clock, and the other nodes that 'sync' to it (master clock) are slaves. At a specified *presentation time*, the audio signal is 'presented' for processing by a receiving node. This ensures that audio is reproduced at the same time by multiple receiving nodes on a network.

2.1.3 Network latency

When audio is transported on a digital network, it experiences delay as it moves from a source to a destination device. This delay is referred to as the network latency. It is the difference between the time at which the source node transmits an audio packet and the time when the packet arrives at the destination node. An audio networking technology endeavors to reduce the network latency, so that audio is delivered in the shortest possible time.

Fonseca attributes latency on a digital audio network to three delay factors, namely [25]:

- *Propagation delay* - which refers to the delay in transfer of (audio) data from the source to its destination. This delay depends on the speed of the transmission cable.
- *Frame delay* - which refers to the delay between when the first bit and last bit of a (data) frame are transmitted. This is dependent on the size of the frame and the bandwidth of transmission.
- *Switch latency* - which refers to the delay introduced by intermediate network switches when they store received data frames. This is known as *buffering latency*. The switch latency also includes the *forwarding latency*, which is a property of the switch and it indicates how long it takes for the switch to process received frames.

These three factors contribute to the overall latency of the network. Although these delay factors cannot be completely eliminated on a digital audio network, the networking technology should ensure that the overall latency is minimal.

In order to attain interoperability between networked audio devices the three requirements (resource allocation, device synchronization, and network latency) have to be addressed by the networking technology. It is also required that the networked devices support similar audio formats. Otherwise a receiving device will be unable to reproduce the audio data sent from a transmitting device.

The available audio networking technologies address these requirements in different ways. In the following section, some of the audio networking technologies currently in use are described, with insights into how they fulfill these requirements.

2.2 Overview of current Audio Networking Technologies

Currently there are a number of audio networking technologies, which ensure that audio that is being transmitted by a source device is distributed and timeously delivered to a number of destination devices, without degradation of the audio quality. Some of these technologies have been standardized by standards bodies such as the Institute of Electrical and Electronics Engineers (IEEE) [12], and others are proprietary technologies that are utilized by specific manufacturers.

Standardization enables interoperability since it provides assurance to manufacturers that by conforming to a standard, their products can exchange audio data with compliant products from other manufacturers.

The audio networking technologies that are described in this chapter are classified into two categories based on their implementation strategy and level of operation. In accordance with the OSI/ISO model, the categories are [26]:

- *Layer 2 audio networking technologies* - enable packet transmission and reception at layer 2 of the OSI/ISO model [27].
- *Layer 3 audio networking technologies* - enable packet transmission and reception at layer 3 of the OSI/ISO model [27].

Although different approaches are used to address the requirements mentioned in section 2.1, the audio networking technologies that are described in this chapter take into consideration the requirements in their design and deployment. The following subsections describe the two categories of audio networking technologies.

2.2.1 Layer 2 Audio Networking Technologies

In this section a number of layer 2 audio networking technologies are described. These include:

- IEEE 1394,
- Ethernet AVB,
- CobraNet,
- RockNet,
- EtherSound.

The descriptions that are provided in this section are intended to provide an overview of these layer 2 audio networking technologies, which are currently available and are being deployed in various audio installations.

2.2.1.1 IEEE 1394

IEEE 1394 is a technology for high-speed data transfers on a serial bus. This technology has been standardized and published by the IEEE Standards Association (IEEE-SA) [28]. The IEEE 1394 standard has evolved over the years since its original publication in 1995. The various IEEE 1394 specifications are:

- IEEE 1394-1995 - is the original IEEE 1394 specification and was published in 1995 [29]. It defines data transfer rates of 100Mbps, 200Mbps, 400Mbps, and a 6-pin cable connector [30].
- IEEE 1394a-2000 - is an amendment to the IEEE-1394-1995 specification, which was published in 2000[31]. This amendment was aimed at addressing performance issues, and improving interoperability between the various vendor implementations of IEEE 1394-1995 that existed at the time. It also defined a 4-pin connector, that was intended for small devices such as hand-held video cameras, by removing the power signals (2 pins) of the 6-pin connectors.
- IEEE 1394b-2002 - was published in 2002 as an amendment to the IEEE 1394a-2000 specification. It enables full-duplex operation, and improved the transfer speed of the serial bus to 800Mbps [32]. IEEE 1394b-2002 also defined a 9-pin connector for IEEE 1394 devices.
- IEEE 1394c-2006 - was published in 2006 in order to meet several application requirements [33]. The improvements added by the IEEE 1394c-2006 specification include interconnecting IEEE 1394 devices with new cable types (such as CAT5e, CAT6, GOF, POF, and GigE) [32].
- IEEE 1394-2008 - combined and amended the various IEEE 1394 standards (IEEE 1394-1995, IEEE 1394a-2000, IEEE 1394b-2002, and IEEE 1394c-2006) into a single document [2].

Presently, the 1394 Trade Association (1394TA) is responsible for compliance and interoperability testing. The 1394TA is also responsible for marketing the IEEE 1394 networking technology [34].

Figure 2.1 shows a simple network of three IEEE 1394 nodes.

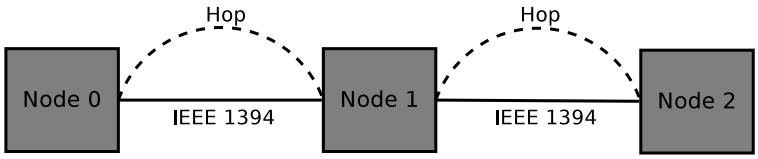


Figure 2.1: IEEE 1394 serial bus interconnections

As shown in Figure 2.1, IEEE 1394 nodes are daisy-chained on a single bus. The IEEE 1394 technology permits a a maximum of 63 nodes per bus. For large installations IEEE 1394 bridges can be used to enable a maximum of 1023 buses to be interconnected, with each bus possibly having as many as 63 devices on it [35]. Figure 2.1 shows three networked nodes, with a single hop between *'Node 0'* and *'Node 1'*, and two hops between *'Node 0'* and *'Node 2'*. A maximum of 16 hops is permitted between a transmitting and a receiving node [36].

When a device is added or removed from the (IEEE 1394) bus, a process known as *bus reset* occurs. At the end of a bus reset:

- each node is assigned a *node ID* that uniquely identifies the node
- the transmission speeds between nodes are determined, and
- every node is aware of the bus topology.

There are two types of transactions that are used for data transfer on the IEEE 1394 serial bus. These IEEE 1394 transactions are:

- *Asynchronous transaction* - guarantees delivery of the transmitted data. Asynchronous transactions are typically used for transmitting control messages, such as register read and write commands. An asynchronous packet is used to transport an asynchronous message to a specified target node, which is identified by its node ID. The target node responds to each received asynchronous control message by sending an acknowledgement and requested data (if needed) to the requester.
- *Isochronous transaction* - ensures that at constant intervals, a constant amount of data is transmitted. Isochronous transactions are used for high speed deterministic data transmission on the IEEE 1394 serial bus, such as real-time audio and/or video streaming. The IEEE 1394 serial bus has a dedicated channel (channel 63) which is used for broadcasting isochronous messages.

The format of the audio data that is transmitted within an isochronous message has been standardized by the IEC 61883-6 specification [37]. The IEC 61883-6 specification is

the "Audio and Music data transmission protocol", which defines how multiple channels of audio can be simultaneously transmitted within *streams*, on the IEEE 1394 serial bus. IEC 61883-1 defines a *Common Isochronous Packet (CIP)* header for transporting the audio data [38]. CIP (among other things) enables a transmitter to specify the nominal sampling rate of the audio data within its payload.

Being a standardized protocol, IEC 61883-6 enables interoperability by guaranteeing that compliant nodes are able to exchange audio data.

IEEE 1394 has been utilized as the audio networking technology of choice for a number of audio device control protocols, such as *mLAN* and *AV/C* [20] [39, Pg 56]. These two audio control protocols will be described later in chapter 3 on *Audio Networking Control Protocols*.

IEEE 1394 resource allocation

On an IEEE 1394 bus, the two resources that are necessary for isochronous data transmission are:

- *channel* - on which to transmit a stream, and
- *bandwidth* - that is sufficient for the transmission.

The node that is responsible for managing these resources is called the *Isochronous Resource Manager (IRM)*. The IRM is determined at bus reset, and it monitors and maintains the allocation of channels and bandwidth for the transmission of isochronous data streams. In order to do this, the IRM implements two registers. These registers are the [40]:

- *CHANNEL_AVAILABLE* register - is a 64-bit register that maps to the 64 channels that are used for transmission on the bus. Initially this register has all of its bits set to '1'. A node wishing to transmit isochronous data will request a channel from the IRM. A successful request will cause the IRM to modify the value of the bit that corresponds to the acquired channel (in the *CHANNEL_AVAILABLE* register) to '0'.
- *BANDWIDTH_AVAILABLE* register - is a 32-bit register that holds the value of the available isochronous transmission bandwidth in *bandwidth units*. Initially the value of the *bw_remaining* field is 4915 (bandwidth units) for isochronous transmission, but it reduces as nodes are granted bandwidth for isochronous transmission. A requesting node indicates how much bandwidth (in bandwidth units) it requires for its isochronous audio data transmission. If the value of the *bw_remaining*

field is more than what is required, that is there is sufficient bandwidth, the *bw_remaining* value is reduced by the required bandwidth and a success response (to the lock transaction) is returned to the requesting node.

A node that wishes to transmit an isochronous stream is required to first acquire a transmission channel from the IRM before proceeding to request bandwidth.

IEEE 1394 device synchronization

Each IEEE 1394 node that is capable of isochronous transactions implements a *CYCLE_TIME* register. In order to synchronize the *CYCLE_TIME* registers of all connected nodes, the *cycle master* broadcasts the value of its *CYCLE_TIME* register within cycle start packets (nominally) once every 125 microseconds. This ensures that the *CYCLE_TIME* registers of the connected (isochronous transaction capable) nodes remain synchronized to that of the cycle master. The cycle start packet also indicates the start of an isochronous stream transmission.

IEEE 1394 network latency

IEEE 1394 is a high speed serial bus with transfer speeds of about 100Mbps, 200Mbps, 400Mbps and 800Mbps. The standard allows for faster speeds of 1.6Gbps and 3.2Gbps [32]. The IEEE 1394-2008 standard defines transfer speeds as high as 4Gbps [41]. Such high speeds allow for fast data transmission on the network, such that latency due to the networking technology (IEEE 1394) is minimal. The IEEE 1394 serial bus allows for interoperability between devices that implement different transfer speeds by ensuring that data is transfered at the lowest speed of the interconnected devices.

2.2.1.2 Ethernet AVB

Ethernet Audio/Video Bridging (AVB) refers to a collection of IEEE 802.1 standards that together provide the necessary quality of service (QoS) for the transmission of time-sensitive data, such as audio, over Ethernet networks. The development of the Ethernet AVB standards was driven by the fact that Ethernet enjoys widespread adoption in the Information Technology (IT) environment. Ethernet AVB is intended to provide deterministic and guaranteed delivery of time-sensitive data on IT networks [42].

Ethernet AVB incorporates Virtual Local Area Networks (VLANs), which allow a physical network to be partitioned such that devices which are physically connected can

be logically isolated into different domains, irrespective of their physical relationships. Thus a number of 'sub-networks' (VLANs) can exist on a single LAN or WAN. For example the ten devices that are shown to be physically connected in Figure 2.2, are isolated into three VLANs. Typically the rules that govern the creation of VLANs are specified by the network switches [43].

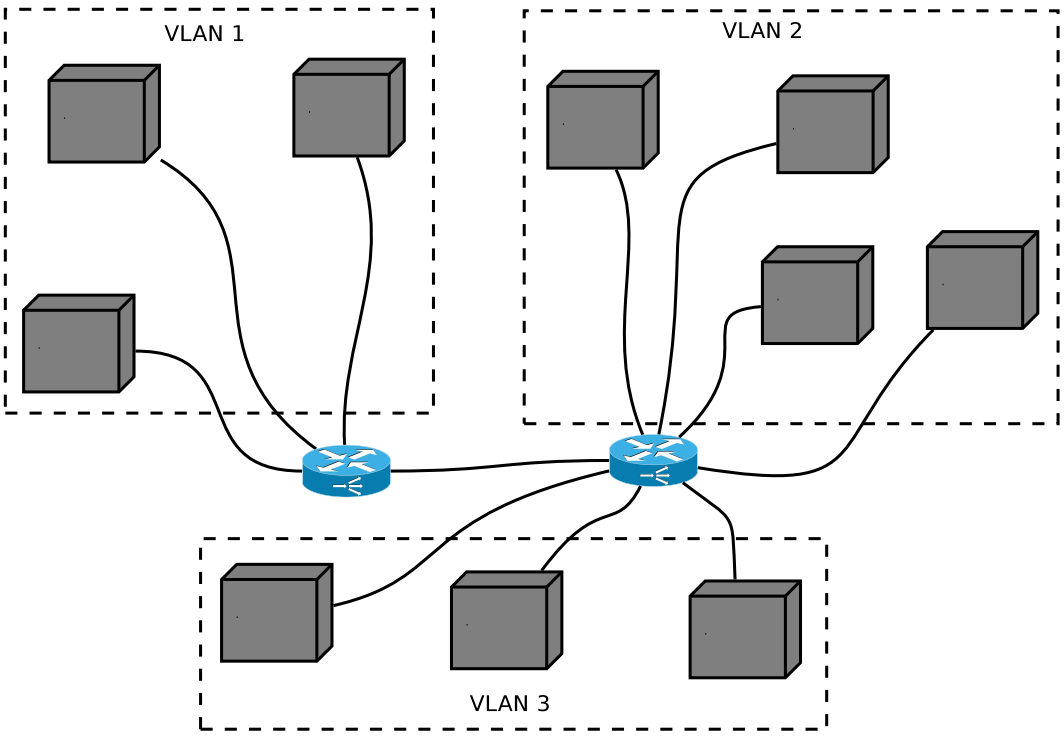


Figure 2.2: VLAN isolates physically connected devices

The documents that make up the Ethernet AVB standard are:

- ***IEEE 802.1Qat*** - the Stream Reservation Protocol (SRP) standard, which is an amendment to the IEEE 802.1Q standard for virtual bridged local area networks [44]. It was published in 2010 by the IEEE. The IEEE 802.1Qat standard defines two roles that exist when a data stream is being transmitted on the network. These are:

  1. *Talker* - a device that is the source or transmitter of a stream on the network.
  2. *Listener* - a device that receives a stream from the network.

  Talker and listener devices are broadly referred to as *endpoints* or e*nd stations*. The SRP protocol provides a mechanism that enables signaling of participating devices so that network resources, such as bandwidth and buffer space on intermediate bridges along the path from talker to listener, can be set aside for a

particular stream. In SRP interactions, a talker regularly publishes information about its available streams on the network, and a listener indicates the particular stream that it is interested in. Also, the intermediate bridges on the path from talker to listener ensure that the necessary resources are available before transmission begins. As SRP signaling messages are being exchanged between talker and listener, the intermediate bridges may modify them (the signaling messages) to indicate whether the network can guarantee resources for the particular talker stream. SRP eliminates packet loss by guaranteeing that prior to a talker transmitting a stream, there are sufficient network resources on the path to the listener.

- ***IEEE 802.1Qav*** - an amendment to the IEEE 802.1Q standard for virtual bridged local area networks, that was published in 2010 by the IEEE as an enhancement for forwarding and queuing of time-sensitive streams [45]. This standard defines a traffic shaping mechanism that ensures smoothing of traffic in order to transmit stream packets at an evenly distributed rate [42]. It does this by adapting the IEEE 802.1Q frame priority tagging scheme, and by defining a forwarding policy for the transmission of frames on the network. This ensures that, although AVB stream data and non-AVB data are transmitted on the network, priority is given to *isochronous* stream data over *asynchronous* data on 'exit' ports of networked endpoints and bridges.

- ***IEEE 802.1AS*** - the IEEE standard for timing and synchronization for time-sensitive applications in bridged local area networks, that was published in 2011 [46]. The IEEE 802.1AS standard was developed to enable networked AVB endpoints to synchronize their clocks. The IEEE 802.1AS ensures that:

  1. a common clock is distributed across the network.
  2. multiple streams, which are distributed across different paths on a network, are presented at the same time.

  Synchronizing the clocks of the networked endpoints ensures clock stability. The IEEE 802.1AS standard is based on the IEEE 1588 standard, which addresses system-wide time synchronization that is in the sub-microsecond accuracy range [47]. On an AVB network, endpoints that share a common clock are said to reside within the same *time domain*. The choice of the common clock that is distributed on an AVB network can either be manually selected, or can be automatically determined by the network.

- ***IEEE 802.1BA*** - the standard for an Audio Video Bridging System, which enables the network to isolate participating AVB endpoints from other devices on

the network [48]. This ensures that endpoints which are capable of reserving network resources, as well as synchronizing their clocks to that of a common clock, are identified. It defines a number of profiles for the features and options that should be implemented by AVB endpoints so that non-AVB compliant nodes (on the network) do not interfere with the operations of the AVB system. By referring to this standard, manufacturers have a standardized conformance requirement for developing AVB compliant devices.

Ethernet AVB networks include bridges and endpoints that cooperate to guarantee the delivery of time-sensitive data. A talker publishes information about the stream(s) it has to offer on the network. The network bridges modify the stream advert(s) to indicate whether there are sufficient resources and more accurately reflect the expected latency across the path, before propagating the advertisement on the network. A listener indicates to the network that it wishes to receive a particular stream by sending a *ready* message, which is propagated to the stream source (talker). As the *ready* indication from the listener is being propagated towards the talker, the network bridges (along the path) reserve the necessary network resources for the stream. The talker begins to transmit when it receives the *ready* message, with the assurance that its stream is guaranteed to be delivered to the listener.

On Ethernet AVB networks, each audio stream is identified by its 64-bit stream ID. The media transport that is used on an Ethernet AVB network is defined in the IEEE 1722 standard [49]. IEEE 1722 is also known as the *Audio/Video Transport Protocol (AVTP)*, and it defines the procedure for exchanging media and timing information between Ethernet AVB end stations. Audio that is formatted according to the IEC 61883-6 specification is transported within the AVTP payload.

The Ethernet AVB technology is used as the transport protocol for the IEEE 1722.1 protocol, which is described in chapter 3 on *Audio Networking Control Protocols*.

Ethernet AVB resource allocation

On Ethernet AVB networks, the reservation of network resources for time-sensitive stream transmission is achieved by two cooperating processes. These processes involve:

1. Signaling participating endpoints and bridges along the path from transmitter to receiver endpoints, requesting them to reserve resources.

2. Applying the resource reservation algorithm.

The signaling mechanism is described by the *Multiple Stream Reservation Protocol (MSRP)*, which is defined in the IEEE 802.1Qat standard [44]. A participating node uses MSRP to reserve network resources. MSRP uses *Multiple Registration Protocol (MRP)* to declare talker and listener attributes on the network [50]. The talker attributes are *talker advertise* and *talker failed*. The listener attributes are *listener ready*, *listener ready failed*, and *listener asking failed*. These attributes are multicast on the Ethernet AVB network.

When signaled (to do so), the network reserves the necessary transmission bandwidth and buffers within participating bridges. Foulkes describes how MSRP uses MRP to declared talker and listener attributes, as well as how these attributes are propagated across the network [1].

The *"Forwarding and Queuing"* algorithm, which is defined in IEEE 802.1Qav, prioritizes AVB traffic over non-AVB traffic [45]. In order to ensure that non-AVB traffic gets a chance to be transmitted on the network, every Ethernet AVB bridge is permitted 75% of its bandwidth to AVB traffic, and the rest to non-AVB traffic [51].

Ethernet AVB device synchronization

The procedure for synchronizing endpoints on an Ethernet AVB network is defined by the IEEE 802.1AS standard [46]. IEEE 802.1AS defines a *generalised Precision Time Protocol (gPTP)* which is a constrained form of the *Precision Time Protocol (PTP)* [52].

The gPTP protocol defines a *time-aware bridged LAN* as a LAN with interconnected:

- *Time-aware end stations* - which are endpoints that are capable of transmitting and receiving timing information by utilizing gPTP,
- *Time-aware bridges* - which are endpoints that have multiple ports and are capable of relaying timing information received on one port to the others.

Time-aware end stations and time-aware bridges make up what is referred to as a *time-aware system*. In a time-aware system gPTP messages (which contain time information) are distributed in order to synchronize participating endpoints to a common clock. The source of the common clock is called a *grandmaster,* and there can be only one grandmaster per *gPTP domain*. A gPTP domain refers to the scope within which gPTP messages are communicated, thus defining the scope of synchronized end points. A *Best Master Clock Algorithm (BMCA)* is used to determine the grandmaster in a gPTP domain [1].

A *master/slave* relationship exists in a time-aware system. This is shown in the gPTP domain depicted in Figure 2.3.

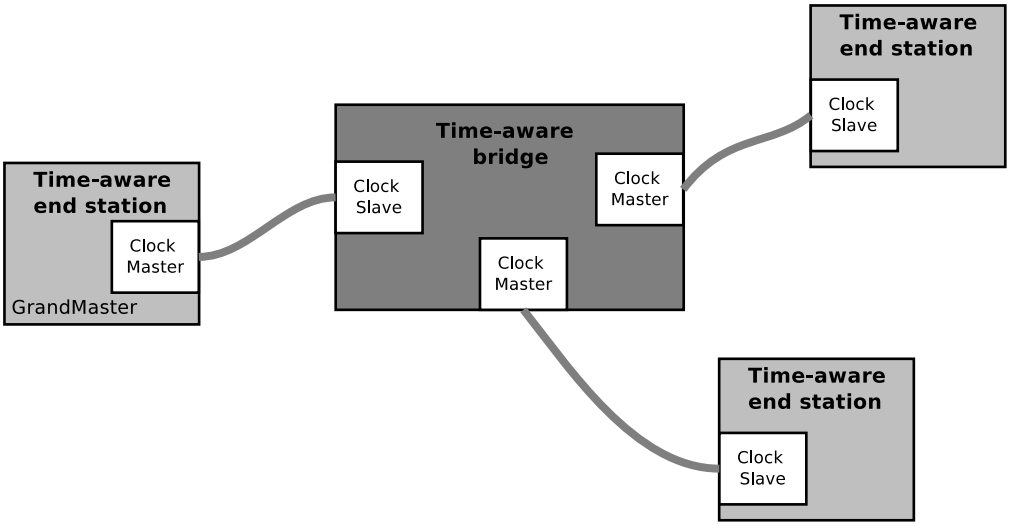


Figure 2.3: Master/slave relationship in a gPTP domain

In Figure 2.3, the time-aware (grandmaster) end station generates the clock information that is used to synchronize the other end stations and the bridge within the gPTP domain. The grandmaster clock is the clock master of the '*Time-aware bridge*' shown in the figure. In turn, the '*Time-aware bridge*' is the master of the other '*Time-aware end stations*' on the network, since it ('*Time-aware bridge*') is closer to the grandmaster than either of the '*Time-aware end stations*'.

The grandmaster transmits its current time within the gPTP messages that it propagates on the network. Each subsequent clock master bridge that receives the gPTP message adjusts the time to include an estimated propagation delay that the message encountered on its way. Then the clock master transmits the gPTP message to its slave endpoints. When a time-aware bridge receives the gPTP message, in addition to adding the propagation delay, it adds the possible delay the message would have encountered as it is being relayed across to its other ports. This additional delay is called the *residence time* [46].

Ethernet AVB network latency

As part of the quality of service (QoS) requirement of media transport on Ethernet AVB, stream packets must arrive within the shortest time as they travel from source to destination endpoints. The prioritization of time-sensitive data over other data that is being transmitted on the network ensures that within the shortest possible time an AVB

bridge would have transmitted a received packet from one port to the other port(s) on the path from talker to listener(s).

The IEEE 802.1Qav standard defines a traffic shaping protocol, which classifies AVB traffic into two classes [53]:

- Class A - with maximum latency of 2 milliseconds when there are 7 hops between transmitting and receiving nodes on a 100Mbps Ethernet connection.
- Class B - with maximum latency of 50 milliseconds when there are 4 hops between transmitting and receiving nodes on a 100Mbps Ethernet connection.

2.2.1.3 CobraNet

CobraNet is a proprietary technology that enables reliable streaming of multiple channels of audio data over Ethernet networks. The CobraNet technology utilizes Ethernet for the transmission of isochronous data, sample clock and control data.

CobraNet can be deployed on a network that either incorporates switches or repeaters (network hubs), but not a mixture of both devices on the same network. When CobraNet is deployed on a switched network, it is possible for the network to handle both CobraNet data and other Ethernet traffic. This is typically not the case in a network with repeaters. The choice of either a switched or repeater network determines how a CobraNet network is designed and deployed.

A CobraNet node comprises [54]:

- A high bandwidth hardware interface - which incorporates a Digital Signal Processor (DSP).
- A protocol stack - that combines multiple channels of isochronous data into an Ethernet packet.
- Device software - that enables device monitoring and management, as well as enabling clock generation and recovery.

Three types of packets are transmitted on a CobraNet network. These are [55]:

- *Beat packets* - these packets contain the network operating parameters, transmission permissions, and the sample clock that is used to synchronize all other devices on the network. The beat packets are multicast on the network, and their time-sensitive nature requires that they are timeously delivered.

- *Isochronous packets* - these packets typically contain the audio data, which are transmitted within *bundles*. A bundle can either be multicast to several networked devices, or unicast to a single networked device. Each bundle comprises multiple channels.

- *Reservation packets* - these are small sized packets that are multicast by each CobraNet device on the network. They contain transmission reservation information, network congestion information and the IP address of the CobraNet device. A CobraNet device transmits one reservation packet every second.

On a full-duplex switched network, CobraNet can transmit 32 channels of (20-bit) audio at 48kHz over a 100Mbit Ethernet link. On repeater Ethernet networks, 64 channels of (20-bit) audio at 48kHz can be transmitted within bundles. Each bundle will contain 8 channels of uncompressed audio [4]. CobraNet is capable of transmitting 16, 20, and 24 bit audio resolutions at 48kHz sampling rate in order to allow for interoperability with different types of devices. CobraNet is also capable of transmitting audio at 96kHz.

CobraNet utilizes the *Simple Network Management Protocol (SNMP)* for device monitoring and control [56]. Since CobraNet utilizes Ethernet for data transmission, it permits other control protocols (that use Ethernet for transport) to be transported on the network. This further enables interoperability with other networked devices.

CobraNet resource allocation

Audio is transmitted within isochronous packets on a CobraNet network. A beat packet signifies the beginning of an isochronous transmission cycle. Typically the isochronous cycle interval is 1.33 milliseconds, but for low-latency modes CobraNet allows isochronous cycle intervals of 0.66 milliseconds and 0.33 milliseconds [55]. In order to handle variations in delivery time and to reorder isochronous packets on a receiving device, CobraNet allows buffering of packets.

The reservation packets, which are transmitted once every second, are used to specify transmission resource reservations and network congestion information [54].

CobraNet device synchronization

Beat packets contain the clock of a participating device known as the '*conductor*' and it is multicast to all devices on the network. Usually, the conductor transmits 750 beat packets per second. Every other device on the network listens for the beat packets. On

receiving a beat packet, the CobraNet device synchronizes its clock to that of the sample clock within the beat packet. A beat packet is typically about 100 bytes in size.

CobraNet network latency

Transmission delay that is less than 400 microseconds is expected for beat packets, so as to ensure that the local clocks of networked devices are synchronized with the conductor's clock [55].

The latency for the transmission of isochronous packets on CobraNet is 5.33 milliseconds when the isochronous cycle interval is 1.33 milliseconds. CobraNet also allows low latency modes of 2.66 milliseconds and 1.33 milliseconds for isochronous packet transmissions when the isochronous cycle intervals are 0.66 milliseconds and 0.33 milliseconds, respectively [55].

2.2.1.4 RockNet

RockNet is a proprietary real-time, low latency audio networking technology that allows for the distribution of multiple channels of audio. It is deployed as a ring network topology that utilizes CAT5 cables to connect a maximum of 99 nodes [57].

It provides redundancy for fail-safe audio transmission by providing two connections for each networked device interface. The use of two network interface connections, together with the ring network topology, reduces the risk of a single point of failure. There are two variations of the RockNet technology, namely:

1. *RockNet 300* - which is able to transmit 160 channels of 24-bit audio at a sampling rate of 48kHz. It is also capable of 96kHz sample rate [57].

2. *RockNet 100* - which is able to transmit 80 channels of 24-bit audio at a sampling rate of 48kHz.

Interoperability exists between devices that implement the above types of RockNet technologies. To enable easy device configuration, RockNet devices have control buttons on their front panel.

RockNet allows for isochronous transmission of audio, together with standardized network traffic (such as TCP/IP) on the same network [5]. Detailed information on the RockNet technology could not be obtained.

RockWorks is a software package that can be used to remotely configure and monitor networked RockNet devices [5].

2.2.1.5 EtherSound

EtherSound is a proprietary audio networking technology that was developed by Digigram to allow for the deterministic transmission of high-quality audio with low-latency over standard Ethernet (IEEE 802.3) networks. It was first publicly presented in 2001 [58].

EtherSound devices can be incorporated into networks of devices with standard Ethernet interfaces and switches. It allows devices to be daisy-chained into a peer-to-peer network topology, as well as permitting star and fault-tolerant ring topologies. A combination of any of these network topologies is also permitted by the EtherSound technology.

There are two variations of the EtherSound networking technology. The choice of which EtherSound technology to deploy depends on the application requirements. The available EtherSound technologies are [59]:

- *EtherSound ES-100 Audio Transport* network - was designed for 100Mbps Ethernet infrastructure. This allows for the transmission of 64 bi-directional channels of 24-bit digital audio at 48kHz sampling rate, together with control and monitoring data.
- *EtherSound ES-Giga System Transport* network - utilizes a 1Gbps dedicated Ethernet infrastructure for the transmission of 512 channels of 24-bit digital audio at a sampling rate of 48kHz, together with 100Mbps of control and monitoring data.

EtherSound devices communicate by transmitting *EtherSound frames*, which are encapsulated within standard Ethernet (IEEE 802.3) frames. An EtherSound frame consists of an [6]:

1. *EtherSound header* - contains protocol specific information, that identifies an EtherSound frame.
2. *EtherSound payload* - contains the data that is being transported on the EtherSound network. The EtherSound payload consists of:
   (a) *Packet header* - that holds the packet type and subtype information.
   (b) *Packet data* - which is the actual data being transported.

The EtherSound protocol defines two types of data packets which are transmitted in the *EtherSound payload* on a 100Mbps network, namely:

1. *Command packet* - which is either a control command, or a request for status information.

2. *Audio packet* - which is used to transmit the channels of 24-bit PCM (digital) audio.

EtherSound devices implement a Field-Programmable Gateway Array (FPGA), which defines an internal database of 256 16-bit device registers. Remote device control and monitoring is achieved by reading and writing to these registers [6].

EtherSound includes a device control protocol and an Application Programming Interface (API) for the protocol. The API can be used by a control application to transmit command instructions to EtherSound devices that are produced by different manufacturers, thus enabling interoperability [59].

EtherSound resource allocation

EtherSound frames are used to transport audio data and clock information between the networked devices. In order to understand how the EtherSound network distributes audio data, an understanding of the various roles that can be fulfilled by the EtherSound devices is required. Figure 2.4 shows a network with five EtherSound devices, each fulfilling a specific role.

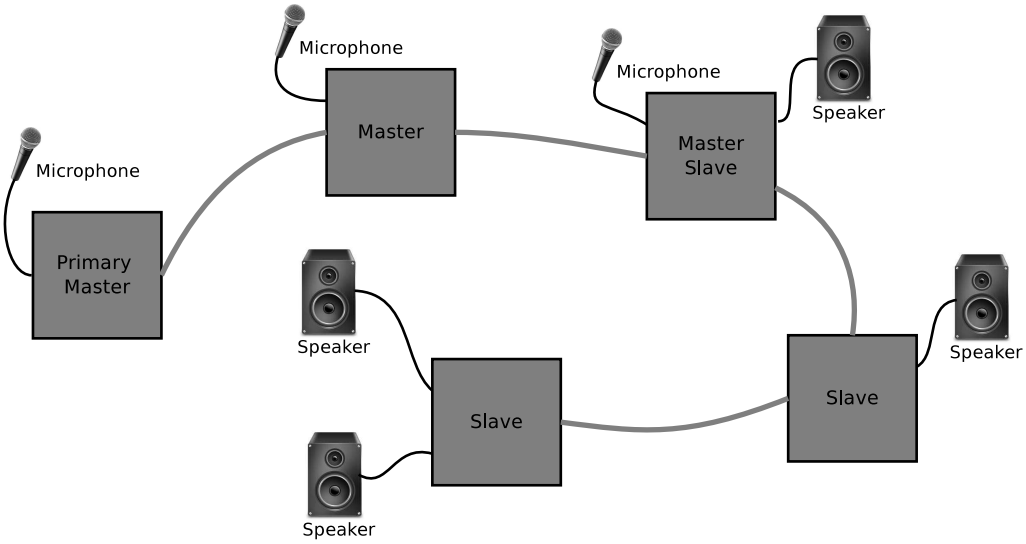


Figure 2.4: EtherSound network

The roles depicted in Figure 2.4 are:

- *Primary Master* - is a source of audio on the network, and the originator of the EtherSound frames that are transmitted downstream.
- *Master* - is an audio source device, and is located downstream from the primary master on the network. It responds (upstream) to control and status information commands from the primary master. A master device adds its audio data to the EtherSound frame as it flows downstream from the primary master.
- *Slave* - is a receiver of audio from the networked devices upstream. It obtains the audio data from EtherSound frames that are transmitted by the primary master on the network. It is also capable of responding upstream to control and status information commands from the primary master.
- *Master/Slave* - is downstream from the primary master and is both a transmitter and receiver of audio data. Like the master and slave devices, it is capable of responding to control and status information commands from the primary master.

EtherSound device synchronization

EtherSound devices synchronize their audio clocks to that of the network clock, which is derived from the primary master [59]. The primary master transmits (downstream) its audio data and its clock within an EtherSound frame.

A master, master/slave, or slave device synchronizes its clock with that obtained from the EtherSound frame transmitted by the primary master.

EtherSound network latency

Network latency on an EtherSound network is independent of the number of audio channels that are being transmitted [60].

The EtherSound ES-100 Audio Transport ensures a 125 microseconds end-to-end transmission time (of 6 samples) at 48kHz sampling rate. Each EtherSound interface in a daisy-chain Ethernet topology adds less than 1.5 microseconds latency [60]. Any switch along the path (from network input to network output) adds between 2 to 20 microseconds latency.

The EtherSound ES-Giga System Transport promises less than half a microsecond latency [60].

2.2.2 Review of Layer 2 Audio Networking Technologies

In the previous section, a number of audio networking technologies were described. Some of them have been standardized, in particular IEEE 1394 and Ethernet AVB, while others are proprietary networking solutions, such as CobraNet, RockNet, and EtherSound. Each technology defines its own transmission frame for audio data and timing information exchange. Some of the prominent features of these networking technologies are highlighted below.

- IEEE 1394
  - Serial bus network technology.
  - Transmits up to 64 channels of uncompressed audio.
  - Audio is formatted according to IEC 61883-6 specification.
  - Variable sampling rates (from 32kHz to 192kHz) for audio that is transported on the bus.
  - Synchronization information is obtained from the timing information that is distributed by the cycle master node.
- Ethernet AVB
  - Spanning tree network topology.
  - Ethernet links with dedicated Ethernet AVB switches.
  - Supports transmission of IEC 61883-6 formated audio within AVTP frames.
  - Supports sampling rates as high as 192kHz.
  - Synchronization is achieved by locking to a common clock that is distributed by the grandmaster node.
  - Media clock synchronization for both IEEE 1394 and AVB is implemented by each receiver, which extracts the presentation time stamps from the transmitted stream and uses these regular time stamps to adjust the media clock frequency to that of the transmitter.
- CobraNet
  - Spanning tree network topology.
  - Ethernet links with standard Ethernet switches and repeaters.
  - 64 channels of uncompressed audio.

  - Sampling rate of 48kHz is typical for audio with a resolution of 20-bit.
  - Networked devices are synchronized by a beat packet, which is originates from a conductor node. 750 beat packets are transmitted (by the conductor) every second.

- RockNet
  - Ring network topology.
  - Uses Ethernet link for data transmission.
  - *RockNet 300* transmits 160 channels of 24-bit audio.
  - *RockNet 100* transmits 80 channels of 24-bit audio.
  - Typically, sampling rate is 48kHz.

- EtherSound
  - Token ring communication.
  - Ethernet links with standard Ethernet switches on the network.
  - 64 bi-directional channels of 24-bit digital audio on 100Mbps Ethernet link.
  - 512 channels of 24-bit digital audio on 1Gbps Ethernet link.
  - Typical sample rate is 48kHz with a resolution of 24-bit.
  - Networked devices synchronize to the clock information obtained from the primary master.
  - One way audio traffic.

Most of the audio networking technologies described in this chapter, with the exception of IEEE 1394, utilize Ethernet links between networked nodes. EtherSound and RockNet permit 24-bit audio at 48kHz sampling rate. Uncompressed audio that is formatted according to IEC 61883-6 is permitted on IEEE 1394 and Ethernet AVB networks.

2.2.2.1 Interoperability on layer 2 networks

It is difficult to attain interoperability between devices that utilize different audio networking technologies. The existence of different network links, packet formats for audio transmission, and synchronization mechanisms, makes it difficult to guarantee reliable transmission of audio across different networking technologies. Even when the same

physical link (for example Ethernet) is used to connect the devices, the format of the frames for audio data transport and the synchronization mechanisms usually differ.

The following section describes a tunneling approach that enables interoperability between devices on IEEE 1394 and Ethernet AVB networks. This approach enables audio that is transmitted by an Ethernet AVB device to be received by an IEEE 1394 device. Likewise, the tunneling approach enables audio that is transmitted by an IEEE 1394 device to be received by an Ethernet AVB device.

2.2.2.2 Tunneling nodes for Layer 2 Interoperability

Foulkes has demonstrated how interoperability can be achieved between two standards-based audio networking technologies, in particular IEEE 1394 and Ethernet AVB [1]. The technique involved the use of *tunneling nodes* that have IEEE 1394 and Ethernet network interfaces.

The use of the tunneling nodes ensured deterministic delivery of audio data from an Ethernet source node to an Ethernet sink node and vice versa. The audio data was transported across an IEEE 1394 network, thus making the stream available to other IEEE 1394 devices. A layout of the network is shown in Figure 2.5.

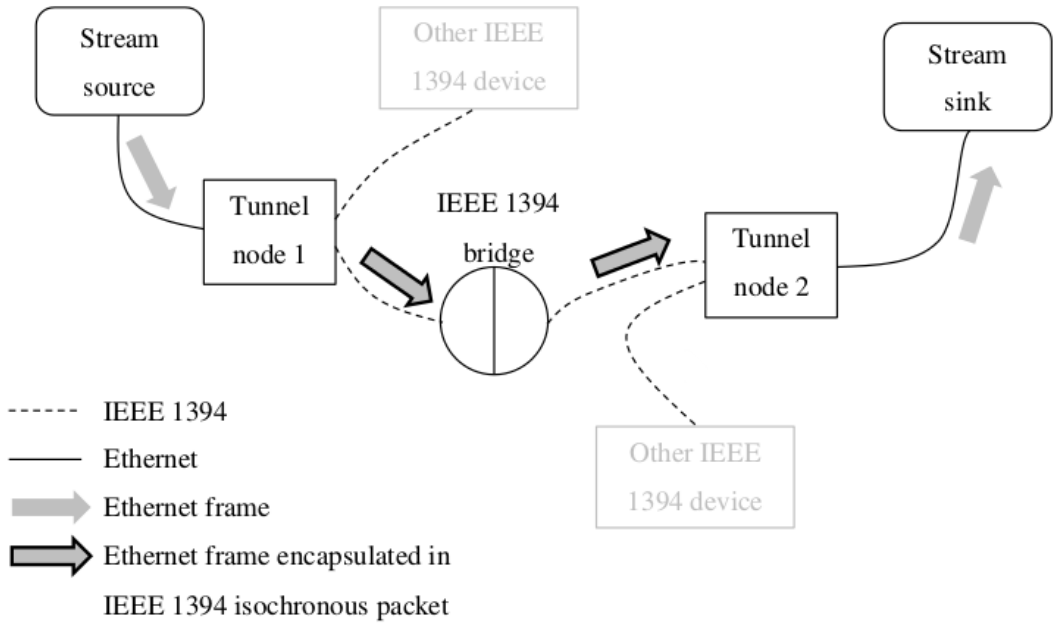


Figure 2.5: Tunneling audio across IEEE 1394 and Ethernet AVB networks [1, pp. 200]

The tunneling nodes (*'Tunnel node 1'* and *'Tunnel node 2'*) in Figure 2.5, each have two network interfaces. *'Tunnel node 1'* has one of its interfaces connected to the *'Stream*

source' (Ethernet AVB) device, and its second interface connected to the *'IEEE 1394 bridge'*. *'Tunnel node 2'* has an interface connected to the *'IEEE 1394 bridge'*, and a second interface connected to the *'Stream sink'* Ethernet AVB device. Besides the tunneling nodes and IEEE 1394 bridge, there are two other IEEE 1394 nodes on the serial bus.

The bandwidth of an IEEE 1394 network determines the maximum size of packets that can be transmitted on the bus. It is possible that a tunneling node receives more Ethernet frames (containing audio data) than it can fit into a single isochronous packet for onward transmission on the IEEE 1394 network. In order to overcome such restrictions, a transmitting tunneling node is able to pack multiple Ethernet frames into a single isochronous packet, if the isochronous packets are larger than the Ethernet frames. Also, the receiving tunneling node is able to unpack and retrieve the Ethernet frames. If the Ethernet frames are larger than an isochronous packet, the transmitting tunneling node is able to fragment the Ethernet frame and transmit the fragments across to the receiving tunneling node. The receiving node is capable of combining the fragments to form the original Ethernet frame.

By utilizing the tunneling nodes, audio data can be transmitted from an IEEE 1394 network to an Ethernet AVB network, and vice versa.

2.2.3 Layer 3 Audio Networking Technologies

A number of audio networking technologies implement audio transport at layer 3 of the OSI/ISO model. Some of these protocols are described in this section. In particular, the following protocols are described:

- Q-LAN
- Ravenna
- Livewire
- Dante

There are some similarities among the above layer 3 audio networking technologies, such at the use of *Internet Group Management Protocol (IGMP)* which enables devices to subscribe to an audio stream, and the use of *Differentiated Services (DiffServ)* to prioritize audio data over other data on the network [61] [62].

The following subsections provides an overview of each of the above audio networking technologies. In particular emphasis is on how these technologies allocate network

resources, the mechanisms they use for device synchronization, and information about the expected network latency.

2.2.3.1 Q-LAN

Q-LAN is a high-resolution low-latency audio distribution technology. It allows for audio and control transport on gigabit (or higher rates) Ethernet infrastructure [7]. Q-LAN is a proprietary solution of QSC that was developed as the audio networking technology for *Q-Sys* [63]. Q-Sys is an integrated system platform for media distribution between QSC audio products [64]. It (Q-LAN) allows for audio distribution between the Q-Sys components, namely*:*

- *Core*,
- *I/O frames,* and
- *User control interfaces.*

The Q-Sys *Core* is a device that is dedicated to audio processing on the Q-LAN network. The audio input and output devices on the Q-Sys network are referred to as *I/O frames*. The *user control interfaces* are the user interfaces for signal monitoring and control. The *Q-Sys designer*, which is software that runs on a PC workstation, fulfills the role of *user control interface*. Any number of these components may be present in a Q-LAN installation.

In order to deal with network failure, Q-LAN makes use of two Ethernet connections between networked devices. Failure on one transmission path causes a seamless routing of the audio through the other path.

Audio is transmitted within stream packets on a Q-LAN network. These stream packets are unidirectional from a transmitter to a receiver, and they are encapsulated within User Datagram Protocol (UDP) frames [65]. The transmission rate of the stream packets is 3000 per second. Each stream connection to an 'I/O device' contains a maximum of 16 channels (in each direction) of 32-bit (floating point formatted) audio. Usually the Q-LAN cores have a maximum capacity of 128 channels in each direction, although up to 512 channels (per direction) can be transmitted with each audio stream containing 8 or more channels [66]. After every 100 stream packets, the receiver sends an acknowledgement to the transmitting device [7]. The UDP datagrams (containing the audio stream packets) are encapsulated within IP packets and unicast to a receiver [67]. Thus in order to route the same audio signals to multiple receivers, separate audio streams will be unicast to each receiving device.

Devices on a Q-LAN network implement the Q-Sys network control protocol, which is an IP-based protocol for device control, monitoring and signal routing.

Q-LAN resource allocation

Q-LAN can be deployed on existing (end-to-end) gigabit Ethernet infrastructure. It is able to ensure the necessary quality of service (QoS) for time-sensitive data transmission, by prioritizing Q-LAN packets over other traffic on the network. It utilizes the Differentiated Services (DiffServ) technique for QoS, and operates using three priority classes [62]. The traffic classes used by Q-LAN are:

- *Expedited Forwarding (EF)* - used for clock transmission
- *Assured Forwarding (AF41)* - used for audio transmission
- *Default classification (0)* - used for control message transmission

Each switch on the Q-LAN network implements a minimum of four queues for each egress (outward) port, in order to handle the network traffic. The transmission selection mechanism used by these switches is the *strict priority selection*, which ensures that high priority traffic is transmitted before the lower priority traffic.

Q-LAN device synchronization

Q-LAN utilizes *Precision Time Protocol (PTP)* for synchronizing the local clocks of each networked device [52]. The devices on the network are synchronized to a common clock of a dedicated device, known as the *grandmaster*. PTP enables the grandmaster to transmit periodic time updates on the network, which are used by the other devices to update their local clocks. A device on the network is also capable of periodically querying the grandmaster for its time.

Q-LAN network latency

Q-LAN imposes restrictions on the number of hops between a transmitting and a receiving device, based on their distance apart. The further away the devices, the fewer the number of hops that are permitted.

The latency on a gigabit Ethernet infrastructure is less than 12 microseconds, and network switches are required to have a latency that is less than 10 microseconds. The total amount of time that is allowed for data transfer on the Q-LAN network is 243 microseconds [68].

2.2.3.2 RAVENNA

The Real-time Audio Video Enhanced Next-generation Network Architecture (RAVENNA) is an Internet Protocol (IP) based real-time media distribution technology [8]. RAVENNA does not define any new protocols, but rather utilizes established protocols and standards that cooperate to enable media streaming on an IP infrastructure.

It was designed as a low latency data transport for high performance audio applications, where tight synchronization and network reliability are required. A RAVENNA network is scalable with the IP infrastructure on which it is deployed, although RAVENNA is IP infrastructure-agnostic [51]. Traditional IP network traffic can co-exist with RAVENNA

If redundancy is required, RAVENNA provides fault tolerance by transmitting the same stream data (with the same time stamp) across separate routes on the network [51]. Both multicast and unicast messaging are permitted on RAVENNA networks.

RAVENNA uses the *Real-time Transport Protocol (RTP)* for streaming media on the network [69]. This enables the media streams to be available to a wide range of applications that support RTP. The media streams are encapsulated within *Real-time Transport Protocol/Audio Video Profile (RTP/AVP)* payload, and are transported within *User Datagram Protocol/Internet Protocol (UDP/IP)* packets [70] [71] [67]. RAVENNA supports 16-bit (*'L16'*) and 24-bit (*'L24'*) digital audio stream formats which are defined in RFC 3551 and RFC 3190, respectively [70] [72].

RAVENNA implements a tight synchronization scheme, and for interoperability with networked devices this synchronization scheme must be implemented by the networked nodes.

RAVENNA uses *Real Time Streaming Protocol (RTSP)* to remotely configure audio stream connections between networked RAVENNA nodes [73]. It also recommends that RAVENNA nodes implement *Hypertext Transfer Protocol (HTTP)* web servers, which allow for remote device configuration and control [74].

RAVENNA resource allocation

RAVENNA uses *Differentiated Services (DiffServ)* to ensure quality of service (QoS) for media streams on a RAVENNA network [62]. This ensures that RAVENNA packets are prioritized over other traffic on the network, thus reducing jitter at receiving nodes and packet loss when there is traffic congestion on the network.

RAVENNA device synchronization

On a RAVENNA network, a *grandmaster* node provides the *master clock* that is used to synchronize every other node on the network. Each RAVENNA node maintains a local clock that is tightly synchronized with the master clock. The master clock can be obtained from either a dedicated clock source such as *Global Positioning System (GPS)*, or a RAVENNA node that assumes the role of grandmaster. When selecting a RAVENNA node to assume the role of grandmaster, RAVENNA uses the *Best Master Clock Algorithm (BMCA)*, which is defined by the Precision Time Protocol (PTP) [52].

RAVENNA utilizes *Precision Time Protocol Version 2 (PTPv2)* for clock distribution on a network [52]. The local clock on each networked node is synchronized with a master clock that is transmitted using PTP. The accuracy of the local clock with respect to the grandmaster's clock is approximately 100 nanoseconds given that there is sufficient support from the network infrastructure.

The master clock is transported within PTP packets. When a node receives a PTP packet, it synchronizes its local clock with the master clock, thus maintaining a tight 'sync' with the grandmaster's clock. The local clock on a RAVENNA node is used to synchronize its media clock. The media clock is used to sample audio that arrives at the analog audio input of the node. Each sample is time stamped using the local clock and stored in a buffer. When the appropriate number of samples have been received, RTP packets are used to transmit the audio samples on the network [69]. On a receiving node, the RTP packets are de-packetized then stored in the receive buffer until their play-out (presentation) time. The play-out time is determined by the local clock on the receiving node.

RAVENNA network latency

The network latency on a RAVENNA network varies, depending on the network infrastructure (that is network bandwidth, capacity of switches, and number of hops), as well as on the number of samples and channels being transmitted. A minimum network latency of approximately 1 millisecond can be expected on a RAVENNA network [51].

2.2.3.3 Livewire

Livewire is a proprietary standards based Internet Protocol (IP) technology that utilizes standard Ethernet infrastructure to provide low latency audio distribution for high

performance applications. Livewire traffic can coexist with traditional Ethernet traffic, including traffic from Voice over IP (VoIP), file transfers, and emails [75].

There are two types of devices that implement Livewire. These are [76]:

- *Hardware nodes* - physical devices that implement Livewire, for example *Axia AES/EBU* digital audio node [77] and *Axia Router Selector Node* [78].
- *PC nodes* - PC workstations that incorporate the Livewire software driver. The software driver appears like a hardware interface on the PC (workstation), while actually streaming audio via the workstation's Ethernet interface.

A Livewire device that has audio streams to offer, advertises its audio sources with a text name (maximum of 24 characters) and an associated numeric identifier on the network. Devices capable of receiving the audio, build up a list of available sources on the network [76].

There are two types of streams that are transmitted by Livewire nodes. These are [9]:

- *Livestreams* - which are small packets that are transmitted frequently, and are optimized for low-latency transmission. Livestream packets have an audio data size of 72 bytes which contains 12 samples of 24-bit audio at a sampling rate of 48kHz. Livewire *hardware nodes* are able to receive the livestream packets on the network, but *PC nodes* are unable to receive these packets because of their fast transmission speeds.
- *Standard streams* - which are much bigger packets in comparison with livestreams. They are also intended for real time transmission. The audio data size of a standard stream packet is either 1440 bytes (for 240 samples of 24-bit audio at 48 kHz sampling rate) or 720 bytes (for 120 samples of 24-bit audio at 48 kHz sampling rate). Livewire utilizes the *Real-Time Protocol (RTP)* over IP to multicast standard streams on the network. Standard streams can be transmitted and received by both hardware nodes and PC nodes.

This technology is able to distribute audio data, together with control messages on the same Ethernet infrastructure. Livewire Routing Control Protocol (LWRP) is implemented by all Livewire devices, and it enables signal routing on a Livewire network. In order to allow for sophisticated device configuration and control that go beyond routing audio signals between connected devices, the *Livewire Control Protocol (LWCP)* can be implemented within the Livewire devices [75].

Livewire resource allocation

Livewire audio streams (livestreams and standard streams) can be transported on the same Ethernet network infrastructure as traditional Ethernet traffic. Hence care must be taken to ensure that the audio data is prioritized over other traffic. Livewire utilizes a quality of service (QoS) approach that involves various components of the network. The QoS approach ensures that [75]:

- the Ethernet switches are able to dedicate an entire port to a single node.
- there are full-duplex collision free Ethernet links that allow the full link bandwidth to be utilized in each direction.
- audio data is prioritized over non-audio traffic on the network.

This QoS approach makes it possible to reliably deliver audio data on a network infrastructure that is shared with standard Ethernet traffic.

Livewire device synchronization

In order to synchronize the devices on a Livewire network, a device designated as the *clock master* distributes its clock at regular intervals.

Each Livewire node implements a *Phase Locked Loop (PLL)*, which is used to recover the local clock from the multicast clock of the master clock. The PLL consists of hardware and software components that work together to ensure that the differential delay is less than 5 microseconds network-wide [9].

The clock master is selected via an arbitration process that ensures that a device with the highest *clock master priority* emerges as network clock master. The clock master priority is in the range ‘0’ to ‘7’. During the clock master arbitration process each competing device multicasts its priority. A clock master priority of ‘0’ indicates that the device should never be assigned the responsibility of clock master, while a ‘7’ indicates that the device should always be designated as the network clock master. Typically a device’s manufacturer configures the default clock master priority to ‘3’.

Livewire network latency

A network latency of 0.75 millisecond can be expected from a Livewire network [79].

2.2.3.4 Dante

Dante is a proprietary media transport technology that was developed by Audinate in 2006 [80]. It allows for low-latency, multi-channel digital audio streaming over Ethernet networks [10]. By utilizing existing Internet Protocol (IP) standards, Dante is able to ensure high-performance audio distribution on networks that transport other Ethernet traffic such as file transfers and emails.

Dante transports control data and high-bandwidth digital audio data, on existing (100Mbps and 1Gbps) Ethernet network infrastructures. The audio data is transported within *User Datagram Protocol (UDP)/ IP* packets [65] [67] [10]. Typically audio streams are unicast from a transmitter to a receiver on a Dante network. However a multicast transmission mode where a transmitter sends data to multiple receivers, is permitted [81].

As many as 1024 bi-directional channels of 24-bit audio sampled at a rate of 48kHz can be transported on a gigabit Ethernet Dante network. If the Ethernet network bandwidth is 100Mbps, Dante allows for up to 96 bi-directional 24-bit audio channels at 48kHz sampling rate [80].

An audio stream on a Dante network is identified by its unique (descriptive) label. The use of descriptive labels allows streams to be easily identified on a control application user interface. A label can be remotely modified, at anytime, by a controller on the network. The stream labels persist even after powering off and restarting the device.

With regard to control workstations, Dante does not require additional hardware. The *Dante Virtual Soundcard* is software that enables a workstation (Windows or Macintosh) to transmit and receive multiple channels of audio via a standard Ethernet interface [82].

Dante resource allocation

Dante implements a quality of service (QoS) approach that allows audio streams to be reliably transmitted on a standard Ethernet infrastructure. *Differentiated Services (DiffServ)* are used to prioritize audio data over IP packets on the network [62] [80].

Dante device synchronization

The Dante audio networking technology utilizes *Precision Time Protocol (PTP)* for clock synchronization [52]. PTP enables the clock of a dedicated clock source device (known as the *grandmaster*) to be distributed to the other devices on the network. The PTP mechanism has been described in section 2.2.4.2.

Dante network latency

Network latency as low as 150 microseconds can be achieved between devices directly connected to each other [81]. On a network with gigabit Ethernet switches (end-to-end), latency of 1 millisecond is achieved when the number of hops is 10 [83].

2.2.4 Review of Layer 3 Audio Networking Technologies

Some of the notable features of each of the layer 3 technologies for audio transport, are highlighted below.

- Q-LAN
  - Is a proprietary transport technology.
  - Distributes audio data over 1000Mbps Ethernet link.
  - Each I/O device has a maximum of 16 input channels and 16 output channels.
  - Each Core device has a maximum of 512 input channels and 512 output channels.
  - Transports 32-bit floating point audio data at 48kHz sampling rate.
  - Prioritizes audio data over non-audio data using the *DiffServ* protocol.
  - Synchronizes networked devices using *PTP*.
  - Network latency is 243 microseconds.
- RAVENNA
  - Is an open standard transport technology.
  - Allows audio data and standard Ethernet traffic to coexist.
  - Uses *RTP* for audio transport.
  - Transports 16-bit uncompressed audio.
  - Transports 24-bit linear encoded audio.
  - Prioritizes audio data over non-audio data using the *DiffServ* protocol.
  - Synchronizes networked devices using *PTP*.
  - Ensures less than 1 millisecond network latency.

- Livewire
  - Is a proprietary audio transport technology.
  - Distributes audio data over 100Mbps or 1000Mbps Ethernet links.
  - Allows audio data and standard Ethernet traffic to coexist.
  - Transports 24-bit audio at a sampling rate of 48kHz.
  - Prioritizes audio data over non-audio data in order to guarantee QoS.
  - Clock master distributes its clock information at regular intervals in order to synchronize the clocks of the other devices on the network.
  - Network latency is 0.75 millisecond.
- Dante
  - Is a proprietary audio transport technology.
  - Distributes audio data over 100Mbps or 1000Mbps Ethernet links.
  - Allows audio data and standard Ethernet traffic to coexist.
  - Transports 24-bit audio at a sampling rate of 48kHz.
  - Transports a maximum of 1024 bi-directional channels on 1000Mbps (gigabit) Ethernet links.
  - Transports a maximum of 96 bi-directional channels on 100Mbps Ethernet links.
  - Prioritizes audio data over non-audio data using the *DiffServ* protocol.
  - Synchronizes networked devices using *PTP*.
  - Network latency of 150 microseconds on point-to-point connections.
  - Network latency less than 1 millisecond when data is transported over ten hops.

There are some similarities between the above layer 3 audio networking technologies. These include:

- The use of Ethernet links for distributing multiple channels of audio data.
- The use of standard Ethernet infrastructures (including network switches).
- Time sensitive (audio) data is prioritized over other standard Ethernet traffic, such as e-mail traffic.

- With the exception of Livewire, a common approach for device synchronization.

In spite of these similarities, the different audio sampling rates, audio formats, device synchronization schemes (for example Livewire), and number of channels within an audio stream, makes it difficult to attain interoperability between the different layer 3 audio transport technologies.

2.2.4.1 Interoperability on layer 3 networks

Interoperability remains a challenge between the various layer 3 IP-based audio transport technologies. In order to solve this problem, the AES-X192 project was initiated within the AES. This project aims to provide a set of recommendations that will provide the necessary QoS for high performance audio streaming over IP-based networks.

By complying with a standard, manufacturers are guaranteed that their devices can exchange audio data with compliant devices from different manufacturers. Hence a number of promoters of different audio networking technologies are actively participating in the development of a standard within the X192 project. This standard is described in the following subsection 2.2.4.2.

2.2.4.2 AES-X192 for Layer 3 Interoperability

The AES-X192 project was initiated in 2010 by the Audio Engineering Society (AES) in an effort to develop a standard for audio interoperability over high performance IP networks [84]. The AES-X192 project investigates the various Internet Protocol (IP) based networking technologies that are deployed in audio networks, in an effort to produce a recommendation that would be published as an AES standards document. This recommendation should describe how interoperability can be achieved between networked audio devices by utilizing existing IP-based network standards.

The AES-X192 document is in the draft stage, but has started to address the following concerns within IP-based audio networks [11]:

- *Synchronization* - the mechanism by which a common time base is distributed to devices on the network. When multiple receivers are synchronized to the same clock, they are able to play back audio (that might have arrived through different network paths from the transmitting device) at the same time.
- *Media clock management* - is concerned with the relationship between a media clock and the (common) time base on the network. The media clock is used by

a transmitting device to sample audio, and by a receiving device to play back the audio streams.

- *Transport* - the technique used for end-to-end transmission of audio and control data on the network. It includes flow control, ordering, and time stamping of packets in high performance audio networks.
- *Encoding* - is concerned with the packetization of audio data, which are transmitted as *streams* on a digital audio network. The encoding of audio data is concerned with payload formats, sample rates, and stream channel counts for the digitized audio data. This ensures that the audio that is transmitted can be reproduced at the receiver(s), irrespective of the manufacturer of the transmitting and receiving devices.
- *Stream description* - is concerned with the format used to describe critical information about an audio stream. Such information includes the stream source information, network address, and encoding format. This information is required for discovery and connection management.
- *Discovery* - the mechanism for determining the available devices and services offered on the network. Networked devices should be able to obtain a list of available devices and services, and should be notified when a device or service is no longer available.
- *Connection management* - the procedure for establishing and destroying audio stream connections between transmitting and receiving devices.

AES-X192 recommends the use of the *Real-time Transport Protocol (RTP)* for audio data transport [69]. It recommends that audio is encoded as 16-bit uncompressed linear audio data samples (defined in RFC 3551 [70]) at 48kHz sampling rate, or 24-bit uncompressed linear audio data samples (defined in RFC 3190 [72]) at a sampling rate of 48kHz. Another requirement is that AES-X192 compliant audio devices should be able to receive audio streams that contain from 1 to 8 channels of audio.

AES-X192 is currently at an advanced stage of development. It promises interoperability between devices that comply with the standard, and quality of service for real-time audio transmission on IP-based networks.

AES-X192 resource allocation

AES-X192 seeks to ensure that an IP network provides the quality of service (QoS) for the transmission of real-time audio. It recommends that audio streams are transmitted

within *IP version 4 (IPv4)* packets [67]. These packets may be multicast or unicast from transmitter to receivers, but networked devices that conform to AES-X192 are required to support *IGMP version 2 (IGMPv2)* or *IGMP version 3 (IGMPv3)* [85] [61].

The QoS strategy ensures that audio data, which is time-critical, is prioritized over non-time-critical traffic. The current AES-X192 documentation suggests that AES-X192 networks will utilize *Differentiated Services (DiffServ)* technique as defined in RFC 2474 to ensure QoS [62]. DiffServ prioritizes IP packets based on the value of the *Differentiated Services Code Point (DSCP)* field, which determines the traffic class of an IP packet. AES-X192 defines three traffic classes, namely:

- *Clock* - used for PTP (clock) messaging and its value is '46'.
- *Media* - used for transmitting media streams and its value is '34'.
- *Best effort* - used for control and status update messaging such as messages used for discovery and connection management. Its value is '0'.

These values are set in the DSCP field within an IP packet. The higher the DSCP value, the more urgent the packet is considered to be. Standard IP traffic such as email, and messaging have a DSCP value of '0'.

AES-X192 device synchronization

Synchronization between networked devices enables multiple receivers of an audio stream to playback the audio at the same time, and at the same rate. The AES-X192 project recommends the use of *Precision Time Protocol (PTP)* defined in IEEE 1588-2008 [52]. PTP provides a means for a common clock to be distributed between *PTP-aware* nodes on a network.

A node on the network, known as the *grandmaster*, distributes its clock to the other nodes. The grandmaster is determined by a *Best Master Clock (BMC)* algorithm, which results in a master/slave hierarchy being established between interconnected nodes. The grandmaster forms the root of the hierarchy and the leaf nodes are slaves. Each master node propagates a PTP message (containing time information) to its slave node. A bridge in the path between grandmaster and leaf nodes, could be either a *Boundary Clock (BC)* or a *Transparent Clock (TC)* [86].

A *boundary clock* is a PTP-aware bridge that receives PTP clock signals on one of its ports, which is a slave to a port on another node. Its other ports are slaves to this port, such that PTP messages received on the port are propagated to the other (slave) ports.

A *transparent clock* (TC) is a PTP-aware bridge that is capable of calculating the time a PTP message has spent in the bridge. This time is called the *residence time*. The TC adds the residence time to the time information within the PTP message before sending it off to its slaves. Figure 2.6 depicts a PTP-aware network.

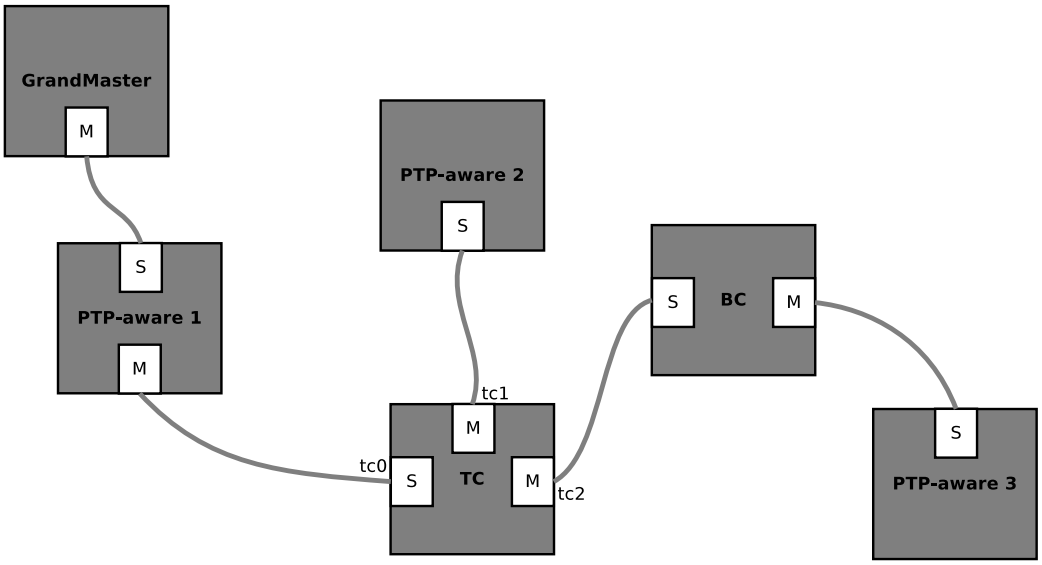


Figure 2.6: PTP-aware network

There are six PTP-aware nodes in the network shown in Figure 2.6. The '*GrandMaster'* is the source of the clock information used to synchronize the networked nodes. A synchronization hierarchy is formed, such that the '*GrandMaster*' clock is received by '*PTP-aware 1*', which is a PTP-aware node and slave to '*GrandMaster'*. The clock of '*PTP-aware 1*' is master to the PTP-aware bridge which has a transparent clock ('*TC'*). '*TC'* clock is in turn the master to the clocks on '*PTP-aware 2*' and boundary clock (*'BC'*). '*tc0'* is the clock information received by *'TC'*. '*tc1'* and '*tc2'* are the clocks that are propagated to '*PTP-aware 2*' and *'BC'*, respectively. The values of '*tc1'* and '*tc2'* includes the residence time on *'TC'*. '*TC'* does not need to be synchronized, that is it should be 'transparent'. However, it is required that '*TC'* is able to determine the residence time, and that it adds the residence time to the clock information that it distributes. When 'BC' receives clock information, it synchronizes its local clock (like any of the other PTP-aware nodes on the network), then it propagates clock information onto its clock slave ('*PTP-aware 3'*).

PTP allows for different modes of operation, clock types, parameter values, attributes and options that result in interoperability becoming a challenge [87]. As a result a particular application context is allowed to define a PTP *profile* which it can utilize. A PTP profile defines a constrained set of operations, attributes, options and modes of operation that can be utilized within a defined context.

In order to enable interoperability between AES-X192 compliant nodes, AES-X192 defines a *media (PTP) profile* that can be used by standard IP nodes that do not have dedicated hardware for time-stamping. It also allows for the Ethernet AVB synchronization protocol (that is the IEEE 802.1AS standard) to be used for high-performance IP networks. A heterogeneous synchronization scheme that will allow a common clock to be shared between nodes that utilize the media profile and those that utilize IEEE 802.1AS synchronization is being developed.

AES-X192 network latency

AES-X192 aims to enable interoperability for high-performance streaming of audio. It defines a packet time latency of 1 millisecond, so that it is suitable for a wide range of audio applications [11]. Packet time latency refers to the amount of time it takes a transmitting device to packetize audio data for transmission. All AES-X192 receivers are required to be able to receive audio packets that have 1 millisecond of audio data. This is expected to allow for interoperability between a variety of audio equipment.

2.3 Audio Networking Technology Interoperability

In section 2.2.2.1, a tunneling approach for interoperability between two layer 2 audio networking technologies, was described. The tunneling approach involved the active participation of a tunneling node, which implements the two audio networking technologies. In section 2.2.4.1, the AES-X192 project was described as a consolidated effort by different audio manufacturers to provide a common set of recommendations that guarantee interoperability on IP-based networks.

However, device control is necessary to ensure that the networked audio devices can be remotely configured when establishing or destroying audio stream connections. Typically when devices are connected on a network, they require user intervention to select which stream(s) should be transmitted on the network, and which stream(s) should be received from the network. In order to allow for this, each device implements an audio control protocol that allows for local and remote device configuration. Also, an audio control protocol ensures that a network controller is able to discover all compliant nodes on the network.

Even if there is network transport interoperability between networked audio devices, each device needs to observe a common control protocol for stream communication. This will be the focus of subsequent chapters.

2.4 Summary

Digital networking technologies provide solutions for interconnecting audio devices. The transmission of audio between networked devices requires that the networking technology is capable of providing the necessary quality of service. This includes being able to guarantee that:

- there are sufficient resources on the network,
- the networked devices are synchronized,
- the network latency is kept to a minimum.

There are a number of audio networking technologies that meet these requirements. Some of the audio networking technologies implement low-latency real-time audio transport on OSI/ISO layer 2. Others implement IP-based high-resolution audio transport on layer 3. In this chapter, these two approaches were described in an effort to give an overview of some of the available audio networking technologies.

The existence of different networking technologies, each defining its own audio sampling rate, audio format, resource allocation scheme, transaction type for audio data transmission, and device synchronization mechanism, makes it difficult for devices that utilize different transport technologies to exchange audio data. Hence interoperability does not exist between devices that implement different audio transport technologies.

Two approaches to audio network interoperability were described in this chapter. One approach involved the extraction and encapsulation of audio data into different packet structures, as the audio is transported from one layer 2 networking technology to another. The second approach is a united effort by promoters of different technologies to standardize the techniques used for audio transport on layer 3 networks. Irrespective of the approach used, an audio device requires a control protocol that will enable it to be configured. Typically, an audio transport technology will allow audio data and control commands to be transported on the same network.

Chapter 3

Audio Network Control Protocols

In the previous chapter, an audio networking technology was described as a means of interconnecting nodes such that audio from a source node can be transported to one or more destination nodes. In order for a networking technology to be able to assume the role of transport protocol for real-time audio transmission, it should be able to ensure deterministic and guaranteed delivery of audio streams without degradation in the audio quality.

Because audio devices are networked as nodes on an audio networking technology, control and monitoring is possible from a remote device. This chapter provides information about some of the available audio control protocols, which make it possible to remotely monitor and adjust parameters of remote audio devices.

3.1 Audio Control Protocols

When devices are networked, it is desirable to be able to remotely control multiple devices that might be some distance from the controller. An audio control protocol enables networked audio devices to interpret the instructions that are addressed to them. The transport protocol (audio networking technology) is used by the audio control protocol to exchange messages between compliant devices on a network. However an audio control protocol is not concerned with how the actual audio streams are transmitted, but rather focuses on how (control and monitoring) messages are exchanged between networked devices.

An audio control protocol will typically define:

- a message packet layout,

- the meaning of the information exchanged between compliant devices,
- the layout of controllable parameters within the device,
- a device discovery mechanism,
- a procedure for connection management.

At present, there are a variety of audio control protocols that are being deployed in different contexts. These protocols include those used in the broadcast industry, hospitality industry, recording and post-production industries, automotive industry, and in the entertainment industry. In these cases, any number of networked audio devices can be controlled from a single control booth. In some cases, such as in the hospitality and tourism industry, multiple (and perhaps mobile hand-held) devices can be used to configure and monitor the networked devices. An audio control protocol should be able to meet these requirements.

Typically, a commercially available audio device implements only one control protocol. Hence the manner in which it can be remotely configured is determined by its control protocol.

An audio control protocol provides a service for a specific application implementation, such that it receives and transmits messages on behalf of the application. This is represented in the form of a diagram in Figure 3.1.

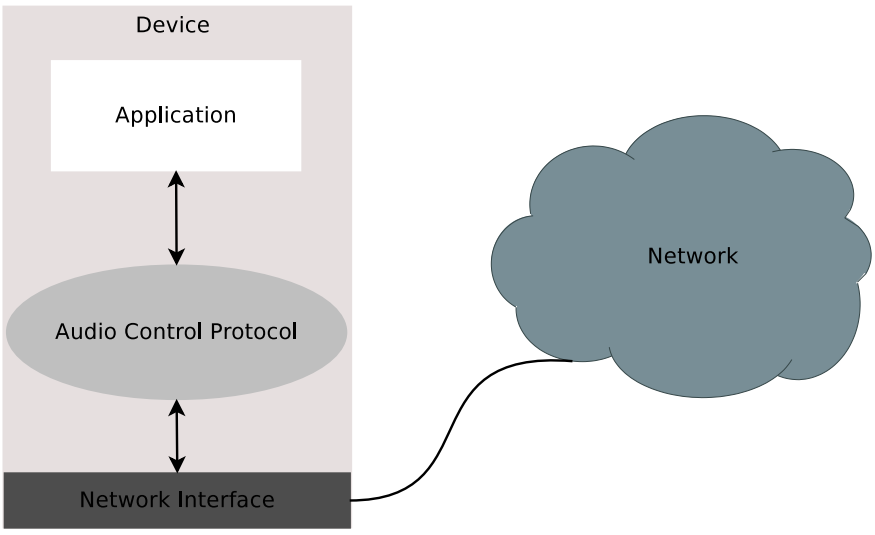


Figure 3.1: Audio control protocol interacts with application

An audio networked device is typically designed in the manner depicted in the above figure. The *'Application'* depicted in Figure 3.1 could be a *networked device controller* that has a:

- graphical interface component, which is used to present network devices (and their attributes) to a user, and
- control component, which implements the logic for controlling the networked devices.

The '*Audio Control Protocol*' receives all messages addressed to the device via its '*Network Interface*'. It gives meaning to a received message and passes it on to the '*Application*'. It also receives instructions from the '*Application*', and creates the appropriate commands for onward transmission to the network.

The audio control protocols that are described in this chapter have been classified into two categories, based on which of the OSI/ISO 7 layers a particular audio control protocol uses for transporting its messages [26]. In this regard, some of the audio control protocols described in this chapter utilize the OSI/ISO *layer 3* (*"Network layer"*) audio transport protocols, and they are typically IP-based. Others utilize the OSI/ISO *layer 2* (*"Data link layer"*) audio transport protocols. These two categories of audio transport protocols have been described in chapter 2.

In section 3.2 a number of layer 3 dependent audio control protocols, are described. Section 3.3 provides a description of some of the available layer 2 dependent audio control protocols.

3.2 Overview of Layer 3 Audio Control Protocols

Some audio control protocols utilize existing OSI/ISO layer 3 protocols for messaging. Such protocols have been classified in this chapter as layer 3 audio control protocols. These include the following protocols:

- Open Sound Control (OSC)
- Architecture for Control Networks (ACN)
- Common control interface for networked audio and video equipment (IEC 62379)
- Audio Engineering Society standard for audio applications of networks - Command, control, and connection management for integrated media (AES-64)
- Open Control Architecture (OCA)

These layer 3 audio control protocols are described in the following subsections.

3.2.1 Open Sound Control (OSC)

Open Sound Control (OSC) is an open specification that was developed within *The Center for New Music and Audio Technologies (CNMAT)* at the University of California Berkeley in 1997 [88]. It specifies a digital media content format for real-time media control messages [14]. Although OSC was initially intended for audio applications, it has been used in such fields as show control, gesture recognition, and robotics.

OSC is transport-independent, and it is concerned with how control information is distributed between networked devices. It has been published as an open specification which developers can utilize, and it does not define a certification process or any royalty for adapting the technology. Hence it has a wide range of applications which include [89]:

- *Sensor/gesture-based electronic musical instruments* - where sensor(s) are used to capture physical activities (such as motion, acceleration, pressure, displacement, flexion, key presses, and switch closures) from a performer. The captured data from the performer are then processed (by mapping them) in real-time to controls on an electronic device [90].
- *Multiple-user shared musical control* - where a number of users interact with an interface in order to control a shared sonic environment in real-time [91].
- *Web interfaces* - where a server that implements OSC is used to send real-time OSC messages to other OSC devices [92].
- *Networked LAN Musical Performance* - where a group of people in a musical collaboration make use of networked computers that are able to remotely control parameters on other computers [93].
- *WAN performance and Telepresence* - where a number of musicians who are located at large physical distances apart, are able to collaborate in a musical production [94].

The OSC specification describes communication between networked devices as conforming to the client/server architecture. It defines two roles between communicating devices, namely [95]:

- *OSC client* - an OSC compliant device that sends a request.
- *OSC server* - an OSC compliant device that processes a request.

It is possible for an OSC device to have both capabilities, that is OSC client and OSC server. An example of this is an OSC streaming device (which has been implemented on a PC) that is capable of discovering other OSC devices on the network, and requesting them to specify their manufacturer and model. The streaming device can also respond to requests to set up its stream connections. Such a device is described in chapter 5.

3.2.1.1 OSC messaging

Communication between OSC clients and OSC servers on a network is performed by exchanging *OSC packets*. An OSC packet consists of:

- a size field, which specifies the number of 8-bit bytes that make up the packet,
- a content field, which could be either an *OSC message* or an *OSC bundle.*

The size of an OSC packet is always a multiple of 4. The first eight bits of an OSC packet's content indicates whether the packet is an OSC message or an OSC bundle [96].

An *OSC message* consists of:

- An *OSC address pattern* - this is a URL-style addressing scheme which is comprised of an *OSC string* that begins with the '/' (forward slash) character. An *OSC string* is a sequence of non-null ASCII character strings that is terminated with a null character, followed by null-character padding to ensure that the number of bits is a multiple of 32.
- An *OSC type tag string* - is a sequence of characters that represent the arguments in the message, and they appear in the order in which the arguments occur. This character sequence (OSC type tag string) starts with a ',' (comma) character, and is followed by any number of OSC defined argument types. The basic OSC argument types are:
  - 32-bit integer denoted with the character 'i'
  - 32-bit float denoted with the character 'f'
  - *OSC string* denoted with the character 's'
  - *OSC-blob* denoted with the character 'b'.

  An OSC server that receives a message containing unfamiliar characters will ignore the message.

- *OSC argument(s)* - any number of arguments may be included in an OSC message. However, the order in which the arguments appear must match the order in which the OSC type tag string is encoded.

An example of the structure of an OSC message is provided in section 3.4.1.

An *OSC bundle* consists of:

- An *OSC string* - which has a value of *'#bundle'*, and is used to identify the OSC packet's content as an OSC bundle.
- An *OSC time tag* - a 64-bit fixed point number that specifies when the *OSC bundle element* (which is described below) should be processed. The most significant 32-bits (of the OSC time tag) represents the number of seconds since *epoch*. The epoch used for this field is January 1, 1900. The least significant 32-bits indicates fractional parts of a second to a precision of 200 picoseconds.
- An *OSC bundle element* - which consists of:
  - *size* - a 32-bit integer that specifies the number of 8-bit bytes that are in the *contents*, and is a multiple of 4.
  - *contents* - are either OSC messages or OSC bundles. It can also consist of both OSC messages and OSC bundles.

Any number of OSC messages and/or OSC bundles can appear as the OSC bundle element. An OSC server processes the OSC messages and/or OSC bundles in the same order as they appear. The OSC bundle element is processed as a single atomic transaction. This means that if any of the contents (OSC messages and/or OSC bundles) within the OSC bundle element fails (cannot be processed), the entire transaction will fail as well. The OSC protocol is further described in section 3.4.1.

3.2.2 Architecture for Control Networks (ACN)

The Architecture for Control Networks (ACN) is a collection of protocols and languages that enable the creation of networked control systems. It was developed by the *Entertainment Services and Technology Association (ESTA)* together with the *Professional Lighting and Sound Association (PLASA)* [97]. ACN was initially intended for use as a lighting system control protocol that enables interoperability between networked devices. However, it was developed as a reliable control data distribution technology for

networked devices in lighting, entertainment technologies and other control networks [98].

ACN specifies a number of independent protocols that can be combined as modular units in order to create a networked system. To ensure consistency and interoperability, ACN specifies the use of *interoperability profiles* which describe how the various modules (ACN protocols) should be combined in a particular application context.

Communication in an ACN network occurs between ACN *components*. An ACN component is an addressable ACN entity (functional unit or endpoint), which could be a program or application, that is capable of receiving and transmitting ACN data. Each component is identified by its *Component Identifier (CID)* [15]. The CID is a 128-bit globally unique *Universally Unique Identifier (UUID)*, and is used by a component for its entire existence [99].

Within an ACN component are a number of controllable units that are known as the component's *properties*. These properties model specific functionalities within a component that can be remotely accessed and modified. ACN specifies the protocols that enable a remote controller to enumerate, monitor and control the properties within an ACN component. These (modular) protocols include [15]:

- *Device Management Protocol (DMP)* [100] - defines the addressing structure necessary for identifying individual properties within a component, and the messages that can be used to efficiently manipulate these properties. The DMP also provides a mechanism that will enable a component to announce changes to the values of its properties as they occur.

- *Device Description Language (DDL)* [101] - is a language for describing *DDL devices* in a manner that will enable DDL *controllers* to interface with them. A *DDL device* refers to an entity that can be monitored and configured remotely. The DDL provides an interface with which a *DDL controller* is able to enumerate the device. Devices that can be described with the same DDL description are said to belong to the same *device class*. Each device class is uniquely identified by its *Device Class Identifier (DCID)*, which is a UUID.

- *Session Data Transport (SDT)* [102] - is used by ACN client protocols (such as DMP) to transport messages (within sessions) on the network. It ensures that the standard network infrastructure is efficiently utilized by 'bundling' multiple client protocol messages (which are typically very small) and transmitting them to a group of components that receive from the same session. A session ensures

ordered delivery of messages through a bi-directional transport connection between a *leader* and one or more *members*. Typically a majority of the traffic in a session is downstream from a *leader* to the other *members* of the session, although the members may also transmit upstream to the leader. By utilizing SDT, multiple messages can be multicast on a network with each component extracting only those messages that are addressed to it.

The unit of a message sent by a client protocol, such as DMP, is known as a *Protocol Data Unit (PDU)*. ACN defines a common PDU that encapsulates a client protocol's PDU. A lower level *Root Layer Protocol (RLP)* is defined in ACN. RLP combines PDUs from higher level protocols into packets for transmission on the network, and it transfers received packets to the appropriate protocol.

ACN utilizes *Service Location Protocol version 2 (SLPv2)* for device discovery [103]. SLP provides a means for ACN components to publish their presence on a network, and to discover other components on the network.

3.2.3 Common Control Interface for Networked Audio and Video Products (IEC 62379)

IEC 62379 is a set of standards that allow for device control and live media streaming over different networking technologies. The IEC 62379 standards were originally intended for radio broadcasting where (at the time) there was a need for audio to be reliably distributed over an *Asynchronous Transfer Mode (ATM)* network, and for remote device control on such networks [16].

Each device on an IEC 62379 network is referred to as a *unit*, and it consists of a number of functional entities called *blocks*. Typically a unit (such as a mixing console) will have a number of interconnected blocks that process an audio signal as it is routed from an input to an output. The output of one block feeds the input of another block. The IEC 62379 standard defines two special types of blocks, namely:

- *input port* - connects the other block(s) within a unit to signals from outside of the unit. It serves as an entry point for external signals to the unit.
- *output port* - is a block that is similar to the input port in that it is a link between the blocks within a unit to outside the unit. It receives signals from blocks within a unit.

The interconnection of blocks within a unit, such that signal that enters the unit from an input port is routed for processing from one block to the other, is called a *processing chain*. Typically the processing chain within a unit indicates the overall functionality of the unit [16]. Figure 3.2 shows a unit with a single processing chain.

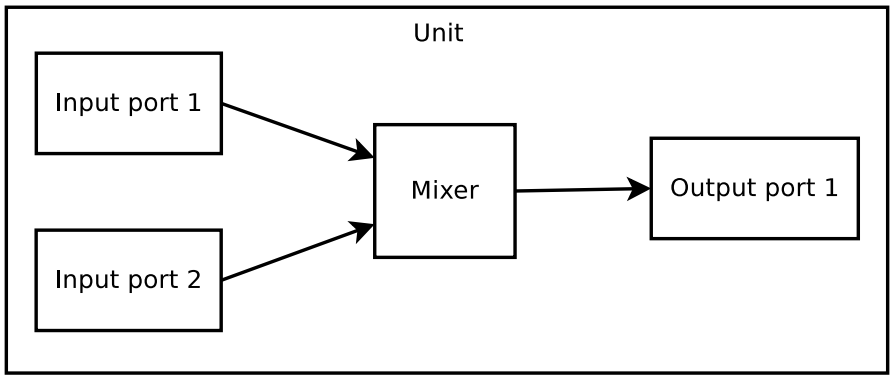


Figure 3.2: IEC 62379 unit with a single processing chain

In Figure 3.2, the '*Unit*' has four blocks. Two of the blocks are input ports ('*Input port 1*' and '*Input port 2*') and they are connected to a '*Mixer*' block, which is then connected to an output port ('*Output port 1*').

Within blocks are control points called *parameters*. The value of a parameter can be adjusted or monitored by a *management terminal* in order to modify or determine (respectively) the state of the unit. A management terminal is a device that enables configuration, monitoring and control of networked IEC 62379 compliant devices.

3.2.3.1 IEC 62379 monitoring and control

For monitoring and controlling networked devices on an IEC 62379 network, the *Simple Network Management Protocol (SNMP)* is used [56]. Each unit possesses a *Management Information Base (MIB)* that consists of a number of *'managed objects'* that are organized in a hierarchical order [104]. Each (managed) *object* is uniquely identified by its *Object Identifier (OID)*, and it (the object) represents the properties within a block.

By performing 'read' and/or 'write' operations on these objects, a management terminal is able to obtain status information about the properties within a block, as well as control those properties. A dot-notation is used to (hierarchically) address objects within a unit. For example, an audio port can be addressed by the following dot-notation [105]:

1.0.62379.2.1.1

Which means:

ISO (**1**)

Standard (**0**)

IEC 62379 (**62379**)

Audio (**2**)

Audio MIB (**1**)

Audio port (**1**)

The IEC 62379 standard restricts possible controls that are permitted by users based on four privilege levels. These privilege levels are [16]:

- *listener* - is the lowest privilege level, and allows users in this category to only possess local control of signals.
- *operator* - is a higher level than the listener privilege level, and users in this category are able to adjust controls that affect other users, such as listeners.
- *supervisor* - is a higher level than the operator privilege level. Instructions that are targeted at controls from users in this group are of higher priority than those of the operator. Such controls could affect other users.
- *maintenance* - is the highest privilege level, and is used for system troubleshooting and upgrades.

A management terminal can only manipulate the value of a parameter based on its privilege level.

In order to announce status information on a network, the IEC 62379 standard specifies that such messages should be broadcast on the network. This enables multiple management terminals to be updated with the same message.

3.2.3.2 IEC 62379 discovery

In order to monitor and control a networked IEC 62379 device, a management terminal typically has to locate and enumerate the parameters within the remote device. The discovery mechanism requires that a management terminal [16]:

- discover the units on the network and add each discovered unit to a list. Each unit is identified by a unit 64-bit address that conforms to the *64-bit Extended Unique Identifier (EUI-64)* specification [106].

- discover the blocks within each unit in its list. Each block within a unit is globally unique and identified by its *block OID*. The block OID provides some indication of the type of block it represents. A unique value called the block ID is used to identify particular blocks within a unit. A list of blocks is created that associates a block with its parent unit.

- discover connections between discovered blocks.

After devices have been discovered and their parameters enumerated, a management terminal can proceed to control them by performing 'read' and 'write' operations.

3.2.4 Audio Engineering Society standard for Command, Control and Connection Management for Integrated Media (AES-64)

The Audio Engineering Society standard for Command, Control and Connection management for Integrated Media (AES-64) is an IP-based peer-to-peer audio network protocol [17]. Being an IP-based protocol, the AES-64 protocol is transport independent and has been deployed on Ethernet and IEEE 1394 networks [107]. It was initiated by UMAN Technologies and called Cross-Fire Network (XFN) [108]. It later became part of the Audio Engineering Society's (AES) AES-X170 project. It has been ratified by the AES and published as AES-64. For the sake of consistency, it will be referred to as AES-64.

Control points within an AES-64 device are known as *parameters*. These parameters can be remotely monitored and controlled. The AES-64 specification defines a number of parameters that can be used to model a device. Each parameter has a 32-bit value associated with it, which is known as the *parameter identifier (parameter ID)*. The parameter ID uniquely identifies a parameter within a device, and can be used to address it (the parameter).

Each AES-64 device consists of:

- An AES-64 protocol stack (commonly referred to as the *XFN stack*) - which is responsible for AES-64 messaging and provides various mechanisms for device discovery, connection management, parameter subscription and notification, and maintaining parameter relationships.

- AES-64 device nodes (commonly referred to as the *XFN node*s) - which represent a particular functional entity within the XFN stack. It is possible for a device to

implement several XFN nodes. For instance an AES-64 proxy device may require multiple XFN nodes to correspond to each physical device that it proxies on the network.

- Fixed (7-level) hierarchical parameter structure - which ensures a consistent addressing scheme for each parameter within an XFN node. Typically each node consists of multiple parameters. The use of a fixed 7-level hierarchy provides a way of modeling parameters according to their functionality.

- AES-64 application - which creates the XFN nodes and corresponding parameters, and utilizes the XFN stack for communication with remote devices. An AES-64 application incorporates other functionalities that occur in response to modification of a device's parameters.

The AES-64 protocol enables complex relationships between parameters within a *group*. This grouping mechanism makes it possible to select parameters, which may reside on the same or different devices, into a collection for the purpose of establishing various relationships between them. In particular, the AES-64 protocol enables [109]:

- *peer-to-peer* relationships between parameters - a change in the value of any parameter in the group will cause a change in the value of the other parameter(s) in the group.

- *master-slave* relationships between parameters - a change in the value of a parameter that has been designated as *master* will cause the other parameters (which are *slaves*) to be modified as well. However, the value of any slave parameter may change without affecting the value of the master parameter or other corresponding slave parameters in the group.

There are two types of peer-to-peer and master-slave parameter relationships. These are:

- *absolute* relationship - in which a change in a member of the group to a new value causes the other parameters (in the relationship) to be updated to the exact same value.

- *relative* relationship - in which a change in the value of a member of the group causes the other parameters (in the relationship) to be modified by the same amount, while maintaining the offset in values that exists between them.

AES-64 includes a mechanism for parameters to subscribe to value changes on other parameters. This mechanism is called a *push* mechanism [110]. The *push* mechanism allows an AES-64 parameter to indicate to a target device, that it is interested in receiving updates whenever the value of a specified parameter changes. This causes the XFN stack within the target device to add the requester's address to its local *push list*. The *push list* is a listing of addresses to which a notification (status change) should be sent. This mechanism prevents a controller from continuously polling the value of a particular parameter, thus providing an efficient way for controllers to be updated.

3.2.4.1 AES-64 messaging

The AES-64 specification defines an AES-64 message packet layout, that is used to transfer AES-64 instructions (commands) and status reports (responses) on a network. In order to remotely address a parameter, an AES-64 controller transmits a message that specifies a [17]:

- 128-bit device ID
- 32-bit node ID
- 104-bit message address block that specifies the 7-level address of the target parameter, or the 32-bit parameter ID of the target parameter.

In order to address multiple parameters with a single AES-64 message, AES-64 defines a *'wildcard'* mechanism. When a wildcard is used at any of the 7-levels in an AES-64 message, it implies that the message is targeted at all the alternatives (implemented in the node) at that level [110].

AES-64 messages are encapsulated within UDP/IP packets, and may be unicast, multicast or broadcast packets [71] [67]. Unicast packets are used for peer-to-peer device parameter communication on the network, while multicast packets are used to address a group of devices. Broadcast packets are used when a message is intended for every device on the network.

Further details about the AES-64 protocol are provided in section 3.4.2.

3.2.5 Open Control Architecture (OCA)

The Open Control Architecture (OCA) is a control and monitoring protocol for media networks [111]. It was developed by the OCA-Alliance which was founded in 2011

[112]. It is designed to operate above various networking technologies including Ethernet AVB and Dante, and is primarily focused on device monitoring and control. The OCA is a successor of the AES24 project [113]. It is currently being ratified by the Audio Engineering Society (AES) as part of the AES-X210 project. In its current form, the OCA protocol has been developed for audio networks, but there are intentions to enable control of video devices as well.

OCA is an object-oriented protocol that defines [114]:

- the type of objects (which are instances of the OCA classes),
- how objects and their attributes are identified,
- the format of the data transfered between objects,
- the procedure for exchanging data when objects communicate.

The OCA specification is being developed for a network that has between 0 and 10,000 devices [18]. The specification consists of three parts [111]:

- *Open Control Framework (OCF)* - which describes the models and mechanisms of the OCA [114].
- *Open Control Class (OCC)* structure - which describes the nature of the OCA class hierarchy [115]. OCA classes are instantiated as objects which represent control points within an OCA device.
- *Open Control Protocols (OCP.<n>)* - which are the protocols that utilize the OCF and OCC for particular application contexts. The *'<n>'* represents an index of the particular protocol. Presently only the *OCP.1* (which is for TCP/IP networks) has been defined [116]. There are plans for specifying the OCP.2 protocol for USB connections, and OCP.3 protocol which will be a "text version for various purposes" [18].

OCA classes are organized in a tree hierarchy with a single root node referred to as a *root class*. Subsequent nodes downstream from the root are other *derived* OCA classes that each inherit from a single parent class. Each node (OCA classes) in the tree hierarchy (except for the root class) inherits from exactly one parent node. A *derived class* is a specialized entity of the parent class and it may redefine any of the methods that it inherited from its parent [18]. Each class is uniquely identified by its *class identifier (class ID)*. A class ID can be used to address a particular class, and it consists of integers in the dotted number notation. By convention the names of standard OCA classes begin with the *"Oca"* characters. An OCA class has a number of [114]:

- *properties* - these are attributes that can be remotely monitored and controlled,
- *methods* - these are used to acquire and/or modify class properties,
- *events* - enable a remote controller to subscribe to a class such that it is notified whenever a change occurs to a particular class property.

The OCA device model consists of objects that perform the following roles. These roles are depicted in the OCA device model of Figure 3.3.

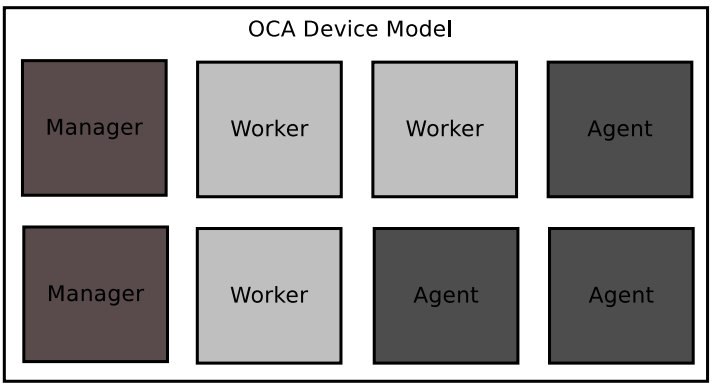


Figure 3.3: OCA device model

The possible roles that can be fulfilled by an OCA object are [114]:

- *Managers* - are objects whose attributes and states reflect the overall status of the device. An OCA device is permitted only one manager object.
- *Workers* - are objects which are control interfaces to the functions within a device. There are different types of workers based on the type of control functions they fulfill. These include:
  - *actuators* - which control application functions, for example switches.
  - *sensors* - which detect and report signal parameters and other values back to controllers, such as signal level sensors.
  - *blocks* - which allow the grouping of objects within a device.
  - *matrices* - which allow objects to be assembled into two-dimensional arrays within a device.
  - *networks* - which describe digital networks to which the device is attached.
- *Agents* - are objects which act as intermediaries that are capable of affecting controls within a device, although they (agents) do not perform a signal processing function.

Each OCA object is uniquely identified by its 32-bit object number (ONo) [18].

3.2.5.1 OCA messaging

Communication between OCA devices occurs between *objects*, which are instances of OCA class(es). The message format used by OCA objects to communicate, depends on the particular OCP protocol. Each OCA message is encapsulated within *Protocol Data Units (PDUs)* for transmission on the network. Most OCA messages are unicast in nature and require that the target OCA object returns an acknowledgement to the source of the message. The exception to this is the *"fast"* message, which is multicast on the network and does not require an acknowledgement. The fast message type is part of the OCA's event subscription mechanism, and is typically used for noncritical traffic such as meter updates on a remote console [18].

An OCA message will typically include [18]:

- a method call,
- a method return status or event notification.

3.3 Overview of Layer 2 Audio Control Protocols

The monitoring and control of networked audio devices can be performed by utilizing OSI/ISO layer 2 audio transport protocols for transporting command messages and status updates. Audio control protocols that do so are classified in this chapter as layer 2 audio control protocols. They include:

- Audio Video Control (AV/C) protocol
- IEEE 1722.1 (AVDECC) protocol
- Music Local Area Network (mLAN) protocol

Each of the above audio control protocols are described in the following sections.

3.3.1 Audio Video Control (AV/C)

Audio Video Control (AV/C) is a protocol that was developed for device control on the IEEE 1394 serial-bus [117]. In its original form, AV/C utilizes IEEE 1394 asynchronous

transactions, which permit direct memory 'reads' and 'writes' to registers on an IEEE 1394 node. Each asynchronous command is acknowledged by the target node. Recently, there has been a demonstration of *AV/C over IP* in an effort to transmit AV/C messages over IP network infrastructure [118]. The 1394 Trade Association (1394TA) maintains the various AV/C standards [119].

AV/C utilizes the *Function Control Protocol (FCP)* command and response messaging mechanism [120]. FCP defines a set of command and response registers. The command register is called the *'FCP_COMMAND'* register, and the response register is known as the *'FCP_RESPONSE'* register. A *controller* (IEEE 1394 node) transmits a command within an FCP frame (which is encapsulated in an IEEE 1394 asynchronous packet) to a *target* node. This command is addressed to the *FCP_COMMAND* register within the target node. The target node responds by performing an asynchronous transaction to the *FCP_RESPONSE* register within the controller. These asynchronous transactions are writes to the particular registers. AV/C defines a command set that utilizes the FCP mechanism for device control.

An AV/C device consists of any number of logical entities called *units*. Depending on the overall purpose of a device it may implement one or more units. Within a unit are a number of functional entities called *subunits*. Figure 3.4 depicts an AV/C unit with its subunits.

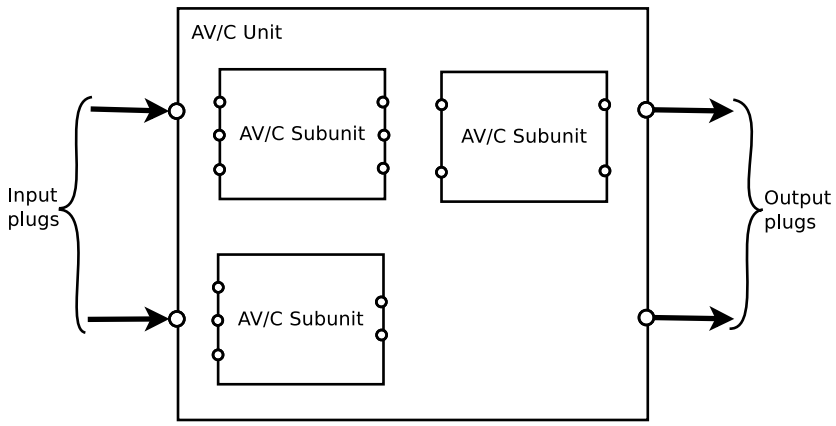


Figure 3.4: Structural layout of an AV/C unit

The AV/C unit shown in Figure 3.4 consists of three subunits. There are different types of AV/C subunits, for example an *AV/C audio subunit* and an *AV/C music subunit* have been defined by the I394TA [121] [122]. Since a unit may have multiple instances of a subunit, a *3-bit subunit identifier (ID)* is used to uniquely specify a particular subunit. The AV/C subunits implement a number of *functional blocks* that perform specific control or signal processing on behalf of the subunit. The types of functional blocks that

exist in a device depends on the subunit within which they reside.

An AV/C unit implements a number of input and output *plugs*, which are virtual endpoints for receiving and transmitting data (respectively). There are different types of unit plugs, namely [117]:

- *serial bus isochronous plugs* - for transmitting isochronous data (typically time-sensitive media streams) from or to the unit.
- *serial bus asynchronous plugs* - for transmitting asynchronous data (typically non-critical command data) from or to the unit.
- *general bus plugs* - make it possible to transmit data between other types of signal transmission buses besides the IEEE 1394 serial bus.
- *external plugs* - transmit data between the unit and any other signal source/destination that is not a bus.

Input plugs act as entry points for data to a unit, and output plugs fulfill the role of exit points of data from the unit. The AV/C specification defines the address that should be used for each of the above plug types. Each unit can have a maximum of 31 input and output plugs.

AV/C subunits implement a number of *source* and *destination* plugs. A subunit source plug is the point of exit of data from the subunit, and the destination plug is the point of entry of data to the subunit. A maximum of 31 source plugs and 31 destination plugs may be implemented on each subunit. A device that requires more plugs could do so by increasing the number of subunits (of the same type) that it implements.

Various connections exist between the unit input plugs, unit output plugs, subunit destination plugs and subunit source plugs. These connections could be fixed, in which case they are created by the manufacturer, and they cannot be modified by a controller. In some cases, it is possible for a controller to remotely modify these connections by sending the appropriate connection commands.

The logical layout of the internal structure of an AV/C device is presented by an interface known as *descriptors and information blocks* (*info blocks)* [123]. An AV/C device's descriptors are presented in a hierarchical structure that describes the parameters that it contains. At the top of the hierarchy is a descriptor known as the *root descriptor*. There are three types of descriptors in AV/C, namely *[124]*:

- *unit/subunit identifier descriptor* - holds information that is associated with the entire unit or subunit.

- *list descriptor* - holds information about the *entry descriptors* that are contained within the unit/subunit.
- *entry descriptor* - is addressable within the descriptor hierarchy, and may contain other list descriptors or info blocks that provide specific information about the unit/subunit.

The use of descriptors and info blocks for presenting a device's information does not constrain a manufacturer on how a device's properties are implemented. It rather provides a consistent way of presenting device information so as to provide a standard way of remotely enumerating the device's properties. The actual storage of these parameters within the device is at the discretion of the device manufacturer. AV/C defines a set of commands that can be used to acquire and modify a device's descriptors and info blocks.

3.3.2 IEEE 1722.1 (AVDECC)

IEEE 1722.1 refers to the IEEE standard protocol for device discovery, connection management and control of IEEE 1722 based devices [19]. It is currently a draft standard, and is commonly referred to as the *Audio Video Device Discovery, Enumeration, Connection Management and Control (AVDECC)* protocol. It is being developed to enable control of networked Ethernet AVB endpoints that are capable of transmitting IEEE 1722 data streams.

IEEE 1722.1 defines three protocols that will enable the discovery, connection management, enumeration and control of AVB endpoints via layer 2 transport. These protocols are:

- *AVDECC Discovery Protocol (ADP)* - which defines a mechanism for device discovery on Ethernet AVB networks. It is used by an AVDECC end station to announce its presence and departure on a network, as well as to discover other networked end stations.
- *AVDECC Connection Management Protocol (ACMP)* - which defines a mechanism for establishing and destroying IEEE 1722 stream connections on an Ethernet AVB network.
- *AVDECC Enumeration and Control Protocol (AECP)* - which defines a mechanism for discovering and controlling the features and functional units within networked endpoints.

AVDECC messages are encapsulated within layer 2 Ethernet frames that are transmitted on an Ethernet AVB network. Each of the AVDECC protocols defines a particular *Protocol Data Unit (PDU),* which is used to encapsulate protocol specific messages.

An end station that is capable of AVDECC messaging is referred to as an *AVDECC entity*. An AVDECC entity is uniquely identified by a 64-bit identifier. The possible roles that an AVDECC entity could fulfill are:

- *AVDECC controller* - is an AVDECC entity that sends commands to control or obtain information about the other AVDECC entities on the network. An *AVDECC proxy* is a special case of the AVDECC controller, which forwards AVDECC messages between OSI/ISO layer 3 and layer 2.
- *AVDECC talker* - is an AVDECC entity that is the source of one or more media streams on the network.
- *AVDECC listener* - is an AVDECC entity that receives one or more media streams from the network.
- *AVDECC interface* - is an entity on the network that is capable of receiving and/or transmitting AVDECC messages but is not an AVDECC controller, talker, or listener.

The IEEE 1722.1 standard also defines a device modeling scheme known as the *AVDECC Entity Model*. This model standardizes the layout of the various properties and controls within an AVDECC entity into a structured hierarchy of addressable objects. At the top level of this hierarchy is an *AEM entity* object that describes the overall functionality of the AVDECC entity.

The IEEE 1722.1 (AVDECC) protocol is further described in section 3.4.3.

3.3.3 Music Local Area Network (mLAN)

The music Local Area Network (mLAN) is an IEEE 1394-based audio network control protocol that allows for the transmission of audio and music control data between networked devices. To enable this, the mLAN protocol defines a number of formats, structures and procedures that make it possible to deploy reliable IEEE 1394 networks within music studios [125]. The mLAN technology enables multiple sequences of audio and music data (such as MIDI and SMPTE) to be 'bundled' together, then transmitted as data streams from a source IEEE 1394 node and appropriately extracted at the receiving

IEEE 1394 node. The media (audio and music data) that are transmitted conform to the *"Audio and Music data transmission protocol"* specification [37]. mLAN also enables synchronization of networked devices by transporting timing information within an asynchronous packet's *Common Isochronous Packet (CIP)* header. CIP is defined in the IEC 61883-1 specification [38].

Two approaches were implemented in the mLAN technology. These are:

- *mLAN version 1*
- *mLAN version 2* called the *enabler/transporter architecture*

In the first generation of mLAN (*mLAN version 1*) the formats, structures and procedures for device control and connection management were implemented in hardware (such as the mLAN-PH1 chip [126], and the mLAN-PH2 chip [127]), and associated firmware within IEEE 1394 mLAN nodes. An mLAN node is regarded as an IEEE 1394 serial bus node that is mLAN compliant. In mLAN version 1, each mLAN node implemented *mLAN plugs* which were abstractions (in software) of physical signal endpoints. The mLAN plugs could be 'source plugs', in which case they transported signals out of the mLAN node, or 'destination plugs' that are the entry points of signals into the mLAN node.

mLAN version 1 defined various input and output registers that are associated with particular input and output plugs on a node. In order to establish a stream, an mLAN controller would perform an asynchronous write transaction on the output plug register associated with a particular *mLAN source plug* on a source node. This write transaction indicated the channel on which the source IEEE 1394 node should transmit its audio stream. The controller then performed an asynchronous write transaction to the input plug register that is associated with a particular *mLAN destination plug* on the destination node, specifying the same channel value as written to the source node.

The states of each plug were stored within non-volatile memory on the mLAN nodes, thus each node was able to restore to its previous state after a power restart.

The mLAN version 1 nodes implemented AV/C *descriptors* and *info block*s to expose the properties of their mLAN plugs [124]. They also implemented AV/C *vendor dependent commands* that made it possible to read and modify mLAN plug information, which were presented using the AV/C descriptor and info block mechanism [117].

Since the descriptor and info block mechanism permits the internal functionality to be device or manufacturer specific, mLAN defined its own procedures but utilized the descriptors and info blocks for control presentation.

This approach suffered from a number of problems, in particular [128]:

- the large amounts of memory required to implement the mLAN plugs on devices contributed to high cost of mLAN compliant devices.
- an mLAN compliant devices would require firmware upgrades in order to comply with bug fixes, upgrades and changes in connection management approach.
- non-mLAN chip manufactures required a considerable amount of effort and expertise in order to get their devices to communicate with mLAN devices.
- device discovery and enumeration was time consuming for networks with many devices.

To address these concerns, the *Enabler/Transporter* architecture (also known as the *Plural-Node* architecture) was developed [129]. The core difference between this approach and the mLAN version 1 approach was the relocation of the mLAN plugs to reside within a control workstation rather than at the mLAN nodes. Thus a workstation, which could be a Macintosh, Windows or Linux PC, is required to establish and destroy connections between networked devices. In order to do this, the controlling workstation implemented an mLAN control capability known as an *'Enabler'*. A networked mLAN node implements the *Audio and Music (A/M)* protocol responsible for encapsulating and extracting audio and music data. These nodes are referred to as mLAN *'transporters'*. Each transporter implements a *'transporter controller interface'* which is used by an enabler to control the transporter. Each transporter is controlled by only one enabler, although an enabler may have multiple transporters under its control. This separation between A/M data encapsulation, extraction and transport, from the actual mLAN plugs abstraction is known as the *plural-node approach*, and the enabler is a *plural node device* since it incorporates many nodes [129].

Within an mLAN enabler are three layers, namely [128]:

- *mLAN plug abstraction layer* - implements all mLAN source and destination plugs of the transporters under the control of the enabler.
- *Audio and music manager layer* - maintains information about the mLAN plug parameters under the control of the enabler.
- *Hardware Abstraction Layer (HAL)* - abstracts information about hardware implementations of the transporter, thus enabling interoperability between transporters that are manufactured by different vendors. The HAL communicates with the

transport controller interface within each of the transporters under the control of the enabler.

A vendor wishing to communicate with mLAN devices would need to implement a transport controller interface and provide the associated HAL for the interface to the enabler as a 'plug-in'. An *Open Generic Transporter (OGT)* has been defined, providing a standard structure for vendors to implement their HAL and associated transport controller interface [130].

3.4 Protocols of Interest

Chapter 1 mentions the fact that the existence of multiple audio control protocols has resulted in an interoperability challenge, which is a situation where networked devices that implement different audio control protocols are unable to communicate with each other. To resolve the protocol interoperability challenge, this research project has proposed the use of a command translator.

An AES-64 protocol controller was available for this research project. There were also AES-64 end point devices. The goal was therefore to test the hypothesis using the controller and layer 2/3 protocols for which:

- there were already device implementations
- there was already take up in the industry and the protocols were likely to be used in future audio devices

In the previous section, several audio control protocols were described. Typically, an audio control protocol defines a standard technique for:

- modeling devices that conform to it,
- discovering compliant devices on the network,
- enumerating the device to determine its capabilities,
- connection management, that is the procedure for establishing and destroying audio stream connections.

This section emphasizes these requirements of an audio control protocol. In particular, further details will be provided on how three of the previously described audio control protocols are able to meet the above requirements. The audio control protocols of interest are:

- OSC,
- AES-64, and
- IEEE 1722.1 / AVDECC

The remaining protocols described in the previous section were not used in this study for the following reasons:

- IEC 62379 - there were no available devices that implement the IEC 62379 protocol, and no implementation code that could have been used to create IEC 62379 compliant devices.
- ACN - there were no audio devices that implement the ACN protocol, and no implementation code that could have been used to create one.
- mLAN - there are no plans to develop mLAN devices in the future.
- AV/C - there are no plans to develop AV/C devices in the future.
- OCA - there were no available devices that implement the OCA protocol, and no implementation code that could have been used to create OCA compliant devices.

It is worth noting that the OSC and AES-64 audio control protocols are dependent on the OSI/ISO layer 3 transport technology, while the IEEE 1722.1 audio control protocol is dependent on OSI/ISO layer 2 transport. Thus the following descriptions will provide insights into the operations of layer 3 and layer 2 transport dependent audio control protocols. A goal of this study was to show how the command translation approach could be used for translation between layer 3 protocols as well as between layer 3 and layer 2 protocols.

The OSC protocol has been chosen because:

- it is being deployed across a wide range of application contexts. These include show control, gesture recognition, and robotics.
- the OSC specification is open and ready available online [96].
- there are many implementations of the OSC protocol that can be used to create OSC applications. These implementations have been written in a number of programming languages including C, C++, Java, Ruby, and Python [131].
- OSC messages can be transported within layer 3 IP packets.

The AES-64 protocol has been chosen because:

- The AES-64 protocol has been published as an open standard within an international standards body, the Audio Engineering Society (AES) [132].
- The AES-64 protocol incorporates features that are common to most contemporary IP control protocols. These include a connection management procedure, device discovery mechanism, device enumeration, and device control. It also incorporates advanced control features such as joins, push notification and grouping. An implementation of AES-64 by Universal Media Access Network (UMAN) was made available for this research [108].
- AES-64 was co-developed with the Rhodes University audio networking research group, thus AES-64 devices were readily available for this research.
- AES-64 messages can be transported within layer 3 UDP/IP packets.

The IEEE 1722.1 (AVDECC) protocol was chosen for the following reasons:

- IEEE 1722.1 is being developed within an international standards body, the Institute of Electrical and Electronics Engineers (IEEE) [12].
- The IEEE 1722.1 specification will be an open standard that can be obtained from the IEEE.
- It utilizes layer 2 transport for message transmission, thus providing an opportunity to test the command translation approach (proposed in this thesis) across audio transport layers.

These three open standard audio control protocols, that is OSC, AES-64 and IEEE 1722.1, will be fully described in the following sections.

3.4.1 Focus on OSC

OSC has been described as a control message format exchange protocol that allows networked devices to communicate according to a client-server architecture. In an OSC network, an OSC client transmits a request to an OSC server, which in turn processes the received request. Although any type of networking topology is permitted by OSC, typically UDP/IP is used for transporting OSC packets. These OSC packets provide an *OSC address pattern* that specifies the particular *OSC method(s)* that should be triggered. OSC packets could be either *OSC messages* or *OSC bundles*.

Communication between OSC devices involves triggering control points, known as *OSC methods*, within the target OSC device. In essence an OSC message specifies the OSC method (in the form of an OSC address pattern) that should be executed. Figure 3.5 depicts the communication between an OSC client and an OSC server when an OSC client sends an OSC message to obtain the name of an OSC server.

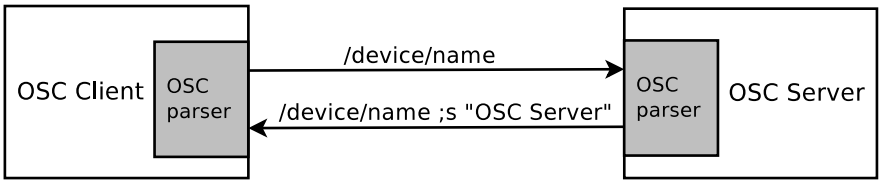


Figure 3.5: OSC client/server communication

In Figure 3.5, the OSC client sends an OSC message requesting the name of the OSC server. Thus the OSC address pattern within the message is *'/device/name'*. The OSC server (in the figure above) responds by sending:

- the same address pattern that was received from the client,
- the argument type (*'s'*) which indicates that the response is an OSC string,
- the string value *"OSC Server"*.

Each OSC device implements an OSC protocol stack, that enables it to parse messages that conform to the OSC specification.

In order to enable time-critical control of networked devices, the OSC specification defines *OSC time tags* which make it possible for a controller device (in the form of an OSC client) to specify when a particular instruction should be executed by the OSC server. OSC time tags are used when the message being transmitted is an *OSC bundle*.

An OSC bundle may comprise nested OSC messages and/or other OSC bundles. All of the OSC instructions within an OSC bundle are handled as an atomic transaction. In order words if one of the instructions within an OSC bundle cannot be executed, the entire 'bundle' of instructions are considered to have failed.

In line with other audio control protocols, OSC provides a means of modeling networked devices. It can be used to discover devices on an network, as well as configure discovered devices such that audio streams can be established and destroyed between devices. The following sections describes how these are accomplished in OSC.

3.4.1.1 Device model

In OSC, the modeling of device controls is achieved by defining the *OSC address space* within the device. An OSC address space incorporates every OSC address pattern that can be used to address an OSC device. It is an hierarchical structured representation and addressing scheme that can be used to access all of the *OSC methods* within the device. Every OSC server is required to implement an OSC address space.

The OSC address hierarchy can be viewed as consisting of nodes, with the top-most node being the *root node.* The root node is identified with the forward slash symbol (*'/'*). The leaf nodes are *OSC methods*, which are trigger points that can be addressed by a remote client in other to fulfill a task. All nodes between the root node and the OSC methods are known as *OSC containers*. An OSC pattern which is used to address an OSC method describes the full-path from root node to the OSC method in a URL-style addressing scheme. Figure 3.6 depicts the OSC address space of a simple OSC server that is capable of receiving a signal via its input and transmitting the received signal via any or both of its outputs. For the sake of this example, each input and output has a mechanism for instructing it to start or stop receiving (in the case of the input) or transmitting (in the case of the output).

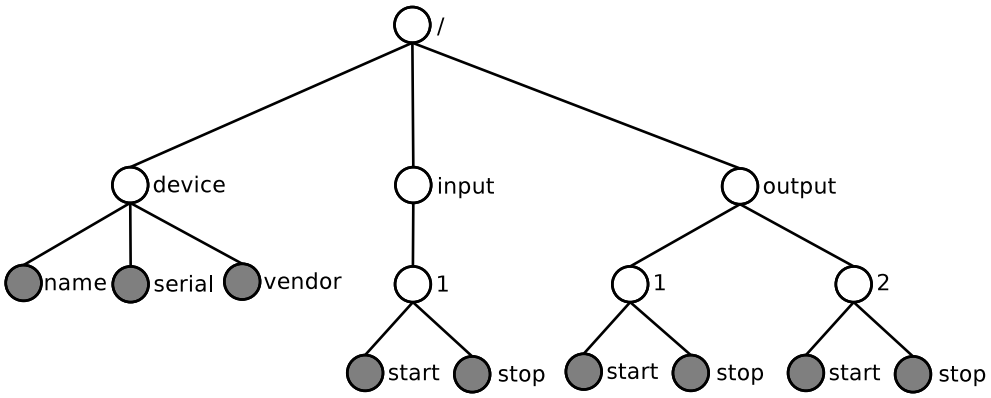


Figure 3.6: OSC address space of a simple OSC server

The OSC server of Figure 3.6 has three OSC containers (*'device'*, *'input'* and *'output*'). The OSC *'device'* container has three OSC methods (*'name'*, *'serial'* and *'vendor'*). The OSC *'input'* container has one 'child' OSC container (*'1'*) associated with it. This OSC '1' container has two OSC methods, namely *'start'* and *'stop'*. The OSC *'input'* container has two child containers associated with it, namely *'1'* and *'2'*. In the same manner as the *'/input/1'* container, the OSC *'/output/1'* and *'/output/2'* containers each have associated *'start'* and *'stop'* OSC methods.

In order to instruct the input to start receiving signal, an OSC message with an OSC

address pattern *'/input/1/start'* would be sent to the server. To stop the same signal, the OSC pattern within the OSC message will be *'/input/1/stop'*.

The process of mapping an OSC address pattern to a particular OSC method is called *matching*, and the processes of invoking (or triggering) the OSC method is called *dispatching*. When an OSC message is dispatched, the application specific implementation is triggered. This implementation is fulfilled by the OSC method. In certain instances (for example WOscLib [133]) the OSC methods are implemented as application callback functions. This approach enables a loose coupling between application implementation and OSC message parsing.

The OSC protocol defines a number of special characters that can be used to enhance pattern matching within an OSC address space. These characters are described in the OSC specification [96].

Typically an OSC parser will match and dispatch OSC messages on behalf of an application. There are a wide range of OSC parsers freely available for download, and a list of some of them can be found at the OSC website [131].

3.4.1.2 Device discovery

OSC is a message content format specification, and does not explicitly define a device discovery procedure [134]. The particular technique used for device discovery is considered to be beyond the scope of the OSC specification, since its primary concern is how control messages can be exchanged between clients and servers.

However the OSC 1.1 specification suggests that a service discovery procedure that is based on the *Domain Name System (DNS)* protocol can be used for OSC device discovery [135]. This technology is known as the *DNS-based Service Discovery (DNS-SD)* [136].

DNS-SD enables networked devices to discover instances of a particular service within a specified DNS domain. A *client* node on the network utilizes DNS-SD to announce its presence on the network, and to discover all instances of a particular *service type* that are present on the network. The entire process of discovery and advertising on the network utilizes standard DNS queries. Each instance of a particular *service type* is described using *DNS-SRV* and *DNS-TXT* records [137] [135].

DNS-SRV describes the format for identifying service instances on a network. It proposes the use of a service identification name of the form shown in Listing 3.1, together with the host IP and port number necessary for communicating with the service instance.

```
<instance>.<service type>.<domain>
```

Listing 3.1: DNS-SRV record syntax

The syntax of the DNS-SRV record shown in Listing 3.1 comprises the following fields:

- *'<instance>'* - a human readable character string that is used to identify a particular service on a device.
- *'<service type>'* - defines the type of service that is being offered by the service instance. A *service type* is comprised of two labels, both beginning with an underscore character ('_'), and separated by a dot ('.'). The first label indicates the application protocol that offers the service (for instance *'_http'* or *'_ipp'*). The second label specifies the transport protocol and would typically be *'_tcp'* or *'_udp'*. Thus typical examples of a service type could be *'_http._tcp'* or *'_ipp._udp'*.
- *'<domain>'* - refers to the *DNS subdomain* where the service names are registered. For example, domain addresses such as *'tester.interop.com'* and *'application.interop.com'* are said to reside in the same (*'interop.com'*) DNS subdomain.

DNS-TXT is used to provide additional information, which is organized as *key/value* pairs. For example if the number of inputs and outputs on a devices were to be included in a DNS-TXT record, it would be of the form shown in Listing 3.2.

```
inputs=2
outputs=3
```

Listing 3.2: DNS-TXT key/value pairs

The *'inputs'* and *'outputs'* are referred to as the *keys*, each having a corresponding *value*. In this instance, the DNS-TXT indicates that the device has two inputs and three outputs.

The OSC 1.1 specification suggests that for discovery of OSC devices on a network, the following OSC service types could be used [134]:

- *'_osc._tcp'* - which indicates an OSC application via TCP transport.
- *'_osc._udp'* - which indicates an OSC application via UDP tranport.

The DNS-TXT record could include the version of the OSC protocol implemented on the device, and the OSC type tags that are supported by the OSC server.

3.4.1.3 Connection management

OSC does not define a connection management procedure. Within the context of audio networks, any application that utilizes OSC is required to define the procedures and commands that should be followed when establishing or destroying audio stream connections. The commands necessary for connection management can be transported within OSC packets.

3.4.2 Focus on AES-64

AES-64 is an IP-based audio control protocol that enables remote device monitoring, configuration and control via transmission of AES-64 messages. The AES-64 messages are encapsulated within UDP datagrams, which are in turn encapsulated within IP packets for transmission on a network.

When a controller transmits an AES-64 message to a target device on the network, the message is addressed to a *parameter* within the target device. These parameters are the control points that determine the state of various properties and attributes of the device. For example a device may possess a parameter that indicates whether a particular input is streaming, and another parameter that indicates what audio format the input can receive. Each parameter can be uniquely addressed in AES-64.

The high-level structure of a UDP/IP packet that is used to transmit an AES-64 message, is depicted in Figure 3.7.

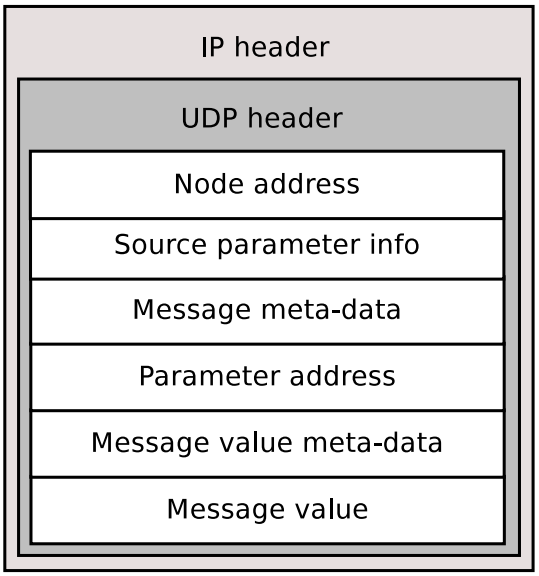


Figure 3.7: High-level layout of a UDP/IP packet that includes an AES-64 message

Figure 3.7 shows the:

- *IP header* - contains the standard *Internet Protocol version 4 (IPv4)* information necessary for routing packets from a source to a destination device on a network [67]. The AES-64 protocol permits *unicast*, *broadcast* and *multicast* addresses within the IP header. A unicast is used for one to one communication, and broadcast is used when the intention is to transmit the AES-64 message to all devices on the network. Multicast transmission is used when the intention is to transmit the AES-64 message only to members of a group, that is members 'listening' on a particular group IP address.
- *UDP header* - provides standard UDP transport of AES-64 messages [71]. The UDP headers contain a communication port number. The port number *'7107'* has been registered for the AES-64 protocol by the *Internet Assigned Numbers Authority (IANA)* [138].
- AES-64 message - contains the AES-64 protocol specific information, including:
  - *Node address* : refers to the fields within an AES-64 packet that specify the particular AES-64 node that the message is addressed to. This includes the following fields:
    - *Destination device ID* - a 128-bit field that specifies the target AES-64 device ID.
    - *Destination node ID* - a 32-bit field that specifies a particular target AES-64 node ID.
    - *Source device ID* - a 128-bit field that specifies the source (or controller's) AES-64 device ID.
    - *Source node ID* - a 32-bit field that specifies a particular source (or controller's) AES-64 node ID.
  - *Source parameter info* - refers to the 32-bit *source parameter ID* field that specifies the parameter (within the source device's node) that generated the request.
  - *Message meta-data* : refers to the fields within an AES-64 packet that provide information about the nature of the message. These include the following fields:
    - *User-level* - an 8-bit field that provides information used to determine the authorization of the command to modify the target parameter.

        * *Message type* - an 8-bit field that indicates whether the parameter address that is contained within the message is a hierarchical 104-bit address or a 32-bit index of the parameter. It also specifies whether a response is required from the target parameter.
        * *Sequence ID* - a 32-bit field that is used to match requests with responses. This field provides reliability for AES-64 messaging, which utilizes UDP transport.
        * *Command executive* - an 8-bit field that indicates the essence of the message.
        * *Command qualifier* - an 8-bit field that indicates the attribute of the parameter that is being targeted.
    - *Parameter address* : there are two types of parameter addresses permitted by the AES-64 protocol. These are:
        * hierarchical addressing - refers to the 104-bit field that describes the full (7-level) parameter address. The 7-level hierarchy is explained in section 3.4.2.1.
        * index addressing - refers to the 32-bit *parameter index* that is unique to each parameter, and is associated with the parameter by the AES-64 protocol stack when the parameter is created. This is further explained in section 3.4.2.1.
    - *Value meta-data* : refers to the 8-bit (*valft*) field that specifies the format of the parameter's value, which is contained within the message.
    - *Value* : is a variable length field that indicates the actual value of a parameter, and is of the format specified by the value meta-data.

A detailed structure of the AES-64 packet can be obtained from the AES-64 specification [17].

AES-64 messages can broadly be considered to be of two types. These are:

- *'Get'* message - used to obtain the value of a parameter. This type of message does not contain any value in the value block of Figure 3.7.
- *'Set'* message - used to modify the value of a parameter. When this type of message is transmitted by a controller, a value is indicated and it will be of the format specified by the value meta-data of Figure 3.7.

The AES-64 specification defines a large number of parameters whose values can be acquired and/or modified by a controller in order to monitor or change the state of an AES-64 device [17]. Associated with each parameter is an application specific *callback* that implements the functionality of the parameter. While the hierarchical addresses provide a way of communicating with a particular parameter, the callback associated with that parameter fulfills the instruction. For example a message to 'set' a *gain* parameter on a particular input signal on a mixer will cause the callback associated with the gain parameter to be executed. It is this callback that will actually cause the new value to be set on the gain, thus resulting in an observable effect. Figure 3.8 shows how an AES-64 device processes a message.

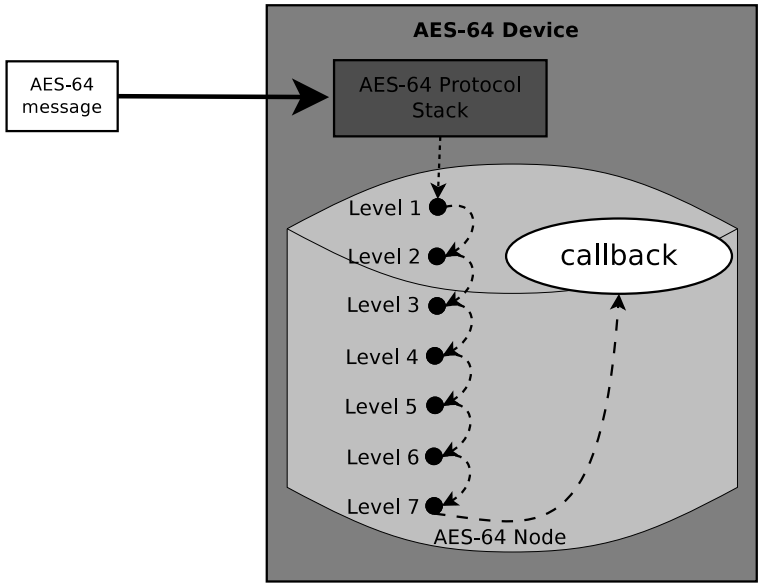


Figure 3.8: AES-64 message triggers a parameter's callback

In Figure 3.8, an AES-64 message is received and processed by the 'AES-64 *Protocol Stack'*. It matches the address within the received message with the specified parameter. This causes the callback associated with the parameter to be triggered.

Every AES-64 device incorporates an AES-64 protocol stack, which is responsible for protocol specific implementations. These include:

- AES-64 command packetization and extraction,
- abstracting AES-64 nodes,
- identifying the parameter(s) that a particular AES-64 message is addressed to,
- enabling discovery by responding to AES-64 discovery requests,

- enabling the parameter grouping mechanism.

Although the AES-64 protocol can be used to control any type of device, its current implementations are targeted at audio devices. The following subsections describe how AES-64 is able to meet some of the requirements of an audio control protocol mentioned in section 3.4.

3.4.2.1 Device model

AES-64 provides a mechanism for modeling compliant devices. The device model relies on the AES-64 protocol stack to create nodes and parameters that adequately model the various functionalities and attributes within the device. Figure 3.9 depicts a conceptual layout of an AES-64 device model.

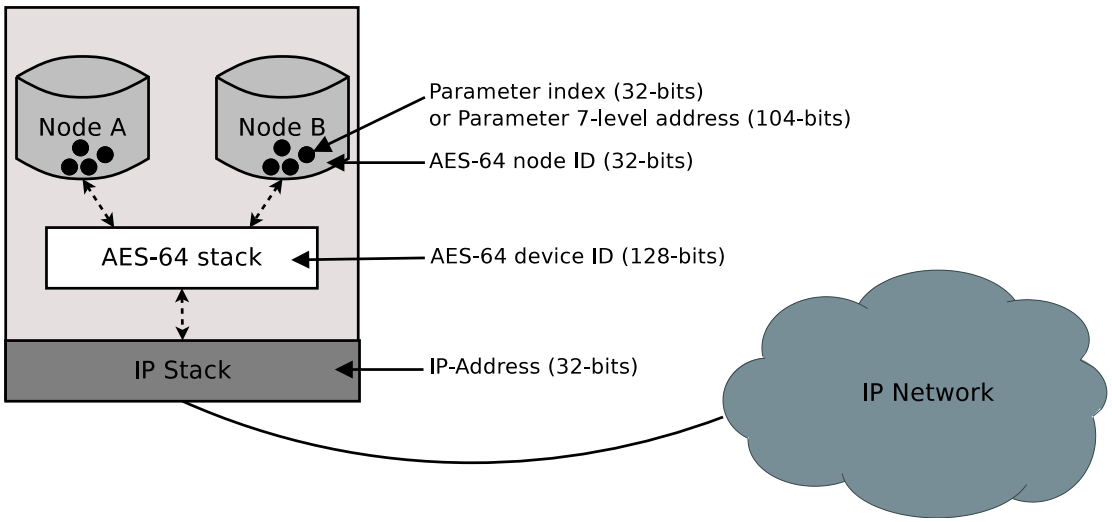


Figure 3.9: AES-64 conceptual device model

At the bottom of the AES-64 device model depicted in Figure 3.9 is the *'IP stack'*, which 'picks' up IP messages from the network. In particular UDP traffic on the designated AES-64 communication port (7107) is 'passed' on to the AES-64 protocol stack. This protocol stack then executes the instruction specified in the *'message meta-data'* for the parameter specified by the *'parameter address'* section of the AES-64 message. The AES-64 message structure has been described in section 3.4.2. The 'AES-64 *stack'* is responsible for determining that the received message is meant for the device by comparing the value of the *destination device ID* field (within the AES-64 message) with its own 128-bits *device ID*.

Above the 'AES-64 *stack'* are a number of functional units known as *nodes*. An AES-64 application (or device) may have any number of nodes, with each node corresponding to

a particular functional entity. However each node will possess a unique 32-bit identifier known as its *node ID*. For example, a proxy device may instantiate multiple nodes, with each node corresponding to each device that it proxies.

The parameters are contained within a node. These parameters are created by the AES-64 stack, and associated with each parameter is a 32-bit unique identifier known as the *parameter ID*. The parameters within an AES-64 node are structured in a fixed 7-level hierarchy that can be used to address the particular parameter. The seven levels allow for a consistent way of presenting parameters to a remote controller.

The seven-levels defined by AES-64 are [17]:

- *Section block* - is the highest level of the hierarchy that describes in the broadest sense a grouping to which a particular parameter can be associated. For instance, input section and output section.
- *Section type* - is a further categorization of the section block into smaller groupings. For example, audio input and video input are both inputs, but have been further classified.
- *Section number* - is a number that is used to categorize a parameter based on the signal path it is associated with. For example for digital audio signals the section number would be the channel number of the signal.
- *Parameter block* - categorizes parameters into their functional blocks. For example a parameter block a group of parameter equalizers that allow for wide ranging equalization of an audio channel.
- *Parameter block index* - specifies particular parameter groupings within the functional blocks. For example, each parameter equalizer type could be assigned an index 1, 2, 3 and so on.
- *Parameter type* - indicates the particular functionality of the parameter. An example would be a gain, or frequency in a parameter equalizer.
- *Parameter index* - is a number that identifies the particular parameter. For instance, if a device has more than one gain parameter, this *parameter index* specifies which of the parameters is being addressed.

Each of the levels listed above has a defined set of values with particular meanings. A full listing of the defined values of each of these levels can be obtained from the AES-64 specification [17]. Figure 3.10 shows the layout of an example 7-level parameter

structure that can be used to access the *name* parameter and *globally unique identifier (GUID)* parameter (such as serial number) on an AES-64 node.

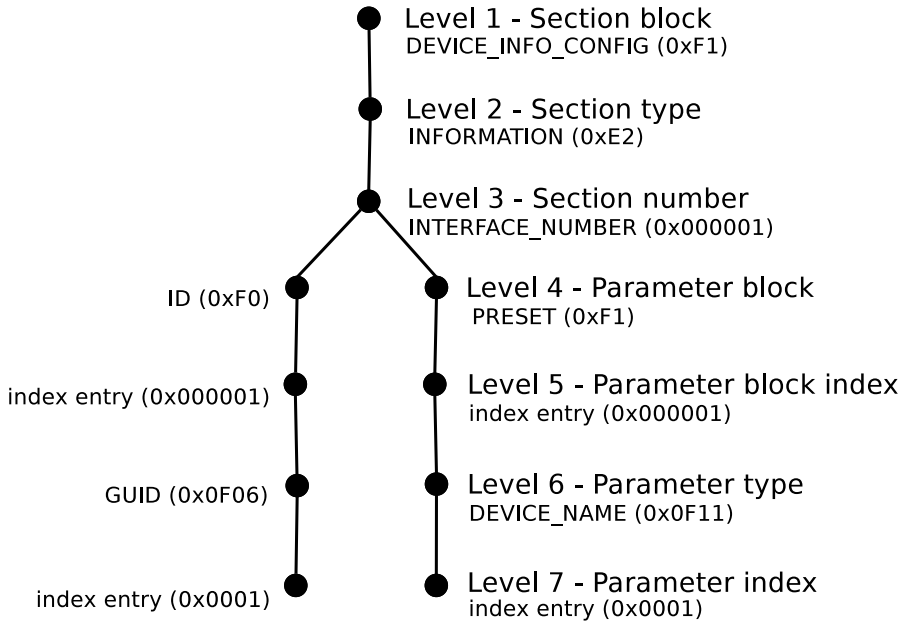


Figure 3.10: AES-64 7-level parameter hierarchy

As shown in Figure 3.10 each tree node in the 7-level parameter tree has a value associated with it. In this case, the values shown are those that are associated with the *'GUID'* parameter (on the left side) and the *'DEVICE_NAME'* parameter (on the right side) of the tree. These values are necessary when addressing a parameter. The structure of the '*Parameter address'* block of Figure 3.7, when addressing the *'DEVICE_NAME'* parameter is shown in Figure 3.11.

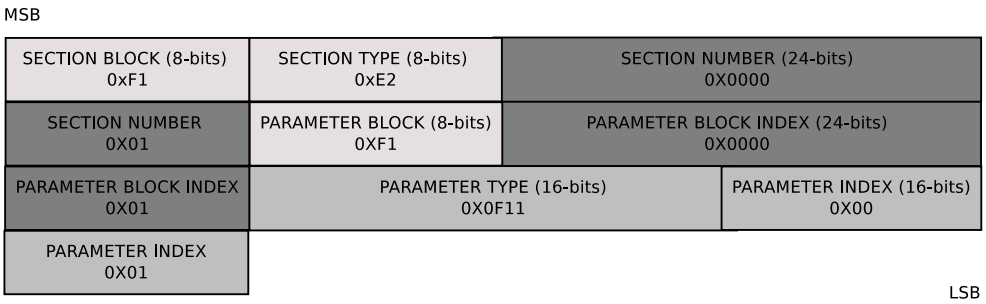


Figure 3.11: An example AES-64 message's parameter address for a *'DEVICE_NAME'* parameter

As mentioned earlier, the 32-bit parameter index that is associated with the parameter at its time of creation can be used instead of the full 7-level address shown in Figure 3.11.

It is possible to address multiple nodes at any level of the 7-level parameter hierarchy, by using the *'wildcard'* mechanism. For instance, if a controller wanted to obtain both parameters shown in the tree structure of Figure 3.10, it could 'wildcard' the *'Parameter block'* (Level 4) of the full 7-level address shown in Figure 3.11.

3.4.2.2 Device discovery

Communication between AES-64 devices is via 'gets' and 'sets' of parameters within a device. This is so since every attribute of a device is modeled as parameter. A controller wishing to discover networked AES-64 devices will need to broadcast a request that will cause all the AES-64 devices to respond to it. In fact, this could be a number of broadcast requests to all of the *device discovery* parameters that are necessary for communication and identifying each AES-64 device on the network. The device discovery parameters include IP address, subnet mask, device type, and device name.

To simplify the discovery process, an enhanced mechanism has been developed for AES-64 bulk value requests. This mechanism, known as the *Universal Snap Group (USG)*, provides an efficient way for a controller to request the values of a group of parameters.

A USG-enabled AES-64 device, implements two data structures within a particular node (functional unit). These are [139]:

- *USG cache list* - is an indexed list of parameters within an AES-64 node, such that each entry within the list maps a parameter's hierarchical address to its parameter ID. In section 3.4.2, it was mentioned that a node's parameters can either be addressed by its hierarchical (7-level) address or by its parameter ID (which is assigned to the parameter by the protocol stack). The USG cache list enables a single index to be mapped to both of these addressing schemes for each parameter within the node.

- *USG buffer pool* - is a fixed sized data store that contains a number of entries called *USG buffers*. Each buffer within the 'pool' is uniquely identified by its *USG buffer ID*, and contains among other attributes a *parameter bitmap*. The parameter bitmap is used to specify (by mapping) which parameters within the USG cache list are associated with the USG buffer.

USG defines two roles between communication devices, namely *USG requester* and *USG target*. A USG requester is a device (such as a controller) that requests the values of a group of parameter values. The USG target is the device that responds to the

request, by returning its parameter values. An AES-64 device can fulfill both roles on the network.

A USG transaction involves the following steps [139]:

- *Initialization* - the USG target creates a USG cache list which contains the full parameter address and parameter 'ID' for all parameters within a node. Then it creates a fixed size USG buffer pool for USG transactions.
- *Parameter bitmap creation* - the USG requester sends a message to the USG target containing the full address for all the parameters it is interested in. The target will create a USG buffer (within the target's USG buffer pool) with a bit map that indicated the indexes of the target's USG cache list that match the required parameter. The target responds to this request by sending a valid USG buffer ID, to the requester.
- *Retrieval of parameter values* - the requester sends a message to the target requesting the values of the parameters associated with a particular USG buffer. It indicates the USG buffer of interest using the USG buffer's ID. In turn, the target returns the values of the parameter associated with the USG buffer. At any time, the requester is permitted to request the values of the parameters associated with the USG buffer. The requester is in a position to match the values to the appropriate parameter.
- *USG buffer deallocation* - since the size of the USG buffer pool is defined at initialization, the pool memory should be adequately managed. Thus when a requester no longer requires updates of the group of parameters within the USG buffer it requested, it should indicate this to the target. The target will then be able to reallocate the USG buffer to another USG query. This is typically the case when USG is used in device discovery.

3.4.2.3 Connection management

The AES-64 protocol does not define a generic connection management procedure. By its nature, AES-64 enables remote device monitoring, configuration, and control by providing a mechanism for acquiring and modifying the values of parameters. In order to enable a remote controller establish or destroy streams, an AES-64 device models the device stream controls as parameters, with associated callbacks that perform application specific functions.

In AES-64, media streams are referred to as *multicores*. There are different types of multicores, for example *audio multicores* and *video multicores*. For audio stream connections, an AES-64 device will implement a number of audio multicores that represent the various input and/or output audio stream connections that the device is capable of transmitting and/or receiving (respectively).

The connection management procedure utilized by the AES-64 protocol typically depends on the audio transport protocol (technology). The two audio transport technologies for which the protocol has been used are IEEE 1394 and Ethernet AVB. The AES-64 connection management procedures on each of these technologies are slightly different and will be described here.

AES-64 connection management procedure for IEEE 1394 devices

In order to establish or destroy a stream connection between a source and a destination node on an IEEE 1394 serial bus, an AES-64 device would implement a:

- *channel* parameter - one for each input and output multicore. This parameter indicates the value of the channel on which a source node is transmitting, or a destination node is receiving an audio stream.
- *start* parameter - one for each input and output multicore , which can have one of two values. In the case of a source device's output multicore, this parameter is used to start or stop the transmission of an audio stream. In the case of a destination device's input multicore, this parameter is used to control when the device should start or stop receiving an audio stream.

The AES-64 IEEE 1394-based connection management requires the controller to [1]:

- acquire the value of the *channel* parameter associated with a particular output multicore on the source node.
- modify the value of the *channel* parameter associated with a particular input multicore on the destination node, to the value obtained from the source node.
- modify the parameter associated with the output multicore (on the source node), that will cause it to start streaming.
- modify the parameter associated with the input multicore (on the destination node), that will cause it to start receiving the stream.

AES-64 connection management procedure for Ethernet AVB devices

Connections on an Ethernet AVB network are between a talker (which is the source of the stream) and one or more listeners (which are the receivers/consumers of the stream).

On an Ethernet AVB network, a talker advertises the characteristics and attributes of the streams it has on offer by utilizing *MSRP*. MSRP has been described in section 2.2.1.2. These attributes include the 64-bit *stream identifier (ID)* that is used to uniquely identify each stream. The announcements (adverts) by the talker are received by each device on the network. This provides a mechanism for a listener to gain knowledge of what streams are available on the network.

A listener that wishes to receive a stream will send a *listener ready* attribute declaration via MSRP to the network. This message is forwarded by the intermediate Ethernet AVB bridges towards the talker. As the message is being propagated, the necessary resources will be reserved. When the talker gets a *listener ready* or *listener ready failed* attribute declaration, it knows that at least one listener on the network is prepared to receive a particular stream indicated by the stream ID. The talker then commences transmission of its stream, assured that it will be delivered to the listener.

In order to support AES-64 connection management of Ethernet AVB devices, a *stream_ID* parameter (with value 0x0D13) has been defined at level-6 (*parameter type*) of the parameter hierarchy. Table 3.1 shows the 7-level hierarchy for the stream_ID parameter of an audio input Ethernet AVB multicore.

Level	Label	Value	Description
1	Section Block	0x01	INPUT_SIGNAL
2	Section Type	0xD1	AUDIO
3	Section Number	0x778000	*Entry index '1'*
4	Parameter Block	0xD1	MULTICORE
5	Parameter Block Index	0x000001	*Entry index '1'*
6	Parameter Type	0x0D13	STREAM_ID
7	Parameter Index	0x0001	*Entry index '1'*

Table 3.1: An example AES-64 7-level hierarchy for a stream ID parameter

The connection management procedure for establishing a stream connection on an Ethernet AVB network is as follows:

- the controller sends an AES-64 message to acquire the value of the stream_ID

parameter associated with an output multicore on the talker. This is accomplished by sending a 'get' message to the talker.

- the controller sends an AES-64 message to modify the value of the stream_ID parameter associated with an input multicore on the listener. This is accomplished by sending a 'set' message to the listener.

On receiving a 'set' message addressed to a particular stream_ID parameter, the listener could proceed to request attachment to the specified stream by utilizing MSRP.

3.4.3 Focus on IEEE 1722.1

The IEEE 1722.1 standard for device discovery, enumeration, connection management and control is also known as the *Audio Video device Discovery, Enumeration, Connection management and Control (AVDECC)* protocol [19]. AVDECC is an OSI/ISO layer 2-based protocol, that is currently being developed within the IEEE [140]. A number of sub-protocols are defined by AVDECC, with each fulfilling a particular requirement. AVDECC is designed for interaction between devices that implement the IEEE 1722 specification, which is also known as the *Audio Video Transport Protocol (AVTP*).

Typically communication on an AVDECC network is between AVDECC controllers, AVDECC listeners, and AVDECC talkers. A single device (known as an AVDECC end station) may fulfill any or a combination of these roles. The AVDECC protocol also defines AVDECC interfaces, as devices that do not fit into the three above-mentioned types. Furthermore, to enable layer 3 communication with AVDECC end stations, AVDECC defines a client-server architecture implemented by the *AVDECC Proxy Protocol (APP)*. The APP protocol is described in chapter 4.

An AVDECC message is transported as a *Protocol Data Unit (PDU)* that is encapsulated within an Ethernet frame. A number of PDUs have been defined by AVDECC, with each PDU depending on the particular sub-protocol message being transported. These PDUs are based on the *Audio Video Transport Protocol Data Unit (AVTPDU)*, which is defined by the IEEE 1722 standard [49, pp. 10]. The general format of an AVDECC message is shown in Figure 3.12.

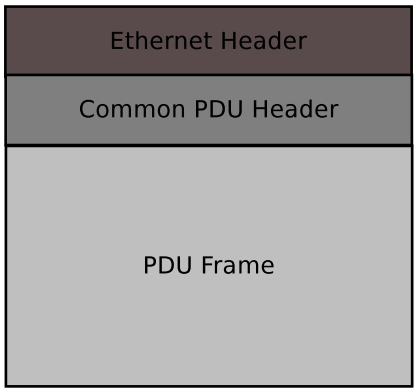


Figure 3.12: Structure of an Ethernet frame with an AVDECC PDU

The *'Ethernet Header'* shown in Figure 3.12 consists of:

- *Destination Medium Access Control (MAC) address* - a 6 octet field that specifies the MAC address to which the Ethernet frame is transported.
- *Source MAC address* - a 6 octet field that specifies the MAC address of the origin of the Ethernet frame being transported.
- *EtherType* - a 2 octet field that provides an indication of which protocol is being transported by an Ethernet frame. AVDECC uses the value *'0x22F0'* for its EtherType field, which is defined by the AVTP protocol [49, pp. 8].

The *'Common PDU Header'* shown in Figure 3.12 consists of the following fields:

- *cd* - is a 1-bit field that indicates whether the PDU frame contains control information or stream data. For AVDECC protocols, the value of this field is *'1'* indicating that the frame contains control information.
- *subtype* - is a 7-bit field that indicates which AVDECC sub-protocol message is contained in the PDU frame. The subtypes defined by the IEEE 1722.1 standard are shown in Table 3.2.

Description	Subtype Value	PDU
AVDECC Discovery Protocol	0x7A	ADPDU
AVDECC Enumeration and Control Protocol	0x7B	AECPDU
AVDECC Connection Management Protocol	0x7C	ACMPDU

Table 3.2: AVDECC subtypes

- *sv* - is a 1-bit field that indicates whether a valid stream ID is specified in the PDU. AVDECC protocols use the value *'0'* for this field indicating that the PDU does not utilize the stream ID field as defined in IEEE 1722.

- *version* -is a 3-bit field that specifies the version of the AVTP protocol being used. AVDECC protocols currently specify a value of *'0'* for this field.

- *message_type* - is a 4-bit field that indicates the particular type of message within the AVDECC sub-protocol that is being transmitted in the PDU. Each of the AVDECC protocols defines a number of possible commands, and this field is used to distinguish between the available commands implemented by a particular AVDECC sub-protocol.

- *status or valid_time* - is a 5-bit field that performs two possible functions. In the case of an ADPDU, this field is referred to as the *'valid_time'* and it indicates the validity of the information transported within an ADPDU in 2-second increments. In the case of AECP and ACMP, this field is referred as the *'status'* field. In AECPDUs this field indicates whether the command was successfully executed, while in ACMPDUs this field is used in response to a command, to indicate the status of the command.

- *control_data_length* - is an 11-bit field that specifies the number of octets that make up the PDU frame.

- *stream_ID* - is a 64-bit field which specifies the IEEE 1722 stream_ID in the IEEE 1722 standard. The AVDECC protocols reuse this field for different purposes. These are shown in Table 3.3.

Protocol	Field name	Meaning
ADP	*entity_GUID*	Indicates the 64-bit GUID of the entity transmitting an ADP ENTITY_AVAILABLE message. The ADP protocol is described later in section 3.4.3.2.
AECP	*target_ID*	Indicates the 64-bit GUID of the target to which the AECP command is addressed. The AECP protocol is described later in section 3.4.3.1.
ACMP	*stream_ID*	Indicates the stream ID of an IEEE 1722 stream that is transmitted by a talker. The ACMP protocol is described later in section 3.4.3.3.

Table 3.3: Meaning of 64-bit stream_ID field

The *'PDU Frame'* shown in Figure 3.12 is a variable length field, and depends on the particular AVDECC sub-protocol being transported.

When an AVDECC message is received by an AVDECC end station, the appropriate sub-protocol is responsible for processing the message. An AVDECC message can be unicast or multicast in nature. A unicast message is used when the message is intended for a single (target) end station. This mode of transmission is used by the AECP protocol commands, except for the AECP identification notification command, which is used to send an unsolicited information update to a controller. Multicast messaging is used when the intention is to make the AVDECC message available to every end station within the AVDECC network, which could be within an AVB time domain. The multicast transmission is used by ADP and ACMP protocols. A multicast MAC address has been reserved for AVDECC communication. This AVDECC multicast MAC is *'91:E0:F0:01:00:00'* for ADP and ACMP messaging, and *'91:E0:F0:01:00:01'* for identification notification.

3.4.3.1 Device model

An AVDECC end station is an AVB endpoint that is capable of transmitting and/or receiving IEEE 1722 streams, and implements one of the AVDECC protocols. It may possess one or more network interfaces. Figure 3.13 shows a conceptual layout of an AVDECC end station.

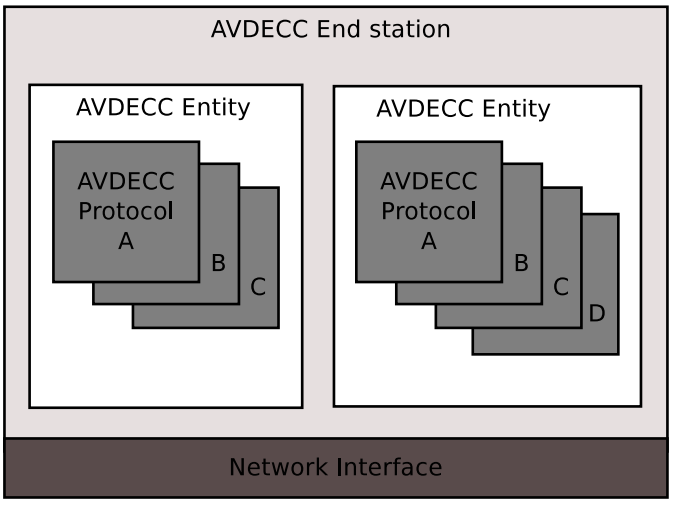


Figure 3.13: Conceptual layout of AVDECC end station

An AVDECC end station may contain any number of functional units known as *AVDECC entities*. An AVDECC entity could be an AVDECC controller, AVDECC talker, AVDECC

listener or AVDECC interface. In Figure 3.13, there are two AVDECC entities depicted in the end station. An AVDECC entity is uniquely identified by its 64-bits entity Globally Unique Identifier (GUID). An AVDECC entity implements one or more of the AVDECC sub-protocols. That is ADP, ACMP, AECP, or it may implement the AVDECC Proxy Protocol (APP).

The AVDECC Entity Model (AEM), which is defined in the IEEE 1722.1 standard, provides a means for modeling AVDECC entities. It describes how the internal components of an AVDECC entity can be structured. AEM provides a hierarchical structure of objects that can be described in a structural manner using *descriptors*. An AVDECC descriptor provides information about an AEM object. At the top of the hierarchy is a particular descriptor known as the *Entity* descriptor, which describes the information about the entire entity, and the various configurations that may exist within the entity. An entity's configuration is described by *Configuration* descriptors. A configuration descriptor represents a particular operation mode of the AVDECC entity. A configuration descriptor may contain any number of *Unit* descriptors and *Control* descriptors, which describe the different functional components that cooperate to fulfill the requirements of a particular configuration. The general hierarchical layout of an AVDECC entity's AEM descriptors is shown in Figure 3.14.

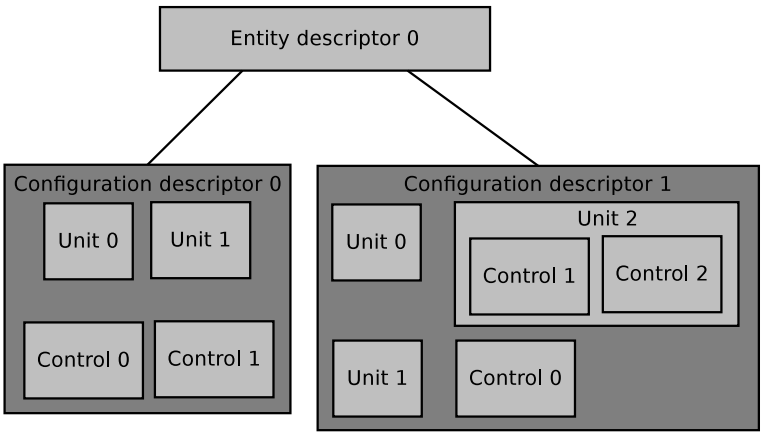


Figure 3.14: Layout of AEM descriptors

Figure 3.14 shows an entity descriptor (*'Entity descriptor 0'*) that describes the two possible modes of operation of a particular AVDECC entity. These modes of operations are modeled as two different configurations (*'Configuration descriptor 0'* and *'Configuration descriptor 1'). 'Configuration descriptor 0'* contains two unit descriptors (*'Unit 0'* and *'Unit 1'*) and two control descriptors (*'Control 0'* and *'Control 1'*). *'Configuration descriptor 1'* contains three unit descriptors (*'Unit 0', 'Unit 1'* and *'Unit 2'*), and a control descriptor (*'Control 0'*). The *'Unit 2'* descriptor contains two control descriptors

(*'Control 1'* and *'Control 2'*).

The IEEE 1722.1 standard defines different types of descriptors that can be used to model the functional components within an AVDECC entity. A *'descriptor_type'* field is used to differentiate between the various types of descriptors. Each descriptor is uniquely identified by its 16-bit index known as the *'descriptor_index'* field. Within an AEM the value of *'descriptor_index'* starts at '0' (zero) for a particular descriptor type and increments with the number of descriptors of the same type per configuration.

To describe signal paths as they convey signals 'into' and 'out of' a unit, the IEEE 1722.1 describes *port* descriptors. Figure 3.15 depicts the AEM for an AVDECC entity. This diagram was created during the research project and was later was modified and adopted by the IEEE 1722.1 working group as the example AEM model in the IEEE 1722.1 standard.

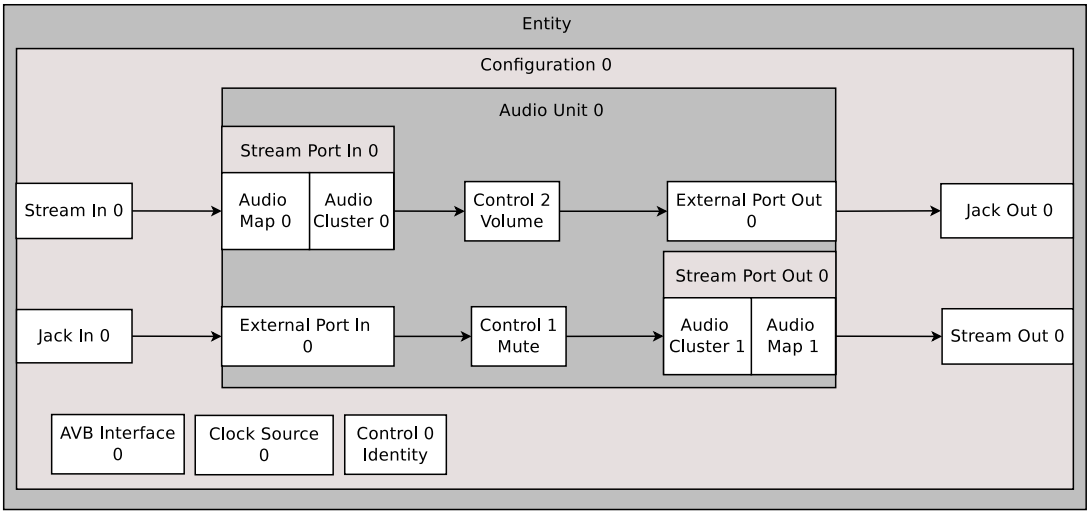


Figure 3.15: Example AEM model for an AVDECC entity

Figure 3.15 shows a single configuration of an entity that is capable of receiving an IEEE 1722 stream and an analog stereo audio signal, then processing the signal within an *audio* unit before transmitting it on an analog stereo output and IEEE 1722 output (respectively). There are two signal paths as audio enters and exits the AVDECC entity depicted in Figure 3.15.

- In the first path, '*Stream In 0'* descriptor describes the received IEEE 1722 stream and routes the stream to the *'Stream Port In 0'* descriptor, which resides within the *'Audio Unit 0'* descriptor. *'Audio Map 0'* makes it possible to trace the static relationship between the channels in a *stream descriptor ('Stream In 0')* and an *audio cluster descriptor* (*'Audio Cluster 0'*). *'Audio Cluster 0'* describes the channel

groupings (in this case two channels of audio) within a stream stream. The two-channel audio signal is grouped by the *'Audio Cluster 0'*, then routed to *'Control 2 Volume'* descriptor that allows for remote control of the gain (volume) of the signal. The signal is then routed to the egress '*External Port Out 0'* descriptor as it makes its way out of the '*Audio Unit 0*' descriptor. On exit from the entity, the signal is routed via the '*Jack Out 0'* descriptor.

- In the second signal path, a stereo audio signal is described by the '*Jack In 0'* descriptor as in the configuration depicted in Figure 3.15. The '*External Port In 0'* descriptor provides information about the audio signal as it enters the *'Audio Unit 0'* descriptor. The signal is routed to the *'Audio Cluster 1'* descriptor (which resides within the *'Stream Port Out 0'* descriptor) via the *'Control 1 Mute'* descriptor, which allows for remote mute control of the audio signal. The *'Audio Map 1'* descriptor maps the stereo audio signal as it exits the entity via the *'Stream Out 0'* descriptor.

In order to enable remote control and monitoring of the various descriptors implemented in an AVDECC entity, the IEEE 1722.1 standard has defined a number of commands that can be used to acquire and modify descriptors, as well as the possible responses that should be expected for each command. These commands and responses are transported within an *AVDECC Enumeration and Control Protocol Data Unit (AECPDU)*. The AECPDU is identified by the value of the *'subtype'* field of the *'Common PDU Header'* of Figure 3.12. The value of the *'subtype'* field for an AECPDU is *'0x7B'*. The *'message type'* field of the *'Common PDU Header'* of Figure 3.12 is used to differentiate between the different types of messages being transported by the AECPDU. In particular, a *'message type'* field value of *'0'* indicates that the AECPDU is transporting an AEM command, while a value of *'1'* indicates that an AEM response is being transported.

Although the fields of the AECP commands and responses that allow for remote access and control of an AEM model vary, they contain a *'command type'* field which indicates the specific type of command or response being transported. An AECP also contains a '*descriptor type'* field, which specifies the particular type of descriptor that is addressed by an AECP command or that is responding to an AECP command. A number of AEM descriptors are defined in the IEEE 1722.1 specification [19].

As depicted in Figure 3.14, the AEM model is structured as a hierarchy of descriptors with the *entity descriptor* being the top-most descriptor. Each entity descriptor has any number of configuration descriptors, depending on the possible modes of operation of the entity.

An AVDECC controller that wishes to enumerate the AEM of an AVDECC entity will need to 'read' the descriptors, one level at a time, in order to get an overall view of the various features and controls that reside within the entity. Figure 3.16 shows how an '*AVDECC Controller'* can enumerate the AEM depicted in Figure 3.15.

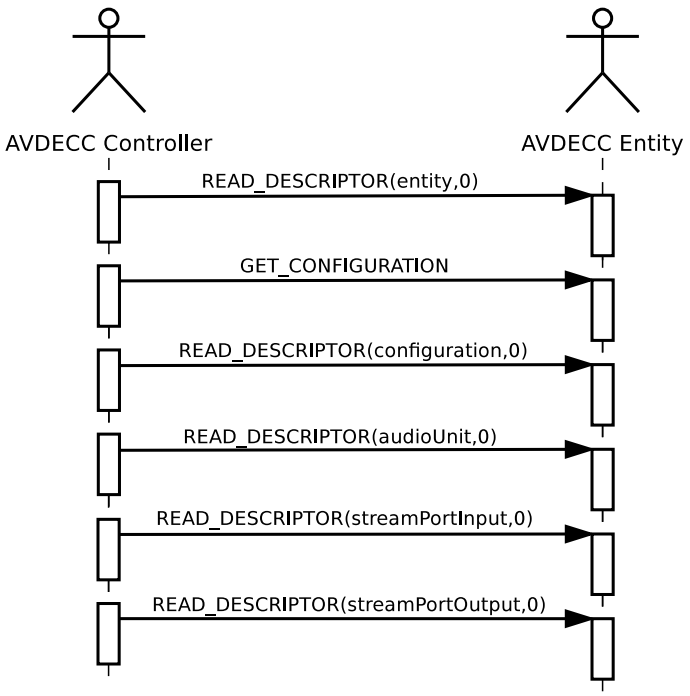


Figure 3.16: Example procedure for enumerating an AEM

As shown in Figure 3.16, the following messages are transmitted by the '*AVDECC Controller'* in order to enumerate the '*AVDECC Entity'*.

- READ_DESCRIPTOR command indicating the target descriptor type as *entity* descriptor ('*0x0000*') and the descriptor index as *'0'*. This command is used to obtain information about the possible configurations that exist within the entity. Its response will indicate that there is a *configuration* descriptor with index '*0*' within the entity descriptor.

- GET_CONFIGURATION command is used to determine the current operation mode of the AVDECC entity. In the case of the AEM depicted in Figure 3.15, it might not be necessary for the controller to issue this command because there is only one possible operation mode, which is represented by '*Configuration 0'*. However, this command will return the index (in this case '*0'*) of the current configuration in use.

- READ_DESCRIPTOR command indicating the target descriptor type as *configuration* descriptor (‘*0x0001*’) and the descriptor index as *‘0’*. This will return the types and the number of instances of each type of the top level descriptors within *‘Configuration 0’*. The returned values are shown in Table 3.4.

Descriptor type	Descriptor value	Number of descriptors
Audio Unit	*0x0002*	1
Stream Input	*0x0005*	1
Stream Output	*0x0006*	1
Jack Input	*0x0007*	1
Jack Output	*0x0008*	1
AVB Interface	*0x0009*	1
Clock Source	*0x000A*	1
Control	*0x001A*	1

Table 3.4: Response to enumeration of the configuration descriptor

The AEM of Figure 3.15 has one of each of the descriptors (at the top level) listed in Table 3.4.

- READ_DESCRIPTOR command indicating the target is the *audio unit* descriptor (‘*0x0002*’) with the descriptor index *‘0’*. The response to this command will provide information about the type and number of each top-level descriptor in the audio unit. This information is presented in Table 3.5.

Descriptor type	Descriptor value	Number of descriptors	Base index
Stream Port Input	*0x000E*	1	0
Stream Port Output	*0x000F*	1	0
External Port Input	*0x0010*	1	0
External Port Output	*0x0011*	1	0
Control	*0x001A*	2	1

Table 3.5: Response to enumeration of the audio unit descriptor

From the response shown in Table 3.5, besides the *control* descriptor which has two instances within the audio unit, each of the other descriptors have a single instance.

The audio unit descriptor also returns the lowest index (known as *base index*) of each of its descriptors. This makes it possible for a controller to accurately specify

the index of a descriptor it wishes to address. The values of the lowest descriptor index (for each descriptor) are also shown in Table 3.5.

- READ_DESCRIPTOR command indicating the target is the *stream port input* descriptor (‘*0x000E*’) with the descriptor index *‘0’*. In response to this command the entity will indicate that it has an *audio cluster* descriptor (‘*0x0014’*) and an *audio map* (‘*0x0017*’) descriptor, and that the base index of each of these descriptors is *‘0’*.

- READ_DESCRIPTOR command indicating the target is the *stream port output* descriptor (‘*0x000F*’) with the descriptor index *‘0’*. In response to this command the entity will indicate that it has an *audio cluster* descriptor (‘*0x0014’*) and an *audio map* (‘*0x0017*’) descriptor, and that the base index of each of these descriptors is ‘*1’*.

The structure of the READ_DESCRIPTOR and GET_CONFIGURATION commands and responses are described in the IEEE 1722.1 standard. These two commands have the values ‘*0x0004*’ and ‘*0x0007*’ respectively.

By utilizing the AECP commands defined in the IEEE 1722.1 standard it is also possible to modify the value of controls within an AVDECC entity. Figure 3.17 shows the sequence of commands that can be used to change the value of the control descriptor labeled ‘*Control 2 Volume’* of Figure 3.15.

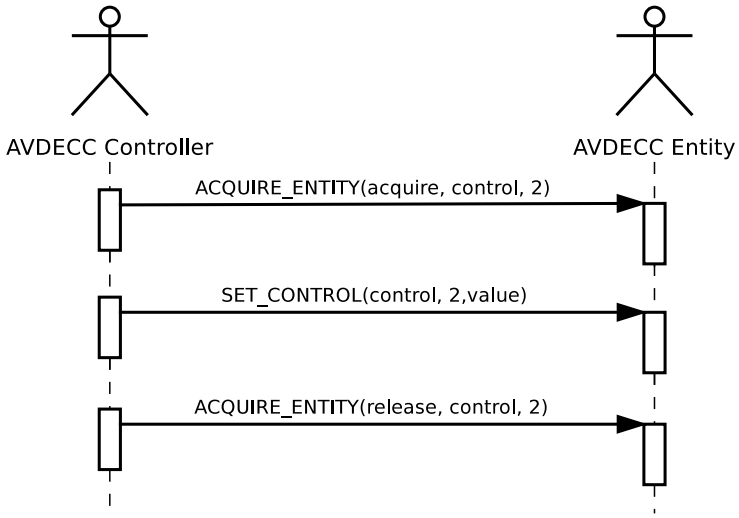


Figure 3.17: AECP commands to change value of AEM control descriptor

The layout of the AECP commands are described in the IEEE 1722.1 specification [19, pp. 172].

In order to change the volume of the control descriptor with index '2' (refer to Figure 3.15), the following AECP commands are issued.

- ACQUIRE_ENTITY command is used to obtain a lock to the descriptor, thus restricting access to modifying the descriptor. This command is sent to the *'AVDECC Entity'* with the *flag* field value of *'0x0000 0000'*, which indicates that it is an acquire command (and not a release command). The *descriptor type* field is set to *'0x001A'* which indicates that it is a CONTROL descriptor, and the *descriptor index* field is set to *'2'* which identifies the particular descriptor.
- SET_CONTROL command is used to modify the value control descriptor. The *descriptor type* and *descriptor index* fields are the same as in the ACQUIRE_ENTITY command, that is *'0x001A'* and *'2'* respectively. The *value* field specifies the new value for the volume control. This command will cause the entity to adjust the specified volume to the new value.
- ACQUIRE_ENTITY command is used to release the lock on the control descriptor. Thus making it available to any other controller that wishes to acquire it. The value of the *flag* field for this release command is *'0x8000 0000'*.

The *'command type'* values for the above AECP commands are defined in the IEEE 1722.1 specification. The value for the ACQUIRE_ENTITY command is *'0x0000'*, and the SET_CONTROL command has its value as *'0x0018'*.

It is possible for any other controller on the network to read a descriptor that has been acquired by another. However, only the controller that acquired the descriptor can modify the descriptor.

3.4.3.2 Device discovery

The IEEE 1722.1 standard defines an *AVDECC Discovery Protocol (ADP)* that will enable an AVDECC entity to announce its presence on a network, announce its departure from the network, and discover other AVDECC entities on the network. In order to do this, ADP defines three message types, namely [19]:

- ENTITY_AVAILABLE - is multicast by an AVDECC entity at regular intervals to indicate its presence on the network.
- ENTITY_DEPARTING - is multicast by an AVDECC entity when it is gracefully leaving the network.

- ENTITY_DISCOVER - is multicast on the network to discover AVDECC entities on the network. An ADP ENTITY_DISCOVER message can be used to discover every AVDECC entity on the network, or to discover a specific AVDECC entity on the network. Whether an ENTITY_DISCOVER message is intended to discover every entity on the network or to discover a specific entity, the target(s) is/are expected to respond by sending an ENTITY_AVAILABLE message.

In order to transmit these ADP messages, the ADP protocol defines an *AVDECC Discovery Protocol Data Unit (ADPDU)*. The structure of the ADPDU is shown in the IEEE 1722.1 standard [19, pp. 26]. When transmitting an ADPDU, an entity specifies its 64-bits entity GUID. The other fields that are contained in an ADPDU include:

- *vendor_id* - is a 32-bit field that identifies the manufacturer of an AVDECC entity.
- *entity_model_id* - is a 32-bit field that can be used by a manufacturer to specify the model of the entity.
- *entity_capabilities* - is a 32-bit field that bitmaps the various features that are supported by the AVDECC entity. This includes indicating whether the entity supports AEM commands and responses, as well as indicating the class of IEEE 1722 streams supported by the entity.
- *talker_stream_sources* - is a 16-bit field that specifies the maximum number of simultaneous IEEE 1722 streams that the AVDECC entity is capable of transmitting.
- *talker_capabilities* - is a 16-bit field that bitmaps the talker features that are supported by the AVDECC entity. This includes indicating whether the entity has audio, MIDI, SMPTE or video stream sources.
- *listener_stream_sinks* - is a 16-bit field that specifies the maximum number of simultaneous IEEE 1722 streams that the AVDECC entity is capable of receiving.
- *listener_capabilities* - is a 16-bit field that bitmaps the listener features that are supported by the AVDECC entity. This includes indicating whether the entity has audio, MIDI, SMPTE or video stream sources.
- *controller_capabilities* - is a 32-bit field that bitmaps the controller features that are supported by the AVDECC entity. In the current draft of the IEEE 1722.1 specification, this field could indicate that an entity is an AVDECC controller, or that it is a layer 3 proxy.

- *available_index* - is a 32-bit field that differentiates between different cycles of the same entity. It starts from '0' and is incremented by '1' on subsequent transmissions. When the entity leaves the network, or after a power restart, this field is reinitialized to '0'.
- *as_grandmaster_id* - is a 64-bit field that specifies which IEEE 802.1AS grandmaster an AVDECC entity is synchronized with. Thus it indicates which IEEE 802.1AS time domain that a particular entity belongs to.
- *association_id* - is a 64-bit field that can be used to group multiple AVDECC entities into a logical collection.

An AVB network that supports the transmission of IEEE 1722 streams is shown in Figure 3.18.

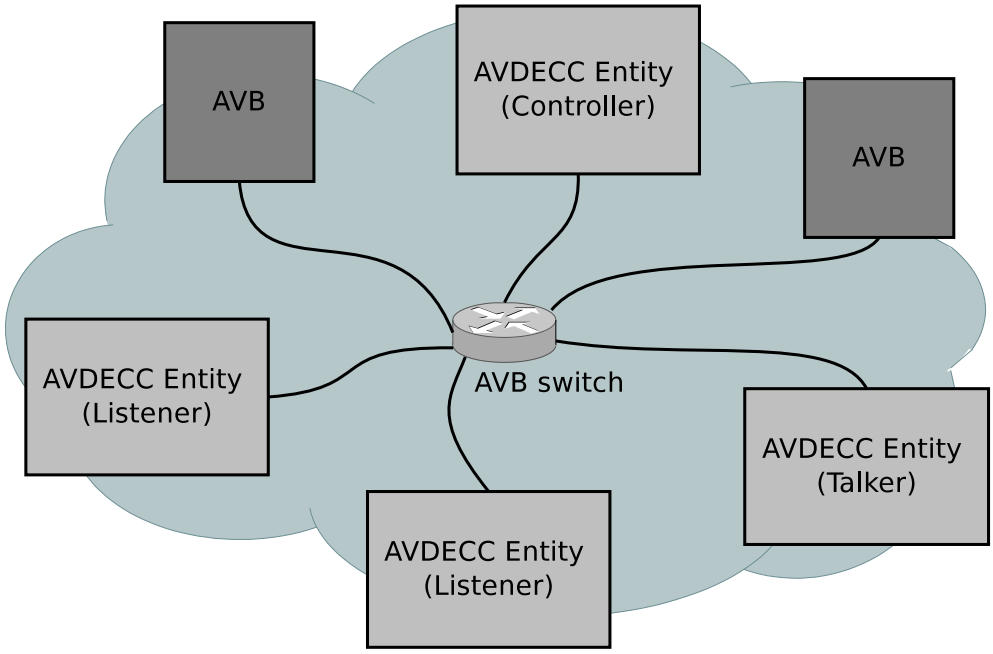


Figure 3.18: AVB network of AVDECC and non-AVDECC compliant devices

Figure 3.18 depicts an AVB network with six AVB endpoints within the same IEEE 802.1AS clock domain. Four of the AVB endpoints are AVDECC entities, and the other two AVB endpoints do not support the AVDECC protocol. When any of the AVDECC entities join the network, they will announce their presence by multicasting an *ENTITY_AVAILABLE* message. This message is only processed by the AVDECC entities (that is the *'AVDECC Entity (Controller)', 'AVDECC Entity (Talker)'* and the two entities labeled *'AVDECC Entity (Listener)'*) on the network. If at any point the *'AVDECC Entity (Controller)'* seeks knowledge of all the AVDECC entities on the network, it

will multicast an *ENTITY_DISCOVER* message. On receiving such a message, each AVDECC entity will respond by multicasting an *ENTITY_AVAILABLE* message. It is possible for the *'AVDECC Entity (Controller)'* to request that only a particular entity should respond to its *ENTITY_DISCOVER* message. It does so by specifying the entity GUID of the entity it is interested in. When an AVDECC entity is gracefully leaving the network, it will multicast an *ENTITY_DEPARTING* message to inform the other entities of its departure.

These ADP messages are transmitted as multicast on the network, and a multicast address has been reserved for ADP communication. This multicast address is *'91-E0-F0-01-00-00'*. ADP entities listen for messages whose *destination MAC address* (within the *Ethernet header* of Figure 3.12) has its value set to this multicast address.

In order to enable AVDECC controllers, listeners and talkers to respond appropriately to ADP messages, the IEEE 1722.1 standard defines a number of state machines. These state machines are:

- *Advertising Entity State Machine* - is implemented by an AVDECC entity to announce its presence on an AVB network. It performs these announcements at regular intervals, determined by the value of the *reannounceTimerTimeout* variable within the state machine. The value of the *available_index* field is incremented with each announcement. On receiving an *ENTITY_DISCOVER* ADP message from a controller, the advertising entity state machine responds as soon as possible, without waiting for its *reannounceTimerTimeout* to lapse.

- *Advertising Interface State Machine* - is implemented for each AVB interface of an AVDECC entity. This state machine utilizes the advertising entity state machine to send *ENTITY_AVAILABLE* ADP messages on the network. Unlike the advertising entity state machine, this state machine keeps track of changes to the IEEE 802.1 AS domain grandmaster, and also track whether the entity's interface link is connected or disconnected form the network. When either of these changes occurs, the advertise interface state machine updates (the value of) the corresponding state variable so that subsequent adverts on the network will indicate the updated value.

- *Discovery State Machine* - is implemented by an AVDECC entity that seeks knowledge of the other AVDECC entities on the network. The discovery state machine keeps a list of entities on the network, and it updates this list by:

  - adding a new entry to the list whenever a new entity announces its presence on the network

  - removing an entry from the list whenever the corresponding entity announces its departure (*ENTITY_DEPARTING*) from the network
  - modifying an entry in the list whenever a previously discovered entity advertises one or more new attributes within its *ENTITY_AVAILABLE* ADP message.

The advertising entity state machine and advertising interface state machine are typically implemented by AVDECC talkers and listeners. A discovery state machine is typically implemented by an AVDECC controller or any other AVDECC entity that requires AVDECC discovery.

3.4.3.3 Connection management

The IEEE 1722.1 standard defines a protocol for connection management known as the *AVDECC Connection Management Protocol (ACMP)*. The ACMP defines the procedure for establishing IEEE 1722 streams, as well as the procedure for destroying such streams. An AVDECC controller is able to utilize ACMP to set up an IEEE 1722 stream that is being offered by an AVDECC talker as the source, and set up a particular sink on an AVDECC listener as the destination of the stream.

ACMP defines a *protocol data unit*, called the *ACMP Data Unit (ACMPDU)*. The ACMPDU is a 44-octect frame, and its structure can be found in the IEEE 1722.1 standard document [19, pp. 263]. The ACMPDU includes the following fields:

- *controller_guid* - a 64-bit field that specifies the entity GUID of the controller that issued the ACMP command. It enables an AVDECC controller to match received responses with its original transmitted command.
- *talker_guid* - a 64-bit field that specifies the entity GUID of the target talker for which an ACMP is intended.
- *listener_guid* - a 64-bit field that specifies the entity GUID of the target listener for which an ACMP is intended.
- *talker_unique_id* - a 16-bit field that is used to identify a particular source stream amongst the various source streams on offer by a talker.
- *listener_unique_id* - a 16-bit field that is used to identify a particular sink stream amongst the various sinks that a listener possesses.

- *stream_dest_mac* - a 48-bit field that specifies the MAC address that should be used for transmitting an IEEE 1722 stream from a talker to one or more listeners.
- *connection_count* - a 16-bit field that provides some information about the probable number of stream connections that a talker assumes have been established with the stream identified by the *talker_unique_id* field. This value is the difference between the number of requests for connections and requests for disconnections, received by the talker entity. If a connection was pulled down without an explicit command being sent to the talker, the value of this (*connection_count*) field will be inaccurate. This could happen if a connection cable was removed unexpectedly.
- *sequence_id* - a 16-bit field that is used to match a command with its appropriate response. It starts with a value '0' and is incremented on each command transmitted, but is reinitialized to '0' after a power up.
- *flags* - a 16-bit field that is a bitmap used to indicate attributes of a particular connection. This includes indicating the type of connection mode being used.

The ACMP defines four connection management modes. These modes are:

- *Fast Connect* - is used by an AVDECC listener to re-establish a connection that was previously connected. This connection mode is used by a listener that is capable of storing its connections in non-volatile or rapid memory before departing from a network. On joining the network, the listener is capable of negotiating for the same stream(s) from the appropriate talker.
- *Fast Disconnect* - is used by an AVDECC listener to destroy a previously established stream connection when it is shutting down cleanly.
- *Controller Connect* - is the connection mode that is used by an AVDECC controller to establish a stream connection between a particular source stream on an AVDECC talker and a sink stream on an AVDECC listener.
- *Controller Disconnect* - is the connection mode that is used by an AVDECC controller to destroy a particular stream connection between an AVDECC talker and an AVDECC listener.

The *'message_type'* field of an ACMP message is used to specify the ACMP command type. Each command type has a timeout associated with it. If a command is issued and no response has been received within the expected timeout period, the entity that sent

the message may retransmit the same message a specified number of times. The number of retransmits depends on the particular role being fulfilled by the entity from where the message originated. That is whether the message originated from a controller, talker or listener state machine.

The ACMP messages are transmitted as multicast on the network. The multicast *destination MAC address* (shown if Figure 3.12) for ACMP transmission is the same as that for ADP (*'91-E0-F0-01-00-00'*).

To fulfill the roles of AVDECC controller, listener or talker, the ACMP protocol defines three state machines. The ACMP state machines are:

- *Controller state machine* - this state machine describes the behavior of an AVDECC controller that is involved in the connection management process. It describes what possible transition states the AVDECC controller could exist in, and how to respond to received commands at each state, as well as how to interpret received responses.
- *Listener state machine* - this state machine describes the behavior of an AVDECC listener that will sink an IEEE 1722 stream. It describes how the listener will behave when it receives connection management commands (from an AVDECC controller) to establish a sink stream, or to destroy a stream connection. It also defines how the listener is expected to respond to such commands.
- *Talker state machine* - this state machine describes the behavior of an AVDECC talker that is the source of one or more IEEE 1722 stream(s). It describes the various states and expected responses from a talker that is participating in the ACMP.

The *controller connect* and *controller disconnect* connection management modes are the typical way for setting up and pulling down (respectively) stream connections. In these modes of operation, an AVDECC controller initiates the connection management procedure. The particular ACMP commands that are exchanged between the AVDECC controller, listener and talker entities depend on whether a stream is being established or destroyed. However, the general procedure used by an AVDECC controller to establish or destroy a stream connection can be achieved in four steps. These are shown in Figure 3.19.

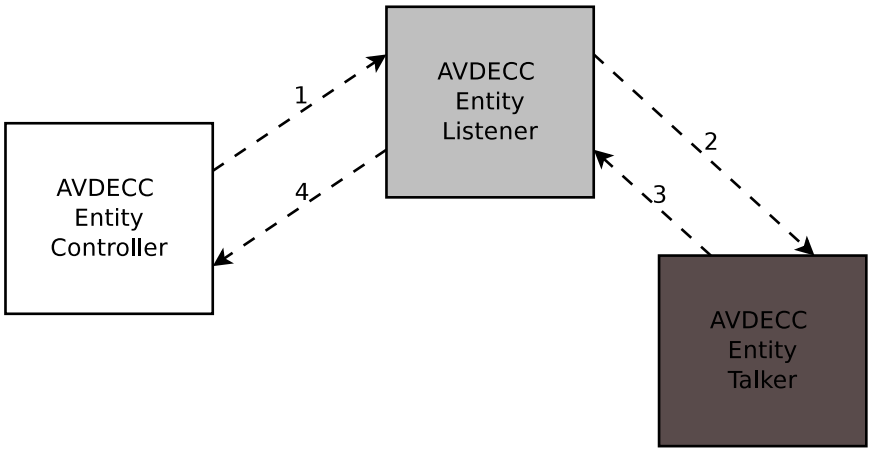


Figure 3.19: ACMP connection management procedure

In Figure 3.19 the AVDECC controller is labeled *'AVDECC Entity Controller'*, the AVDECC listener is labeled *'AVDECC Entity Listener',* and the AVDECC talker is labeled *'AVDECC Entity Talker'*. The steps in the connection management procedure shown in Figure 3.19 are:

1. The AVDECC controller utilizes its *controller state machine* to issue an ACMP command, and indicates whether an IEEE 1722 stream should be established (connect) or destroyed (disconnect) between the AVDECC listener and the AVDECC talker. The AVDECC listener and the particular sink (stream destination) on the listener are specified by the *listener_guid* and *listener_unique_id* fields (respectively) in the ACMPDU frame. Similarly the AVDECC talker and the particular stream source on the talker are specified by the *talker_guid* and *talker_unique_id* fields (respectively) in the ACMPDU frame. The controller will also specify its entity GUID in the *controller_guid* field of the ACMPDU.

2. The AVDECC listener's *listener state machine* processes the received command to verify that its GUID is specified in the command. Then it sends either a request for the stream (in the case of a 'connect' message) or an indication that it is no longer interested in the stream (in the case of a 'disconnect' message), to the AVDECC talker.

3. The AVDECC talker's *talker state machine* receives and processes the request for stream connection from the listener. If the received message is a request for connection to the stream (specified in the *stream_ID* field) that is not already being sinked to another entity on the network, the talker will register the stream on the AVB network via MSRP. If the message is to 'disconnect' a previously requested stream and the AVDECC listener is the only end station listening, the

talker withdraws the stream via MSRP. This will make the Ethernet AVB network resources that were being utilized by the stream available for other stream connections. MSRP has been described in chapter 2. Then the talker returns an ACMP response to the AVDECC listener indicating (in the *status* field) whether or not it was able to process the request. For a successful connect request, the talker will specify the multicast MAC address of the stream within the *stream_dest_mac* field of the ACMPDU.

4. The AVDECC listener's *listener state machine* receives and processes the response from the talker. If the response is for an earlier request for connection that was sent by the listener, the AVDECC listener proceeds to request attachment to the specified stream via MSRP. The AVDECC listener then sends a response to the AVDECC controller which started the connection management procedure. The message to the controller will indicate whether the connection management procedure was successful.

The above illustration assumes that each message is successfully transmitted and the appropriate responses are received within the defined timeout periods. Thus there are no retries of any of the ACMP commands indicated in the above sequence.

3.5 Summary

An audio control protocol defines a:

- message format for control and response information exchanges
- structured representation of the properties and features of compliant devices
- procedure for compliant devices to discover each other
- technique for enumerating the various controls and features within a compliant device
- procedure for establishing and destroying stream connections.

There exist a range of audio control protocols that enable remote device configuration, monitoring and control, and some of them have been described in this chapter. While some of the protocols have been designed to utilize OSI/ISO layer 3 transport protocols for messaging, as is the case with OSC and AES-64, others such as AV/C and

IEEE 1722.1 (AVDECC) utilize OSI/ISO layer 2 transport protocols for exchanging their messages.

There was a particular focus on the OSC, AES-64 and AVDECC protocols which were determined to be the viable protocols for this study on interoperability. From discussions of these three audio control protocols certain similarities can be noticed, particularly between the layer 3 IP-based protocols, that is between OSC and AES-64.

These similarities include:

- Device modeling - the three control protocols provide a hierarchical structure for modeling the controls and features that exist within a device. In particular OSC and AES-64 model a device as consisting of addressable control points called *OSC methods* and AES-64 *parameters*, respectively. The hierarchical structure for device modeling in AVDECC is called an AEM model, and it allows for remote control and monitoring by sending AECP commands to the appropriate *descriptors*.

- Application robustness - all three protocols enable loose coupling between what is application specific implementation, and what the protocol defines. The OSC protocol does this by enabling an application to perform specific tasks when an *OSC method* is dispatched. An AES-64 device is able to do this within the *call-backs* associated with each parameter. A similar mechanism is provided by the ACMP protocol's *processResponse* call, which enables a controller to perform device specific tasks when it receives an ACMP response [19, pp. 272].

- Message structure - the fundamental OSC and AES-64 messages posses a similar structure of the form:

  <*address to target control point*><*argument type*><*value*>

  In the case of OSC, this takes the specific form:

  <OSC address pattern><argument OSC type tag><argument value>

  With AES-64, it takes the form:

  <device ID + node ID + parameter address><value format>< value>

- Control architecture - a similar control architecture exists for the three protocols. Dedicated roles are defined for networked devices, where one device issues a request for service and another provides the service. In OSC this takes the form

of an *OSC client* requesting a service, and an *OSC server* providing the service. In AES-64 this takes the form of a *requester* (which could be a network controller) issuing a command, and a *target* fulfilling the request specified in the command. AVDECC implements a similar architecture, where an AVDECC *controller* issues discovery, enumeration, control or connection management commands and the targeted AVDECC *listener* or *talker* responds to the commands.

In spite of these similarities, networked audio devices can only interact when they implement the same control protocol. The analysis of protocols in this chapter has highlighted similarities, and thus provided an indication of the possibility of protocol command translation as a way to enable interoperability.

Chapter 4

Approaches to Networked Audio Device Interoperability

In the previous chapter, various audio control protocols were described. These ranged from layer 3 (IP-based) control messaging protocols (such as AES-64 and OSC) to layer 2 control messaging protocols (such as AV/C and IEEE 1722.1). An audio control protocol defines a particular message structure that is used for exchanging control commands and attribute information between networked devices. When audio devices are networked, communication is restricted to compliant devices. That is, only devices that implement the same audio control protocol are able to communicate with each other.

This chapter describes the problem known as the *'interoperability challenge'*, which arises when devices of different audio control protocols are interconnected. This is followed by a description of some of the approaches to providing a solution to the interoperability challenge. Finally, the chapter describes the command translation approach, which this research proposes as a solution to the interoperability challenge.

4.1 Control Protocol Interoperability Challenge

An audio networking technology provides the transport mechanism for exchanging digital audio data between networked devices. Such a technology should provide the necessary quality of service (QoS) required for time-sensitive data transmission. In chapter 2, some of the available audio networking technologies were described.

Above the audio transport technology is a control protocol that enables remote device configuration and monitoring of control points within an audio device. An audio device implements a protocol stack or parser that receives, transmits and processes messages

that are formatted according to the particular protocol. As a result, communication is restricted to devices that implement the same protocol. For example, since AES-64 and OSC allow for IP-based messaging over Ethernet links, devices that conform to these two protocols can be deployed on the same transport technology (for example an Ethernet AVB). The transport technology (Ethernet AVB) ensures that network resources can be reserved and that the time information (necessary for synchronization) can be exchanged between the networked devices. This enables deterministic and guaranteed delivery of the audio data between participating devices on the network. However configuring the AES-64 and OSC devices to stream audio between each other may require at least two controllers. One of the controllers will implement the AES-64 protocol to control the AES-64 devices on the network, while the second controller will implement the OSC protocol so as to enable configuring the OSC devices. This situation is depicted for an Ethernet AVB network in Figure 4.1.

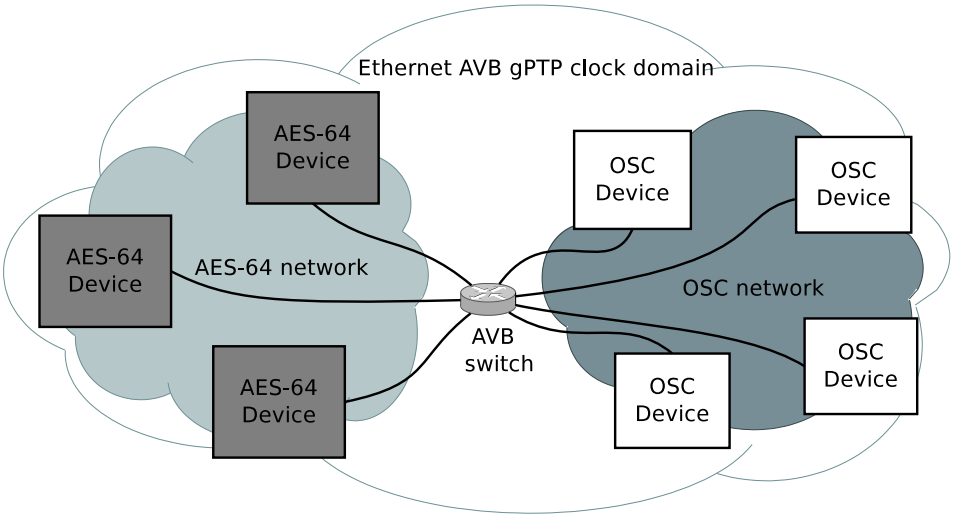


Figure 4.1: AES-64 and OSC devices are unable to communicate

The devices shown in Figure 4.1 reside within the same Ethernet AVB clock domain. This means that they are being synchronized by the same IEEE 802.1AS grandmaster. There are two audio control networks shown in Figure 4.1. One of the audio control networks consists of all the AES-64 devices, and the other audio control network consists of all the OSC devices. The devices shown in Figure 4.1 cannot communicate across the 'boundary' of their audio control protocol.

The above situation has been introduced as the *interoperability challenge* in chapter 1. In this case, audio stream connections cannot be established between interconnected devices because they implement different control protocols. This results from the inability to set up the stream connections by sending the same audio control messages to

the networked devices. In the scenario depicted in Figure 4.1, the networked devices cannot communicate even when they use the same transport technology because they implement different audio control protocols.

In order to control and monitor all the devices on the network (as depicted in Figure 4.1), a network controller application should be able to communicate using AES-64 and OSC control protocols. The situation gets more complicated when there are large numbers of audio control protocols implemented by the different networked devices. This would require that the network controller implements as many protocols as are represented by the devices on the network.

Typically, commercially available audio devices implement a single audio control protocol. Usually the controller application intended for such devices also implements a single protocol. Such a network controller will be unable to configure and monitor other devices that do not conform to the audio control protocol it implements.

The interoperability challenge currently prevails in the audio networking industry. As a result, audio engineers are restricted by the types of devices that can be purchased when designing an audio network. Even when there might be benefits to using devices that conform to a different audio control protocol, an audio engineer is hampered by the interoperability challenge, and forced to use only devices that conform to a particular audio control protocol. This research proposes a solution that will solve this problem.

4.2 Solutions for Interoperability

Various solutions to the interoperability challenge have been proposed, and in some cases implemented. This section describes some of these approaches.

4.2.1 Hardware abstraction plug-in approach - mLAN

The mLAN version 2 audio control protocol, which was described earlier in section 3.3.3 on page 72, is an IEEE 1394-based audio networking and control protocol [128]. It implements an *enabler/transporter* architecture, which is also referred to as the *plural-node* architecture. The plural-node architecture considered interoperability between mLAN devices and non-mLAN compliant devices.

Figure 4.2 depicts an mLAN enabler/transporter network that consists of an enabler and three transporter nodes.

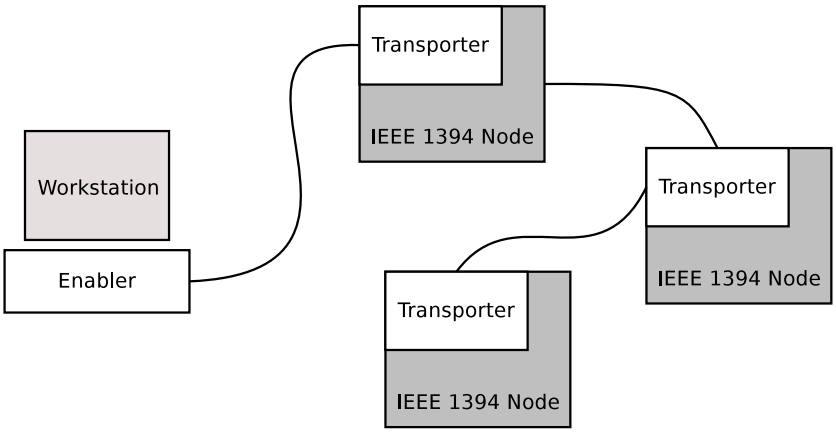


Figure 4.2: mLAN enabler/transporter network

An *enabler* resides within a workstation, which could be a Windows PC, Mac, or Linux workstation, and it is responsible for configuring and controlling the transporter (IEEE 1394) nodes. A *transporter* resides within the networked mLAN compliant nodes, and it enables device configuration by implementing and exposing device registers that can be accessed by the enabler via a *transport controller interface*. An enabler will typically control multiple transporters, but each transporter can only be controlled by a single enabler.

mLAN utilizes IEEE 1394 isochronous transactions for audio stream transmission, and asynchronous transactions are used to control messaging. On an mLAN network, the endpoints of an isochronous audio stream are abstracted as *mLAN plugs*. The mLAN plugs reside within the enabler, and they enable connection management. This separation of roles, whereby the enabler exists on a workstation for device monitoring and control and the transporter resides on the actual device, is referred to as the *plural-node* architecture.

In order to understand how the plural-node architecture provides for interoperability between mLAN compliant and non-mLAN compliant nodes, the mLAN enabler and transporter are described here.

mLAN enabler

An mLAN enabler consists of three layers that each contribute to its over all functionality of configuring transporter nodes. These three layers are [128]:

- *mLAN plug abstraction layer* - is the top layer of the mLAN enabler. The mLAN plug abstraction layer interfaces with an application on the workstation that utilizes the mLAN enabler. It defines an Application Programming Interface (API)

that an application can use to interact with the mLAN plugs. These mLAN plugs allow the application to manage device connections.

- *A/M manager layer* - refers to the audio and music layer of the mLAN enabler. It interfaces with the mLAN plug abstraction layer via a defined API. This layer is concerned with audio and music data parameter manipulations. Each transporter that is being controlled by the enabler, is abstracted as *transporter objects* within the A/M manager layer. The transporter objects make it possible for the enabler to maintain and monitor the state of the corresponding transporter node.

- *Hardware Abstraction Layer (HAL)* - this layer interacts with the A/M manager layer and the *transporter control interface* that resides within the transporter nodes. It abstracts manufacturer specific implementations of audio and music data encapsulation and extraction. The abstractions are implemented as plug-ins that are incorporated into the enabler.

The enabler is able to achieve interoperability between mLAN devices that are implemented by different manufacturers. This is precisely the function of the HAL layer. Each manufacturer incorporates a device-specific HAL plug-in for communication with the manufacturer's device.

mLAN transporter

An mLAN transporter implements a module known as the *mLAN node controller* that enables it to encapsulate and extract audio and music data within isochronous packets. An mLAN transporter consists of [128]:

- the *IEEE 1394 layer* - this implements the IEEE 1394 serial bus requirements, as well as the *mLAN unit directory*, which enables the identification of mLAN transporter nodes, and other software diagnostics.

- *Transporter control interface layer* - maps IEEE 1394 registers for audio and music data processing to particular addresses within the transporter. This enables IEEE 1394 asynchronous transactions to have the desired effects on the transporter's stream processing.

- *A/M protocol layer* - implemented in hardware (for example the Yamaha's NC1, PH1 [126] and PH2 chips [127]) or in software (for example the BridgeCo's DM1000 chip [35]) to enable the packetization and extraction of audio and music data according to the A/M specification [37].

An mLAN transporter may implement a *Connectionless Isochronous Transmission (CIT) manager* layer [35]. The CIT makes it possible for a transporter to save its plug configurations within non-volatile memory known as *boot parameter memory*, so that the transporter does not require the enabler to reconfigure its plugs after each power down.

The enabler/transporter architecture is designed such that the enabler, which is the control point for mLAN nodes on the network, can be utilized in order to attain interoperability with non-mLAN nodes. This is the situation depicted in Figure 4.3.

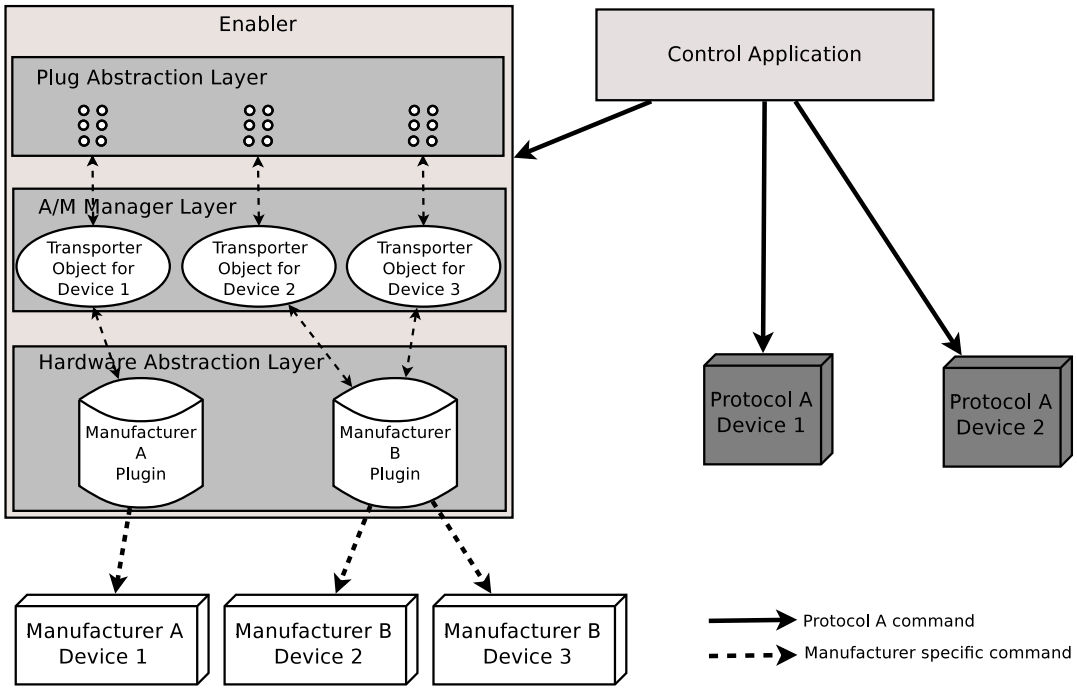


Figure 4.3: mLAN enabler/transporter architecture

In Figure 4.3, the *'Control Application'* is responsible for establishing and destroying stream connections between devices that implement a particular audio control protocol (referred to as Protocol A in the figure). This means that the *'Control Application'* is able to perform connection management between Protocol A nodes (*'Protocol A Device 1'* and *'Protocol A Device 2'*). With the assistance of the enabler, the *'Control Application'* is also able to perform connection management operations on the networked mLAN transporter nodes (*'Manufacturer A Device 1'*, *'Manufacturer B Device 2'*, and *'Manufacturer B Device 3'*). The *'Control Application'* transmits the appropriate Protocol A connection management commands to the devices on the network. The enabler ensures that the 'Protocol A' commands that are directed at the plug abstraction layer, result in the appropriate manufacturer specific commands being sent to the corresponding transporter node.

The enabler shown in Figure 4.3 controls the three mLAN transporter nodes. Two of these nodes are manufactured by the same hardware vendor (Manufacturer B), and the third node is manufactured by a different hardware vendor (Manufacturer A). Each of these manufacturers have created a HAL plug-in, which is incorporated into the enabler's HAL layer. At the enabler's *'A/M Manager Layer'*, three transporter objects are created. Each of these transporter objects corresponds to a particular transporter node under the control of the enabler. At the *'Plug Abstraction Layer'*, the enabler abstracts the end-points of audio sequences and MIDI subsequences in a manner that conforms to Protocol A. The enabler exposes these endpoints ('plugs') to the *'Control Application'* in the same manner that the Protocol A devices expose their stream connection endpoints.

The *'Control Application'* can establish audio stream connections by connecting the enabler's 'Protocol A endpoints' with the endpoints that reside within the Protocol A devices. In this way, the enabler is able to achieve interoperability between devices that implement the mLAN protocol and devices that implement other audio control protocols. The *'Control Application'* and the *'Enabler'* depicted in Figure 4.3 may reside on the same workstation.

4.2.2 Layer 3 common specification approach - AES-X192

The AES-X192 task group is currently developing a specification that will allow for interoperability between networked audio devices via IP-based messaging [11]. It utilizes existing protocols and standards. AES-X192 standardizes the procedure for:

- setting up and destroying audio stream connections
- exchange of timing information necessary for synchronization
- describing stream characteristics and encoding

Of particular interest to this research project is how stream connections can be remotely configured on networked audio devices such that audio that is transmitted by a source device, can be received by a destination device.

The AES-X192 project recommends the use of *Session Initiation Protocol (SIP)* for connection management [141]. SIP provides signaling for setting up or tearing down connections between participants known as *user agents (UA)* [142].

SIP interactions are between a *User Agent Client (UAC)* that sends a request, and a *User Agent Server (UAS)* that processes and responds to the request. Typically a single UA

is capable of fulfilling both roles. Each UA registers its location on a dedicated SIP server known as the *Registrar* server. This (registrar) server stores a database of all SIP services being offered on the network, and their corresponding location.

Associated with each UA is a *Uniform Resource Identifier (URI)* that can be used to identify the particular UA on the network [143].

The SIP URI recommended by AES-X192 is of the form:

sip:<username>@<host>

The '*sip:*' indicates that the URI is defined for the SIP protocol. The *'<username>'* identifies the particular instance on the *'<host>'*.

Each SIP response has a status code and reason phrase that are associated with it [142]. The status codes are defined in the SIP standard documents [141] [144] [145] [146]. The reason phrase provide a description of the code as human-readable text.

Figure 4.4 depicts communication using SIP.

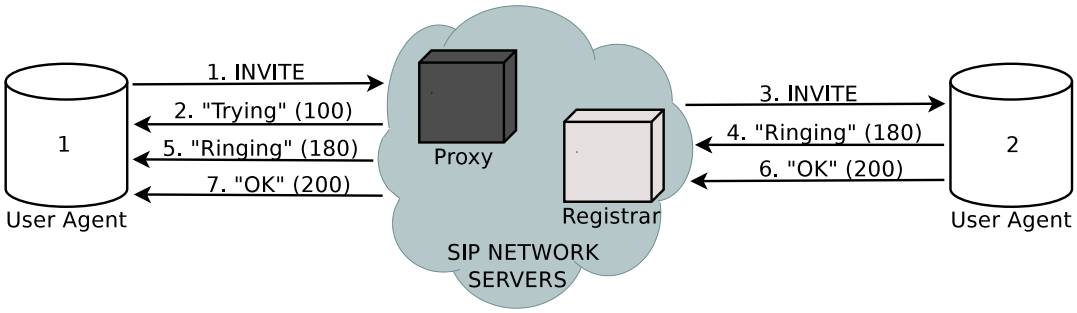


Figure 4.4: SIP communication between user agents

Figure 4.4 depicts a three-way handshaking approach to initiating a SIP connection. The steps are:

1. *'User Agent 1'* sends an *INVITE* request to the *'Proxy'* server. It will specify the SIP URI of the device it wishes to communicate with (that is the URI on '*User Agent 2'*).

2. The *'Proxy'* server sends a *"Trying"* response (with code 100) to '*User Agent 1'*, which indicates that it is processing the request. Then it queries the '*Registrar'* server for the location of '*User Agent 2'*.

3. After gaining knowledge of the location of the intended target, the *'Proxy'* server sends the *INVITE* request to '*User Agent 2'*.

4. *'User Agent 2'* sends a "Ringing" response (with code 180) to the *'Proxy'* indicating that connection is being established.

5. The 'Proxy' relays the "Ringing" response to *'User Agent 1'*.

6. *'User Agent 2'* sends an *"OK"* response (with code 200) to the *'Proxy'* which means that the connection has been established at *'User Agent 2'*.

7. The *'Proxy'* relays the *"OK"* response to *'User Agent 1'*. The "OK" message means that *'User Agent 2'* is ready to receive data.

The transaction illustrated in Figure 4.4 is the SIP request *INVITE* method, and it is used to establish a connection. The transaction for pulling down a connection is implemented by the SIP request *BYE* method.

SIP is transport protocol independent and it has been used with TCP and UDP protocols [147] [71]. It is widely deployed and understood by network administrators, thus making it an attractive solution for interoperability between devices on IP-based audio networks.

By defining a common connection management procedure, each AES-X192 compliant device is guaranteed interoperability with other (AES-X192) compliant devices on the network. The AES-X192 task group proposes that an X192 compliance mode be added to devices.

4.2.3 AVDECC Proxy Protocol

The IEEE 1722.1 standard defines an AVDECC Proxy Protocol (APP) that will enable AVDECC messaging (which is a layer 2-based messaging) across layer 3 networks [19]. This will ensure interoperability between AVDECC end stations across different network transport protocols, that is layer 3 and layer 2 transport technologies.

The APP approach allows for AVDECC interoperability over layer 3 transport technology. In order to enable cross-network communication, the APP defines two distinct roles between communicating devices. These are:

- *AVDECC Proxy Server (APS)* - an AVDECC entity that is capable of receiving ADP, ACMP, and AECP messages (refer to section 3.4.3) via layer 3 messaging, then forwarding the received messages to an AVDECC layer 2 network.

- *AVDECC Proxy Client (APC)* - an AVDECC entity that communicates via layer 3 AVDECC messaging, and utilizes the APS for layer 2 communication.

By utilizing an AVDECC Proxy Server (APS), an AVDECC end station that implements the APC state machine is able to communicate with other AVDECC end stations across an intermediate layer 3 network. This approach is intended for interoperability between networked AVDECC end stations.

APP allows forwarding of layer 3 messages to layer 2 (AVDECC) networks by defining an *AVDECC Proxy Protocol Data Unit (APPDU)*, which is used for APC and APS communication. The APPDU encapsulates ADPDU, ACMPDU or AECPDU (refer to section 3.4.3) protocol data units, as it is being transported across a layer 3 network.

The communication between AVDECC entities that utilize APP is represented in the form of a diagram in Figure 4.5.

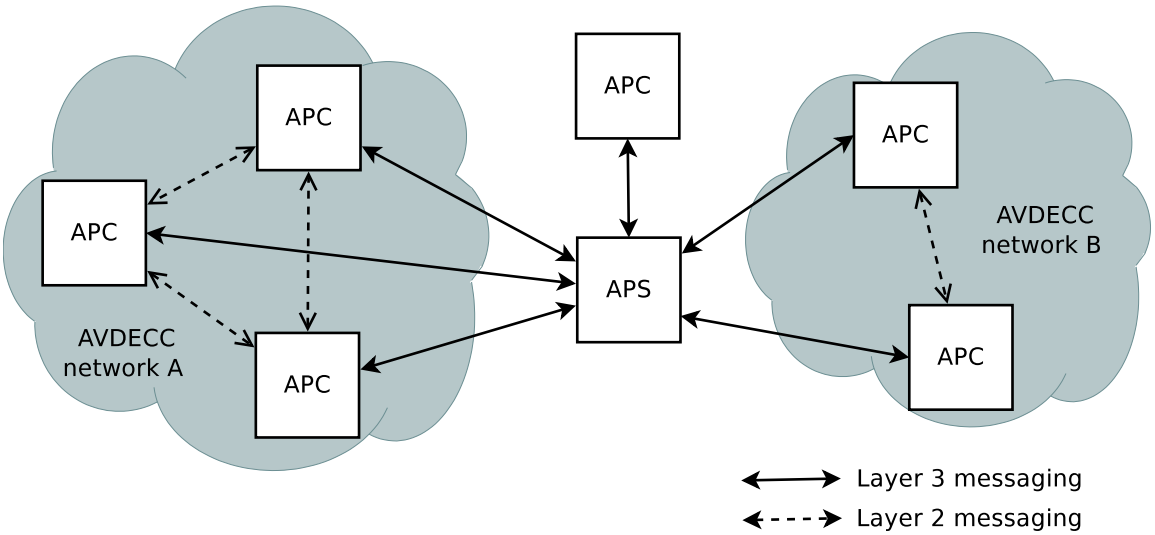


Figure 4.5: APC and APS communication

In Figure 4.5, *'AVDECC network A'* consists of three APC AVDECC end stations, and *'AVDECC network B'* consists of two APC AVDECC end stations. Communication between end stations that reside in the same network, is via layer 2 (AVDECC) messaging. In order for any of the APCs in *'AVDECC network A'* to communicate with an APC in *'AVDECC network B'*, it will encapsulate its message within a layer 3 APPDU message and address it to the APS. The APS is responsible for forwarding layer 3 APPDU messages between *'AVDECC network A'* and *'AVDECC network B'*. It is also possible to have a device, for example a web-based controller, that implements APC and is able to communicate with AVDECC end stations via the APS.

The APP defines two state machines that enable an APC or APS to appropriately respond to received messages. The *AVDECC proxy client state machine* implements the functionality of an APC, and an *AVDECC proxy server state machine* implements the functionality of an APS.

APP utilizes the HTTP CONNECT method to initiate the tunneling of (APPDU) messages between an APS and an APC [74].

An APC has to discover the available APS on the network. The IEEE 1722.1 standard recommends the use of DNS-SD for discovery. DNS-SD requires that a particular service type is being advertised and/or discovered on the network depending on whether the participating device is a client discovering one or more service(s), or a server publishing its service(s). A DNS-TXT record is used to provide further information about the service being offered. DNS-TXT record information is presented as key/value pairs.

An APS advertises a DNS-SD service type *'_avdecc._tcp'*, which utilizes TCP/IP for communication and a textual description of *"AVDECC Proxy"*. It uses port *'17221'* for communication.

4.3 Command translation for Interoperability

The previous section described some approaches to attain interoperability between networked audio devices. The mLAN approach involves the use of a central mLAN compliant network configuration and monitoring application called an enabler, which resides within a workstation. The enabler abstracts the endpoints of audio sequences and MIDI subsequences in order to conform to a protocol, and can thus be used for connection management of mLAN compliant transporter nodes and devices that comply with that protocol.

The AES-X192 approach requires that networked devices conform to the same (IP-based) network interoperability requirements. It recommends that networked audio devices implement SIP for connection management. The AES-X192 approach ensures that irrespective of the underlying audio protocol of the networked device, SIP can be used to establish and destroy audio streams on the device.

The AVDECC proxy protocol approach only allows interoperability between AVDECC end stations. It utilizes a client/server architecture, which uses HTTP to initiate tunneling of APPDU messages.

This research proposes the use of a protocol command translator to enable device interoperability. An initial investigation of this command translator approach was implemented on an IEEE 1394 network of AV/C and AES-64 devices. Those results were documented in a paper titled *"A Proxy Approach for Interoperability and Common Control of Networked Digital Audio Devices"* [148]. This research project provides further experimentation and implementations that include:

- Command translation between devices on an Ethernet AVB network where the devices implement layer 2 and layer 3 protocols.
- Centralized control of networked devices that conform to different audio control protocols
- Quantitative analysis of the protocol command translation approach

The command translator acts as a proxy that enables communication between devices that implement different audio control protocols. The proxy enables devices conforming to protocol A to appear as if they were devices conforming to protocol B to an external controller. The proxy should be able to:

- discover the devices (conforming to protocol A) that it proxies,
- enable a controller that implements a different protocol (protocol B) to discover the proxied devices,
- receive protocol messages on behalf of the proxied devices,
- translate the received messages from one protocol to another,
- transmit the translated messages to the intended target.

Figure 4.6 depicts how a proxy is used for command translation between networked devices.

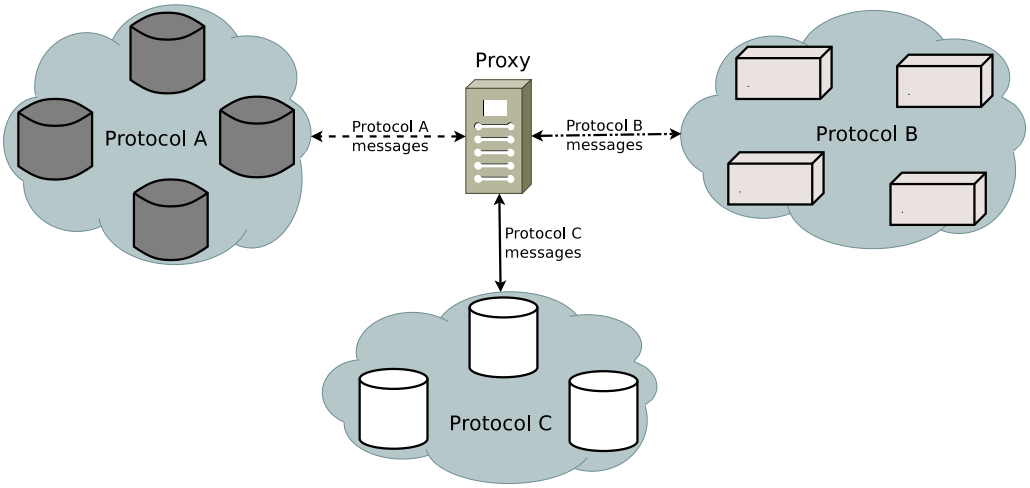


Figure 4.6: Command translation for interoperability

Figure 4.6 shows three networks that are distinguished by the audio control protocol they implement. The three networks are ‘Protocol A’, ‘Protocol B’, and ‘Protocol C’.

The objects within each of these networks communicate with each other by exchanging their own particular protocol messages.

In Figure 4.6, the 'Proxy' is used to translate control messages ('Protocol A messages', 'Protocol B messages', and 'Protocol C messages') so as to enable communication between the devices that conform to the three protocols.

The proxy (command translation) approach ensures that networked devices communicate using the particular audio control protocol that they implement. When compared with the plural-node architecture, this approach eliminates the need for a manufacturer of an mLAN transporter to create a HAL for their devices (refer to section 4.2.1). Secondly, it eliminates the need to modify the enabler so that it can abstract the stream endpoints in a manner that will allow the network controller to interact with them.

The proxy is able to translate commands between audio control protocols that are implemented on different transport technologies. This approach avoids the requirement that networked devices conform to the same transport protocol and implement the same procedure for connection management, as is the case with the AES-X192 approach to interoperability (refer to section 4.2.2).

The command translation approach implemented by the proxy enables communication between devices on layer 2 and layer 3 networks as is the case for APP. However it does not place restrictions on the particular audio control protocol that should be used for communication between the networked devices and the proxy, as is the case of the APP approach.

The proxy makes it possible for networked devices to discover other devices that implement disparate audio control protocols. It makes it possible for a single network controller to configure all of the networked devices, since the messages of the network controller will be appropriately translated by the proxy, as they are forwarded to their targets.

4.4 Summary

An audio control protocol enables remote monitoring and configuration of networked devices on an audio network. Several audio control protocols are currently being used. Audio devices can only interact with other networked devices that implement the same control protocol. Hence interoperability between devices that implement different control protocols remains a challenge.

A number of audio control protocols have incorporated solutions that provide a way to enable interoperability with non-compliant devices. Yet these solutions impose certain restrictions. The mLAN approach requires the construction of a HAL by each manufacturer of a transporter, and the enabler can only provide an abstraction of one protocol. The AES-X192 and APP approaches require protocol conformance.

These restrictions can be overcome with the use of a command translator on the network. The command translator implements the audio control protocols of the networked devices, and should be capable of discovering and communicating with the networked devices according to their particular control protocol.

Chapter 5

Layer 3 end station implementation - OSC

This research project has proposed the command translation approach as a solution for interoperability between networked audio devices. A description of the approach has been provided in the previous chapter. In order to investigate the command translation approach, devices that implement different audio control protocols need to be networked together with the command translator. The Open Sound Control (OSC) protocol was selected as one of the audio control protocols for this investigation (the reasons for this have been provided in chapter 3).

OSC is a specification that defines message formating syntax for communication between devices. The communication between OSC devices conforms to the client-server architecture where the following roles are defined in OSC:

- *OSC client* - an OSC device that sends out a command or request for a service
- *OSC server* - an OSC device that executes the requested instruction or performs the requested task

Both roles (client and server) can be performed by the same device. That is, an OSC device can be both an OSC client (with its ability to transmit requests) and an OSC server (with its ability to process requests). For further information on OSC refer to chapter 3.

OSC is transport independent, and OSC messages can be encapsulated within OSI/ISO layer 3 Internet Protocol (IP) packets. Most OSC implementations utilize IP for OSC messaging. Hence OSC has been classified as a *layer 3 audio control protocol* (refer to chapter 3).

In this investigation, a decision was made to utilize Ethernet AVB for audio streaming since:

- Ethernet AVB is rapidly becoming the transport technology of choice for many audio/video manufacturers.
- There was access to workstation-based implementation of Ethernet AVB [1]. This Ethernet AVB implementation was used in this research to implement layer 2 and layer 3 protocols.

As a result, the Ethernet AVB technology was selected as the audio transport technology of choice. However at the time of this investigation there were no commercially available Ethernet AVB capable OSC devices. As a result, an OSC server that is capable of streaming audio on an Ethernet AVB network was created. The audio streams conform to the IEEE 1722 standard also known as the *Audio Video Transport Protocol (AVTP)* [49]. Devices that transmit IEEE 1722 streams are referred to as *AVTP end stations*.

The OSC server, which was created in this investigation, is an AVTP end station that is capable of performing the roles of *AVTP talker* and *AVTP listener*. An AVTP talker is the source of an audio stream on an Ethernet AVB network, and an AVTP listener receives an audio stream from an Ethernet AVB network. The term *'AVTP end station'* refers to a device on the Ethernet AVB network that is capable of fulfilling the roles of AVTP talker and AVTP listener.

This chapter provides an overview of the OSC server's design. It describes the various components that together make up the server. This is followed by some design and implementation details that describe the various aspects of the OSC server, including the:

- device discovery implementation with regards to the OSC server,
- OSC message handling,
- OSC address space that was created for the (OSC) server,
- procedures and features of the server that enable connection management, and
- internal routing of signals within the server.

The overall goal with this implementation, was to create an OSC server that is capable of transmitting and receiving IEEE 1722 streams on an Ethernet AVB network, as well

as enabling remote control of its connection management *OSC methods* in order to set up the IEEE 1722 streams.

In this chapter, the term '*AVB audio stream'* refers to audio streams that conform to the IEEE 1722 standard and that are being transported on an Ethernet AVB network. Also '*AVB*' refers to the Ethernet AVB networking technology that was described in chapter 2 on page 13.

5.1 OSC Server Overview

The OSC server, that was implemented as part of this research project, is an Ethernet AVB end station that is capable of transmitting and receiving IEEE 1722 audio streams.

The design of the (OSC) server focuses on three important aspects: device discovery, connection management and internal routing. These (with respect to the OSC server) are described below.

- Device discovery - refers to how the OSC server can be discovered by a controller or other OSC devices on the network.
- Connection management - refers to the procedure for making and breaking stream connections between the OSC server and a remote device.
- Internal routing - refers to the internal routing of audio between the various inputs and outputs of the OSC server.

In this investigation, a PC implementation of the OSC server was created. The implementation is aided by the C++ programming language, thus allowing for an object-oriented programming model. Figure 5.1 shows an overall layout of the OSC server implementation. As shown in the figure, there is an implementation platform, above which are three components (*Device discovery component*, *AVB component* and *OSC parser component*). Above these three components is the OSC service which implements the OSC methods that execute OSC requests.

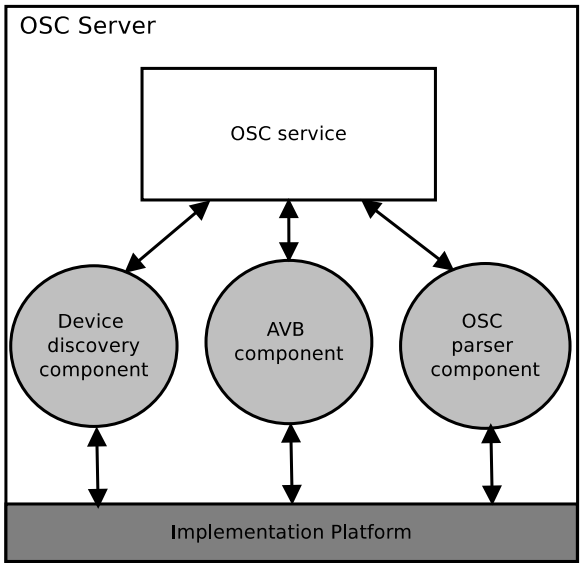


Figure 5.1: Overview of OSC server components

In Figure 5.1 the arrows illustrate the interactions between the various components and the OSC service (above them), and the implementation platform (below them). These are explained in the following sections.

5.1.1 Implementation Platform

The *Implementation Platform* forms the basis for the components above it (see Figure 5.1). The OSC server has been developed on *Ubuntu Linux* (*version 11.10* and kernel *version 3.0.0-14-generic*) as its platform. The platform provides an IP stack that is utilized by the service discovery and OSC parser components. It also provides the necessary interface for the Ethernet AVB (kernel) modules to interact on a network.

5.1.2 Device discovery component

This component provides a mechanism for informing other devices of the presence of the OSC server on a network. It also ensures that other networked devices are aware of the OSC server's departure from the network. In the current implementation of the OSC server, the *avahi* library (*version 0.6.30*) is utilized by this component [149].

Avahi is an implementation of the zero configuration networking technology [149]. Zero configuration networking includes multicast DNS (mDNS) and DNS Service Discovery (DNS-SD). mDNS enables networked devices to acquire an IP address without manual configuration or the use of a dedicated DNS server. DNS-SD provides a mechanism for devices to discover available services on the network.

5.1.3 AVB component

The *AVB component* handles AVB interactions between the OSC server and the Ethernet AVB network. There are Linux kernel modules that can be utilized to implement Ethernet AVB capabilities. These modules are [1]:

- *kmrp* module - implements the Multiple Registration Protocol (MRP) as defined in IEEE 802.1ak [50]. MRP enables applications to declare and register attributes, it also maintains the state of the attributes on a bridged LAN.
- *kmmrp* module - implements the Multiple MAC Registration Protocol (MMRP) defined in IEEE 802.1ak [50]. It is an MRP application that is used to register and deregister MAC address information on a bridged LAN.
- *kmvrp* module - implements the Multiple VLAN Registration Protocol (MVRP) defined in IEEE 802.1ak [50]. It is an MRP application that enables end-stations and bridges to declare and withdraw attributes when joining or leaving a VLAN.
- *kmsrp* module - implements the Multiple Stream Registration Protocol (MSRP) defined in IEEE 802.1Qat [44]. MSRP utilizes MRP, MMRP, and MVRP to declare stream attributes, which are used by the network to ensure that the necessary resources are reserved for the transmission of the stream.
- *faq* module - implements the Forwarding and Queuing procedure defined in IEEE 802.1Qav [45]. This module implements a buffer that holds Ethernet frames which are destined for transmission via an Ethernet interface.

The specific implementation of these Linux kernel modules are described in [1].

All interactions between these modules and the server is via this component. The AVB component also ensures that the encapsulation of audio streams is in accordance with IEEE 1722.

5.1.4 OSC parser component

The *OSC parser component* is responsible for interpreting received OSC messages, and formulating OSC messages in response to a received query. All OSC messages exchanged with the OSC server are encapsulated within UDP/IP packets. The OSC parser component is responsible for triggering the appropriate OSC method (within the OSC address space) that matches a received OSC address pattern.

5.1.5 OSC service

The *OSC service* implements the OSC server's OSC address space and combines the functionalities of the three components, in order to meet the requirement of the OSC server. For instance in order to execute a request from a remote OSC client to advertise it's available (IEEE 1722) streams on the network, the OSC server receives the message (via the OSC parser component), and parses it to trigger the appropriate OSC method within its OSC address space. The OSC method will send an 'advertise stream' request to the relevant Ethernet AVB module (via the AVB component).

5.2 OSC Server capabilities

The OSC server is capable of:

- Receiving an AVB audio stream connection, thus performing the role of AVB listener
- Transmitting audio via an AVB stream connection, thus performing the role of AVB talker
- Internal routing of audio, including routing:
  - analog audio input plug to IEEE 1722 output plug
  - IEEE 1722 input plug to analog audio output plug

The term *plug* as used here refers to the endpoint of an audio connection. A plug will consist of the source or destination of a mono or stereo audio channel.

5.3 OSC Server Implementation Layout

The three components depicted in Figure 5.1 give an overview of the various functional units that together form the OSC server. This section describes in detail the OSC server implementation.

Figure 5.2 depicts the OSC server in the form of a class diagram.

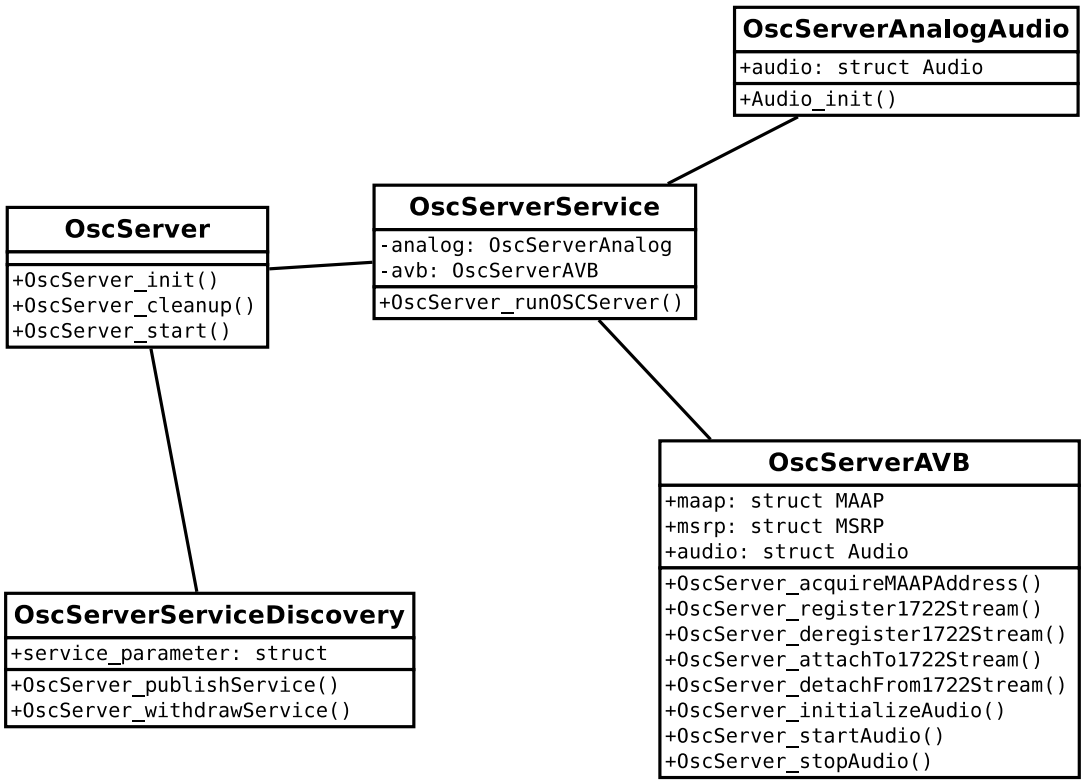


Figure 5.2: OSC server class diagram

The initialization and starting of the OSC server is implemented in the *OscServer* class of Figure 5.2. At startup, the *OscServer_init* method of the *OscServer* class is called to create an instance of the *OscServer*. This instance (of the *OscServer*) is started by calling the *OscServer_start* method. In order to properly shut down the server, an *OscServer_cleanup* method has also been defined within the *OscServer* class.

The device discovery component (of Figure 5.1) is implemented by the *OscServerServiceDiscovery* class. The *OscServerService* class abstracts the OSC service component of Figure 5.1. The *OscServer_runOSCServer()* method within the *OscServerService* class starts up the OSC parser component (of Figure 5.1). The *OscServerAVB* class abstracts the AVB component, thus enabling AVB interaction between the server and the Ethernet AVB network. An *OscServerAnalogAudio* class has been created as an abstraction that handles stereo analog audio input and output plugs.

The following sections provide a detailed description of the various classes (shown in Figure 5.2).

5.4 Device Discovery

Device discovery refers to the mechanism used by networked devices to discover each other. Within a network of audio devices, a controller requires knowledge of the other devices it is capable of controlling/configuring. These will be all the network devices that conform to the same control protocol as the controller.

In this implementation, DNS service discovery (DNS-SD) is the device discovery mechanism used. DNS-SD allows for a networked device to announce its presence when it joins a network, and announce it's departure when it leaves the network (*gracefully*). Graceful departure refers to a device adhering to the proper procedure for leaving a network. This is in contrast to a situation where the power plug on a networked device is accidentally pulled out.

The OSC server uses *avahi* for its DNS-SD implementation [149]. *Avahi* is a Linux library (with API) that allows for publishing and browsing of DNS-SD services. Each service has a particular *service type* by which it is identified. Additional information about the service is obtained from the DNS records. DNS-SD is described in detail in section 3.4.1.2 on page 80.

The *OscServerServiceDiscovery* class of Figure 5.2 implements device discovery within the OSC server. This class exposes two functions (*OscServer_publishService* and *OscServer_withdrawService*) to the *OscServer* class. The *OscServer_publishService* is used to announce that a device with a particular service type is now available on the network. A controller that is listening for announcements of this service type (on the network) gets this information and can probe the device for further information. For instance it could proceed to discover the features, capabilities and controls within the device. This process is called device enumeration.

The *OscServer_withdrawService* is used to announce that the device is leaving the network. A controller can use such information to update its list of available devices.

In order to use the two functions mentioned above, an object of the *service_parameter* data type (shown in Listing 5.1) is passed to the appropriate function. This *service_parameter* is used to provide device discovery specific information (device name and UDP port number) that can be retrieved by a remote device via DNS-SD.

```
struct service_parameter{
        char* name;
        int port;
};
```

Listing 5.1: Data structure for service type

The variables in the *service_parameter* data type comprise minimal the device information, which is published on the network, thus making this information accessible to a remote controller. The *name* field holds a reference to a descriptive name used to identify the device, and the *port* is used to specify the UDP port on which the OSC server is listening for OSC messages. Following this it is possible to enumerate further device information.

The publishing and withdrawing of services make use of the *avahi* library, and are described in the following subsections.

5.4.1 Publishing of OSC server

The publishing of an OSC service is implemented by the *OscServer_publishService* function, which is passed an instance of the *service_parameter* data structure (refer to Listing 5.1). This function makes use of the *avahi* API following the steps below:

1. It creates an instance of the *avahi* client with the *avahi_client_new* method. This client is used to get a handle on the avahi library. Associated with the client is a callback that is triggered after initialization of the handle to the *AvahiClient*, and this callback incorporates an error handling mechanism.

2. It creates an instance of the *AvahiEntityGroup* using the *avahi_entity_group_new* API function, and associates it with a callback that is triggered whenever there is a change in the entity group.

3. It specifies details of the OSC service and adds it to the entity group (created above) by utilizing the *avahi_entry_group_add_service* method of the avahi API. Such details include the:

   - service type - which in this case is *'_osc._udp'*
   - name - is the *name* field in the *service_parameter* of Listing 5.1
   - port - is the *port* field in the *service_parameter* of Listing 5.1

4. It registers the service by calling the *avahi_entity_group_commit* API function.

At the end of this process all controllers browsing for the *'_osc._udp'* service type, will be informed that the OSC server is available on the network. There is an Avahi capability to 'resolve' a service type announcement. Refer to the avahi API for more information about the functionalities of the avahi library [150].

5.4.2 Withdrawing of OSC service

The *OscServer_withdrawService* method of the *OscServerServiceDiscovery* class enables an entity to withdraw its service. This method withdraws the (*'_osc._udp'*) service by announcing its departure from the network, following steps below:

1. It frees the entity group by calling the *avahi_entity_group_free* API method.
2. It deletes the *AvahiClient* by calling the *avahi_client_free* API method.

At the end of this process all controllers on the network will be informed that the OSC server has left the network. This is accomplished via a browser callback that is associated with each avahi client that is browsing for OSC services on the network. The browser callback of such a client (avahi client browsing for OSC services) is triggered whenever a departure announcement is made on the network.

5.5 OSC Address Space for OSC Server

In OSC the entire hierarchical layout of the various paths from *root* node to *leaf* nodes within an OSC server is known as its *OSC address space*. The root node is the topmost node of the hierarchy and the leaf nodes are the *OSC methods* that form trigger points. When an OSC client requests a service or directs a query at an OSC server, it sends an OSC message that is addressed to an OSC method (leaf node on the address space hierarchy). OSC provides a mechanism for a single OSC message to address multiple OSC methods.

The OSC address space created for the OSC server can be classified into three categories:

- OSC generic properties
- Device properties
- AVB properties

The OSC address space is implemented in the *OscServerService* class of Figure 5.2. When the *OscServer_runOSCServer* method of the *OSCServerService* class is called, the OSC address space is created and the OSC server waits for its OSC methods to be triggered.

The following subsections describe the OSC address space based on the three categories mentioned.

5.5.1 OSC address space for OSC generic properties

This category includes the OSC methods that reveal information about the OSC protocol implemented by the OSC server. Currently the OSC server implements two methods in this category. These are shown in Listing 5.2.

```
/osc/version
/osc/ping
```

Listing 5.2: OSC address space for OSC generic information

The *version* method is used to inquire about the OSC version implemented by the OSC server. The server implements version 1.1 of the OSC specification [96].

The *ping* method is used to detect the availability of the server on a network. This could be used by a controller to probe whether the server is still accessible or active.

5.5.2 OSC address space for device properties

The OSC methods within this category are those that provide device specific information about the OSC server. These include information such as the server's name, it's IP address, and the total number of audio inputs on the server. Listing 5.3 shows the OSC methods in this category.

```
/device/name
/device/type
/device/ip
/device/sources
/device/sinks
/device/source/name
/device/sink/name
/device/sink/source
```

Listing 5.3: OSC address space for device specific information

The OSC methods referred to in the following discussion refer to those listed in Listing 5.3.

The *name* method is used to obtain the name of the OSC server. The server will return a string that contains its name whenever this method is triggered. This value can also be set remotely, in which case the remote controller will provide the new name as an argument (of type string) to the *name* OSC method. Listing 5.4 depicts such a message.

```
/device/name <name>
```

Listing 5.4: OSC message to set the OSC server's name from a remote controller

The *type* method is used to determine the functionality of the server. Currently the OSC server responds by returning a value that indicates that it is an AVB end station. This is a read only value, and cannot be changed remotely by a controller.

The *sources* and *sinks* methods are used to determine the total number of audio outputs and the total number of inputs, respectively. These are read-only values, that is they cannot be changed remotely. These totals (number of outputs and number of inputs) include both the analog audio and AVB audio streams.

The *source/name* method returns the name of a particular audio source (output), and the *sink/name* method returns the name of a particular audio sink (input). Each input and output on the server has a name assigned to it for easily identification. To obtain the name of a particular audio input or output, an *index* of the input or output must be specified as an integer argument within the OSC message. The index starts from '1' to the total number of inputs or outputs on the server. Listing 5.5 depicts an OSC message to obtain the name of an output on the OSC server.

```
/device/source/name <index>
```

Listing 5.5: OSC message to get the name of a source

The names of the audio inputs and outputs can be set remotely by a controller. To do this a second (string) argument is included in the message to the *name* method. Listing 5.6 shows the syntax of an OSC message that modifies the name of an output on the server. Changes to the names of inputs or outputs on the server are non-persistent, hence are lost when the OSC server is restarted.

```
/device/source/name <index> <newName>
```

Listing 5.6: OSC message to set a source name

The *sink/source* method is used for internal routing within the server. The method (*sink/-source*) is described in section 5.7.

5.5.3 OSC address space for AVB properties

The OSC methods in this category relate to the AVB properties on the OSC server. These methods are shown in Listing 5.7 on the next page.

```
/avb/sources
/avb/source/name
/avb/source/type
/avb/source/id
/avb/source/channels
/avb/source/format
/avb/source/start
/avb/source/stop
/avb/source/state
/avb/source/advertise
/avb/source/withdraw
/avb/sinks
/avb/sink/name
/avb/sink/type
/avb/sink/id
/avb/sink/channels
/avb/sink/format
/avb/sink/start
/avb/sink/stop
/avb/sink/state
/avb/sink/listen
/avb/sink/destroy
```

Listing 5.7: OSC methods for AVB stream information

Detailed descriptions of the OSC methods, which are depicted in Listing 5.7, are provided later in section 5.6.2 on page 143.

The following section describes how to establish or destroy a stream connection on the OSC server.

5.6 Connection Management

Connection management describes the procedure for establishing and destroying stream connections between networked devices. The OSC server can perform the role of AVTP talker, in which case it is the origin (source) of an audio stream on an Ethernet AVB network. The server can also perform the role of AVTP listener where it becomes the destination (sink) of an IEEE 1722 audio stream.

In order to establish an audio stream connection between an AVTP talker and AVTP listener, the following steps have to be followed [1]:

- The talker acquires a multicast MAC address using MAAP.
- The talker announces its AVTP audio stream by sending a *talker advertise* attribute while utilizing MSRP. This attribute contains the properties of the audio stream, and registers the stream on the network.
- A listener that requires the advertised stream sends a *listener ready* attribute (utilizing MSRP), which specifies the stream ID of the stream it wants to receive.
- When the talker receives a *listener ready* (or *listener ready failed*) attribute (via MSRP) and if the stream ID of this attribute matches one of its advertised streams, the talker starts to stream.

A set of OSC methods have been created in order to control this connection management process. Before describing these OSC methods, the design of the connection management capabilities (of the OSC server) is provided in the following section.

5.6.1 Implementing connection management capabilities in the OSC server

For the implementation of the roles of AVTP talker and listener, the *OSC parser component* and *AVB component* of Figure 5.1 are utilized. All OSC messages are parsed by the OSC parser component. The acquisition of a multicast MAC address, as well as the MSRP interactions are handled by the AVB component. A multicast MAC address enables multiple AVTP listeners to receive an audio stream from an AVTP talker. When setting up or destroying a stream connection, both of these components (OSC parser and AVB) are used. For instance, when an OSC message to advertise a stream is received by the OSC server, the following steps ensue:

- the OSC parser component scrutinizes the message in order to trigger the appropriate OSC method within the server's OSC address space. In this case it will be a call to the '/avb/source/advertise' method.
- the advertise method utilizes the AVB component to register its stream on the network. It does this by calling the MSRP module, which subsequently calls MRP to issue a talker advertise attribute.

Figure 5.3 shows the two classes of the OSC server's class diagram (Figure 5.2 on page 134) that are of interest to this discussion.

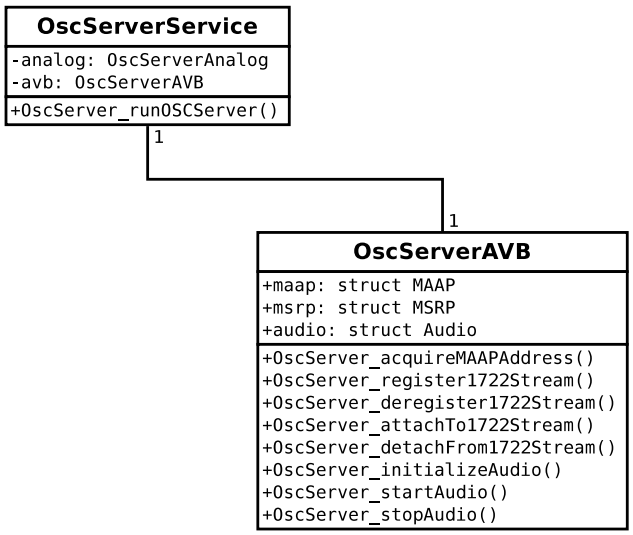


Figure 5.3: OSC server's classes for AVB interaction

The OSC parser component runs within the *OscServer_runOSCServer* method of the *OscServerService* class. It is this method that tallies (matches) received OSC messages with the appropriate OSC address patterns in the server's OSC address space.

The *OscServerService* constructor creates an instance of the *OscServerAVB* class which implements the AVB component. The *OscServerAVB* class implements methods for:

- acquiring a range of multicast MAC addresses by utilizing MAAP. This is implemented in the *OscServer_acquireMAAPAddress* method.
- network resource reservation (for its audio streams) by utilizing MSRP, which enables it to performs the roles of AVB talker or listener. Advertising and withdrawing of the available of AVB streams on the AVB talker are fulfilled by the *OscServer_register1722Stream* and *OscServer_deregister1722Stream* methods (respectively). The AVB listener role involves indicating interest or disinterest in a stream. This role is fulfilled by the *OscServer_attachTo1722Stream* and *OscServer_detachFrom1722Stream* methods.
- initializing the required audio buffers, as well as initiating and terminating the transmission of audio. These are fulfilled by the *OscServer_initializeAudio*, *OscServer_startAudio*, and *OscServer_stopAudio* methods respectively.

In order to enable connection management the OSC server implements a number of OSC methods. These methods are described in the following section.

5.6.2 OSC methods for connection management

The OSC methods referred to in the following discussion refer to those listed in Listing 5.7.

The *sources* and *sinks* methods are used to obtain the number of AVB output and input streams (respectively) that exist on the OSC server. In the current implementation, the OSC server transmits two AVB audio source streams and accepts a maximum of two AVB audio sink streams. These values (sources/sinks) cannot be modified by a remote controller.

The *name* methods return a string that describes a particular source or sink stream. A controller that seeks to determine the name of an audio source stream will send an OSC message of the form shown in Listing 5.8. To obtain the name of the first AVB audio source stream, the value of the *index* argument (in Listing 5.8) will be specified as '1', and to obtain the value of the second AVB audio source stream the *index* argument will be '2'.

```
/avb/source/name <index>
```

Listing 5.8: OSC message to get the name of an AVB source audio stream

A similar syntax is used to obtain the names of the AVB audio sink streams. The names of the stream sources/sinks can be changed by a remote controller. To do this, the controller specifies a name argument as depicted in Listing 5.9.

```
/avb/source/name <index> <name>
```

Listing 5.9: OSC message to set the name of an AVB source audio stream

The *type* methods are used to determine the type of AVB stream presented by the *sources/sinks* methods. This provides a means to determine the type of data being transported within the AVB stream. Hence by querying this value, a controller is able to distinguish between an audio and a video stream. In the case of the OSC server, the streams are audio AVB streams. In order to determine the stream type of a source or sink, the OSC message is passed an *index* (as an argument) to specify the particular source or sink. Listing 5.10 depicts the syntax of an OSC message to a *type* method within the *source* OSC container.

```
/avb/source/type <index>
```

Listing 5.10: OSC message to get the type of AVB stream

Each AVB audio stream has a unique stream identifier (stream ID). The *id* methods are used to determine the stream ID of a particular AVB audio source or sink. The stream IDs are 64-bit values that consist of :

- a 48-bit field obtained from the server's MAC address and hence unique to the server, and
- a 16-bit field that is a unique identifier for each stream, and hence distinguishes multiple streams from the same OSC server

The AVB stream IDs also provide a mechanism with which to associate talker and listener streams. The syntax of the OSC message used to obtain the stream ID of a source AVB stream is shown in Listing 5.11. The *index* argument specifies the particular stream of interest, and has a value in the range '1' to the number of *sources* ('2').

```
/avb/source/id <index>
```

Listing 5.11: OSC message to get the AVB source audio stream ID

The stream ID's of the source streams are acquired when the *OscServerAVB* class (of Figure 5.2) is initialized by the *OscServerService* class. Once the stream ID's have been set (at initialization), they cannot be modified. The OSC server acquires its 48-bit MAC address by utilizing the MAAP module, which forms part of the AVB component of Figure 5.1. . The 16-bit unique stream identifier is unique for each audio stream on the OSC server.

The format of the OSC message used to obtain the stream ID of an AVB sink stream is similar to that in Listing 5.11, except that it is addressed to the *sink* container. It is also possible to set the value of an AVB sink stream, although this is not supported for a source stream. This capability is required when setting up the OSC server as an AVB listener. The source stream ID on an AVTP talker will be the same as that of an AVTP listener that sinks the same stream. The syntax for setting a listener's sink stream is shown in Listing 5.12.

```
/avb/sink/id <index> <streamID>
```

Listing 5.12: OSC message to set a listener's sink stream ID

An AVB stream consists of a number of channels of audio. The number of channels within an AVB stream can be obtained from the *channels* methods. The syntax is shown in Listing 5.13, and a similar message can be addressed to the sink container. The *index* argument is in the range '1' to the number of streams, and it specifies a particular source or sink stream. The OSC server has two channels of audio within each of it's AVB source streams.

```
/avb/source/channels <index>
```

Listing 5.13: OSC message to get the number of audio channels within an AVB stream

The audio format of the AVB streams can be obtained by sending a message of the form shown in Listing 5.14. The *index* argument is in the range '1' to the number of streams, and is used to indicate the particular source stream. The OSC server transmits raw AM824 audio. A similar syntax is used to obtain the audio format of a sink stream.

```
/avb/source/format <index>
```

Listing 5.14: OSC message to get the audio format of an AVB stream

The *start* methods of the *source* and *sink* containers are used to start transmitting or receiving an audio stream. The *stop* methods are also defined to allow a controller to stop the transmission or reception of audio streams. The syntax for starting and stopping an audio source stream is shown in Listing 5.15. A similar syntax is used for the sink streams. The *index* argument is used to indicate the particular source/sink and has a value that ranges from '1' to number of sources/sinks.

```
/avb/source/start <index>
/avb/source/stop <index>
```

Listing 5.15: OSC message to start and stop an audio stream

To determine whether an audio stream has started, a controller transmits a message to the *state* method. This method can be used to inquire about a source or sink stream, and its syntax is as shown in Listing 5.16. A similar syntax is used to obtain the state of a sink stream. The *index* argument is used to specify a particular source/sink stream.

```
/avb/source/state <index>
```

Listing 5.16: OSC message to get the state of an audio stream

The *advertise* method of the *source* container causes the OSC server to advertise/register a particular stream on the Ethernet AVB network. If successful, this method results in the AVB network reserving adequate resources for the transmission of a stream. An *index* argument specifying the source stream to advertise, is passed to this method. The OSC message is of the form shown in Listing 5.17.

```
/avb/source/advertise <index>
```

Listing 5.17: OSC message to advertise an AVB stream

Similar to the *advertise* method, the *withdraw* method within the *source* container causes the server to indicate to the Ethernet AVB network that it is withdrawing an existing stream. This will cause the network to deallocate all previously allocated resources for that stream. This makes them (the resources) available to the network for any other stream request. The OSC message that triggers the *withdraw* method is of the form depicted in Listing 5.18. The *index* argument indicates which stream to withdraw.

```
/avb/source/withdraw <index>
```

Listing 5.18: OSC message to withdraw an existing AVB stream

On the *sink* container is a *listen* method that causes the server to request attachment to a stream that was previously advertised on the AVB network by an AVTP talker. The syntax of the *listen* method is shown in Listing 5.19. The *index* argument specifies which of the audio sinks should request attachment to a talker stream, and it (*index*) has a value from '1' to the number of audio sinks returned by the *sinks* method.

```
/avb/sink/listen <index>
```

Listing 5.19: OSC message to request attachment to an AVB stream

In order to stop listening, that is detach from a previously attached stream on the AVB network, the *destroy* method is implemented within the OSC server. A message to this method is of the form shown in Listing 5.20. The *index* argument is an integer value that indicates the sink that should be detached from a source audio stream.

```
/avb/sink/destroy <index>
```

Listing 5.20: OSC message to request detachment form an AVB stream

The following subsections describe how the OSC server utilizes the above methods in

fulfilling it's talker and listener features.

5.6.3 OSC server as AVTP talker

The OSC server has a number of OSC methods that enable it perform the role of AVTP talker. As an AVTP talker, the server becomes the source of AVB audio streams that conform to IEEE 1722. Listing 5.7 on page 140 shows a full listing of the server's AVB related OSC methods, but of particular relevance in this section are those (OSC methods) of the *source* OSC container. These methods are shown in Listing 5.21.

```
/avb/sources
/avb/source/name
/avb/source/type
/avb/source/id
/avb/source/channels
/avb/source/format
/avb/source/start
/avb/source/stop
/avb/source/state
/avb/source/advertise
/avb/source/withdraw
```

Listing 5.21: OSC server's AVB source methods

The OSC methods referred to in the following discussion refer to those listed in Listing 5.21.

The role of AVB talker as performed by the OSC server is described under the following headings:

5.6.3.1 Stream identification

The OSC server possesses two AVB source streams. A controller is able to determine the number of source streams available on the server by triggering the *sources* method. A call to this method will return the value '2' which is the maximum possible value of an index argument used to query the server's AVB streams.

When the *OscServerAVB* class is instantiated, it acquires a range of multicast MAC addresses by utilizing MAAP. The *OscServer_acquireMAAPAddress* method, shown in

Figure 5.3 on page 142, implements this functionality. The acquired multicast MAC addresses together with the 16-bit unique ID fields are used to generate two AVB stream IDs for the server's AVB streams. The first stream is assigned a value of '1' as the value of its 16-bit unique ID field, and the second stream is assigned the value '2' for the same field.

A remote controller is able to determine the stream IDs of the server's sources by sending a message to the *id* method with an index (value) argument. The index specifies which of the two streams it wishes to determine its stream ID. The index of the first stream is '1' and the second stream has an index value of '2'.

5.6.3.2 Stream enumeration

Each stream has a name associated with it. To get the name of a stream, the controller sends a message to trigger the *name* method. It must specify the index of the stream whose name it wants to obtain. The first stream has the default name of "1722 Output Stream 1", and the second stream's default name is "1722 Output Stream 2". These names allow the controller to properly present the stream on a graphical interface, if for instance the controller is running on a PC. Other properties such as the type of AVB stream and the audio formats are retrieved by calling the *type* and *format* methods. This information enables a controller to acquire more details about the sort of media that is transported in the AVB stream. The OSC server transmits two channels of audio within its AVB streams. A controller is able to determine the number of audio channels within each of the server's audio source streams by triggering the *channels* method.

5.6.3.3 Stream advertising

A controller can request the OSC server to advertise any of its source streams on the AVB network by triggering the server's source stream *advertise* method. This method causes the *register stream request* primitive of MSRP to be triggered for a particular stream on the AVB network. The *advertise* method does this by calling the *OscServer_register1722Stream* of the *OscServerAVB* in Figure 5.3 on page 142. If successful, this effectively results in the network reserving sufficient resources for the transmission of the specified stream. It also enables all AVTP listeners (on the network) to gain knowledge of the characteristics of the stream. A message that triggers the *withdraw* method ensures that the previously reserved network resources, for a particular stream, are released. This is achieved by calling the *OscServer_deregister1722Stream* of the

OscServerAVB class (of Figure 5.3), which calls the *deregister stream request* primitive of MSRP.

5.6.3.4 Stream transmission

Within the OSC server, audio signals are placed in buffers before transmission on the AVB network. These buffers are initialized when the *OscServerAVB_initializeAudio* method is called. The server implements two types of buffers for audio stream transmission. One buffer is for receiving audio (the *receive buffer*) and is utilized by the sinks on the server. The other buffer is for transmitting audio (the *transmit buffer*) and is utilized by the sources on the server.

The actual transmission of audio can be regulated (remotely) by a controller. The controller does this by sending an OSC message to either the *start* or *stop* OSC methods. The *start* method is used to commence the transmission of the audio from the server. The *stop* method is used to stop the transmission of audio from the server. These methods are associated with the source's transmit buffer.

As depicted in Figure 5.3, the *OscServerAVB* class implements the *OscServer_startAudio* and *OscServer_stopAudio* methods to start and stop (respectively) the transmission of audio within AVB streams.

The *state* method enables a remote controller to determine whether a particular source has started streaming audio.

5.6.4 OSC server as AVTP listener

The OSC server is capable of performing the role of AVTP listener. In accomplishing this role, the server acts as the destination (sink) of AVB audio streams that conform to IEEE 1722. A controller is able to configure the server to receive audio streams from the AVB network by sending OSC messages that match the OSC methods associated with the *sink* OSC container. The OSC methods of interest in this discussion are shown in Listing 5.22.

```
/avb/sinks
/avb/sink/name
/avb/sink/type
/avb/sink/id
/avb/sink/channels
/avb/sink/format
/avb/sink/start
/avb/sink/stop
/avb/sink/state
/avb/sink/listen
/avb/sink/destroy
```

Listing 5.22: OSC server's AVB sink methods

The OSC methods referred to in the following discussion refer to those listed in Listing 5.22.

The role of AVB listener as performed by the OSC server is described under the following headings:

5.6.4.1 Stream identification

The server implements the *id* method to enable a remote controller 'get' and 'set' any of its sink stream ID. The stream ID of a sink is set when a connection is being established (with the sink). If no connection has been established with a particular sink, a message to determine its stream ID will return a zero value (*0x00000000*). When a zero value is returned in response to a 'get' *id* method, a remote controller interprets this as an indication that the sink is not receiving an audio stream.

5.6.4.2 Stream enumeration

The OSC server implements two AVB audio sinks and the number of implemented sinks on the server can be retrieved by triggering the *sinks* method. The first audio sink stream is named "1722 Input Stream 1", and the second is "1722 Input Stream 2". The names of each stream can be obtained by triggering the *name* method with the index of the particular sink stream passed as an argument.

Although these streams are presented by the server, they only represent the maximum number of AVB sink streams that the OSC server can receive. Hence a query to determine the *state* of each stream (with index '1' for the first stream and index '2' for

the second) will indicate that the stream has not started and a *start* command will fail. In fact, any message to methods that enumerate a sink stream before a connection is establish (with an AVB stream) will fail. These include OSC messages to determine the:

- stream ID on a sink stream by triggering the *id* method,
- number of audio channels within a sink stream by triggering the *channels* method, and
- audio format by triggering the *format* method.

5.6.4.3 Stream attachment

In order to sink an AVB stream, the server requires knowledge of the stream's stream ID which uniquely identifies the stream on the network. This is achieved by triggering the *id* method on a sink stream, in order to set the sink's stream ID.

A message to the sink's *listen* method will cause the *OscServer_attachTo1722Stream* method (of the *OscServerAVB* class) to be called. This results in the MSRP *register attach request* service primitive to be declared with a *ready* declaration type.

The *destroy* method causes the server to indicate (via MSRP) that it has no interest in a particular stream. The *OscServer_detachFrom1722Stream* method is used to annouce this lack of interest in a stream. In this case the MSRP *deregister attach request* service primitive is declared on the network.

5.6.4.4 Stream reception

A call to a sink stream's *start* method will cause the server to begin buffering the received audio (on a particular sink) into its *receive buffer*. When the *stop* method is triggered, the server terminates the buffering of audio from the specified sink stream. The *OscServerAVB* class implements the *OscServer_startAudio* and *OscServer_stopAudio* in order to facilitates these two processes (starting and stopping the buffering of audio).

The sink's *start* and *stop* methods utilize the receive buffer which is initialized when the *OscServer_initializeAudio* method is called (by the *OscServerAVB* constructor).

5.7 Internal Audio Signal Routing

The OSC server has a stereo analog audio input and a stereo analog audio output that is abstracted by the *OscServerAnalogAudio* class of Figure 5.2. When this class is initialized by the *OscServerService* class, it gets a handle to the ALSA audio driver on the Linux PC [151]. The ALSA driver enables the server to capture audio from its stereo analog input, and playback audio on its stereo analog output.

Listing 5.23 shows the relevant OSC methods for routing audio signals between the inputs and outputs of the server.

```
/device/sources
/device/source/name
/device/sinks
/device/sink/name
/device/sink/source
```

Listing 5.23: OSC methods for internal signal routing

The OSC methods referred to in the following discussion refer to those listed in Listing 5.23.

As described in section 5.5.2, the audio inputs and outputs on the OSC server can be determined by triggering the *sinks* and *sources* OSC methods, respectively. Each of the inputs and outputs have a name associated with them, which can be retrieved by triggering the *name* method of the appropriate OSC container. The *name* method can also be used by a controller to set the names of the sinks or sources.

In order to allow for patching between the various inputs and outputs on the OSC server, a *source* method has been implemented within the sink OSC container. This method allows a controller to determine which output a particular input is routed to. Thus it allows the controller to either get the current input/output patch, or set the input/output patch.

A controller seeking knowledge of which output an input is currently patched to, will send an OSC message of the form depicted in Listing 5.24. The index is an integer value that ranges between ‘1’ and the maximum number of inputs obtained from the *sinks* method.

```
/device/sink/source <index>
```

Listing 5.24: OSC message to get the input/output patch

The value returned is an integer value which represents the index of the source. This value is between '1' and the number of sources obtained from the '/device/sources' method.

The *source* method within the *sink* OSC container can also be used to route signals between the inputs and outputs of the OSC server. The syntax for routing signals is shown in Listing 5.25.

```
/device/sink/source <inputIndex> <outputIndex> <enable>
```

Listing 5.25: OSC message to set the input/output patch

The *inputIndex* and *outputIndex* specifies the inputs and outputs that should be patched. The *inputIndex* and *outputIndex* have integer values between '1' and the values returned by the *sinks* and *sources* methods, respectively. The *enable* variable has a value of '1' to establish a connection, or '0' to destroy a connection.

5.8 Summary

OSC provides a mechanism for formating messages for transmission on a network, irrespective of the transmission technology. AVB networks are designed to ensure that time-sensitive data (such as audio) can be transmitted with the best possible quality of service by ensuring deterministic and guaranteed delivery of stream data. AVTP end stations are capable of transmitting audio streams that conform to the IEEE 1722 standard on an Ethernet AVB network.

This chapter described an OSC server that is capable of transmitting audio streams on an Ethernet AVB network. There was an explanation of the various components that together enable the OSC server to function as an AVTP end station. Also discussed was the server's OSC message handling mechanism, connection management on an Ethernet AVB network, as well as internal routing within the OSC server.

Chapter 6

Layer 3 Proxy Implementation

Following the creation of the OSC server, which has been described in the previous chapter, an Ethernet AVB network of AES-64 and OSC devices was setup in order to investigate the command translation approach (that has been described in chapter 4). Seeing that the audio streaming technology (Ethernet AVB) and networking infrastructure (Ethernet) were the same, there emerged a need to be able to stream audio between the networked Ethernet AVB end stations irrespective of the audio control protocol implemented by each end station. Furthermore there was a desire to have these networked end stations being controlled from the same network controller.

This chapter describes how the command translation approach, implemented as a proxy, can be used to enable common control, connection management and interoperability between devices that conform to different audio control protocols, in particular the AES-64 and OSC protocols.

6.1 Introduction

Ethernet AVB makes it possible for time-sensitive data, such as audio and video, to be reliably transmitted over Ethernet. However, to allow for networked devices to be remotely configured for streaming, each device implements a control protocol.

An audio control protocol will typically define its own syntax and semantics for commands and responses, as well as procedures for establishing and destroying stream connections. Very often, each equipment manufacturer implements a proprietary control protocol on their devices. As a result there currently exists a large number of disparate audio control protocols, given the wide range of audio equipment commercially available. Typically, each audio device only implements one control protocol. As a result,

although it is possible to physically network devices on the same networking technology (such as IEEE 1394, Ethernet, USB), there remains the problem of interoperability between devices that implement different control protocols.

This was the problem when the OSC server was connected to AES-64 devices on an Ethernet AVB network. Although AES-64 defines a procedure for connection management, and the OSC server implements a number of OSC methods that enable connection management, setting up audio stream connections between devices that conform to both protocols remained a challenge. Also required was a single controller that is capable of configuring both devices (AES-64 devices and OSC servers) such that audio can be streamed between them.

To overcome the interoperability challenge, which was described in chapter 4, this chapter descibes an implementation of the command translation approach in the form of a proxy. The OSC proxy implementation that is described in this chapter enables connection management and control of AES-64 devices and OSC servers on an Ethernet AVB network.

6.2 The Proxy Approach

An OSC proxy has been created to allow AES-64 messages to be translated to appropriate OSC messages in order to enable connection management and control of OSC servers via AES-64 messaging. This enables an AES-64 controller to configure an OSC server. The proxy also relays OSC responses (messages) to the AES-64 controller.

A logical layout of the interaction between an AES-64 device, the OSC proxy and an OSC server is provided in Figure 6.1.

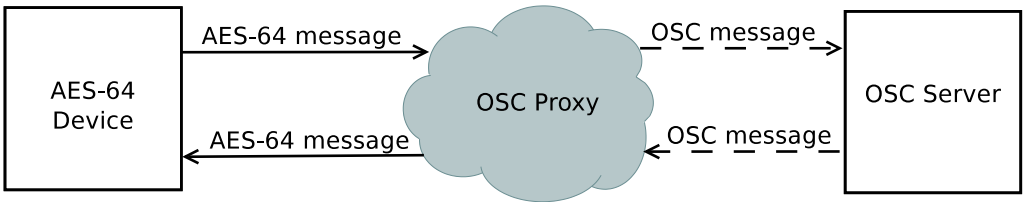


Figure 6.1: Logical layout of interaction with OSC proxy

The interaction depicted in Figure 6.1 is as follows:

- an AES-64 device issues an AES-64 message addressed to an OSC server,
- the OSC proxy receives the message on behalf of the OSC server, then translates it to an OSC message,

- the OSC proxy sends the translated OSC message to the appropriate OSC server,
- the OSC server receives and parses the OSC message, which will cause an OSC method to be called,
- if a response is required (for instance if the original AES-64 message was a *'get'* value message), the OSC server returns a response to the proxy,
- the proxy retrieves the response from the OSC message, then encapsulates it within an AES-64 message as a response, and
- finally the AES-64 response message is sent to the AES-64 device.

In the above interaction, the OSC proxy assumes the role of destination of the AES-64 messages and the source of the OSC messages, which are addressed to the OSC server. This ensures that all AES-64 messages (from the AES-64 device) and all OSC responses (from the OSC server) go via the proxy.

The proxy also enables a common AES-64 controller to control AES-64 devices and OSC servers on a network. The common control of networked AES-64 devices and OSC servers, with the aid of the OSC proxy, is illustrated in Figure 6.2.

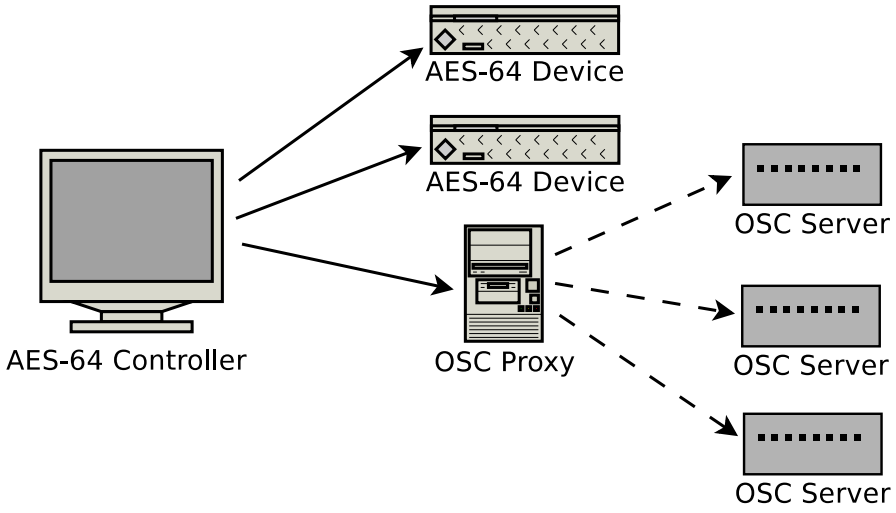


Figure 6.2: Common control of AES-64 devices and OSC servers

In Figure 6.2 the solid lines represent AES-64 messages, and the dotted lines represent OSC messages. The 'AES-64 Controller' sends the same AES-64 message to all AES-64 devices on the network. The proxy receives the message, translates it to the appropriate OSC message and then sends the OSC message to the OSC servers on the network. The 'AES-64 Controller' can control AES-64 and OSC Server devices, and enable streaming between the different device types.

The next section provides details about the requirements and design decisions that were considered when the proxy was being developed.

6.3 OSC Proxy Design

The OSC proxy was designed to enable AES-64 control of networked OSC servers. It is capable of receiving AES-64 messages, and then translating the received message to the appropriate OSC message(s), and vice versa. The design requirements of the OSC proxy are depicted in the form of a use-case diagram in Figure 6.3.

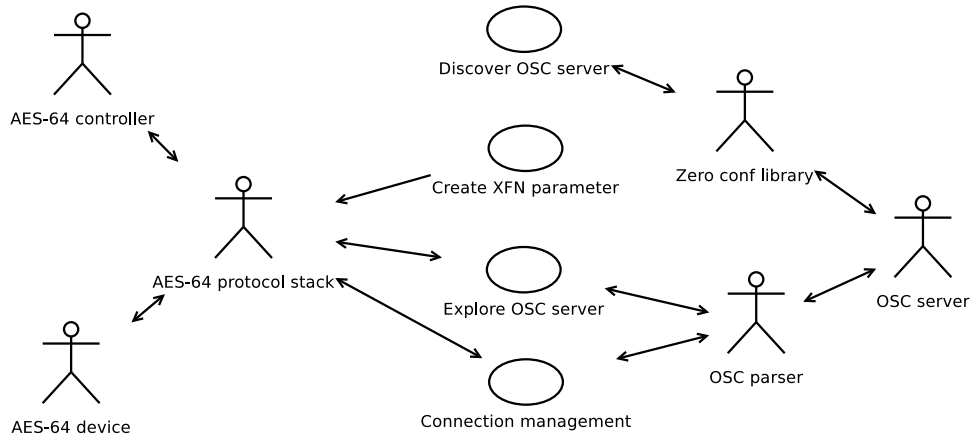


Figure 6.3: OSC proxy use-case diagram

From Figure 6.3 all AES-64 communication with the proxy is handled by the 'AES-64 protocol stack'. This includes messages from either the remote 'AES-64 Controller' or an 'AES-64 device' on the network. The proxy uses the 'OSC parser' for all OSC communication with the 'OSC server'. The parser encapsulates OSC methods within OSC packets for onward transmission to the 'OSC server'. It also causes the appropriate OSC method (within the proxy) to be called when a response is received from the 'OSC server'. The 'Zero conf library' is an implementation of zero configuration networking which includes DNS-SD (which has been described in section 3.4.1.2 on page 80), and it is used for discovering the 'OSC server'.

The proxy is designed in such a way that it runs a single AES-64 stack above its IP stack. For each discovered OSC server, the proxy creates an AES-64 node above its AES-64 stack. Each AES-64 node is uniquely identified by an AES-64 node ID. This is illustrated in Figure 6.4.

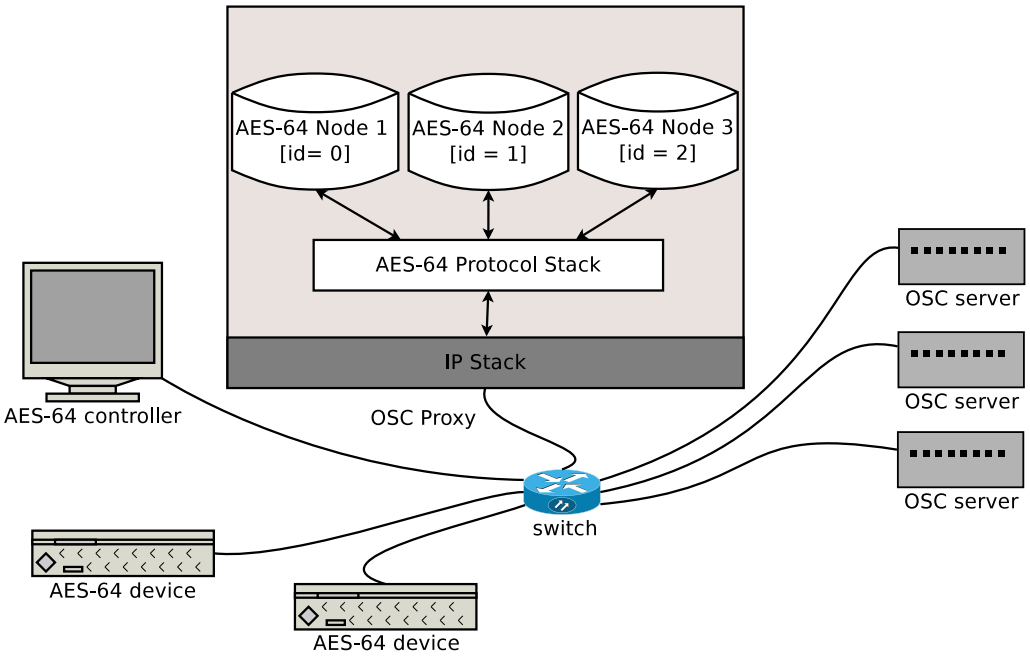


Figure 6.4: OSC proxy creates an AES-64 node for each discovered OSC server

The 'OSC Proxy' in Figure 6.4 has discovered all three OSC servers on the network, and has created the corresponding AES-64 nodes ('AES-64 Node 1', 'AES-64 Node 2', and 'AES-64 Node 3') above its 'AES-64 Protocol Stack'. Each node models the control features on an OSC server as AES-64 parameters. This makes it possible for the 'AES-64 controller' to access the OSC server's features.

The 'AES-64 controller' is able to address an AES-64 message to an OSC server by sending it to the IP-address of the proxy and the AES-64 node ID that corresponds to the particular server. To a remote AES-64 device, the OSC device appears like any other AES-64 device on the network.

The next section gives more details on how the OSC proxy has been implemented to meet its requirements.

6.4 OSC Proxy Implementation

The OSC proxy that has been created is capable of:

- discovering OSC servers,
- exposing the parameters on each discovered OSC server in AES-64 terms, and
- enabling connection management of audio streams on the OSC servers.

The following sections describe how the proxy accomplishes these requirements. *'AVB streams'* is used to describe audio streams on an Ethernet AVB network that conform to the IEEE 1722 standard.

6.4.1 OSC server discovery

The discovery mechanism implemented by the OSC proxy is DNS-SD [152]. DNS-SD enables a device to advertise its available services on a network, and other devices on the network to discover the advertised service. DNS-SD has been described in section 3.4.1.2 on page 80.

At start up, the proxy attempts to discover all OSC devices on the network. It utilizes the *avahi* (version 0.6.25) library to discover OSC services of type *'_osc._udp'* on the network [150]. The proxy interprets each discovered instance of this (OSC) *service type* as an OSC server on the network. The term *service type*, as used in zero configuration networking, is an application protocol name that gives information about what protocol a particular service implements and how to communicate with the service [152].

The proxy adheres to the following steps when attempting to discover instances of the OSC service on the network:

- The proxy creates an instance of the *AvahiClient* by calling the *avahi_client_new()* function. A callback is passed as an argument to this function. Whenever the state of the avaihi client changes, this callback is triggered. On successful initialization of the client, the callback is triggered with the *AVAHI_CLIENT_S_RUNNING* state value, which is an indication that the avahi server is in operation.

- On successful initialization of the *AvahiClient*, the proxy creates a service browser object by calling the *avahi_service_browser_new()* function. The service type argument passed to this function is *_osc._udp*, and it indicates that the proxy is only interested in discovering devices that implement this OSC service on the network. There is a callback argument that is associated with the service browser object. This callback is actuated whenever a browser event occurs. Such browser events include announcements from *avahi* clients indicating that they are either available on (or departing from) the network.

- For each discovered OSC service (of type *_osc._udp*), an *avahi* service resolver object is created with *avahi_service_resolver_new()*. A call to this function enables the proxy to obtain the DNS records of a discovered service instance. A

callback is passed to the *avahi_service_resolver_new()* function, and the callback is actuated if the resolver failed or succeeded in obtaining the DNS records.

The resolver enables the proxy to obtain further details beyond the presence of the service on the network. This includes device specific information such as IP address, port number, and common name (nickname) of the device publishing the service (instance) on the network.

After a service instance has been resolved, the proxy proceeds to create an AES-64 node which represents the resolved service instance.

6.4.2 AES-64 parameters for OSC server

The current implementation of the OSC proxy uses version 1.0.6 of the AES-64 protocol stack, which is known as the *xfndll*. The protocol stack is initialized at startup of the OSC proxy. For each discovered OSC server on the network, the proxy creates an abstraction of the server in the form of an AES-64 node. Each AES-64 node has a unique node ID associated with it, and consists of a number of addressable AES-64 parameters which are organized in a 7-level hierarchy. Each parameter has an associated callback that handles AES-64 messages addressed to it. Refer to chapter 3 (section 3.4.2.1 on page 86) for a description of the AES-64 parameter structure.

When the OSC proxy receives an AES-64 message addressed to a particular parameter within one of its AES-64 nodes, the callback associated with the parameter translates the message, then sends the appropriate OSC message to the corresponding physical device (OSC server).

At level-6 of an AES-64 parameter's hierarchy is the AES-64 parameter type (refer to section 3.4.2.1 on page 86 in chapter 3). An AES-64 parameter type describes the kind of control or feature represented by the parameter. It is possible to have any number of parameters with the same parameter type within an AES-64 node. At level-7 of the parameter hierarchy, a unique parameter index is used to distinguish between different parameters of the same parameter type.

A number of parameter types were created within the AES-64 nodes that correspond to OSC servers on the network. These parameter types are described below.

6.4.2.1 Device discovery parameter types

The various device discovery parameter types enable an AES-64 controller to discover the AES-64 nodes that reside within the proxy. These parameter type are:

- XFN_PTYPE_IP_ADDRESS - contains the IP address of the host device.
- XFN_PTYPE_SUBNET_MASK - holds the network subnet where the host resides.
- XFN_PTYPE_DEVICE_NAME - indicates the name of the device, for instance "UMAN Eval board".
- XFN_PTYPE_DEVICE_TYPE - holds one of the defined AES-64 device types. For instance a proxy device has a defined AES-64 device type within the AES-64 protocol stack.
- XFN_PTYPE_XFN_BOUND - indicates that a particular IP interface is capable of streaming media.

6.4.2.2 Input parameter types

A number of input parameters were created within the proxy's AES-64 nodes. The input parameter types reside within the input section block of the AES-64 node's parameter hierarchy. They are used to model the input features on the OSC server. The parameter types in this category are:

- XFN_PTYPE_MULTICORE_RUNNING_STATE - indicates whether an active connection exists on a particular input stream.
- XFN_PTYPE_MULTICORE_TYPE - indicates the type of multicore such as Ethernet AVB audio multicore, Ethernet AVB video multicore, or IEEE 1394 audio multicore.
- XFN_PTYPE_MULTICORE_NAME - contains the name of the multicore as a string. For instance "Input Multicore 1".
- XFN_PTYPE_MULTICORE_START - this parameter can be used to start or stop the input multicore stream.
- XFN_PTYPE_STREAM_ID - holds the stream ID of the input multicore.
- XFN_PTYPE_LISTEN - this parameter is a listen parameter, which is used to indicate interest in an AVB stream by interacting with MSRP.

6.4.2.3 Output parameter types

A number of output parameters were created within the proxy's AES-64 nodes. The output parameter types reside within the output section block of the AES-64 node's parameter hierarchy, and they model the outputs features on the OSC server. The parameter types in this category are:

- XFN_PTYPE_MULTICORE_RUNNING_STATE- indicates whether an active connection exists with a particular output stream.
- XFN_PTYPE_MULTICORE_TYPE - indicates the type of multicore such as Ethernet AVB audio multicore, Ethernet AVB video multicore, or IEEE 1394 audio multicore.
- XFN_PTYPE_MULTICORE_NAME - indicates the name of the multicore as a string. For instance "Output Multicore 1".
- XFN_PTYPE_MULTICORE_START - this parameter can be used to start or stop the output multicore stream.
- XFN_PTYPE_STREAM_ID - holds the stream ID of the output multicore.
- XFN_PTYPE_ADVERTISE - an advertise parameter, which enables advertising of an AVB stream by interacting with MSRP.

6.4.2.4 Internal routing matrix parameter types

To enable routing of signals between the inputs and outputs on the OSC server, internal routing matrix parameter types were defined. These parameter types are:

- XFN_PTYPE_MATRIX_PIN_NAME - this parameter exists for both input and output multicores. It holds the names (as a string) of the inputs and outputs. For instance "Analog Input 1", "Multicore Input 1 Pin 1" , "Analog Output 1" or "Multicore Output 1 Pin 1".
- XFN_PTYPE_CROSSPOINT_ENABLE - indicates a cross point between an input and an output. A value of '1' indicates that the cross-point has been enabled, and a connection should be established between the corresponding input and output. A '0' value means that the connection between the input and output should be destroyed.

Within the callback functions that are associated with each of the parameters, the proxy translates the messages from AES-64 to OSC. In the next section the connection management of networked AES-64 devices and OSC servers, as implemented by the OSC proxy, is described.

6.4.3 OSC proxy for connection management

Connection management is concerned with establishing and destroying audio stream connections between networked devices. The OSC proxy enables connection management between AES-64 devices and OSC servers on the same network. In particular this proxy was designed to enable connection management on AES-64 and OSC end stations on an Ethernet AVB network. Thus the OSC server is capable of fulfilling the role of AVB talker or AVB listener.

The following discussions describe the interactions that occur when the proxy is used to set up an OSC server as an AVB listener and AVB talker.

6.4.3.1 Setting up OSC server as AVB listener

The interactions that occur when the OSC proxy is used to establish an audio stream between an AES-64 device (as AVB talker) and OSC server (as AVB listener) are depicted in the form of a sequence diagram in Figure 6.5.

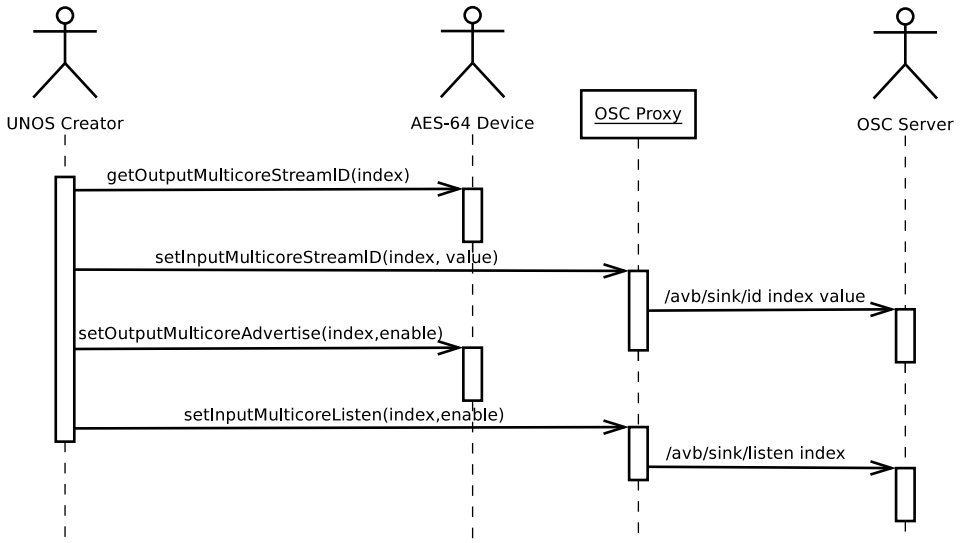


Figure 6.5: OSC proxy enables OSC server to fulfill the role of AVB listener

The actor *'UNOS Creator'* is an AES-64 network connection management application. In Figure 6.5, *'UNOS Creator'* establishes a stream connection between the 'AES-64

Device' and the *'OSC Server'*. To establish a stream connection, the following steps are followed:

- *'UNOS Creator'* sends an AES-64 message to obtain the stream ID of a particular output AVB stream on the AVB talker, which in this case is the 'AES-64 *Device'*.
- *'UNOS Creator'* sends an AES-64 message to the OSC proxy, to set the stream ID on a particular input AVB stream on the *'OSC Server'*. It indicates the input stream index and the value of the stream ID.
- The *'OSC Proxy'* translates the received AES-64 message to an OSC message, then sends it to the *'OSC Server'*. The *'OSC Proxy'* includes the input stream index and the value of the stream ID in the OSC message.
- *'UNOS Creator'* sends an AES-64 message to *enable* the advertise parameter within the 'AES-64 *Device'*. This causes the 'AES-64 *Device'* to register its stream on the Ethernet AVB network via MSRP.
- *'UNOS Creator'* sends an AES-64 message to the *'OSC Proxy'* to *enable* the listen parameter on the Ethernet AVB input.
- The *'OSC Proxy'* translates the AES-64 message to an OSC message and sends it to the *'OSC Server'*. This causes the *'OSC Server'* to indicate interest in receiving the stream from the Ethernet AVB network, via MSRP.

A similar sequence can be used to destroy a stream connection. However when destroying a stream connection, the modifications to the sequence diagram of Figure 6.5 are:

- the *setOutpuMulticoreAdvertise()* function is sent from *'UNOS Creator'* to the 'AES-64 *Device'* with a *disable* argument.
- the *setInpuMulticoreListen()* function is sent from *'UNOS Creator'* to the *'OSC Proxy'* with a *disable* argument.
- the *'/avb/sink/listen index'* message from the *'OSC Proxy'* to the *'OSC Server'* becomes a *'/avb/sink/destroy index'*.

6.4.3.2 Setting up OSC server as AVB talker

The OSC proxy is utilized by UNOS Creator (an AES-64 connection management application) to setup an OSC server as AVB talker. The sequence of interactions that occur when UNOS Creator establishes an audio stream connection between an OSC server (as AVB talker) and an AES-64 device (as AVB talker), via the proxy, is shown in Figure 6.6.

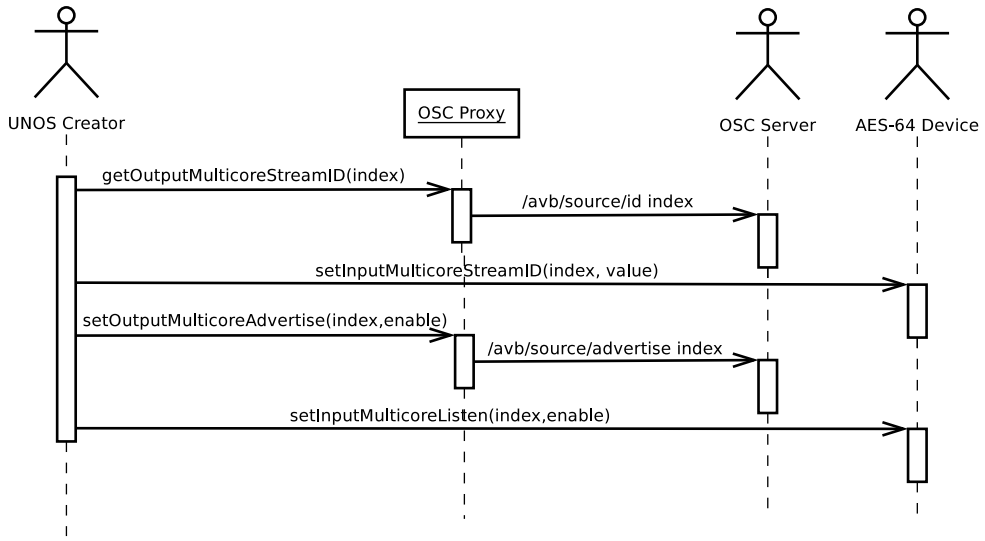


Figure 6.6: OSC proxy enables OSC server to fulfill the role of AVB talker

The interaction between the various actors (*'UNOS Creator'*, *'OSC Server'* and *'AES-64 Device'*) with the *'OSC Proxy'* depicted in Figure 6.6, is as follows:

- *'UNOS Creator'* sends an AES-64 message to the *'OSC Proxy'* to obtain the stream ID on an output (source) AVB stream on the *'OSC Server'*.
- The *'OSC Proxy'* translates the AES-64 message to an OSC message to obtain the stream ID of a source AVB stream on the *'OSC Server'*.
- *'UNOS Creator'* sends an AES-64 message to set the stream ID on an input of the *'AES-64 Device'*.
- *'UNOS Creator'* sends an AES-64 message to the *'OSC Proxy'* to *enable* AVB stream advertisement.
- The *'OSC Proxy'* translates the received AES-64 message to an OSC message, then sends it to the *'OSC Server'*. This causes the *'OSC Server'* to advertise its stream on the Ethernet AVB network via MSRP.

- *'UNOS Creator'* sends an AES-64 message to the 'AES-64 *Device'* to *enable* its listen parameter. This causes the 'AES-64 *Device'* to indicate interest in the AVB stream, via MSRP.

To destroy a stream connection, a similar sequence to Figure 6.6 is followed, except that:

- the *setOutputMulticoreAdvertise()* AES-64 message from *'UNOS Creator'* to the *'OSC Proxy'* is passed a *destroy* argument rather than the *enable*.
- the *'OSC Proxy'* translates this message to an *'/avb/source/withdraw index'* message and sends it to the *'OSC Server'*. This will cause the *'OSC Server'* to declare a withdrawal of its stream from the Ethernet AVB network via MSRP.
- the *setInputMulticoreListen()* AES-64 message from *'UNOS Creator'* to the 'AES-64 *Device'* is passed a *destroy* argument.

6.5 Layout of the OSC proxy Implementation

The proxy implementation is depicted in form of a class diagram in Figure 6.7.

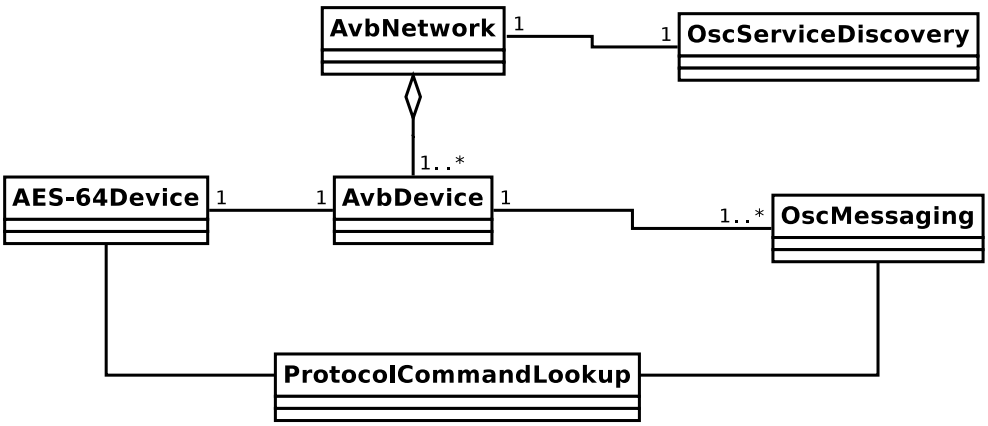


Figure 6.7: OSC proxy class diagram

The proxy was implemented for connection management and device control of networked AVB end stations, hence the class names *'AvbNetwork'* and *'AvbDevice'* shown in Figure 6.7. An *'AvbNetwork'* object abstracts the Ethernet AVB network, and an *'AvbDevice'* object abstracts an AVB end station.

At start up, the proxy creates an instance of the *'AVBNetwork'* object. The *'AVBNetwork'* object is responsible for creating the *'OscServiceDiscovery'* object, which is used

to browse for instances of the OSC service type (*'_osc._udp'*). Each instance of the OSC service type is resolved, and considered (by the proxy) to be an OSC server. The *'AVB-Network'* object holds a list of discovered devices. Each device in this list is uniquely identified by its IP address, and nickname. For each new OSC server discovered, an *'AVBDevice'* object is created and added to the AVB devices list.

For OSC communication with the OSC server, the proxy instantiates an *'OscMessaging'* object. The *'OscMessaging'* object is used by the proxy to receive and transmit OSC messages on the network.

A one-to-one relationship exists between an *'AvbDevice'* object and an *'AES-64Device'* object, since the proxy models every *'AvbDevice'* (in the AVB device list) as an *'AES-64Device'*.

The *'ProtocolCommandLookup'* class that is implemented by the proxy, maps each AES-64 message with the appropriate OSC message, and vice versa. It matches the OSC commands of the *'OscMessaging'* object to corresponding AES-64 callbacks of the *'AES-64Device'* object.

The integration of other control protocols into the proxy entails:

- defining a set of connection management and control commands for the protocol, and
- mapping of the new command set to those in the *'ProtocolCommandLookup'*.

This design enables the proxy to be adopted to new protocols by performing an update to the *'ProtocolCommandLookup'* class. In order to incorporate a new protocol, the new protocol messages will have to be appropriately mapped to those of the already existing protocols. For example, the lookup table (mapping) for command translation between two protocols (AES-64 and OSC) that is implemented by the *'ProtocolCommandLookup'* class could be of the form shown in Table 6.1.

Index	Description	AES-64 callback	OSC method
1	Obtain device name	*getDeviceName()*	*/device/name*
2	Obtain IP address	*getDeviceIPAddress()*	*/device/ip*
3	Modify stream ID	*setAVBStreamId()*	*/avb/streamId*

Table 6.1: Mapping table for command translation

Table 6.1 shows the associated AES-64 callback and OSC method that corresponds to each command index in a lookup table. When an AES-64 *getDeviceIPAddress()* callback is triggered, the command translator maps it to the corresponding OSC method

with the same command index (in this case '2'). Then it transmits an OSC message that will cause the */device/ip* OSC method to be dispatched within the target OSC server. Similarly when an OSC message that is addressed to the */avb/streamId* method is called, the command translation table matches it to the appropriate AES-64 message that will cause the *setEthernetAVBStreamId()* callback within the target AES-64 device to be triggered. The *index* associated with each command allows a particular protocol command to be translated to another protocol command.

In order to incorporate another protocol (Protocol X) into the lookup table, the equivalent 'Protocol X' commands are mapped to those already defined for AES-64 and OSC. The new map will be of the form shown in Table 6.2.

Index	Description	AES-64 callback	OSC method	Protocol X
1	Obtain device name	*getDeviceName()*	*/device/name*	*discoverXName()*
2	Obtain IP address	*getDeviceIPAddress()*	*/device/ip*	*discoverXIp()*
3	Modify stream ID	*setAVBStreamId()*	*/avb/streamId*	*adjustXAVBStreamId()*

Table 6.2: Modified mapping table to incorporate Protocol X

Table 6.2 illustrates the specific 'Protocol X' execution procedure for each of the commands already defined for AES-64 and OSC. A *discoverXName()* Protocol X procedure will result in the command translator sending an AES-64 message that will trigger the *getDeviceName()* callback within the AES-64 target. In the same manner, when an OSC */device/ip* method is dispatched, it will cause the command translator to issue a 'Protocol X' message that will cause a target 'Protocol X' device to execute its *discoverXIP()* procedure.

Using this approach any number of audio control protocols can be incorporated into the proxy for command translation.

6.6 Tests and Results

The effectiveness of the OSC proxy was tested with an AES-64 network monitor, configuration, connection and control manager called UNOS Creator [153]. The test bed topology is shown in Figure 6.8.

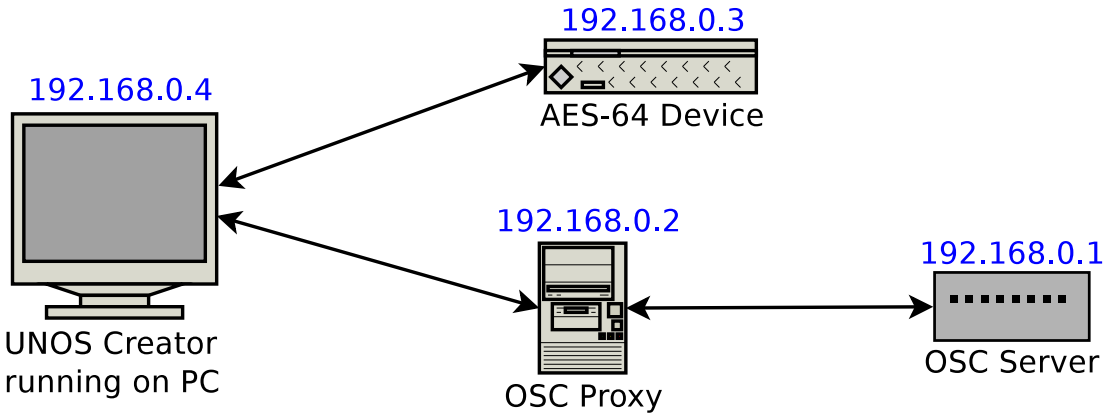


Figure 6.8: Test bed network topology

The 'OSC Server' shown in Figure 6.8 has been described in chapter 5. The 'AES-64 Device' was also implemented as a (virtual) device on a PC [1]. Both of these devices are Ethernet AVB compliant.

UNOS Creator was used to investigate:

- discovery of OSC servers via the OSC proxy, and
- connection management between networked devices that implement AES-64 and OSC protocols.

The results are described in the following subsections.

6.6.1 Device discovery via OSC proxy

When UNOS Creator is started, it broadcasts an AES-64 device discovery message on the network. Each AES-64 node, representing an AES-64 device on the network, responds to the AES-64 device discovery message. The response includes AES-64 discovery information such as IP address, subnet mask, device name, and the type of device. Figure 6.9 shows UNOS Creator's device discovery view for the test bed network of Figure 6.8.

Figure 6.9: UNOS Creator networked devices view

AES-64 is being standardized by the Audio Engineering Society (AES) as part of the AES-X170 project. Hence in Figure 6.9, X170 refers to AES-64 and an AES-64 device exposes its name as *"X170 Virtual Device"*.

To the right of the screen, within UNOS Creator's device discovery view, is the network subnet of discovered devices. Figure 6.9 displays only one subnet (*'192.168.0.0'*), which implies that all discovered AES-64 nodes reside on the same subnet.

Since UNOS Creator also runs the AES-64 protocol stack, it is discovered and displayed in the networked devices view as *'UNOS Creator 192.168.0.4'*.

The 'OSC Server' of Figure 6.8 is discovered by UNOS Creator via the proxy, hence its IP address is *'192.168.0.2'*, which is the IP address of the 'OSC Proxy'. Thus all AES-64 messages to the 'OSC Server' are addressed to the proxy.

6.6.2 Connection management via OSC proxy

Figure 6.10 is a screen shot of UNOS Creator when an audio stream connection is established between the 'AES-64 Device' and the 'OSC Server'. The 'AES-64 Device' was set up as an AVB talker and the 'OSC Server' as the AVB listener.

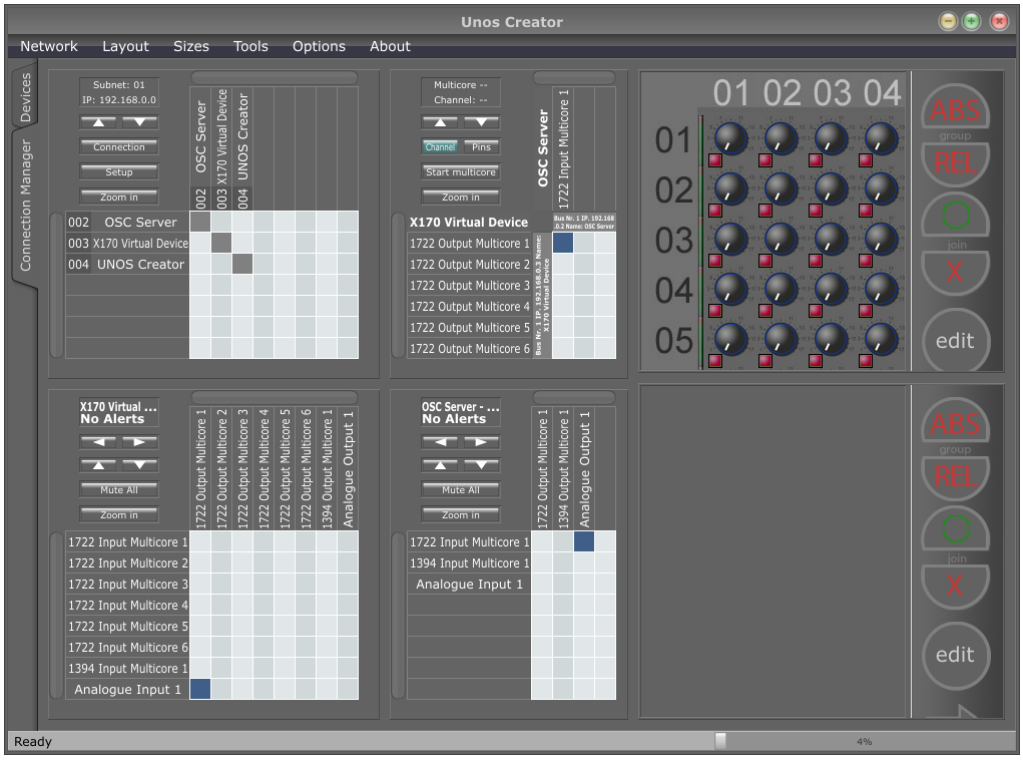


Figure 6.10: UNOS Creator's connection manager view

The top left matrix is UNOS Creator's device matrix, and it shows the discovered AES-64 devices. Clicking on a cross-point in this matrix establishes the device on the left of the cross point as the source device, and the device above the cross point as the destination device.

The top middle matrix is the multicore matrix. To the left of the multicore matrix are the output multicores of the source device. At the top of the multicore matrix are the input multicores of the destination device. In the context of AES-64, a multicore is a term that describe the end-points of an audio or video stream. In Figure 6.10, the selected cross-point indicates that a connection has been established between the output multicore *"1722 Output Multicore 1*" (of the *"X170 Virtual Device"*) and the input multicore *"1722 Input Multicore 1"* of the *"OSC Server"*. A connection can be destroyed by deselecting the cross-point between an output and input. Each of the multicores on the devices used in this test contains two channels of audio.

In UNOS Creator, an internal routing matrix exposes the stream inputs and outputs on a particular device, and it is used to patch a signal from a particular input to one or more outputs on the device.

The bottom left matrix depicts the internal routing matrix of the source device, which in this case is the *"X170 Virtual Device"*. In Figure 6.10, an analogue audio input

signal (*"Analogue Input 1"*) from an audio source (for instance an MP3 player) has been patched to the output (*"1722 Output Multicore 1"*) of the same device.

The internal routing matrix of the *"OSC Server"* is shown in the bottom middle matrix of Figure 6.10. In the figure, the OSC server's input (*"1722 Input Multicore 1"*) is patched to its analogue output (*"Analogue Output 1"*) which is attached to a speaker.

6.7 Qualitative Analysis

On an Ethernet AVB network, an AVB talker advertises its available streams via MSRP. An AVB listener indicates to the AVB talker that it is ready to receive an audio stream. If sufficient network resources have been reserved for the stream, the AVB talker can start transmitting. Refer to section 2.2.1.2 on page 22 for a description on the role of MSRP for enabling streaming on an Ethernet AVB network.

An Ethernet AVB network controller (such as UNOS Creator) can be used to configure the AVB talker and AVB listener in order to establish (or destroy) a stream connection. The AES-64 messages issued by UNOS Creator in order to set up AVB talker and AVB listeners are [1]:

- AES-64 *get value* message to obtain the stream ID of an output multicore from the AVB talker.
- AES-64 *set value* message to modify the value of the stream ID on an AVB listener's input multicore.
- AES-64 *set value* message to the AVB talker to advertise its output stream via MSRP.
- AES-64 *set value* message addressed to the listen parameter of the input multicore on the AVB listener, causing it to request attachment to the stream on offer by the talker via MSRP.

By utilizing the proxy, UNOS Creator is capable of configuring AVB talker and AVB listener end stations according to the above steps.

The role of the command translator (that is proxy) is to:

- discover networked audio devices
- model the discovered devices in terms of a command control protocol, that is the protocol implemented by the network controller

- expose the controls within each discovered device to the network controller
- receive messages, which conform to the common control protocol, on behalf of the discovered devices
- translate the received messages to the appropriate protocol messages
- transmit the translated messages to the target device(s)
- receive a response from the target device(s)
- translate the received response to the appropriate message of the common control protocol
- transmit the response to the common network controller

In the current implementation these functions are fulfilled by a proxy that is located on a separate PC (workstation) from the control application (as shown in Figure 6.8). However, it is possible to host the command translator within the same host PC as the control application. Such an implementation could utilize the same network interface for receiving and transmitting messages for both the network control application and the command translator. Figure 6.11 shows an example layout of how the command translator can be incorporated into the same host PC as an AES-64 network controller.

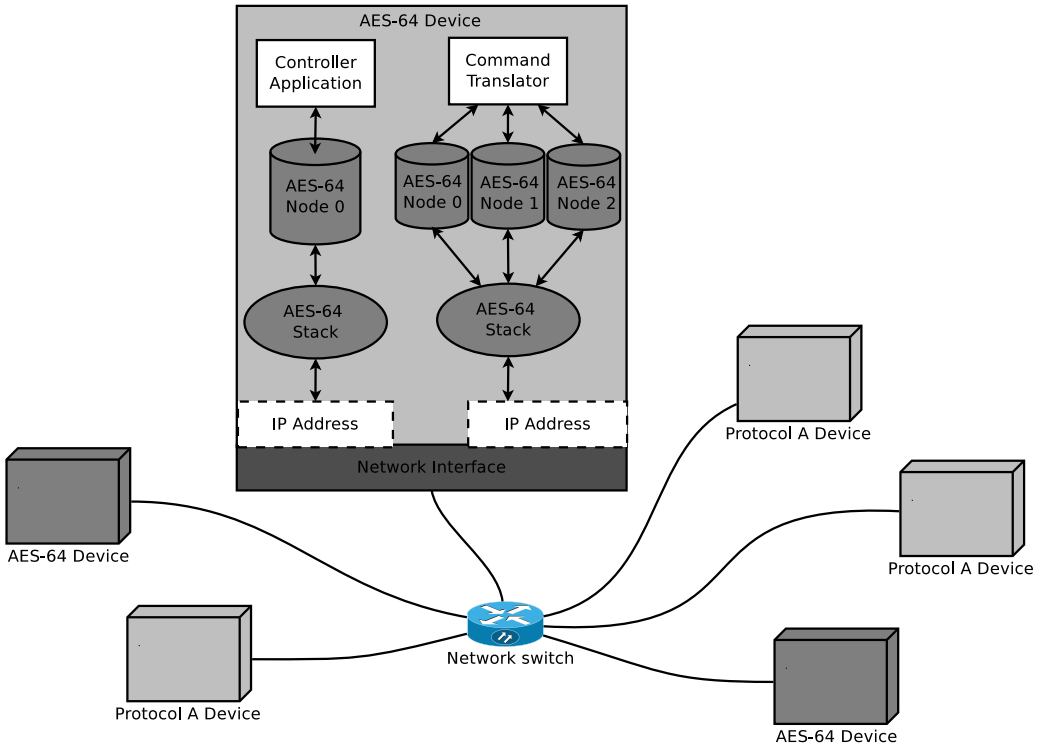


Figure 6.11: Integrated command translator for AES-64 network control

Figure 6.11 depicts a network of six devices, including an AES-64 network control application. Three of the networked devices implement AES-64, while the other three implement a different ('Protocol A') control protocol. The use of a command translator will allow interoperability between the devices on the network. In Figure 6.11, the command translator runs on the same PC workstation as the controller application. A single network interface card is used, but two instances of the AES-64 protocol stack run on the PC. Each AES-64 stack is bound to a different IP address, although the same network interface in used. By binding to a different IP address, the *'Controller Application'* and *'Command Translator'* appear as if they were located on different AES-64 devices. The *'Command Translator'* proxies the three 'Protocol A' devices, hence it creates three AES-64 nodes above its protocol stack. The *'Command Translator'* is able to fulfill its role while located on the same PC as the *'Controller Application'*, and it interacts with the *'Controller Application'* as if it was located on a different PC. This approach ensures that the connection management application does not depend on a remote networked device.

6.8 Summary

There are a wide range of disparate audio control protocols, which are used by networked audio devices for remote configuration, monitoring and control. Interoperability and common control of networked devices that conform to different audio control protocols remains a challenge. This chapter has described the use of a proxy for command translation from one layer 3 audio control protocol to another.

An OSC proxy was created to enable connection management and common control of devices that implement the AES-64 and OSC protocols. In this chapter the requirements, design and implementation of the OSC proxy have been described in detail.

Tests were conducted to determine the effectiveness the OSC proxy when it is used to enable interoperability between networked OSC and AES-64 devices. The proxy proved to be capable of enabling audio stream connections between the networked devices.

Although these tests have been conducted for OSC and AES-64 command translation, the inclusion of other control protocols was considered in the design of the proxy. This will allow for adaptation of the proxy to different audio networks.

Chapter 7

Layer 2 end station Implementation - AVDECC

Ethernet AVB allows for the networking of AVTP endpoints (also referred to as AVTP end stations). An AVTP end station is a device that is capable of transmitting data streams that conform to the IEEE 1722 standard.

IEEE 1722.1 is an audio control protocol that will enable remote monitoring, configuration and control of AVTP end stations. IEEE 1722.1 messages are exchanged between networked IEEE 1722.1 compliant devices. An IEEE 1722.1 message is encapsulated within an OSI/ISO layer 2 Ethernet frame, thus IEEE 1722.1 has been described as a *layer 2 audio control protocol* in chapter 3.

At the start of this research project, there were no commercially available IEEE 1722.1 compliant devices. Also there were no software library that could be used to develop a IEEE 1722.1 compliant device, although there were lots of interest in the development of the IEEE 1722.1 standard. Hence the first step was to develop a software library that will allow for the creation of IEEE 1722.1 audio streaming devices.

This chapter provides an overview of the IEEE 1722.1 standard and describes the implementation of the (IEEE 1722.1) software library that was created in the course of this research project. It also describes a layer 2 end station that runs on a PC workstation, implements the IEEE 1722.1 standard, and is capable of streaming IEEE 1722 audio on an Ethernet AVB network. In the following discussions, the term *'AVB'* refers to Ethernet AVB.

7.1 Introduction

A digital audio network device typically implements a control protocol that enables it to be remotely monitored and configured. An audio control protocol will usually define the procedure for establishing and destroying audio stream connections. This procedure is known as connection management. Other common features of control protocols include:

- a device discovery mechanism,
- support for the enumeration of a device's features, and
- control commands for manipulating the enumerated features.

The IEEE 1722.1 standard defines an *Audio Video device Discovery, Enumeration, Connection management and Control* (*AVDECC*) protocol. The goal of AVDECC is to standardize a procedure for achieving the above mentioned features (of control protocols) on IEEE 1722 (AVTP) devices. AVDECC views networked IEEE 1722 devices as fulfilling one or more of three different roles, namely [19]:

- *AVDECC controller* - device that sends enquiry and control AVDECC messages to AVDECC talkers and AVDECC listeners. An AVDECC controller is a device that sends AVDECC messages to configure AVDECC listeners or AVDECC talkers.
- *AVDECC listener* - device that receives one or more audio stream(s) from the network. It utilizes MSRP to indicate interest in a particular stream on offer by a talker.
- *AVDECC talker* - device that is the source of one or more audio stream(s) on the network. An AVDECC talker utilizes MSRP to ensure that the necessary network resources are available before it commences the transmission of media streams on the network.
- *AVDECC interface* - device that is capable of AVDECC messaging, but does not necessarily fulfill any of the above roles.

Devices that conform to the AVDECC protocol are broadly referred to as AVDECC end stations. At the time of writing, the current draft (draft 19) of the IEEE 1722.1 standard defines a number of (layer 2) sub-protocols that fulfill different aspects of AVDECC. Each of these sub-protocols defines a data unit (DU) that describes the structure and meaning of its messages. A protocol's DU is encapsulated within a layer 2 (AVDECC) message, and is transmitted on the network.

These AVDECC sub-protocols are [19]:

- *AVDECC Discovery Protocol (ADP)* - is a layer 2 protocol that enables networked AVDECC devices to be discovered. ADP defines an *advertising state machine*, which is utilized by an AVDECC end station to announce its presence on a network. It also defines a *discovery state machine*, which enables an AVDECC controller to discover other AVDECC end stations on the network. ADP defines the ADPDU, which provides a standard ADP message encapsulation for transmission on the network. All ADP messages are multicast in nature.

- *AVDECC Enumeration and Control Protocol (AECP)* - is a layer 2 protocol that enables remote device enumeration and control. AECP defines the message structure for discovering the functional units and features within an AVDECC end station. It also defines the structure of layer 2 commands that should be used to modify these features. An AECP command is encapsulated within an AECPDU, and unicast from one AVDECC end station to another on the network.

- *AVDECC Connection Management Protocol (ACMP)* - is a layer 2 protocol that defines the procedures for establishing and destroying stream connections. ACMP defines three state machines that enable AVDECC end stations to fulfill the roles of AVDECC controller, AVDECC talker, and AVDECC listener. The ACMP protocol also defines an ACMPDU, which is used to transmit the various ACMP commands and responses necessary for establishing and destroying stream connections. All ACMP messages are unicast on the network.

An AVDECC end station is uniquely identified by a 64-bit unique identifier, known as it's *entity GUID*. This entity GUID is used for unicast communication. AVDECC defines multicast MAC addresses used for multicast messaging [19, pp. 268].

In order to provide a standardized way for AVDECC end stations to expose their functional units and features to a remote controller, the IEEE 1722.1 standard defines an *AVDECC Entity Model (AEM)*. The AEM defines a hierarchical structure that enables AVDECC end stations to represent their control features, capabilities, and functionalities. AEM can be used to trace the relationship between the features within an end station. AVDECC defines a number of commands that can be used to enquire about a control/feature, and to modify it. AEM commands and responses are encapsulated within the AECPDU of the AECP protocol.

In the course of this research, an AVDECC end station was implemented, that is capable of streaming audio an an Ethernet AVB network. In order to implement the AVDECC end station, a software library that implements the AVDECC protocol has been created. This library is known as *libavdecc,* and it initially implemented on the Linux platform

(*kernel version 3.0.0-17*). To allow for a wider use of the software, it has also been ported to the *Windows* platform. The following section describes *libavdecc* with the intention of providing further detail about the nature of various components that make up the AVDECC protocol. Although these descriptions refer to *Linux* version of the (*libavdecc*) software, the overall architecture of the *Windows* and *Linux* versions are the same.

7.2 AVDECC library

The AVDECC Linux software library implementation (*libavdecc*) that has been developed in the course of this research, was designed to enable a software developer to create an application that conforms to the IEEE 1722.1 standard. An AVDECC controller can utilize *libavdecc* to discover AVDECC talkers and listeners, as well as to transmit AVDECC connection management and control instructions. A software developer is able to create an AVDECC talker and/or AVDECC listener, that runs on a Linux PC, by utilizing *libavdecc*.

The *libavdecc* library was designed to consist of a number of modules, with each module implementing an aspect of the AVDECC protocol. This modular design allows developers to utilize only the modules that are required by their applications. Figure 7.1 shows the design layout of the *libavdecc* library.

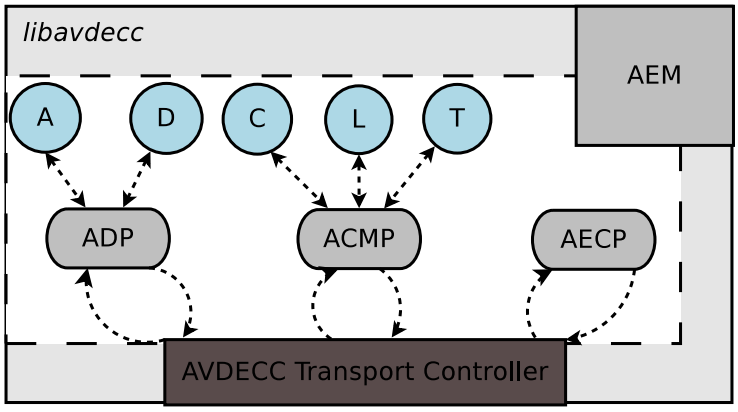


Figure 7.1: Logical layout of the *libavdecc* implementation

Figure 7.1 shows the:

- *'AVDECC Transport Controller'* module - this is the transport module implemented by *libavdecc*, and it is used by all of the other modules (*ADP'*, *'ACMP'*, and *'AECP'*). Currently this module is Linux platform dependent. To use *libavdecc*

on any other platform (such as Windows or Mac OS) requires a port of this module.

- *'ADP'* module - is an implementation of the AVDECC Discovery Protocol. Associated with it are the *advertising* and *discovery* state machines, which are shown as the circles labeled *'A'* and *'D'*, respectively.
- *'ACMP'* module - is *libavdecc*'s implementation of the AVDECC Connection Management Protocol. Associated with *'ACMP'* are the *controller*, *listener* and *talker* state machines, which are the circles labeled *'C'*, *'L'*, and *'T'*, respectively.
- *'AECP'* module - implements the AVDECC Enumeration and Control Protocol. It enables the encapsulation and extraction of AECP commands and responses within AECPDUs.
- *'AEM'* container - is an implementation of the AVDECC Entity Model. It defines a number of data structures that lay out the functional units within an AVDECC entity.

The following sections provide more information about the *libavdecc* modules.

7.2.1 AVDECC Transport Controller module

The *AVDECC Transport Controller* is responsible for receiving and transmitting all AVDECC messages from and to the network, respectively. All AVDECC messages received or transmitted by this module are layer 2 packets. There are two types of packets that the AVDECC transport controller is interested in. These are:

- Unicast packets that are addressed to the network interface hardware address of the host.
- Multicast packets that are sent to the multicast address reserved for AVDECC messaging.

The current draft of the IEEE 1722.1 standard defines *91-E0-F0-01-00-00* as the AVDECC multicast MAC address for device discovery and connection management [19, pp. 268].

Every device that utilizes *libavdecc* is required to initialize this (*AVDECC transport controller*) module for communication with the network. To do this, *libavdecc* provides an initialization function which is shown in Listing 7.1. The Listing also shows the transport controller module's cleanup function, which is used to deallocate resources utilized by the transport controller.

```
avdecc_transport_controller_init(
    avdecc_transport_controller* avdecc_tc)
avdecc_transport_controller_cleanup(
    avdecc_transport_controller* avdecc_tc)
```

Listing 7.1: Initialization and cleanup functions of the AVDECC transport controller module

An *avdecc_transport_controller* data structure is an instance of the transport controller module, and is passed as an argument to the above.

Each AVDECC packet that is received by the transport controller module consists of a protocol Data Unit (DU), which is encapsulated within the transport layer header (Ethernet header). Within each DU is a *'subtype'* field that is used to differentiate between AVDECC sub-protocols. The values of the *subtype* fields as defined by the AVDECC protocol are shown in Table 7.1.

Subtype Value	**Protocol**
0x7A	ADP
0x7B	AECP
0x7C	ACMP

Table 7.1: AVDECC protocol subtypes

When the AVDECC transport controller receives a message from the network, it strips the packet of its transport (Ethernet) header. Then it passes on the DU to the appropriate protocol module based on the DU's *subtype* field. The AVDECC transport controller module provides a function that is used by an AVDECC sub-protocol module to indicate interest in a particular type of AVDECC message. This function is shown in Listing 7.2.

```
avdecc_transport_controller_register_protocol()
```

Listing 7.2: AVDECC transport controller function for registering a sub-protocol

A module utilizing the AVDECC transport controller uses this function to indicate the type of message it is interested in. A callback is passed as an argument to this function, and it is triggered by the transport controller whenever a message of the specified type is received.

In order to transmit AVDECC messages, a sub-protocol (that is ADP, ACMP, or AECP) module utilizes the transport controller by sending its DU to the transport controller. The AVDECC transport controller adds the transport layer header to the received DU, and

sends the packet onto the network. The AVDECC transport controller module provides a function that is utilized by the other AVDECC modules to pass their DUs to the transport controller, for onward transmission on the network. The function is:

```
avdecc_transport_controller_send_data()
```

Listing 7.3: AVDECC transport controller function for transmitting messages on the network

A module utilizing this function passes as arguments the data unit it wishes to transmit, and an indication of the size of the data. The size is dependent on the particular protocol DU being transmitted.

7.2.2 ADP module

The *ADP* module implements the AVDECC Discovery Protocol (ADP) [19]. The ADP protocol defines a data unit known as ADPDU, and two state machines (*advertising* and *discovery* state machines) that enable AVDECC end stations to be discovered on a network.

The ADP module as implemented by *libavdecc* receives an ADP message from the AVDECC transport controller, then passes it to the appropriate state machine for processing. The ADP module determines which state machine should process a received ADP message based on the *'message type'* field. Table 7.2 shows the various ADP message types (as defined by IEEE 1722.1), and the *libavdecc* state machine responsible for processing them.

Value	Meaning	State Machine
0	Available	Discovery
1	Departing	Discovery
2	Discover	Advertising

Table 7.2: ADP message types

The *libavdecc*'s implementation of the *advertising* and *discovery* state machines are described in section 7.2.2.1 and section 7.2.2.2, respectively. *libavdecc* implements the ADP module as the base module for the advertise and discovery state machines. Although either of these ADP state machines can be utilized independently (of each other), they both require the ADP module to have been initialized. The *libavdecc* library

provides a function that can be used by an application to initialize the ADP module, and another for cleaning up resources used by the ADP module. These two functions are:

```
avdecc_adp_init()
avdecc_adp_cleanup()
```

Listing 7.4: ADP module functions for initialization and cleanup

An *avdecc_adp* structure is passed as an argument to the functions above.

All ADP messages are multicast announcements, and the AVDECC multicast MAC address is used to transmit an ADP message on the network. The *libavdecc*'s implementation of the ADP *advertising* and *discovery* state machines are described in the following sections.

7.2.2.1 Advertising state machine

The *advertising state machine* is utilized by an AVDECC end station to announce its presence on a network. It implements a re-announce timer, which ensures that the end station multicasts its advertise messages at regular intervals. An ADP advertise message includes a *valid_time* field, which indicates how long the announcement is valid for. If no re-announcement is received from the end station with this state machine, a controller can assume that the end station is no longer available on the network.

When an ADP discovery message is received from the ADP module, the advertise state machine re-announces its presence on the network. Thus making its presence known to the transmitter of the discovery message. When the end station is leaving the network, the advertising state machine announces its departure by multicasting an ADP 'departing' message.

The *libavdecc* library provides a function for initializing the advertising state machine, and another for deallocating resources used by the advertising state machine. These two functions are:

```
avdecc_adp_advertise_sm_init()
avdecc_adp_advertise_sm_cleanup()
```

Listing 7.5: *libavdecc*'s advertise state machine functions

A data structure *(avdecc_adp_advertise_sm*), which is an instance of the advertise state machine, is passed as an argument to either of these functions.

7.2.2.2 Discovery state machine

The *discovery state machine* is used by an AVDECC end station (typically an AVDECC controller) to discover the other end stations on the network. It does this by multicasting an ADP discover message on the network.

The discovery state machine holds a list of discovered end stations, and it modifies this list whenever a new end station is discovered or an end station has become unavailable. Whenever a re-announce message is received from an end station, the discovery state machine updates its list. This ensures that the list of discovered end stations contains the most recent information about each discovered end station.

There are two ways in which the discovery state machine determines whether an AVDECC end station is available or not. These are:

1. Each end station multicasts an ADP departing message on the network when it has been shut down gracefully. Upon receiving an ADP departing message, the discovery state machine removes the end station from its list of discovered end stations.

2. Each entry that is stored in the list of discovered end stations has a 'time-to-live' value associated with it. This 'time-to-live' value resides in the *'valid time'* field of the ADP advertise message, and it indicates how long an announcement is valid. If within the specified time, an ADP available message is not received from the same end station, the discovery state machine assumes that the end station is no longer available, and it is removed from the list.

libavdecc provides a function that can be used by an application to initialize a discovery state machine, and a function to deallocate resources used by this state machine. These functions are:

```
avdecc_adp_discovery_sm_init()
avdecc_adp_discovery_cleanup()
```

Listing 7.6: *libavdecc*'s discovery state machine functions

A data structure (*avdecc_adp_discovery_sm*) which represents an instance of the discovery state machine is passed as an argument to these functions. Associated with the discovery state machine's initialization function (*avdecc_adp_discovery_sm_init( )*) is a callback function that is passed as an argument by an application to the discovery state machine. The discovery state machine calls the application's callback function whenever an ADP available message or an ADP departing message is received. This enables

the application to be informed whenever an end station becomes available, leaves the network, or updates its information.

7.2.3 ACMP module

The *libavdecc* software library implements the AVDECC Connection Management Protocol (ACMP) as a module. This (*ACMP*) module receives connection management messages from the AVDECC transport controller. Each message received by the ACMP module conforms to the ACMP Data Unit (ACMPDU) defined by the AVDECC protocol [19, pp. 187].

The *libavdecc* library provides a function for initializing the ACMP module, as well as a function to enable an application to properly destroy and deallocate all resources being utilized by the ACMP module. These functions are:

```
avdecc_acmp_init()
avdecc_acmp_cleanup()
```

Listing 7.7: *libavdecc*'s ACMP initialization and destroy functions

The two functions take an instance of the ACMP data structure (*avdecc_acmp*) as an argument.

The *libavdecc* software library implements three state machines that are associated with the ACMP module. Each of these state machines can be enabled independently of each other, but each of them requires the ACMP module for receiving and transmitting ACMPDUs. Thus an end station that utilizes any of these state machines is required to have already initialized the ACMP module. The three ACMP state machines are:

- *Controller state machine* - handles AVDECC controller messages on an AVDECC end station,
- *Listener state machine* - handles AVDECC listener messages on an AVDECC end station, and
- *Talker state machine* - handles AVDECC talker messages on an AVDECC end station.

The ACMP module forwards an ACMPDU to a state machine based on the *'message type'* field of a received ACMP command or response. Table 7.3 shows the possible ACMP message types (commands and responses) received by the ACMP module, and the state machine responsible for handling them.

ACMP state machine	Message type
Controller	GET_TX_STATE_RESPONSE
	CONNECT_RX_RESPONSE
	DISCONNECT_RX_RESPONSE
	GET_RX_STATE_RESPONSE
	GET_TX_CONNECTION_RESPONSE
Listener	CONNECT_TX_RESPONSE
	DISCONNECT_TX_RESPONSE
	CONNECT_RX_COMMAND
	DISCONNECT_RX_COMMAND
	GET_RX_STATE_COMMAND
Talker	CONNECT_TX_COMMAND
	DISCONNECT_TX_COMMAND
	GET_TX_STATE_COMMAND
	GET_TX_CONNECTION_COMMAND

Table 7.3: *libavdecc*'s ACMP state machines and the ACMP messages they handle

The identifying values of the ACMP message types shown in Table 7.3 are defined in the IEEE 1722.1 standard document [19]. All ACMP commands and responses are multicast on the network.

The following sections provides more details about the ACMP state machines, as implemented in *libavdecc*.

7.2.3.1 Controller state machine

This state machine enables an AVDECC controller to transmit and receive connection management commands to and from AVDECC talkers and listeners. The controller state machine is able to retry a command when no response has been received within its (the command's) timeout period. These timeouts are defined by the AVDECC protocol, and they depend on the command's *'message type*' value [19, pp. 192]. The controller state machine is also able to associate a received response with the appropriate initiating command.

libavdecc provides a function that can be used by an application to initialize the controller state machine, and another function for deallocating resources used by the controller state machine. These functions are shown in Listing 7.8.

```
avdecc_acmp_controller_sm_init()
avdecc_acmp_controller_sm_cleanup()
```

Listing 7.8: *libavdecc*'s controller state machine functions

A data structure (*avdecc_acmp_controller_sm*), which is an instance of the controller state machine, is passed as an argument to the functions shown in Listing 7.8.

7.2.3.2 Listener state machine

The listener state machine is used by an AVDECC listener to process connection management commands. It enables an AVDECC listener to respond (appropriately) to received commands. These could be commands to:

- establish stream connections,
- destroy previously established stream connections, or
- determine the state of a particular input stream connection on the listener.

When an AVDECC listener receives an ACMP command to establish or destroy a stream connection, it utilizes MSRP to indicate to the (Ethernet AVB) network that it is interested (or not interested, in the case of a disconnect command) in a particular stream.

The listener state machine monitors commands that it has transmitted. This enables it to match a received ACMP response with the appropriate initiating ACMP command.

libavdecc provides a function that can be used by an application to initialize the listener state machine, and another function to deallocate resources used by the listener state machine. These functions are shown in Listing 7.9.

```
avdecc_acmp_listener_sm_init()
avdecc_acmp_listener_sm_cleanup()
```

Listing 7.9: *libavdecc*'s listener state machine functions

The functions shown in Listing 7.9 accepts as an argument a (*avdecc_acmp_listener_sm*) data structure that represents an instance of the listener state machine.

7.2.3.3 Talker state machine

libavdecc implements a talker state machine, which is used by an AVDECC talker to respond to AVDECC messages from AVDECC listeners and controllers. The talker state machine responds to ACMP commands to:

- establish a stream connection,
- destroy a stream connection,
- determine the state of an output on the talker, or
- determine the state of a stream connection.

The talker state machine utilizes MSRP for reserving network resources for its streams.

libavdecc provides a function that can be used by an application to initialize the talker state machine, and another function to deallocate resources used by the talker state machine. These functions are shown in Listing 7.10.

```
avdecc_acmp_talker_sm_init()
avdecc_acmp_talker_sm_cleanup()
```

Listing 7.10: *libavdecc*'s talker state machine functions

An instance of the talker state machine in the form of a (*avdecc_acmp_talker_sm*) data structure, is passed as an argument to the above functions.

7.2.4 AECP module

The AVDECC Enumeration and Control Protocol (AECP) is implemented in *libavdecc* as the *AECP* module. The *libavdecc*'s transport controller passes all the AECP Data Units (AECPDUs) it receives to this (AECP) module. Once initialized, the AECP module handles all messages that involve enumeration and control of the features on an AVDECC end station. It also processes all AECP messages to provide information about the capabilities and functional units within an end station.

The *libavdecc* library provides functions that allow an application to initialize and destroy the AECP module. These functions are shown in Listing 7.11.

```
avdecc_aecp_init()
avdecc_aecp_cleanup()
```

Listing 7.11: *libavdecc*'s AECP functions

An instance of the AECP module, in the form of a data structure (*avdecc_aecp*), is passed to the functions shown in Listing 7.11.

The AECP module interacts with the AVDECC Entity Model (AEM), and can be used to query and manipulate the AEM. All messages handled by this module conform to one of the formats of the AECP Data Unit (AECPDU) [19, pp. 258-267].

An AVDECC end station can utilize this module to allow for remote control and monitoring. AECP messages are typically unicast to a particular end station.

7.2.5 AEM container

The *libavdecc* library implements an *AEM container*, which provides a data structure for creating an AVDECC Entity Model (AEM). The *AEM container* consists of a number of C-structures that can be used to model the internal units within an AVDECC end station as defined the IEEE 1722.1 AEM [19, pp. 31-107]. AEM is described in section 3.4.3.1 on page 96.

The AEM container data structure allows a software developer to model an AVDECC entity by instantiating the appropriate data structures that corresponds to the descriptors that should be included in the entity's AEM. These data structures are implemented in *libavdecc*.

By utilizing the AEM container API, libavdecc is able to expose the AEM created for an entity to an AVDECC controller. This interaction is depicted in Figure 7.2.

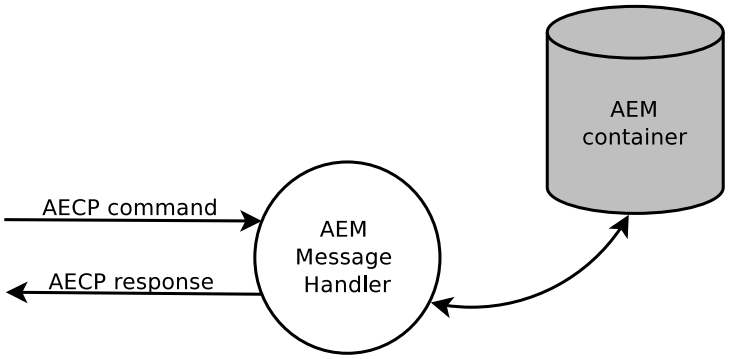


Figure 7.2: *libavdecc* exposes AEM of an AVDECC entity

When an *'AECP command'* is received by *'AEM Message Handler'* shown in Figure 7.2, it interacts with the *'AEM container'*, then returns the appropriate *'AEM response'*. For instance a READ_DESCRIPTOR command (which was described in section 3.4.3.1 on page 96) would cause *'AEM Message Handler'* to obtain the value of the specified descriptor and return it (the descriptor) in the *'AEM response'*. Each AEM descriptor is identified by the type and 16-bit index.

libavdecc provides a number of functions that can be used to build-up the AEM container. These are:

aem_create_container();
aem_delete_container();
aem_add_descriptor(void descriptor);*
aem_delete_descriptor(uint16 type, uint16 index);

The *aem_create_container()* function creates the AEM container, which is effectively a linked list, while *aem_delete_container()* is used to delete the container including all of its contents (descriptors). *aem_add_descriptor(void* descriptor)* is used to add any of the descriptor data structures to the AEM container, while *aem_delete_descriptor(uint16 type, uint16 index)* removes a particular descriptor (identified by the *type* and *index* arguments) from the AEM container.

7.3 Transform based description of *libavdecc*

Following the description of the components that make up *libavdecc* in section 7.2, this section describes how *libavdecc* processes the layer 2 packets that it receives from the network. Figure 7.3 depicts *libavdecc* in the form of a transformation schema based on the Ward and Mellor approach [154].

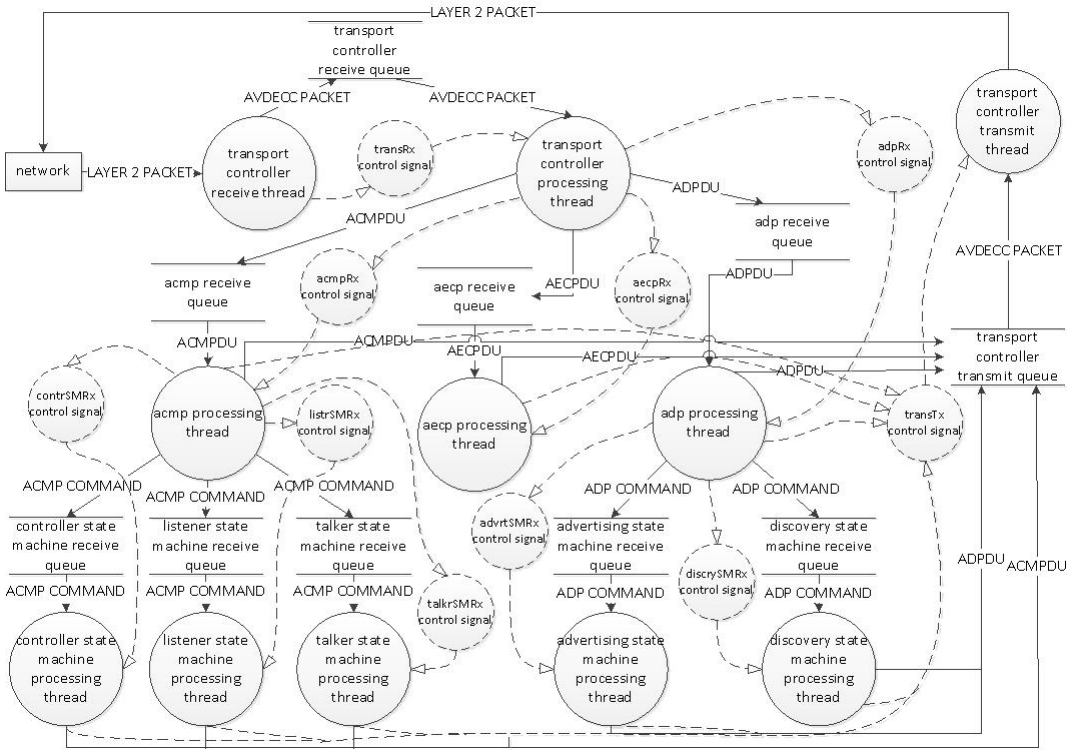


Figure 7.3: Transformation schema of *libavdecc*

In Figure 7.3, the arrows represent message *flows*, and the circles represent processing points. These processing points are known as *transforms* , and they have been implemented as processing threads. The parallel line structures (in the figure) represent *stores*, which are queues that hold items that are processed by the transforms (threads). The rectangle at the top-left of Figure 7.3 represents the external network from and to which layer 2 packets are received and transmitted, respectively.

The activities of the *libavdecc* depicted in Figure7.3, are as follows:

- The '*transport controller receive thread*' receives a layer 2 AVDECC packet from the '*network'*, adds it to the '*transport controller receive queue*', then it signals the '*transport controller processing thread*'.

- The '*transport controller processing thread*' is responsible for processing AVDECC packets from the '*transport controller receive queue*' . Depending on the *subtype* field of the AVDECC message, the '*transport controller processing thread*' adds the data unit (DU) into either the '*adp receive queue*', '*aecp receive queue*', or '*acmp receive queue*', then it signals the appropriate thread.

- the '*transport controller transmit thread*' is signaled whenever an AVDECC packet has been added to the '*transport controller transmit queue*', and it is responsible for transmitting the (AVDECC) packets to the network.

- For ADP message processing, the '*adp processing thread*' receives ADPDUs from the '*adp receive queue*'. Based on the *message-type* field, it adds the ADP command into either the '*advertising state machine receive queue*' store or '*discovery state machine receive queue*' store. Then it signals the appropriate processing thread that handles the ADP command store.

- The '*advertising state machine processing thread*' implements the ADP advertising state machine that has been described in section 7.2.2.1. It receives ADP commands from the '*advertising state machine receive queue*' when signaled by the '*adp processing thread*'. The '*advertising state machine processing thread*' adds its response ADPDU to the '*transport controller transmit queue*' and signals the '*transport controller transmit thread*' to inform it that there is a data waiting for onward transmission to the network.

- The '*discovery state machine processing thread*' implements the ADP discovery state machine which was described in section 7.2.2.2. It receives its ADP commands from the '*discovery state machine receive queue*'when signaled by the '*adp processing thread*'. It adds its responses to the '*transport controller transmit*

queue' and informs the '*transport controller transmit thread*' that there is data for onward transmission to the network with the '*transTx control signal'*.

- For AECP message processing, the '*aecp processing thread*' retrieves an AECPDU from the '*aecp receive queue*' when it has been signaled with the '*aecpRx control signal'*. The manner in which the AECPDUs are processed depends on their *message-type* field. The current *libavdecc* implementation is designed to process only AECPDUs whose message type is *AEM_COMMAND* [19, pp. 259]. Such messages are addressed to the AEM model for processing (which has been described in section 7.2.5. All responses from the '*aecp processing thread*' are added to the '*transport controller transmit queue*' and the '*transport controller transmit thread*' is signaled to inform it that it has data ready for transmission on the network.

- For ACMP message processing, the '*acmp processing thread*' receives ACMPDUs from the '*acmp receive queue*'. Based on the *message-type* field, it adds the ACMP command into either the '*controller state machine receive queue*', '*listener state machine receive queue*', or '*talker state machine receive queue*' store, then it signals the processing thread for the appropriate state machine.

- The '*controller state machine processing thread*' implements the ACMP controller state machine that has been described in section 7.2.3.1, and it is signaled by the '*contrSMRx control signal'*. When signaled, it retrieves an ACMP command from the '*controller state machine receive queue*' and processes it. It transmits its response on the network by adding it to the '*transport controller transmit queue*' and signaling the '*transport controller transmit thread*'.

- The '*listener state machine processing thread*' implements the ACMP listener state machine that has been described in section 7.2.3.2, and it is signaled by the '*listrSMRx control signal'*. It transmits its response on the network by adding it to the '*transport controller transmit queue*' and signaling the '*transport controller transmit thread*'.

- The '*talker state machine processing thread*' implements the ACMP talker state machine that has been described in section 7.2.3.3, and it is signaled by the '*talkrSMRx control signal'*. It transmits its response on the network by adding it to the '*transport controller transmit queue*' and signaling the '*transport controller transmit thread*'.

The *libavdecc* only 'picks up' layer 2 packets that are either addressed to the Ethernet

hardware address of its host, or to the AVDECC multicast MAC address (refer to section 7.2.1).

7.4 AVDECC end station

In the design of the layer 2 (AVDECC) end station, the requirement was that the end station should be capable of being discovered in accordance with the AVDECC Discovery Protocol (ADP), and also that the end station complies with the AVDECC Connection Management Protocol (ACMP). Figure 7.4 shows the interaction between the AVDECC end station implementation and the external libraries it utilizes.

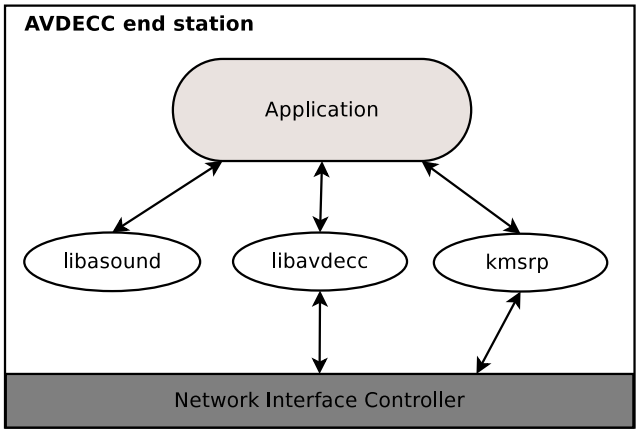


Figure 7.4: Overview of AVDECC end station

The AVDECC end station runs on a Linux PC, and is capable of receiving and transmitting audio streams from and to (respectively) an Ethernet AVB network. In order to do this, the end station utilizes the following software libraries:

- *libavdecc* for discovery and connection management
- *kmsrp* for audio stream resource reservation on the Ethernet AVB network
- *libasound* for analog audio capture and playback

The *libavdecc* software library has been described in section 7.2, and provides the AVDECC functionality for discovery and connection management.

The *kmsrp* software library is an implementation of the MSRP functionality, which:

- enables an Ethernet AVB device to request that the necessary resources it requires for transmitting a stream are reserved on the Ethernet AVB network.

- ensures that the source stream is advertised (at regular intervals) on the network.
- enables an Ethernet AVB device to indicate interest in a particular stream [155].

libasound is the Advanced Linux Sound Architecture (ALSA) software library, that enables a Linux PC to capture and playback audio from it's analog audio inputs and outputs, respectively [151].

With regards to audio inputs and outputs, the AVDECC end station has:

- one analog audio stereo input
- one analog audio stereo output
- one Ethernet AVB source
- one Ethernet AVB sink

These are shown in Figure 7.5.

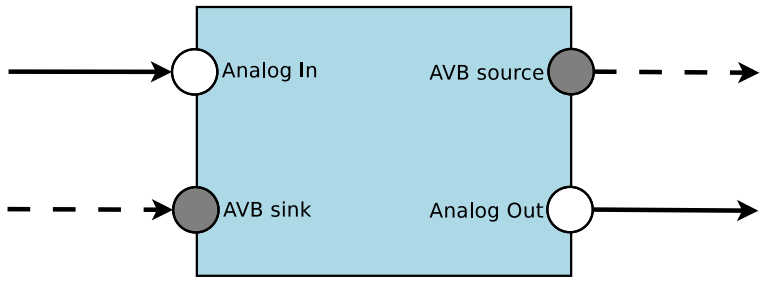


Figure 7.5: Conceptual view of inputs and outputs on the AVDECC end station

The audio input, output, sink stream, and source stream shown in Figure 7.5 are stereo (2 channels). Routing of audio between the inputs and outputs by a remote controller has not been implemented on the AVDECC end station. However, when the end station has reserved the necessary network resources for its source stream, its stereo *'Analog In'* input stream is routed to its *'AVB source'*. In a similar manner, a 2-channel audio stream at the end station's *'AVB sink'* is routed to its *'Analog Out'*.

7.4.1 Discovering the AVDECC end station

The AVDECC end station implements device discovery as defined by the AVDECC protocol, and utilizes the *libavdecc*'s advertising state machine, which has been described in section 7.2.2.1. At regular intervals, the AVDECC end station advertises its presence on

the network, and it is capable of responding to ENTITY_DISCOVERY messages from an AVDECC controller. Figure 7.6 shows the device discovery interaction between an 'AVDECC controller' and the 'AVDECC end station'.

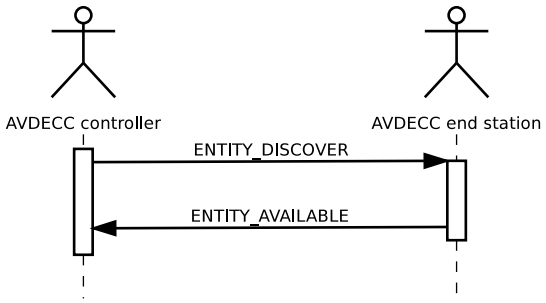


Figure 7.6: AVDECC device discovery mechanism

In order to discover AVDECC end stations on a network, the following steps are performed:

- An AVDECC controller multicasts an AVDECC message with a message type of 'ENTITY_DISCOVER' as shown in Figure 7.6.
- The message is received by the *AVDECC transport controller* module within a participating end station.
- The ADPDU is passed on to the *AVDECC adp* module.
- The ADP module passes the message to its *advertising state machine.*
- The advertising state machine sends a response with a message type of 'ENTITY_AVAILABLE' to its *AVDECC adp* module.
- The ADP module passes the ADPDU to the AVDECC transport controller of its host.
- The transport controller multicasts the response on the network.
- This response is received by the remote AVDECC controller.

7.4.2 Connection management on AVDECC end station

Connection management is the process of establishing and destroying stream connections. In the case of the AVDECC end station, these streams are audio stream connections. The AVDECC end station conforms to the AVDECC Connection Management

Protocol (ACMP) defined in the IEEE 1722.1 standard, and it utilizes *libavdecc* for connection management.

The AVDECC end station can fulfill the role of AVDECC listener (that is the destination of an audio stream) and AVDECC talker (that is the source of an audio stream). In order to fulfill the role of AVDECC listener, the end station utilizes the *listener state machine* that has been described in section 7.2.3.2. To assume the role of AVDECC talker, the end station utilizes the *talker state machine* that has been described in section 7.2.3.3.

libavdecc implements *controller connect* and *controller disconnect* modes for connection management, which are defined by the AVDECC protocol [19, pp. 245]. In these modes of connection management, all commands to either establish or destroy a stream connection are addressed to the AVDECC listener. The listener is responsible for issuing the appropriate command to the AVDECC talker, indicating that it wishes to receive a particular Ethernet AVB source stream. To establish a connection, the *'message type'* field of the ACMP message that is transmitted by the AVDECC controller to the AVDECC listener is CONNECT_RX_COMMAND. The *controller connect mode* for connection management that is defined by the AVDECC protocol, is shown in Figure 7.7.

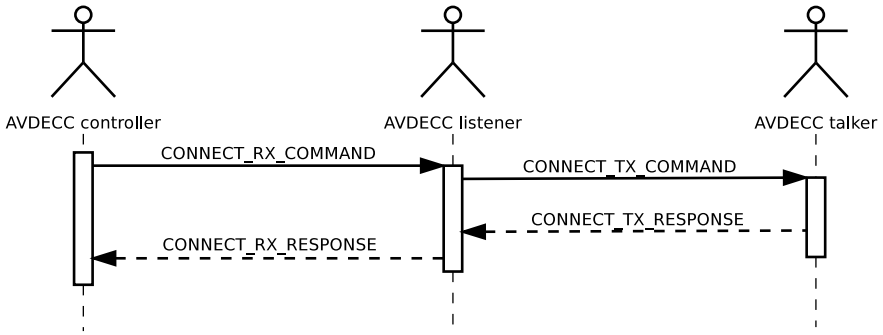


Figure 7.7: AVDECC controller connect mode for connection management

In Figure 7.7, when the AVDECC listener receives a CONNECT_RX_COMMAND command from the controller, it issues a CONNECT_TX_COMMAND to the AVDECC talker. In response, the AVDECC talker sends a CONNECT_TX_RESPONSE to the AVDECC listener. The listener in turn responds to the AVDECC controller's command to establish a connection by sending a CONNECT_RX_RESPONSE to the controller.

A similar procedure is followed for destroying a stream connection, and this is shown in Figure 7.8.

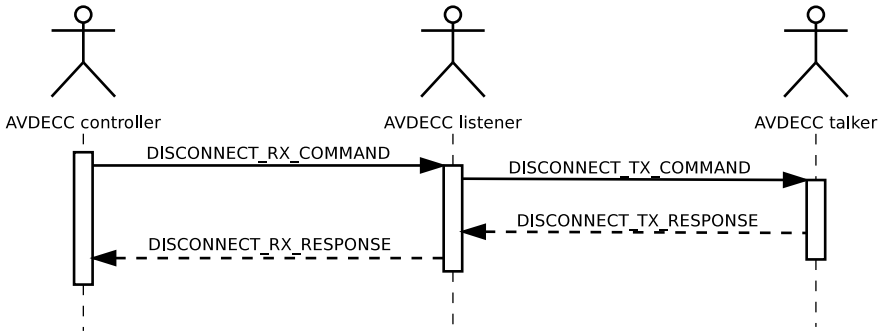


Figure 7.8: AVDECC controller disconnect mode for connection management

The connection management commands and responses shown in Figure 7.7 and Figure 7.8 are encapsulated within an ACMPDU. The ACMPDU is defined in the IEEE 1722.1 document [19, pp. 239]. For this discussion, the particular ACMPDU fields that are of interest are:

- *controller_guid* - is the 64-bit unique identifier of the AVDECC controller issuing a command.
- *talker_guid* - is the 64-bit unique identifier of the AVDECC talker.
- *listener_guid* - is the 64-bit unique identifier of the AVDECC listener.
- *talker_unique_id* - is a 16-bit field which specifies the particular source stream on the AVDECC talker.
- *listener_unique_id* - is a 16-bit field which specifies the particular sink stream on the AVDECC listener.

The status of an issued command is obtained from the *status* field of the corresponding response.

The following sections will describe how the AVDECC end station, utilizing *libavdecc*, is able to fulfill the role of AVDECC listener and talker.

7.4.2.1 AVDECC end station as AVDECC listener

By utilizing the *listener state machine* implemented in *libavdecc*, the AVDECC end station is able to fulfill the role of AVDECC listener. The *listener state machine* enables the end station to receive and process ACMP commands that are addressed to it, as well as adequately respond to the received command.

Connection management as implemented by AVDECC end station conforms with the ACMP *controller connect* and *controller disconnect* modes that were described in section 7.4.2. When the AVDECC listener receives an ACMP connect command (CONNECT_RX_COMMAND) that has a '*listener_guid*' field the same as its (the listener's) own, it is required to return an ACMP (CONNECT_RX_RESPONSE) response to the AVDECC controller which is identified by the '*controller_guid*' field in the ACMPDU. In between receiving this command and transmitting a response, the following occurs:

- the listener multicasts an ACMP (CONNECT_TX_COMMAND) command to an AVDECC talker (identified by the *'talker_guid'* field), indicating its intention to receive from one of its source streams (identified by the *'talker_unique_id'* field)
- upon receiving a response from the talker, the listener determines whether or not the talker is capable of transmitting the stream, by inspecting the *'status'* field of the received response
- if the '*status'* field of the ACMP response from the talker indicates 'SUCCESS', the listener interprets this as implying that the talker is in a state that it can start transmitting the stream. In this case, the listener:
  - obtains the stream ID of the talker's source stream from the received response
  - utilizes MSRP to indicate to the Ethernet AVB network that it is interested in receiving a particular stream (which is uniquely identified by the obtained stream ID) from the network
  - routes its '*AVB sink*' input (identified by the *'listener_unique_id'* field of the ACMP command it received from the controller) to its '*Analog Out*' output shown in Figure 7.5. The AVDECC listener utilizes *libasound* to transmit the audio it receives from its *'AVB sink'* input to its *'Analog Out'*
  - returns an ACMP (CONNECT_RX_RESPONSE) response, with its *'status'* field indicating 'SUCCESS', to the AVDECC controller
- if the *'status'* field of the ACMP (CONNECT_RX_RESPONSE) response from the talker does not indicate 'SUCCESS', an ACMP (CONNECT_RX_RESPONSE) response is returned by the listener to the AVDECC controller. The *'status'* field of this response is the same as that returned by the AVDECC talker's (CONNECT_TX_RESPONSE) response.

When a controller receives an ACMP (CONNECT_RX_RESPONSE) response with a *'status'* field of 'SUCCESS', it interprets this as implying that an audio stream connection has been established between the talker and listener.

7.4.2.2 AVDECC end station as AVDECC talker

The AVDECC end station created is able to fulfill the role of IEEE 1722 audio stream source. As shown in Figure 7.5, the AVDECC end station has only one source stream ('*AVB source*').

At startup, the end station:

- utilizes Multicast Address Acquisition Protocol (*MAAP*) to obtain a multicast MAC address for its IEEE 1722 source stream, and
- generates a 64-bit unique identifier (known as *stream ID*) for its source stream.

The AVDECC end station utilizes the *talker state machine*, implemented by *libavdecc*, to fulfill the role of AVDECC talker. The *talker state machine* enables the end station to receive and process ACMP commands that are addressed to it, as well as adequately respond to the received commands.

The AVDECC end station conforms to the ACMP *controller connect* and *controller disconnect* modes that were described in section 7.4.2. When an AVDECC talker end station receives an ACMP connect command (CONNECT_TX_COMMAND) that has a *'talker_guid'* field that is the same as its own entity GUID, it is required to return an ACMP (CONNECT_TX_RESPONSE) response to the AVDECC listener which is identified by the *'listener_guid'* field. In between receiving this command and transmitting a response, the following occurs:

- the AVDECC talker utilizes MSRP to request resource reservation for its stream
- if the MSRP stream reservation request is successful, the talker:
  - adds the 64-bit stream ID of its source stream to a response ACMPDU. The source stream is identified by the *'talker_unique_id'* field.
  - adds the multicast MAC address (obtained via *MAAP*) associated with its source stream, to the response ACMPDU
  - utilizes *libasound* to capture stereo audio from its '*Analog In*' input, then routes the (2-channel) audio to its '*AVB source*' output

  - multicasts an ACMP (CONNECT_TX_RESPONSE) response with a *status* field that indicates 'SUCCESS'

- if the MSRP stream reservation request failed, or if the specified source does not exist, an appropriate *'status'* field value is transmitted in the ACMP (CONNECT_TX_RESPONSE) response.

A listener receiving an ACMP (CONNECT_TX_RESPONSE) response with a *status* field of 'SUCCESS', interprets this as implying that the AVDECC talker has reserved the necessary network resources for its stream, and has started streaming audio from its source stream.

7.5 Summary

IEEE 1722.1 defines layer 2 protocols that will enable the discovery, enumeration, control and connection management of devices on an Ethernet AVB network. These protocols are commonly referred to as the AVDECC protocol. The IEEE 1722.1 standard requires that compliant networked devices are able to transmit and/or receive data according to the format defined by the IEEE 1722 standard. This chapter described the implementation and design of a software library known as *libavdecc*, which implements the IEEE 1722.1 standard. *libavdecc* is implemented in a modular structure that allows a software developer to utilize only those components that are required. The components of *libavdecc* implement the various sub-protocols defined by AVDECC. These include the:

- AVDECC Discovery Protocol (ADP)
- AVDECC Enumeration and Control Protocol (AECP)
- AVDECC Connection Management Protocol (ACMP)

The *libavdecc* library implements the ADP and ACMP state machines, which enables an AVDECC device to appropriately process and respond to commands.

The implementation of an AVDECC end station was also described in the chapter. This AVDECC end station runs on a Linux PC, and is capable of transmitting and receiving audio streams to and from (respectively) an Ethernet AVB network. The end station fulfills the role of AVDECC talker and listener by utilizing the *libavdecc* library. Thus it is able to receive device discovery and connection management AVDECC commands from a remote AVDECC controller, and respond appropriately in compliance with the IEEE 1722.1 standard.

Chapter 8

Layer 2/Layer 3 Proxy Implementation

The previous chapter described the implementation of an AVDECC end station that can be remotely configured to transmit and receive IEEE 1722 streams on an Ethernet AVB network. At the time of the implementation there were no commercially available AVDECC end stations, as a result the IEEE 1722.1 standard was implemented as a software library that was used to create an AVDECC compliant audio streaming PC (workstation) device.

It may be desirable to enable interoperability between the AVDECC end station and Ethernet AVB end stations that implement other control protocols. Such that an IEEE 1722 audio stream that is transmitted by an AVDECC end station can be received by another AVDECC end station as well as the other (non-AVDECC compliant) end stations on the network. In the same manner it may be desirable for the IEEE 1722 audio stream that is transmitted by any of the other (non-AVDECC compliant) end stations to be received by the AVDECC end stations and non-AVDECC compliant end stations on the network. The procedure for setting up such stream connections is known as connection management.

The command translator approach (which was described in chapter 4) could allow for such interoperability and common control between the AVDECC end station and Ethernet AVB end stations that implement other audio control protocols.

In chapter 3 a *layer 2 audio control protocol* was described as an audio control protocol that encapsulates its control messages within OSI layer 2 packets, as is the case with IEEE 1722.1. A *layer 3 audio control protocol* was described as an audio control protocol that encapsulates its control messages within OSI layer 3 packets, in the manner that AES-64 does.

Connection management is being used as the criteria for investigating the command

translation approach for interoperability between Ethernet AVB end stations that implement a layer 2 audio control protocol and those that implement a layer 3 audio control protocol.

In this chapter the command translator approach is investigated to determine whether it is capable of:

- enabling connection management of networked devices that implement a layer 2 audio control protocol
- enabling connection management of networked devices that implement layer 2 and layer 3 audio control protocols

8.1 Introduction

An AVDECC end station is capable of transmitting and/or receiving IEEE 1722 audio streams on an Ethernet AVB network. In order to allow for remote configuration and control, it implements the IEEE 1722.1 standard (also known as the AVDECC protocol). An AVDECC controller is an end station that can remotely configure an AVDECC talker to be the source of an (IEEE 1722) audio stream, and configure an AVDECC listener to receive the audio stream. The control messages transmitted by the AVDECC controller are encapsulated within OSI layer 2 (Ethernet) packets, hence the AVDECC protocol has been classified as a *layer 2 audio control protocol* (refer to chapter 3).

AVDECC end stations (controllers, talkers, and listeners) are incorporated into an Ethernet AVB network. There may be desirable Ethernet AVB end stations that implement other audio control protocols, which a sound engineer wishes to include into the network. Communication between these non-AVDECC compliant end stations is defined by their audio control protocol. It is possible that the other audio control protocols utilize a different OSI layer for communication, such as the OSI layer 3. For instance, the AES-64 audio control protocol utilizes the Internet Protocol (IP), which is an OSI layer 3 protocol, for transporting its messages. Note that the AES-64 has been described as a *layer 3 audio control protocol* because it encapsulates its messages within IP packets.

In the scenario above, both AVDECC and AES-64 end stations on the Ethernet AVB network are capable of transmitting and receiving IEEE 1722 audio streams, it might be desirable for audio stream connections to exist between the end stations, irrespective of the audio control protocol that they implement. That is, for audio streams transmitted by an AVDECC end station to be received by an AES-64 end station, and for streams

transmitted by an AES-64 end station to be received by an AVDECC end station. This would require configuring participating end stations to transmit and/or receive the audio stream(s). This problem has been described as the *interoperability challenge* in chapter 4.

In order to allow for common control and interoperability between networked audio devices, in particular layer 2 and layer 3 audio control protocols, the command translation approach of chapter 4 can be used. To demonstrate this (command translation) solution, a proxy that is capable of communication with layer 2 and layer 3 Ethernet AVB end stations was created. Figure 8.1 depicts the conceptual layout of a proxy that can facilitate interoperability such an Ethernet AVB network.

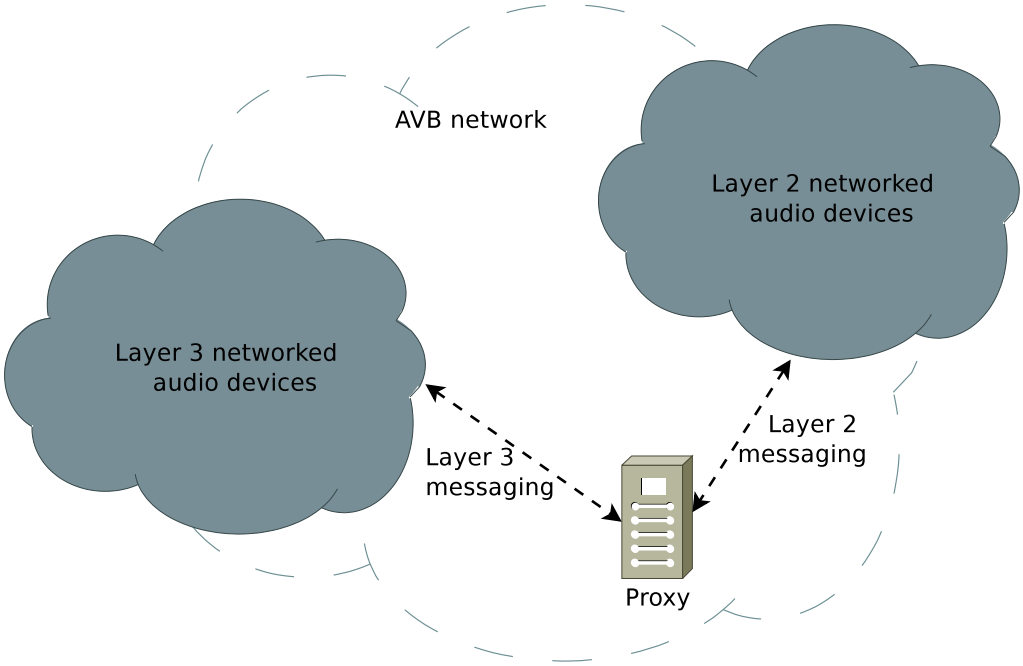


Figure 8.1: Conceptual view of layer 2 proxy approach

As shown in Figure 8.1, a '*Proxy*' is situated between a layer 3 and a layer 2 network of audio devices. The '*Proxy'* receives a layer 3 message, translates it into the appropriate layer 2 message, then transmits the layer 2 message to the layer 2 network. In the same manner, it receives a layer 2 message, translates it into the appropriate layer 3 message, then transmits it to the layer 3 network. This proxy approach for device interoperability also allows for common control of the networked layer 2 and layer 3 devices.

In this chapter, a description of the design and implementation of an AVDECC (layer 2) proxy in provided. The implementation allows a device that implements a layer 3 protocol (AES-64) to control AVDECC end stations. It translates layer 3 IP messages to layer 2 Ethernet messages. The proxy enables an AES-64-based network monitor and

device control application to establish and destroy audio stream connections between AVDECC end stations, as well as between AES-64 and AVDECC end stations.

The motivations for using AES-64 in this investigation were:

- AES-64 was co-developed with the Rhodes University audio networking research group, where the project is situated.
- AES-64 devices were readily available.
- The AES-64 protocol has been standardized by the Audio Engineering Society (AES).
- The AES-64 protocol incorporates features that are common to most contemporary IP control protocols. These include a connection management procedure, device discovery mechanism, device enumeration, and device control.
- The AES-64 protocol incorporates advanced audio networking features such as joins, modifiers and grouping [109].
- AES-64 defines a fixed 7-level hierarchy for parameter modeling, which provide a consistent way to represent and address device controls [17].
- There is an AES-64-based network monitor and control application (UNOS Vision) which is provided freely by its developer - UMAN [156].

The following section describes the design of the AVDECC proxy for device interoperability and common control of layer 2 and layer 3 networked audio devices.

8.2 AVDECC Proxy Design

The AVDECC proxy created in this investigation is capable of layer 3 (IP packets) and layer 2 (Ethernet frames) communication. It can receive a command within an IP message, and translate it to the corresponding Ethernet layer 2 message. It is also capable of receiving a command encapsulated within a layer 2 frame, and translates it into the appropriate layer 3 IP message. In particular the AVDECC proxy was created for AES-64 (layer 3) to AVDECC (layer 2) communication.

The interaction between a layer 3 protocol (AES-64) and a layer 2 protocol (AVDECC), via the proxy, is shown in Figure 8.2.

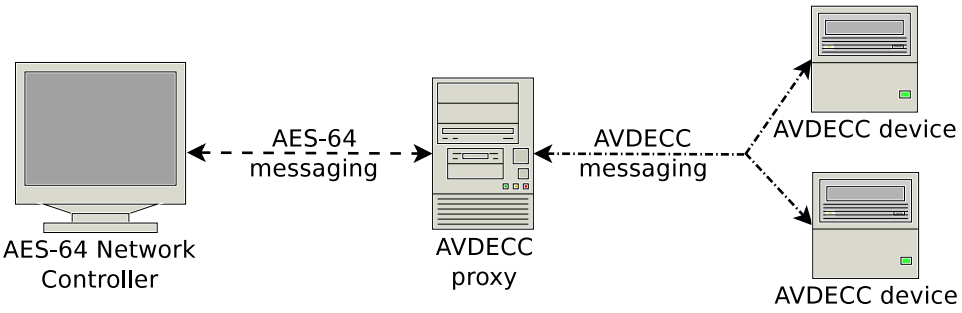


Figure 8.2: AVDECC proxy for integrated network communication

From Figure 8.2, when the '*AVDECC proxy*' receives an AES-64 message from the '*AES-64 Network Controller*', it translates the message to a corresponding AVDECC message, which it then sends to the appropriate '*AVDECC device*'. The AES-64 message could be an instruction to establish a connection between an AVDECC talker and an AVDECC listener. Any response from the '*AVDECC device*' is received by the '*AVDECC proxy*', translated to the appropriate AES-64 message, then it is sent to the '*AES-64 Network Controller*'.

It is also possible to use the AVDECC proxy to allow for common control of networked AES-64 and AVDECC devices. This is shown in Figure 8.3.

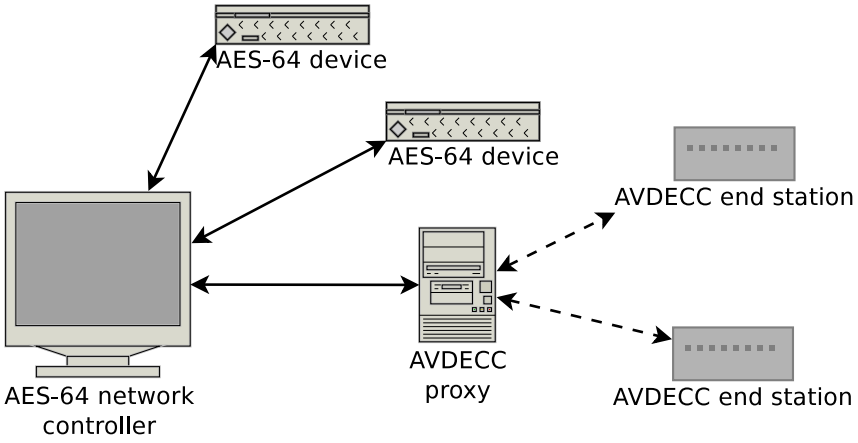


Figure 8.3: Common control of networked layer 2 and layer 3 devices

In Figure 8.3 the arrows with solid lines represent layer 3 (IP-based) AES-64 messages, and the arrows with broken lines represent layer 2 AVDECC messages.

In the figure, an '*AES-64 network controller*' transmits an AES-64 message to set the mute control of all devices on the network. Note that this is possible in AES-64 when the mute parameter on each of the networked device has been '*joined*'. A join is when parameters are grouped in a relationship, such that a change to the value of one parameter could result in a change to the others in the group [109].

The '*AVDECC proxy*' receives AES-64 messages on behalf of the AVDECC end stations. It translates the mute (AES-64) message to the corresponding AVDECC command and sends it to the appropriate '*AVDECC end station*'. Thus enabling the networked AES-64 and AVDECC devices to be controlled by a common '*AES-64 network controller*'.

The AVDECC proxy is capable of:

- discovering AVDECC end stations on a network,
- exposing the discovered AVDECC end stations to AES-64 devices and AES-64 network controllers, and
- enabling connection management over AVDECC end stations.

The requirements of the proxy are depicted in the form of a use case diagram in Figure 8.4.

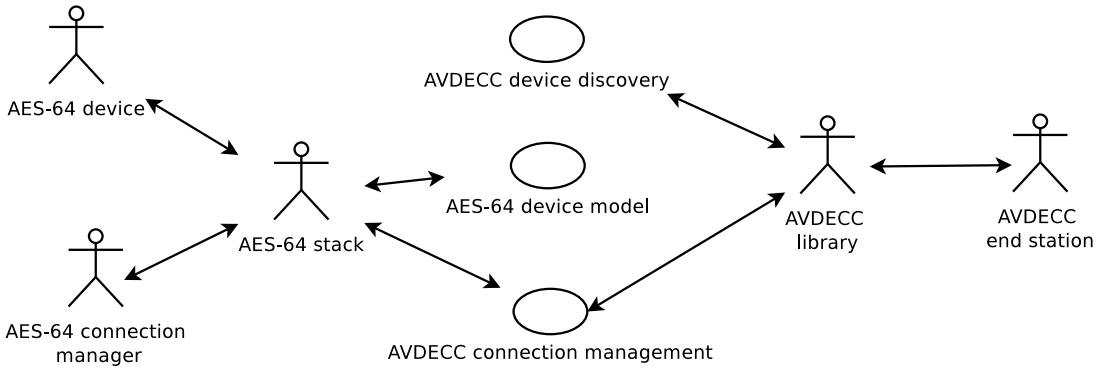


Figure 8.4: Use case diagram of AVDECC proxy

The proxy utilizes the '*AES-64 stack*' for AES-64 messaging with the '*AES-64 connection manager*'. It utilizes the '*AVDECC library*' for layer 2 (AVDECC) communication with the '*AVDECC end station*'. The '*AVDECC library*' refers to the software library ('*libavdecc*'), which is an implementation of the AVDECC protocol. The *libavdecc* software library has been described in chapter 7 on page 175.

The proxy is capable of discovering all AVDECC end stations by utilizing the ADP component of the '*AVDECC library*' to discover AVDECC end stations on the network. For each discovered AVDECC end station, the proxy creates an AES-64 device node, which is discoverable and accessible from a remote AES-64 device such as the '*AES-64 connection manager*' in Figure 8.4.

Utilizing the proxy, the '*AES-64 connection manager*' is able to establish and destroy audio stream connections on the '*AVDECC end station*'. The connection manager is also able to establish and destroy audio stream connections between an '*AES-64 device*' and an '*AVDECC end station*' via the proxy. The AVDECC proxy is able to fulfill the connection management procedure by utilizing the ACMP component of the '*AVDECC library*'.

The overall structural layout of the AVDECC proxy running on a PC is shown in Figure 8.5.

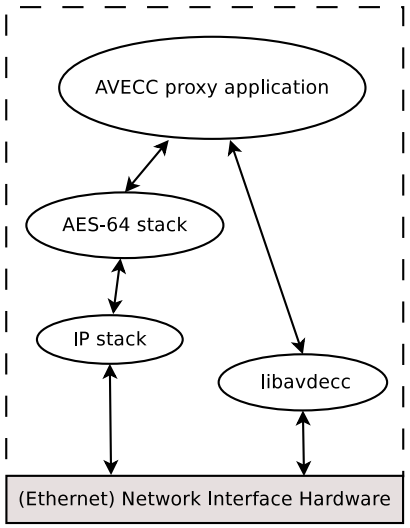


Figure 8.5: Structural layout of AVDECC proxy

As shown in Figure 8.5 the '*AVDECC proxy application*' utilizes the '*libavdecc'* software library for layer 2 (AVDECC) messaging, and it utilizes the '*AES-64 stack*' for layer 3 (AES-64) messaging. AES-64 is an IP-based protocol, hence all AES-64 messages are via the host's '*IP stack*'. The translation of commands from AES-64 to AVDECC (and vice versa) are implemented within the '*AVDECC proxy application*'. The following section provides details of the implementation of the AVDECC proxy.

8.3 AVDECC Proxy Implementation

The AVDECC proxy has been implemented as a Linux (*kernel version 3.0.0-21*) PC application that enables the discovery and connection management of AVDECC end stations via AES-64 messaging. Thus it is capable of both AES-64 and AVDECC communication. For AES-64 messaging the AVDECC proxy utilizes the *AES-64 stack* (*version 1.0.6*), and for AVDECC messaging it utilizes the *libavdecc* (*version 0.1*) software

library that has been described in section 7.2 on page 178. The AVDECC proxy is depicted in the form of a class diagram in Figure 8.6.

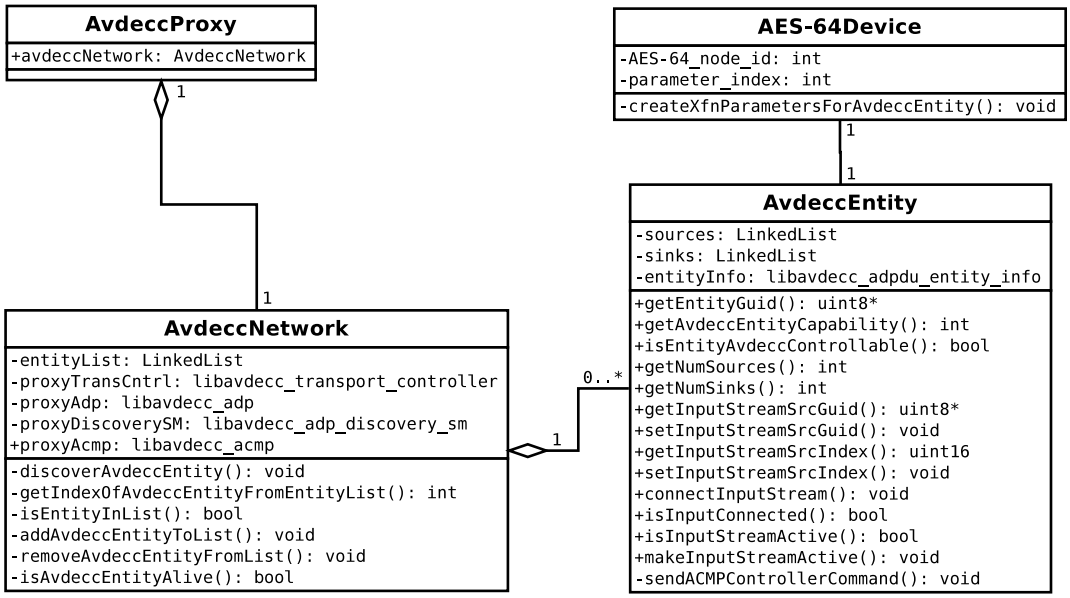


Figure 8.6: Class diagram for AVDECC proxy

An object of the '*AvdeccProxy*' class is the proxy's start-up object which creates the '*AvdeccNetwork'*. The '*AvdeccNetwork'* object initializes *libavdecc*'s transport controller module and the *AES-64 stack* within its constructor. It also initializes the ADP module, the ACMP module and ADP discovery state machine. Following this, it multicasts a *discovery* AVDECC message on the network. Each AVDECC end station on the network responds to this *discovery* message by sending an AVDECC ADP *available* message, which contains (discovery) information about the end station (entity). The information returned in an ADP message includes a [19]:

- 32-bit vendor ID field
- 32-bit entity model ID field
- 16-bit field that specifies the number of source streams on the entity
- 16-bit field that specifies the number of listener streams on the entity
- 16-bit field that specifies the entity's talker capabilities
- 16-bit field that specifies the entity's listener capabilities
- 16-bit field that specifies the entity's controller capabilities

The AVDECC proxy holds a list of discovered entities. Each entity is uniquely identified by its *entity GUID*, which is a 64-bit value that is a combination of the *vendor ID* and *entity model ID* fields [19].

On receiving an ADP *available* message, the proxy checks its list to determine whether the received *entity GUID* corresponds with that of an end station that already exists in its list of discovered entities. If this entity is not in its list, the proxy creates a new instance of the '*AvdeccEntity'* class, which is an abstraction of the actual device, then adds it to the list.

For each discovered end station, the ACMP *controller, listener* and *talker* state machines are created within the constructor of its corresponding '*AvdeccEntity'* class. These state machines enable the proxy to fulfill the roles of controller, listener and talker on behalf of the particular end station. The '*AvdeccEntity'* constructor also creates an object of the '*AES-64Device'* class. A one-to-one relationship exists between an '*AvdeccEntity'* object and an '*AES-64Device'* object.

The '*AES-64Device'* class creates a number of AES-64 parameters which adequately model the connection management features of the AVDECC end station in AES-64 terms. The parameters created are described in section 8.3.1. All AES-64 interactions with an AVDECC end station are achieved by communicating with its corresponding '*AES-64Device'* object. In order to configure a device as the source or destination of a stream connection, typically a network controller has to discover the device. The device discovery process as implemented by the AVDECC proxy is described in more detail in section 8.3.2.

The following sections provide more details of how the (AVDECC) proxy models an AVDECC end station in AES-64 terms, discovers an AVDECC end station, as well as how it enables connection management.

8.3.1 AES-64 parameters for AVDECC end stations

When the AVDECC proxy discovers an AVDECC end station, it models the AVDECC end station in terms of AES-64. To do this, the proxy creates an *AES-64 node* for each AVDECC end station that it discovers. Within each *AES-64 node* are the parameters associated with the particular device. These parameters are structured according to a fixed 7-level hierarchy, that provide both a logical grouping and an addressing scheme for the parameters. Refer to chapter 3 on page 53 for a detailed description of the AES-64 protocol.

Figure 8.7 shows the design layout of *AES-64 nodes* within an AES-64 device.

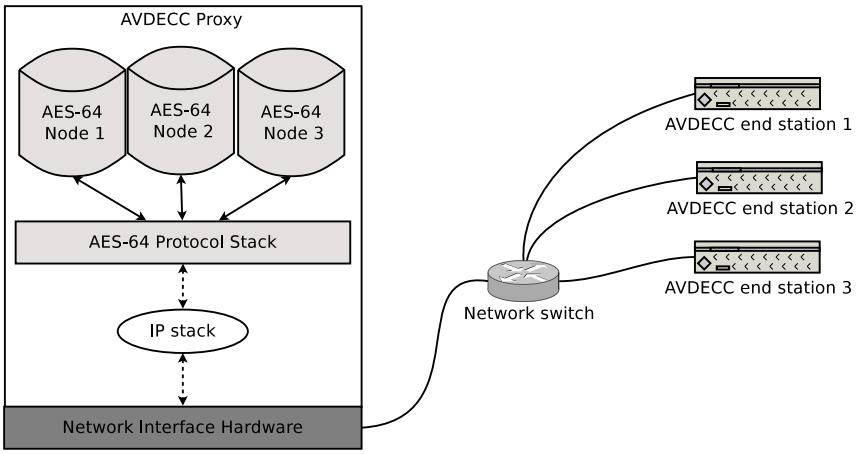


Figure 8.7: AVDECC proxy models AVDECC end stations in terms of AES-64

In Figure 8.7, the '*AVDECC Proxy*' is on the same network as three AVDECC end stations ('*AVDECC end station 1*', '*AVDECC end station 2*', and '*AVDECC end station 3*'). When the proxy discovers an AVDECC end station, it creates an AES-64 node above its *'AES-64 Protocol Stack*'. In the figure, it creates '*AES-64 Node 1*', '*AES-64 Node 2*' and '*AES-64 Node 3*' to correspond with the three AVDECC end stations discovered.

Within each of the AES-64 nodes that models an AVDECC end station, the proxy creates a number of AES-64 parameters that enable AES-64 device discovery and connection management of the corresponding AVDECC end station. These parameters are described in section 8.3.1.1 and section 8.3.1.2.

8.3.1.1 Device discovery parameters for an AVDECC end station

To enable an AES-64 device (or controller) to remotely discover an AVDECC end station, the AVDECC proxy creates a number of AES-64 *device discovery parameters.* These parameters are:

- XFN_PTYPE_IP_ADDRESS - contains the IP address of the host device (PC running the proxy application).
- XFN_PTYPE_SUBNET_MASK - holds the network subnet where the proxy resides.
- XFN_PTYPE_DEVICE_NAME - indicates the name of the AVDECC end station as a string. Currently, the proxy returns the entity GUID of the AVDECC entity as its device name.

- XFN_PTYPE_DEVICE_TYPE - holds one of the defined AES-64 device types. Within the AES-64 protocol stack, an AES-64 device type has been defined for an AVDECC end station.
- XFN_PTYPE_XFN_BOUND - indicates that the IP interface is capable of streaming media.
- XFN_PTYPE_AVDECC_ENTITY_GUID - contains the 64-bits unique identifier of the AVDECC end station.

These parameters are created within each of the *AES-64 nodes* of Figure 8.7, and they are returned in response to an AES-64 device discovery message from a remote controller. A description of how these parameters are used for device discovery is given in section 8.3.2.

8.3.1.2 Connection management parameters for an AVDECC end station

The AVDECC proxy creates a number of input and output parameters that enable an AES-64 controller remotely configure an AVDECC end station in order to establish or destroy audio stream connections. The parameters created are described below.

- Input parameters
  - XFN_PTYPE_MULTICORE_TYPE - indicates the type of multicore. The *AES-64 protocol stack* defines a number of multicore types. The AES-64 term *multicore* describes the endpoint of a media stream. A multicore contains a number of channels of media. The input multicores created by the proxy are of type '*Ethernet AVB Audio Multicore*'.
  - XFN_PTYPE_MULTICORE_NAME - contains the name of the multicore as a string. For example "1722 Input Multicore 1".
  - XFN_PTYPE_STREAM_ID - holds the 64-bit stream ID of the input multicore. At start up, the value of an input multicore's stream ID is zero, until a stream connection has been established.
  - XFN_PTYPE_LISTEN - this parameter is a *listen* parameter, which is used to indicate interest in an Ethernet AVB stream by interacting with MSRP. The connection management sequence is described in section 5.6 on page 140.
  - XFN_PTYPE_AVDECC_SRC_ENTITY_GUID - this is the 64-bit entity GUID of the talker whose source audio stream is sinked to the particular input multicore.

  - XFN_PTYPE_AVDECC_SRC_MULTICORE_ID - this is the 16-bit source multicore index on the talker whose audio stream is sinked to the particular input multicore.
  - XFN_PTYPE_AVDECC_CONTROLLER_CONNECT - this is a *connect* parameter, which is used to establish or destroy an audio stream connection. The *libavdecc* library will be used to perform the connect/disconnect.

- Output parameters

  - XFN_PTYPE_MULTICORE_TYPE - indicates the type of multicore. The proxy creates an output multicore of type *'Ethernet AVB Audio Multicore'* for an AVDECC end station.
  - XFN_PTYPE_MULTICORE_NAME - indicates the name of the multicore as a string. For instance "1722 Output Multicore 1".
  - XFN_PTYPE_STREAM_ID - holds the 64-bit stream ID of the output multicore. The output multicore's stream ID cannot be modified remotely.
  - XFN_PTYPE_ADVERTISE - an *advertise* parameter, which enables advertising of an Ethernet AVB stream this is done by interacting with MSRP.

With an AVDECC end station's connection management features exposed as AES-64 (input and output) parameters, an AES-64 controller is able to configure the end station as AVDECC talker or AVDECC listener for a particular connection. These parameters also enable an AES-64 controller to determine the state of a particular stream connection. A description of how these parameters are used to make stream connections is given in section 8.3.3.

8.3.2 Device discovery of AVDECC end stations

At start up, the AVDECC proxy discovers AVDECC end stations on the network. It does this in accordance with the AVDECC discovery protocol by utilizing the *libavdecc*'s ADP module, which has been described in section 7.2.2 on page 181.

Figure 8.8 shows how the AVDECC proxy utilizes *libavdecc* to discovery an AVDECC end station on the network.

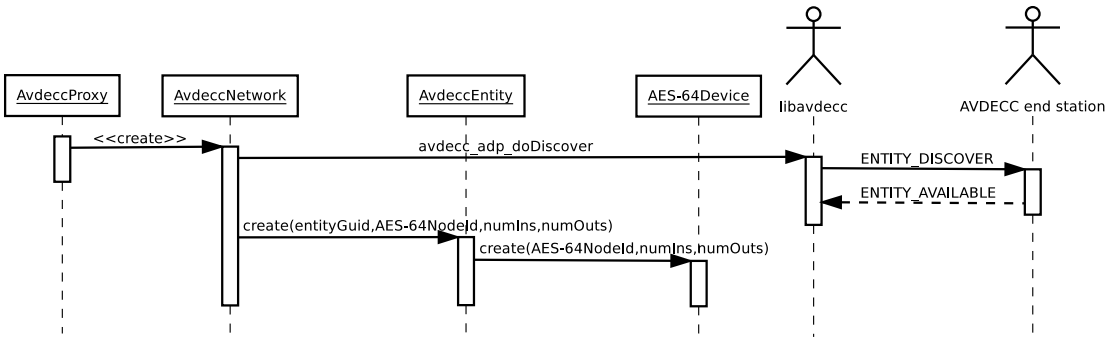


Figure 8.8: AVDECC proxy utilizes *libavdecc* to discover AVDECC end stations

As shown in Figure 8.8 when the '*AvdeccNetwork*' object is created, the following occurs:

- The '*AvdeccNetwork*' object calls the *avdecc_adp_doDiscover()* function of the *discovery state machine* of *libavdecc* to discover AVDECC end stations on the network. The *discovery state machine* multicasts an ADP *ENTITY_DISCOVER* message on the network. An AVDECC end station on the network responds to this message by transmitting an ADP *ENTITY_AVAILABLE* message on the network.

- When the '*AvdeccNetwork*' object is notified (by *libavdecc*) that a new end station has been discovered, it proceeds to create an '*AvdeccEntity*' object, which abstracts the discovered end station. In addition to the entity GUID (*entityGuid*), number of sinks (*numIns*), and number of sources (*numOuts*) parameters that are passed as arguments when creating the '*AvdeccEntity*' object, the '*AvdeccNetwork*' object also passes an AES-64 node ID (*AES-64NodeId*) argument which is assigned to the AES-64 node that corresponds to the '*AvdeccEntity*' object.

- The '*AvdeccEntity*' object models its features in AES-64 terms by creating the '*AES-64Device*' object, and passes the AES-64 node ID (*AES-64NodeId*) that uniquely identifies this (AES-64) node as an argument. Also passed to the '*AES-64Device*' object are the number of stream inputs (*numIns*) and the number of stream outputs (*numOuts*) that the end station possesses. The created '*AES-64Device*' object is responsible for AES-64 messaging on behalf of the corresponding AVDECC end station.

The '*AES-64Device*' object abstracts the AVDECC end station by creating a number of AES-64 parameters that are accessible to an AES-64 network controller. These parameters have been described in section 8.3.1 on page 208.

The AVDECC proxy enables AVDECC end stations to be discovered by an AES-64 network controller. Figure 8.9 shows how an AES-64 network controller (‘*UNOS Vision*’) is able to discover the AVDECC end stations on the network.

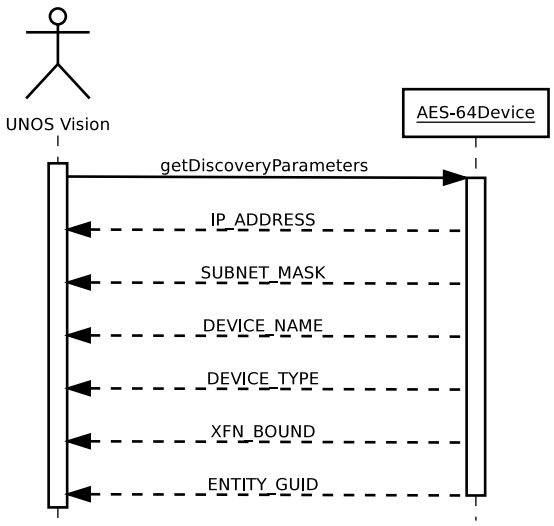


Figure 8.9: AVDECC proxy enables AES-64 discovery of AVDECC end stations

When the ‘*UNOS Vision*’ actor in Figure 8.9 broadcast an AES-64 message to discover AES-64 devices on the network, the ‘*AES-64Device*’ object responds by sending unicast AES-64 messages with the values of the AES-64 device discovery parameters that have been described in section 8.3.1.1.

Having exposed AVDECC end stations on the network to an AES-64 network controller, the AVDECC proxy should be capable of enabling connection management of the AVDECC end stations when requested by the network controller. Section 8.3.3 describes how this is achieved by the AVDECC proxy.

8.3.3 Connection management procedure for AVDECC end stations

The AVDECC proxy was created to enable connection management between AES-64 devices and AVDECC end stations on a network. It (AVDECC proxy) utilizes the three connection management state machines of *libavdecc* to fulfill different tasks. These are:

- *controller state machine* - to enable connection management between two end stations that implement the AVDECC protocol, as well as to enable the proxy to obtain information about the state of a source or sink stream connection.

- *listener state machine* - to enable the proxy to fulfill connection management between an AES-64 device (as listener) and an AVDECC end station (as talker).
- *talker state machine* - to enable the proxy to fulfill connection management commands between an AES-64 device (as talker) and an AVDECC end station (as listener).

When an AES-64 network controller (such as UNOS Vision) is utilizing the proxy, the procedure for connection management depends on whether the:

- source and destination devices are both AVDECC end stations,
- AVDECC end station is the source of the audio stream, and
- AVDECC end station is the destination of the audio stream.

Section 8.3.3.1, section 8.3.3.2 and section 8.3.3.3 provide details the procedures for each of the above mentioned features. In these discussions, *UNOS Vision* which is an AES-64 network configuration, monitoring and device control application is used as the network controller [156].

8.3.3.1 Connection management procedure between two AVDECC end stations

The steps followed by UNOS Vision when establishing a stream connection between two AVDECC end stations are:

- *UNOS Vision* sends an AES-64 message to the AVDECC talker's AES-64 node (*AES-64Device*), in order to obtain the entity GUID. That is, UNOS Vision sends a message to '*get'* the *XFN_PTYPE_AVDECC_GUID* parameter of the AVDECC talker.
- *UNOS Vision* sends an AES-64 message to *'set'* the *source entity GUID* parameter of the AVDECC listener's input multicore. That is, to modify the value of the input multicore's *XFN_PTYPE_AVDECC_SRC_ENTITY_GUID* parameter.
- *UNOS Vision* sends an AES-64 message to *'set'* the *source stream index* parameter of the AVDECC listener's input multicore. That is, to modify the value of the input multicore's *XFN_PTYPE_AVDECC_SRC_MULTICORE_ID* parameter.

- *UNOS Vision* sends an AES-64 message to the *controller connect* parameter of the AVDECC listener's input multicore, indicating that it should establish ('*enable*') a stream connection. The *controller connect* parameter is modeled as the AES-64 XFN_PTYPE_AVDECC_CONTROLLER_CONNECT parameter of the AVDECC listener's input multicore.

The interaction between the AES-64 network controller (*UNOS Vision*) and the proxy that enables the establishment an audio stream connection, is depicted in the form of a sequence diagram in Figure 8.10.

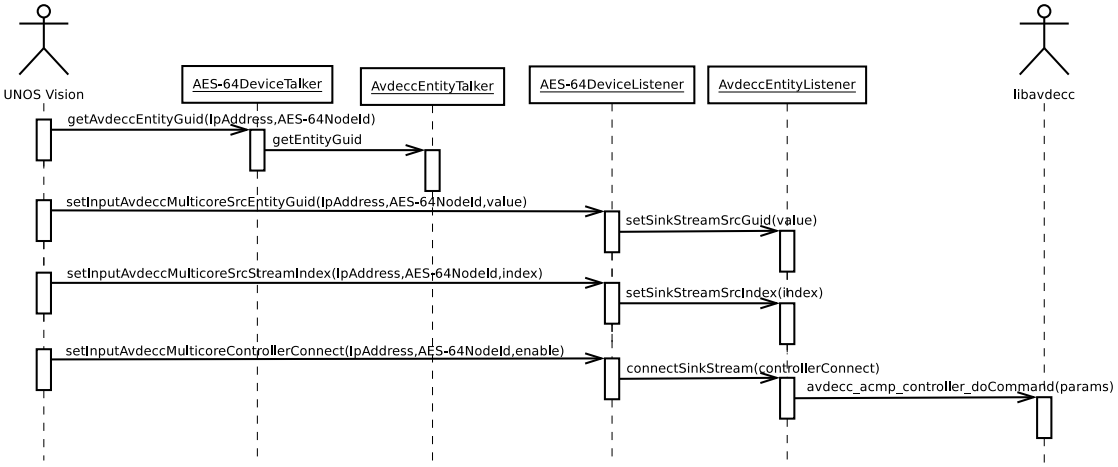


Figure 8.10: Sequence diagram for establishing an audio stream connection between AVDECC end stations

In Figure 8.10, an AES-64 message that is addressed to an AVDECC end station is processed by its corresponding 'AES-64Device' object (within the AVDECC proxy). In turn, the 'AES-64Device' object calls the appropriate method of its corresponding 'AvdeccEntity' object, which then calls a function of the '*libavdecc*' actor to execute an AVDECC command.

The interactions in Figure 8.10 are:

- The '*AES-64DeviceTalker*' object of the AVDECC talker (identified by its *AES-64NodeId*) receives a message from the '*UNOS Vision*' actor, which is effectively a *getAvdeccEntityGuid()* instruction. This causes it to obtain the talker's *entity GUID* by calling the *getEntityGUID()* method of its corresponding *'AvdeccEntityTalker'* object. The *'AvdeccEntityTalker'* object obtains the *entity GUID* of the physical end station (that it abstracts) during the device discovery process that has been described in section 8.3.2.

- The '*AES-64DeviceListener'* object of the AVDECC listener (identified by its *AES-64NodeId*) receives a message from UNOS Vision, which is effectively a *setInputAvdeccMulticoreSrcEntityGuid()* instruction. The '*AES-64DeviceListener'* object calls the *setSinkStreamSrcGuid()* method of its corresponding '*AvdeccEntityListener*' object, in response to the received instruction. The *value* argument is the entity GUID that was obtained (by UNOS Vision) from the AVDECC talker.

- The '*AES-64DeviceListener'* object of the AVDECC listener (identified by its *AES-64NodeId*) receives a message from UNOS Vision, which is effectively a *setInputAvdeccMulticoreSrcStreamIndex()* instruction. The '*AES-64DeviceListener'* object to call the *setSinkStreamSrcIndex()* method of its corresponding '*AvdeccEntityListener*' object, in response to the received instruction. The *index* argument specifies a particular source on the AVDECC talker.

- The '*AES-64DeviceListener'* object of the AVDECC listener (identified by its *AES-64NodeId*) receives a message from UNOS Vision, which is effectively a *setInputAvdeccMulticoreControllerConnect()* instruction. This causes the '*AES-64DeviceListener'* object to call the *connectSinkStream()* method of its corresponding '*AvdeccEntityListener*' object. The *controllerConnect* argument indicates that the connection management procedure is according to the *controller connect mode* described in chapter 7 section 7.4.2 on page 194. On receiving the instruction to establish a stream according to the controller connect mode of (AVDECC) connection management, the '*AvdeccEntityListener*' object calls the *libavdecc*'s function (*avdecc_acmp_controller_doCommand()*) with a *params* argument. The *params* argument includes the:

  - message type (for *controller connect* this value is '1'),
  - talker's entity GUID,
  - index of the source stream on the talker,
  - the listener's entity GUID, and
  - the index of the sink stream on the listener.

Using the *params* argument, the *libavdecc's controller state machine* associated with the '*AvdeccEntity'* object sends an ACMP *CONNECT_RX_COMMAND* message to the physical AVDECC end station that it abstracts. This then starts the connection management sequence that was described in chapter 7 section 7.4.2 on page 194.

In order to destroy an audio stream connection:

- *'UNOS Vision'* sends an AES-64 message that calls the *setInputAvdeccMulticoreControllerConnect()* method of the '*AES-64DeviceListener'* object with a '*disable'* argument. If the argument value is '1', it implies an '*enable'*, while a value of '0' implies a *'disable*'.

- This results in the '*AES-64DeviceListener'* object calling the *connectSinkStream()* method of the '*AvdeccEntityListener'* object with a *controllerDisconnect* argument.

- The '*AvdeccEntityListener*' object then calls the *libavdecc*'s *controller state machine* function with a *controller disconnect* (of value'0') message type as part of the *params* argument.

The connection management procedure depicted in Figure 8.10 enables UNOS Vision to establish an audio stream connection between AVDECC listener and talker end stations (via the AVDECC proxy) according to the *controller connect mode* that has been described in section 7.4.2 on page 194. However there are instances where a sound engineer wishes to establish or destroy audio stream connections between an AVDECC end station and an AES-64 device on an Ethernet AVB network. Section 8.3.3.2 describes how an AES-64 network controller would establish (or destroy) audio stream connections between an AES-64/Ethernet AVB device (as AVB listener) and an AVDECC end station (as AVB talker) via the AVDECC proxy. Section 8.3.3.3 provides a description of how the AVDECC proxy is used by an AES-64 network controller to establish or destroy audio stream connections between an AVDECC end station (as AVB listener) and an AES-64/Ethernet AVB device (as AVB talker).

8.3.3.2 Connection management procedure between AES-64 and AVDECC end stations

The AVDECC proxy is capable of enabling connection management between an AES-64 device and an AVDECC end station. In this case, an AES-64 network controller (such as UNOS Vision) issues AES-64 commands to the proxy in order to configure the appropriate connection management parameters on the AVDECC end station.

The interaction that takes place when UNOS Vision is establishing an audio stream connection (via the AVDECC proxy) between an AES-64 device (as AVB listener) and an AVDECC end station (as AVB talker) is shown in the form of a sequence diagram in Figure 8.11.

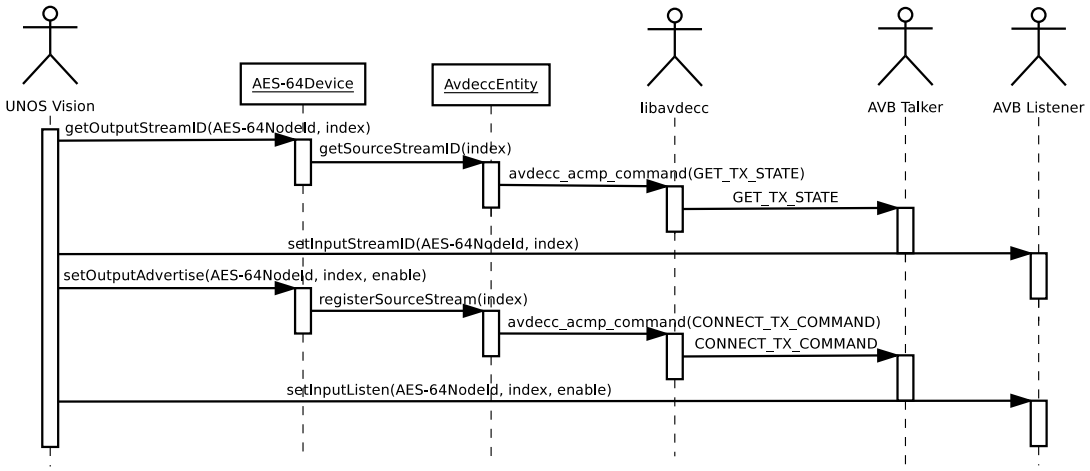


Figure 8.11: Sequence diagram of connection management procedure with AVDECC end station as AVB talker

The transactions in Figure 8.11 are:

- The '*AES-64Device*' object within the AVDECC proxy (which is uniquely identified by the *AES-64NodeId* argument) receives an AES-64 message that triggers its *getOutputStreamID()* callback. This method is passed an *index* argument, which specifies the particular output stream.
- The '*AES-64Device*' object calls the *getSourceStreamID()* method of its corresponding *'AvdeccEntity*' object, while specifying the source stream of interest with the *index* argument.
- The '*AvdeccEntity*' object calls the *avdecc_acmp_command()* function of *libavdecc* in order to multicast an ACMP message (*GET_TX_STATE*) to the *'AVB Talker'* actor.
- The *avdecc_acmp_command()* utilizes the *libavdecc*'s *controller state machine* to multicast the *GET_TX_STATE* command to the *'AVB Talker'* actor.
- Having obtained the stream ID of the AVDECC end station, *'UNOS Vision'* sends an AES-64 message which will trigger the *setInputStreamID()* callback within the (AES-64) *'AVB Listener'*. The particular input multicore is specified with the *index* argument.

When the AES-64 network controller ('*UNOS Vision*' actor in Figure 8.11) sends an AES-64 message to the AVDECC proxy in order to '*enable'* the *advertise* parameter of the '*AVB Talker*' actor, the following occurs:

- The *setOutputAdvertise()* method of the '*AES-64Device*' object, which is uniquely identified by the *AES-64NodeId* argument, is called with an *index* argument specifying the particular output stream. The '*enable'* argument indicates that the command is meant to establish a connection. If the network controller is attempting to destroy a stream connection, a '*disable'* argument value is transmitted. A '1' indicates '*enable'* and a '0' indicates '*disable'*.

- The '*AES-64Device*' object calls the *registerSourceStream()* method of its corresponding *'AvdeccEntity*' object, while specifying the source stream of interest with the *index* argument. The call from the '*AES-64Device'* to the '*AvdeccEntity'* is a *deregisterSourceStream()* when a connection is being destroyed.

- The '*AvdeccEntity*' object calls the *avdecc_acmp_command()* function of *libavdecc* in order to multicast an ACMP (*CONNECT_TX_COMMAND*) message, which is 'picked up' by the 'AVB Talker' actor. When destroying a stream connection, this message is replaced with an ACMP *DISCONNECT_TX_COMMAND* message.

- The *avdecc_acmp_command()* utilizes the *libavdecc*'s *listener state machine* to multicast the *CONNECT_TX_COMMAND* command to the *'AVB Talker'* actor. The *listener state machine* multicasts a *DISCONNECT_TX_COMMAND* message when destroying a stream connection. This will cause the *'AVB Talker'* to advertise its stream via MSRP.

- '*UNOS Vision'* sends an AES-64 message which will trigger the *setInputListen()* callback within the (AES-64) *'AVB Listener'*. The input multicore is specified with the *index* argument. This callback utilizes MSRP to indicate to the network that the *'AVB Listener'* is interested in the particular stream.

When an AVDECC end station receives an ACMP message with the *message type* field of *CONNECT_TX_COMMAND* or *DISCONNECT_TX_COMMAND*, it proceeds to either request resources for transmitting its stream from the Ethernet AVB network, or to release previously acquired resources from the Ethernet AVB network, respectively.

The AVDECC proxy can also be used for connection management with the AVDECC end station as AVB listener, and an AES-64 device as AVB talker. This scenario is described in the following section.

8.3.3.3 Connection management procedure with AVDECC end station as AVB listener

In order to set up an audio stream connection between an AVDECC end station (as AVB listener) and an AES-64 device (as AVB talker), a number of steps are followed. These steps are shown in the form of a sequence diagram in Figure 8.12.

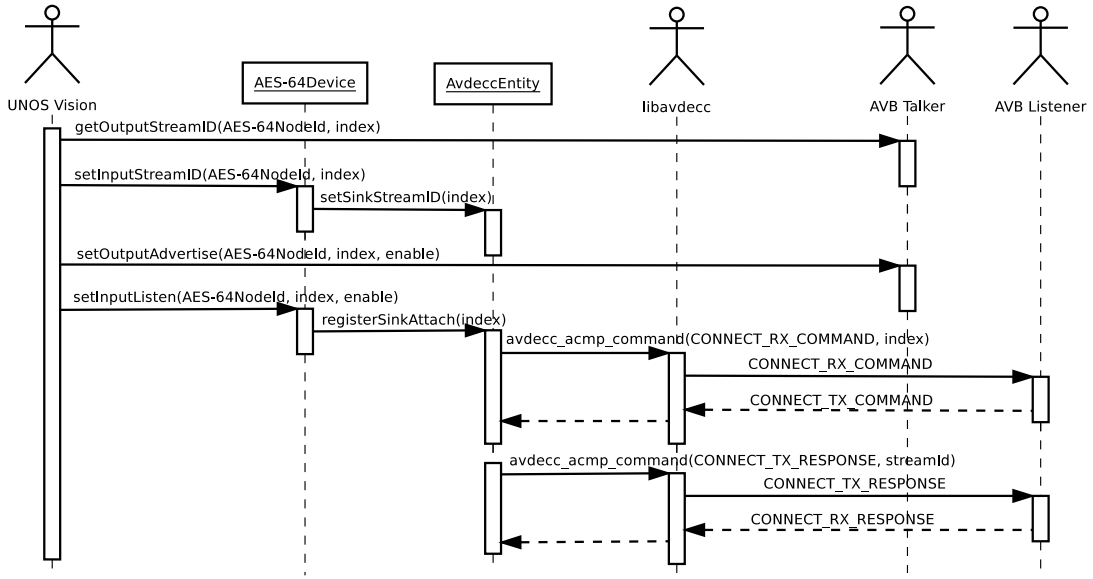


Figure 8.12: Sequence diagram of connection management procedure with AVDECC end station as AVB listener

In Figure 8.12, *'UNOS Vision'* sends an AES-64 message that is effectively a *getOutput-StreamID()* instruction to the (AES-64) '*AVB Talker'*. It indicates the particular *index* of Ethernet AVB stream source (output).

After obtaining the source stream ID, *'UNOS Vision'* sends a *setInputStreamID()* instruction to the '*AES-64Device*' object (within the AVDECC proxy), which causes the *setSinkStreamID()* function of the '*AvdeccEntity*' object to action.

When the '*AES-64Device*' object receives an AES-64 message to *'set'* its input stream ID, it stores the stream ID as an attribute of the '*AvdeccEntity*' object, and it indicates the particular *index* of the Ethernet AVB stream sink (input).

After obtaining and setting the stream ID on the *'AVB Talker'* and *'AVB Listener'* end stations (respectively), *'UNOS Vision'* sends an AES-64 message to the *'AVB Talker'*. This message causes the setOutputAdvertise() callback within the *'AVB Talker'* to be triggered, which causes the *'AVB Talker'* to advertise its stream by utilizing MSRP. This is followed by '*UNOS Vision*' issuing an AES-64 message to enable the *listen* parameter within the *'AES-64Device'* object.

When the AVDECC proxy receives an AES-64 message to '*enable'* the *listen* parameter of the AVDECC end station, the following occurs:

- The *setInputListen()* method of the '*AES-64Device*' object, which is uniquely identified by the *AES-64NodeId* argument, is called with an *index* argument specifying the input. The '*enable'* argument indicates that the command is meant to establish a connection. If the controller wishes to destroy a stream connection it sends a '*disable'* argument. A '1' indicates '*enable'* and a '0' indicates '*disable'*.

- The '*AES-64Device*' object calls the *registerSinkAttach()* method of its corresponding *'AvdeccEntity*' object. It specifies the sink stream of interest with the *index* argument. This call from the '*AES-64Device'* to the '*AvdeccEntity'* is a *deregisterSinkAttach()* when a connection is being destroyed.

- The '*AvdeccEntity*' object calls the *avdecc_acmp_command()* function that is implemented by the *'libavdecc'* actor in order to multicast an ACMP (*CONNECT_RX_COMMAND*) message to the '*AVB Listener*' actor. It utilizes its *controller state machine* of the *libavdecc* library to do this. The message type of the ACMP message is *DISCONNECT_RX_COMMAND,* when destroying a stream connection. The '*AvdeccEntity*' object specifies its *entity GUID* as the *talker GUID* in the ACMP message that is sent to the end station.

- In response to a received *CONNECT_RX_COMMAND* (when establishing a stream connection) ACMP message, the AVDECC end station (which is represented by the *'AVB Listener*' actor) multicasts a *CONNECT_TX_COMMAND* message. This response (*CONNECT_TX_COMMAND* message) is received by the *'AvdeccEntity'* object's *talker state machine*.

- On receiving a *CONNECT_TX_COMMAND* ACMP message from the '*AVB Listener*' actor, the *'AvdeccEntity'* object utilizes its *talker state machine* to multicast an ACMP *CONNECT_TX_RESPONSE* message to the '*AVB Listener'* actor. It does this by utilizing the *avdecc_acmp_command()* function of the *libavdecc* library. The multicast message specifies the stream ID of the source stream that was previously '*set*' for the sink.

- The '*AVB Listener'* actor multicasts an ACMP *CONNECT_RX_RESPONSE* message, which is received by the '*AvdeccEntity*' object's *controller state machine*.

When the AVDECC end station ('*AVB Listener'* actor in Figure 8.12) receives an ACMP message of type *CONNECT_TX_RESPONSE* it proceeds with requesting attachment to

a stream on the Ethernet AVB network by utilizing MSRP. If the received message is an ACMP message of type *DISCONNECT_TX_RESPONSE*, the '*AVB Listener*' detaches itself from the network stream via MSRP.

Following the implementation of the AVDECC proxy, a number of tests were conducted. These are described in section 8.4.

8.4 Testing and Results

From the use case diagram of Figure 8.4, the proxy is required to:

- enable the discovering of AVDECC end stations, and
- enable connection management of AVDECC end stations.

To determine whether the proxy meets these requirements, two scenarios were tested. These scenarios are:

1. Integrating layer 2 (AVDECC) devices into a layer 3 (AES-64) network. This will demonstrate layer 3 control of layer 2 devices by utilizing the AVDECC proxy. This is described in section 8.4.1.
2. Common control of layer 2 (AVDECC) and layer 3 (AES-64) devices. This will demonstrate connection management of layer 2 and layer 3 devices by a layer 3 network controller with the aid of the AVDECC proxy. This is described in section 8.4.2 on page 225.

These test were conducted to determine the effectiveness of the AVDECC proxy. In the following subsections, the proxy is located on a separate PC workstation from the control application. However, it is possible to have the proxy implemented within the same PC as the control application. This approach of having a common host PC for an AES-64 controller application and a command translator (proxy) has been described in subsection 6.7 on page 172.

8.4.1 Integrating layer 2 devices into a layer 3 network

A test bed was set up to investigate whether the AVDECC proxy is capable of enabling connection management between (layer 2) AVDECC end stations, when instructed by a (layer 3) AES-64 network controller. Figure 8.13 depicts the test bed topology.

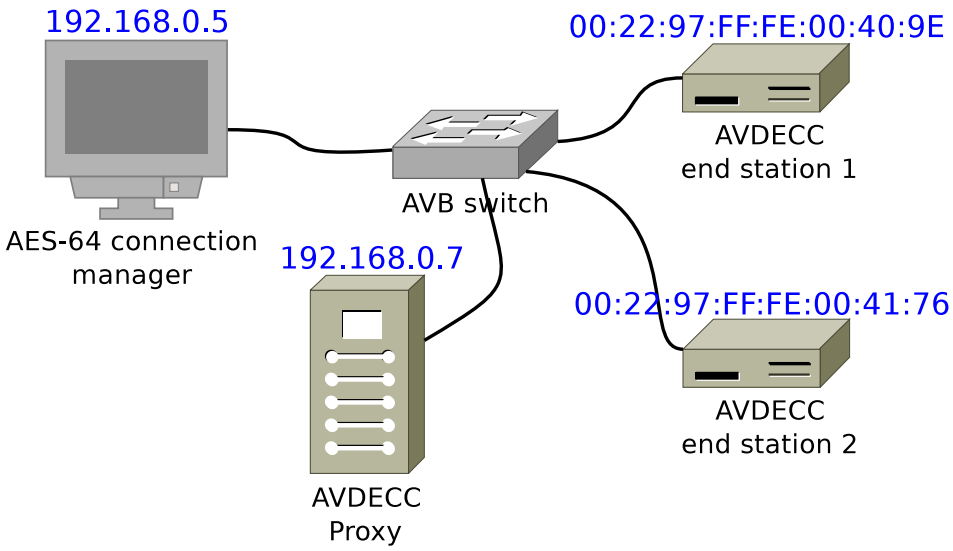


Figure 8.13: Test bed topology for integrating layer 2 devices

The test bed consists of:

- an '*AES-64 connection manager*',
- two AVDECC end stations ('*AVDECC end station 1*' and '*AVDECC end station 2*'),
- an '*AVDECC proxy'* , and
- an *'AVB switch*'.

The '*AES-64 connection manager*' is an AES-64 network configuration, monitoring and device control Windows PC application called UNOS Vision. The AVDECC end stations are *XMOS/Attero Tech Low-cost AVB Audio Endpoint* modules, which are running the XMOS AVDECC end station software [157]. The '*AVDECC proxy'* is a *Linux (kernel version 3.0.0-22-generic)* PC application running on an *Ubuntu 12.04* operating system. The '*AVB switch*' is a *LabX Titanium 411 AVB Ethernet Bridge* which implements the Ethernet AVB protocol [158].

Section 8.4.1.1 describes how the proxy was used to enable device discovery of the XMOS/Attero Tech boards (layer 2 devices). Section 8.4.1.2 describes how the proxy was used to enable connection management between the networked (XMOS/Attero Tech) devices.

8.4.1.1 Discovering layer 2 devices

With the AVDECC proxy on the network, UNOS Vision is able to discover the XMOS/Attero Tech devices. Figure 8.14 shows a screenshot of UNOS' device discovery view.

UNOS Vision discovers the AVDECC end stations on the network in the same manner it would discover any AES-64 device on the network. It does so by broadcasting an AES-64 device discovery message. The proxy responds to this message by sending its (the proxy's) IP address and a different *AES-64 node ID* for each device. Hence in Figure 8.14 both AVDECC end stations discovered by UNOS Vision have the same IP address '*192.168.0.7*'.

In the current implementation, UNOS Vision displays an end station's *entity GUID* as its display name.

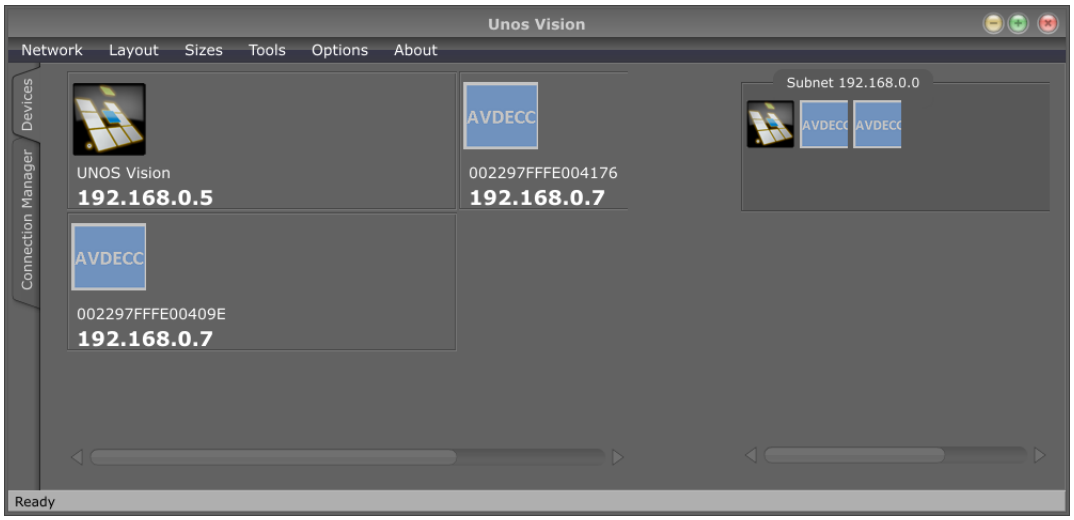


Figure 8.14: UNOS Vision discovers AVDECC end stations

8.4.1.2 Connection management between layer 2 devices

The screen shot in Figure 8.15 shows UNOS Vision when a user establishes a connection between the two AVDECC end stations (in Figure 8.13). This causes the source AVDECC end station to assume the role of AVB talker and the destination AVDECC end station to assume the role of AVB listener.

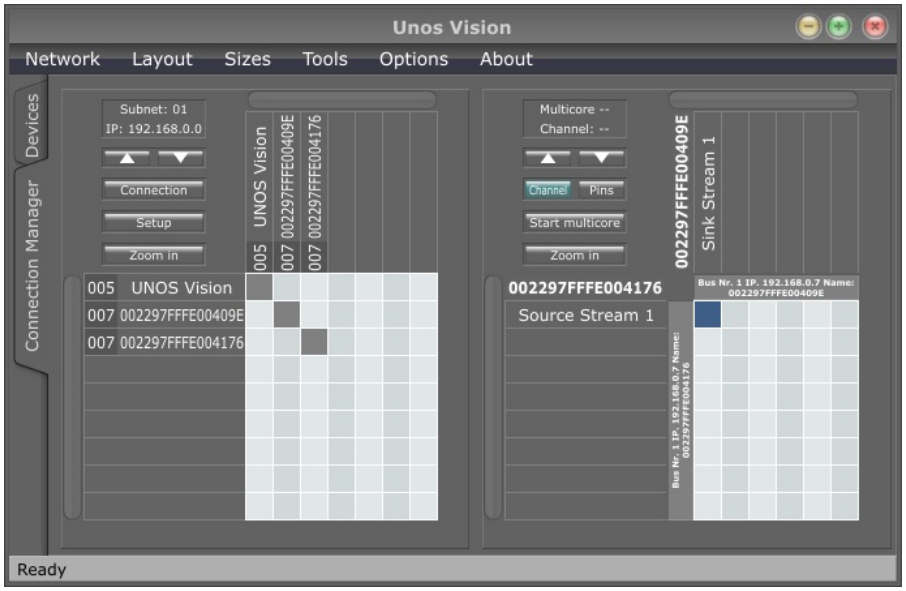


Figure 8.15: UNOS establishes a stream connection between AVDECC end stations

When the cross point in the (*multicores*) matrix on the right of the screenshot (shown in Figure 8.15) is clicked, UNOS Vision proceeds with its connection management procedure which has been described in section 8.3.3.1 on page 214.

8.4.2 Common control of layer 2 and layer 3 devices

To investigate the control of layer 2 and layer 3 devices an Ethernet AVB network was setup to include AES-64 and AVDECC devices. The test bed topology is shown in Figure 8.16.

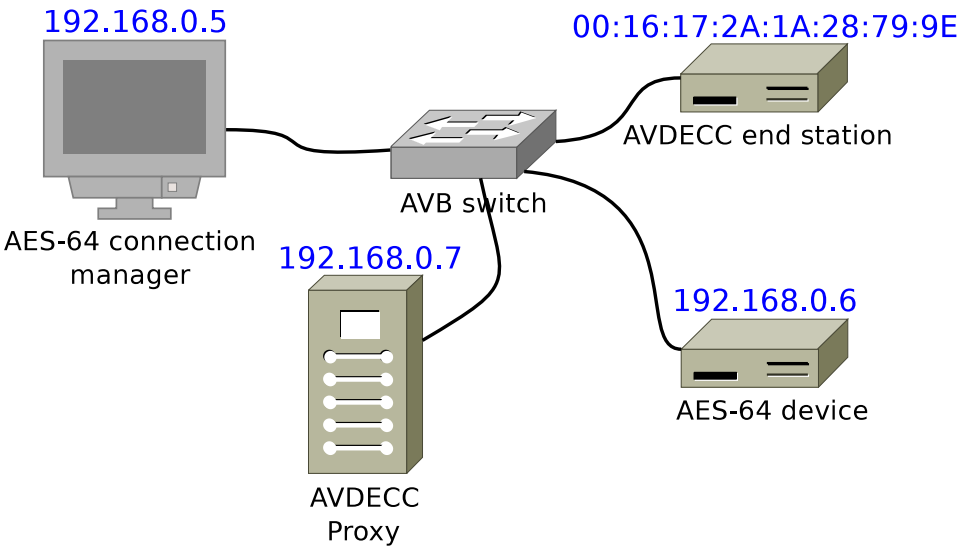


Figure 8.16: Test bed topology for common control of networked layer 2 and layer 3 devices

The test bed consists of an:

- '*AES-64 connection manager*' - an AES-64 network configuration, monitoring and device control software (*UNOS Vision*) running on a Windows PC,
- '*AVB switch*' - a LabX Titanium 411 Ethernet AVB switch which implements the Ethernet AVB protocol [**?**],
- '*AVDECC proxy'* - the AVDECC proxy that has been described in this chapter, running on a PC,
- '*AVDECC end station*' - the AVDECC end station that was described in chapter 7 on page 175, and
- '*AES-64 device*' - an AES-64 Ethernet AVB endpoint running on a PC [1].

The test was conducted to determine whether the AVDECC proxy enables:

- *UNOS Vision* to discovery the AVDECC end station on the network
- *UNOS Vision* to establish and destroy stream connections between the networked devices

Section 8.4.2.1 describes how the AVDECC proxy is used to enable device discovery, and section 8.4.2.2 describes how it is used to enable connection management between AES-64 and AVDECC devices.

8.4.2.1 Discovering layer 2 and layer 3 devices

UNOS Vision is able to discover the networked (AES-64 and AVDECC) devices. Figure 8.17 is a screenshot of the UNOS Vision's device view.

Figure 8.17: Layer 2 and layer 3 device discovery

UNOS Vision discovers networked AES-64 devices by broadcasting an AES-64 discovery message. Each AES-64 device on the network responds to the AES-64 discovery broadcast message. The AVDECC proxy responds by returning its (the proxy's) IP address as the destination address for the AVDECC end station on the network of Figure 8.16. Thus any communication with the AVDECC end station (from UNOS Vision) is addressed to the AVDECC proxy. For multiple AVDECC end stations on the network, the proxy responds to UNOS by returning the same IP address, but different AES-64 node IDs for each of the end stations. The proxy returns the *entity GUID* of the AVDECC end station as the end station's name.

8.4.2.2 Connection management between layer 2 and layer 3 devices

In order to establish a stream connection between AVB devices, UNOS Vision adheres to the following steps:

- *UNOS Vision* sends an AES-64 command to '*get*' the stream ID of a particular source stream on the AVB talker
- *UNOS Vision* sends an AES-64 command to '*set*' the stream ID of a particular sink stream on the AVB listener

- *UNOS Vision* sends an AES-64 command to '*enable'* the *advertise* parameter on a particular source stream on the AVB talker
- *UNOS Vision* sends an AES-64 command to '*enable'* the *listen* parameter on a particular sink stream on the AVB listener

The above steps are depicted in the form of a sequence diagram in Figure 8.18.

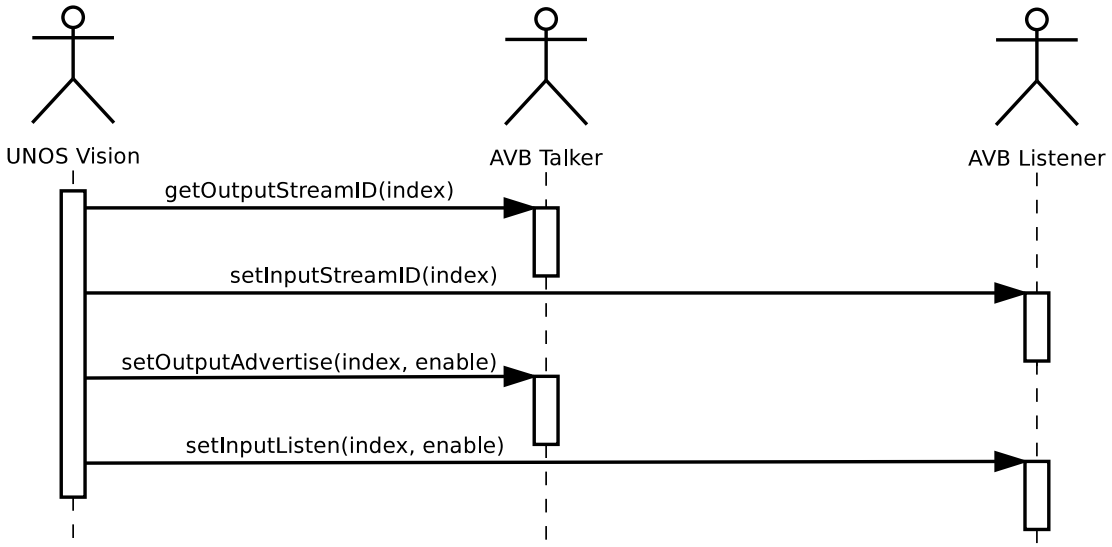


Figure 8.18: Sequence diagram of UNOS Vision's connection management procedure for AVB devices

In order to destroy a stream connection, UNOS Vision sends an AES-64 command to:

- '*disable'* the *advertise* parameter on a particular source stream on the AVB talker
- '*disable'* the *listen* parameter on a particular sink stream on the AVB listener

To test connection management on the testbed (network) shown in Figure 8.16, there are two possible scenarios. The first scenario is when the AVDECC end station assumes the role of AVB talker, that is the source of the Ethernet AVB audio stream. The second scenario is when the AVDECC end station assumes the role of AVB listener. These two scenarios are described below.

AVDECC end station as AVB talker

Connection management was investigated with the AVDECC end station as the source of the Ethernet AVB audio stream. Figure 8.19 shows UNOS Vision's connection management view, when an audio stream connection is established between the *'AVDECC end station'* and the *'AES-64 device*' depicted in Figure 8.16.

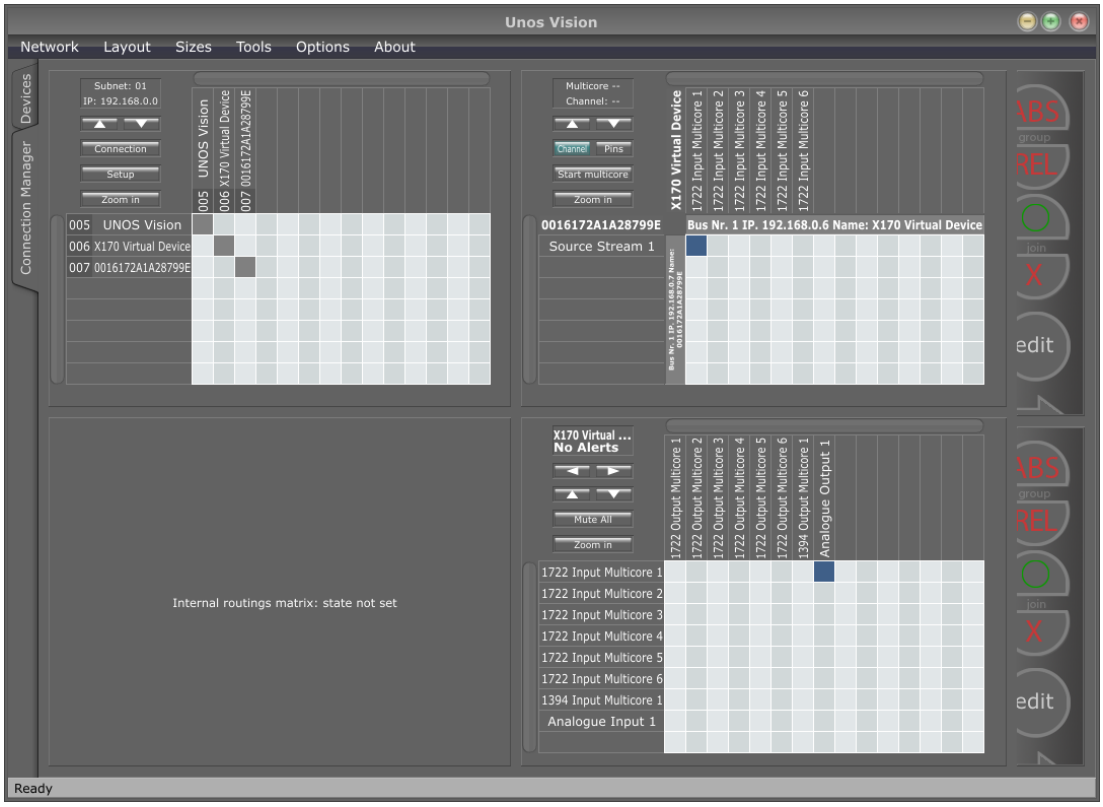


Figure 8.19: Connection management with AVDECC end station as AVB talker

In UNOS Vision's connection management view, the matrix at the top left is the *devices matrix*, and it is used to select the source and destination devices between which to make a connection. The bottom left matrix is the *source internal routing matrix*, and it displays the inputs and outputs on the source device. The bottom right matrix is the *destination internal routing matrix*, and it displays inputs and outputs on the destination device. The top right matrix is called the *multicore matrix*, and it displays the input and output Ethernet AVB streams. In the *multicores matrix*, the source device's outputs are listed on the left and the destination device's inputs are listed at the top.

When the AVDECC end station is selected as the source and the AES-64 device as the destination of a stream connection, the source and destination internal routing matrices are updated as shown in Figure 8.19. The AVDECC proxy does not enable control of internal signal routing within the AVDECC end station, hence there are no inputs and outputs in the *source internal routing matrix* of Figure 8.19. This is because in this investigation connection management between devices that implement different control protocols is considered as the criteria for interoperability. This means that if it is possible to set up audio stream connections between devices that implement different audio control protocols, then the interoperability challenge (described in chapter 4) does not exist between the devices. Also the AVDECC end stations are capable of transmitting

and receiving audio steams without necessarily requiring to perform any internal routings. Figure 7.5 on page 193 shows the layout of the AVDECC end station.

The source Ethernet AVB multicore of the AVDECC end station is shown in the multicores matrix as '*Source Stream 1*', and the sink multicores of the AES-64 device are shown as '*1722 Input Multicore 1'* to '*1722 Input Multicore 6'*. In Figure 8.19 the cross point between the AVDECC end station's '*Source Stream 1*' and the AES-64 device's '*1722 Input Multicore 1'* has been clicked ('enabled') in order to establish an audio stream connection. Both '*Source Stream 1*' and '*1722 Input Multicore 1'* are 2-channel audio streams. The connection management procedure followed when the cross point is clicked has been described in section 8.18 on page 228.

In Figure 8.19, the *destination internal routing matrix* shows that the stereo '*1722 Input Multicore 1'* audio stream has been routed to the '*Analogue Output 1*' stereo output on the AES-64 device. Thus the sound can be heard on a stereo speaker connected to the analogue output of the AES-64 device.

AVDECC end station as AVB listener

The AVDECC proxy was tested to investigate whether it is capable of enabling connection management between an AES-64 device (as AVB talker) and an AVDECC end station (as listener). Figure 8.20 is a screenshot of UNOS Vision's connection management view when a stream connection is established between an AES-64 device and an AVDECC end station. The network topology used in this test is shown in Figure 8.16.

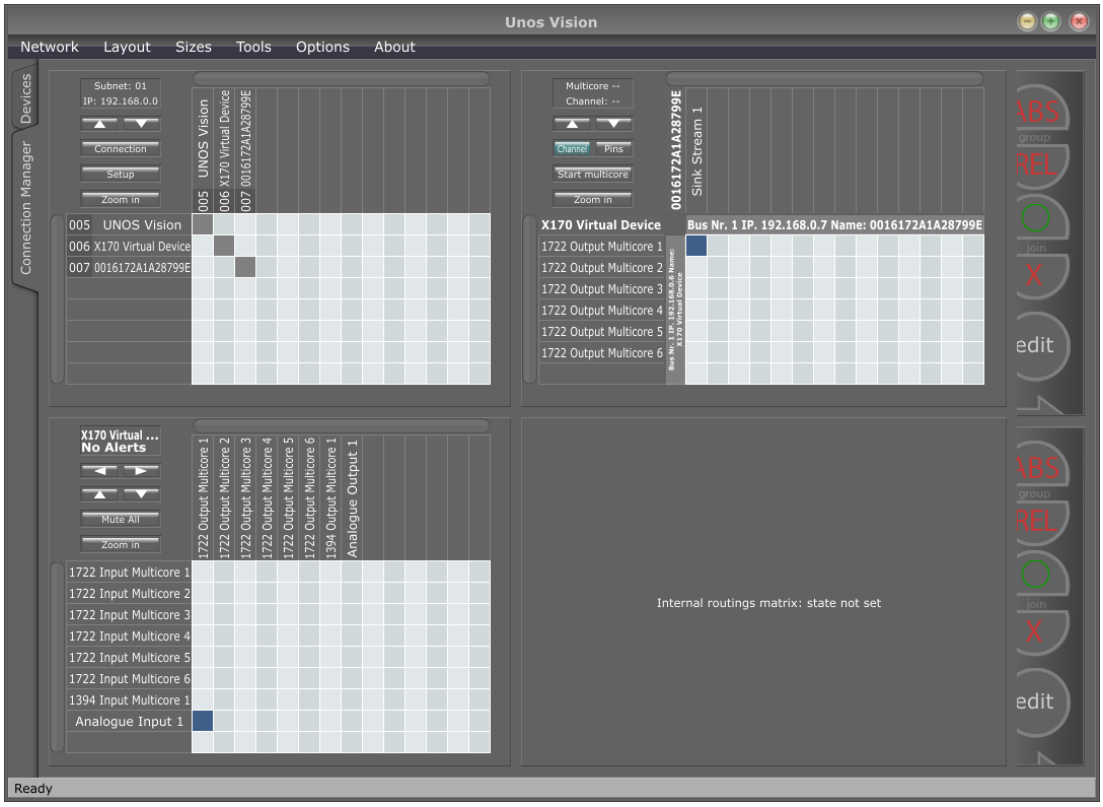


Figure 8.20: Connection management with AVDECC end station as AVB listener

In Figure 8.20, the AES-64 device has been selected as the source device, and the AVDECC end station has been selected as the destination of a stream connection. The *source internal routing matrix* shows the AES-64 device's inputs and outputs. As shown in the figure, the '*Analogue Input 1*' has been routed to the '*1722 Output Multicore 1*' of the AES-64 device. Both '*Analogue Input 1*' and '*1722 Output Multicore1*' are stereo (2-channel) audio streams. The *destination internal routing matrix* for the AVDECC end station is empty since this investigation considers connection management between networked audio devices as the criteria for interoperability. Also the AVDECC end stations are capable of transmitting and receiving audio steams without necessarily requiring to perform any internal routings. Figure 7.5 on page 193 shows the layout of the AVDECC end station.

In Figure 8.20, the source Ethernet AVB multicores of the AES-64 device are shown as '*1722 Output Multicore 1'* to '*1722 Output Multicore 6'* and the sink Ethernet AVB multicore of the AVDECC end station is shown in the multicores matrix as '*Sink Stream 1*'. The cross point between the AES-64 device's '*1722 Output Multicore 1'* and the AVDECC end station's '*Sink Stream 1*' has been clicked ('enabled') in order to establish an audio stream connection. The connection management procedure followed when the cross point is clicked has been described in section 8.18 on page 228.

8.5 Summary

The command translation approach entails that a control message is translated from one audio control protocol to another. In this chapter the approach was investigated between two audio control protocols on an Ethernet AVB network.

The investigation involved the design and implementation of a proxy that is capable of receiving a layer 3 (AES-64) message and translating it to the corresponding layer 2 (AVDECC) message, and vice versa. In particular the messages were connection management instructions, such that an AES-64 connection manager was able to establish and destroy audio stream connections between the networked Ethernet AVB devices.

A number of tests were conducted to determine the effectiveness of the proxy. The results obtained revealed that by utilizing the layer 2/layer 3 proxy, an AES-64 connection manager is able to discover AVDECC end stations, as well as establish and destroy audio stream connections between:

- AVDECC end stations on an Ethernet AVB network
- AVDECC and AES-64 end stations on an Ethernet AVB network

The command translation functionality that is implemented by this proxy can be incorporated into the same host PC as the network control application. This has been described in section 6.7 on page 172, and it will enable the command translator utilize the same processing power of the PC that runs the control application.

Chapter 9

Quantitative Analysis

Chapter 6 investigated how a proxy that is capable of protocol command translation, can be used to effectively enable interoperability between AES-64 and OSC devices on an Ethernet AVB network. The networked devices utilized the same layer 3 (UDP/IP) transport protocol for messaging. Ethernet AVB enabled the transmission of audio streams between the devices.

In chapter 8 the command translation solution to interoperability of networked devices was investigated for devices that implement different transport protocols. A proxy was able to perform command translation between AES-64 and IEEE 1722.1 (AVDECC) messages. IEEE 1722.1 utilizes layer 2 transport for messaging, which is different from the layer 3 transport that is used by AES-64.

In chapter 6 and chapter 8, the effectiveness of the proxy was demonstrated by a network controller that was used to establish and destroy audio stream connections between the networked end stations.

This chapter investigates the efficiency of the command translators (proxies) that were developed as part of this research project, in order to determine whether the command translation approach is a viable solution for commercial installation. The connection management procedures for establishing and destroying audio stream connections with the aid of a proxy, were timed in order to determine the overhead added by the proxy.

9.1 Introduction

When an audio stream is transmitted by a source device (on a network) and received by a destination device, interoperability is said to exist between both devices. Establishing a stream connection between the source and destination devices requires that:

- the source device is configured to transmit a particular stream
- the destination device is configured to receive the specified stream.

The procedure for configuring the source and destination devices, is known as connection management. Connection management procedures differ from one audio control protocol to another. It has been demonstrated that interoperability between devices that implement different audio control protocols can be achieved with the aid of a proxy. The proxy translates protocol commands from one audio control protocol to another. Refer to chapter 6 and chapter 8 for implementation details of the proxies that were created in this research project.

In order to investigate the efficiency of the proxy, it is necessary to determine the amount of time overhead that is added by the proxy when it is used to enable connection management. In order to consider the proxy as a viable solution for commercial deployment, it is required that the proxy does not add noticeable delays due to its command translation operations.

When a human observer utilizes a graphical user interface (GUI) audio network control application for connection management, two types of stimuli can be perceived. These are:

- visual stimuli - refers to external changes that are perceived with human eye.
- auditory stimuli - refers to external changes that are perceived by the ears.

Typically the user of the connection management control application expects feedback as quickly as possible. The feedback could be a visual feedback from the GUI screen, or an auditory feedback from a speaker that is attached to a receiving device.

The above scenario is shown in Figure 9.1.

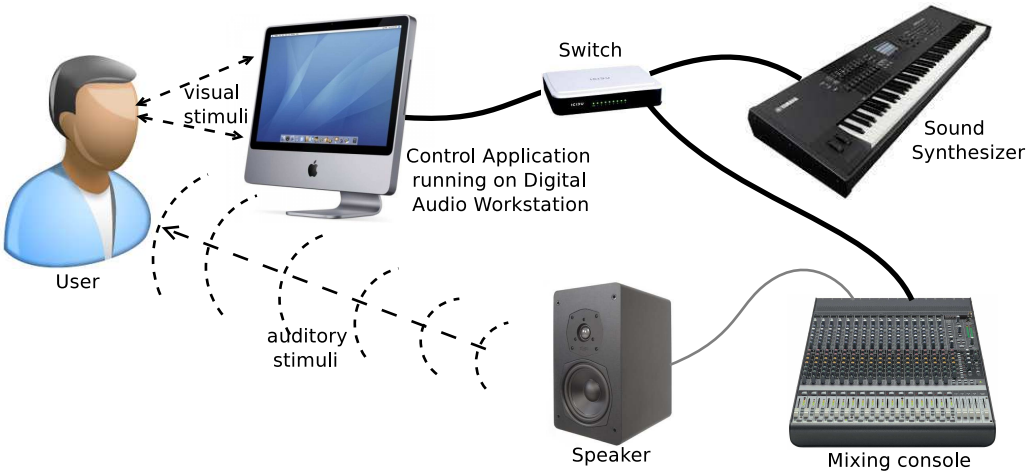


Figure 9.1: Sound engineer's visual and auditory perception

In Figure 9.1, the sound engineer is depicted as a '*User*' that utilizes the '*Control Application running on a Digital Audio Workstation*' to establish an audio stream connection between a '*Sound Synthesizer*' and a '*Mixing console*'. The control application, mixing console and sound synthesizer are networked. When an audio stream connection is established between the synthesizer (as source device) and the mixing console as destination device, the sound engineer is able to observe the visual stimuli from the screen of the digital audio workstation and the auditory stimuli from the speaker.

Usually software developers endeavor to keep the interval between the 'button click event' (by 'User' in Figure 9.1) and the perceived response stimuli as minimal as possible. The following subsections provide some information about human perception times for visual and audio stimuli.

9.1.1 Visual stimuli perception time

The Modeling Human Processor (MHP) provides a model for analyzing how humans respond to external visual stimuli [159]. MHP categorizes human cognitive actions into three processors. The MHP processors are:

- *perceptual processor* - relates to how stimuli are perceived
- *cognitive processor* - relates to how the perceived stimuli are interpreted based on long-term information in memory
- *motor processor* - relates to how humans respond to stimuli

When designing a graphical audio network controller, it is important to consider the *perceptual processor cycle time*. Perceptual processor cycle time refers to the time within which multiple events can be distinguished by an observer. This is illustrated in Figure 9.2.

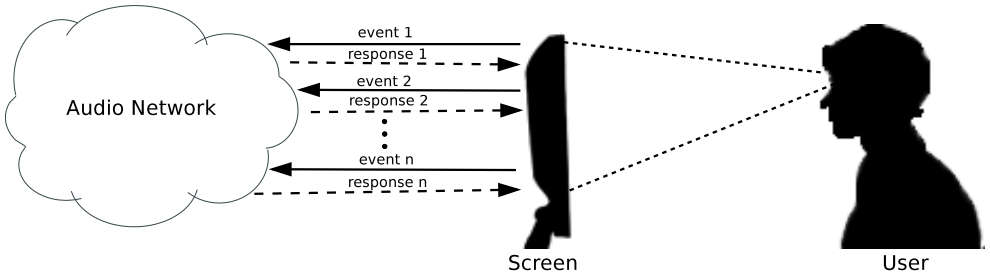


Figure 9.2: User managing network

Figure 9.2 shows a '*User'* making an audio stream connection between devices within an '*Audio Network*'. A workstation (PC) which runs the network connection manager software is used by the '*User'* to discover the devices, as well as to establish stream connections between the devices. The user observes the effects of connection patches made via the PC '*Screen*' shown in the figure.

In Figure 9.2, an action by the '*User'* causes *'n'* events ('*event 1*', '*event* 2' to '*event n*') to be sent from a PC (which is represented by the '*Screen*'). These events are shown to have corresponding responses ('*response 1*', '*response 2*', '*response n*'), which are observed by the '*User'*. The '*Screen'* displays the effect of an action when a response has been received.

Assuming that the time at which '*response 1*' is observed on the screen is noted as $\boldsymbol{T_1}$, and the time at which '*response n*' is observed is noted as $\boldsymbol{T_n}$, the time difference between the two actions is $\boldsymbol{T_n - T_1}$.

The graphical display of network control software will typically show the connection management procedure (for establishing or destroying a stream connection) as a single action. For instance, although the procedure for establishing an audio stream connection (such as those described in section 8.3.3) involves a number of transaction steps, the graphical display of a network controller and monitoring software might present a single button to initiate the transaction steps. An application developer will typically endeavor to keep the time taken between the transaction steps to a minimum so that they appear as a single action to an observer.

The MHP model indicates that the perceptual processor cycle time is 100 milliseconds. This means that when a series of visual events occur within a 100 millisecond interval,

they appear as a single visual event to an observer [159]. With regard to the graphical display example, the transaction steps that are involved in establishing an audio stream connection should take place within a 100 millisecond interval if there are visual responses and if these responses are to be observed as a single event. This means that the '*time taken*' ($\boldsymbol{T_n}$ **-** $\boldsymbol{T_1}$) should not exceed 100 milliseconds.

9.1.2 Auditory stimuli perception time

In a paper titled *"Problems and Prospects for Intimate Musical Control of Computers",* Wessel and Wright suggest that the allowed time for auditory feedback should not exceed 10 milliseconds [160]. The paper describes the use of computers as musical instruments. The time limit for auditory feedback from a computer that is referred to by Wessel and Wright is illustrated in Figure 9.3.

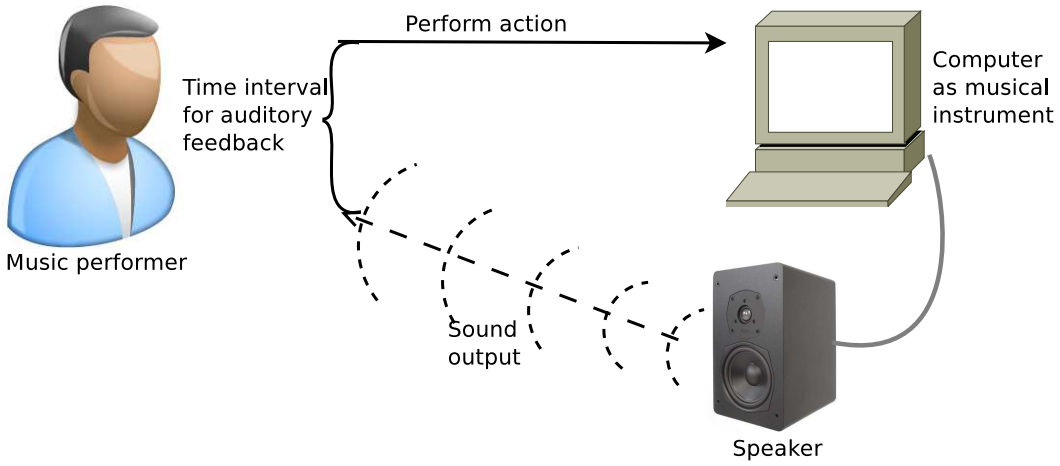


Figure 9.3: Auditory feedback from computer as a musical instrument

In Figure 9.3, the '*Music performer*' gestures the computer in order to output a sound through the '*Speaker'*, which is connected to the computer. The computer processes the gesture it receives via a gestural interface, which could be a digitizing tablet, then it performs a 'generative algorithm' that results in the intended sound(s) being produced through the speaker. Wessel and Wright indicates that the (time) interval between the time when the performer's gesture is captured by the computer and the time when sound is heard by the performer, should not exceed 10 milliseconds.

In addition to this, Moore states that when two auditory stimuli are perceived within 30 milliseconds, they are perceived by humans to occur at the same time [161]. Thus humans detect multiple auditory stimuli that occur within 30 milliseconds as occurring at the same time.

9.1.3 Perception time criterion for quantitative analysis

From the discussions on visual and auditory stimuli perception by humans, which are described in section 9.1.1 and section 9.1.2 (respectively), it can be deduced that the minimum time within which multiple auditory stimuli are distinguishable (30 milliseconds) is much less than for visual stimuli (100 milliseconds). In the case of auditory stimuli, the time for a system to respond to a controlling action should not exceed 10 milliseconds.

In the following investigations, 10 milliseconds will be used as the criterion for acceptable network latency for the completion of a connection management request. This value is the recommended response time for auditory response as prescribed by Wessel and Wright. Since humans are more sensitive to the timing of auditory stimuli than visual stimuli, this value (10 milliseconds) falls within an acceptable response time for visual stimuli.

9.2 Quantitative analysis of Layer 3 Proxy

This section describes the investigation that was conducted to determine the performance of the proxy that was described in chapter 6. The two scenarios used in this analysis were:

- quantitative analysis of Ethernet AVB network containing OSC end stations
- quantitative analysis of Ethernet AVB network containing OSC and AES-64 end stations.

In both scenarios:

- UNOS Vision was used for device configuration and control. UNOS Vision ran on a PC (workstation) for which the specification was:

Processor	Intel Core (i7) @ 2.93GHz
Operating System	Windows 7 Professional (64-bits)
Ethernet Speed	1000Mbps

- The OSC and AES-64 end stations were implemented as PC audio streaming devices. An identical PC specification was used for both end stations. The specification was:

Processor	Intel Core Quad (Q9400) @ 2.66GHz
Operating System	Linux Ubuntu 10.10 (32-bits)
Ethernet Speed	1000Mbps

- The proxy was deployed on the same PC for each of the investigations. The specification of the PC used was:

Processor	Intel Quad-Core (i7) @ 2.70GHz
Operating System	Linux Ubuntu 12.04 (32-bits)
Ethernet Speed	1000Mbps

Figure 9.4 shows the test bed network topology used.

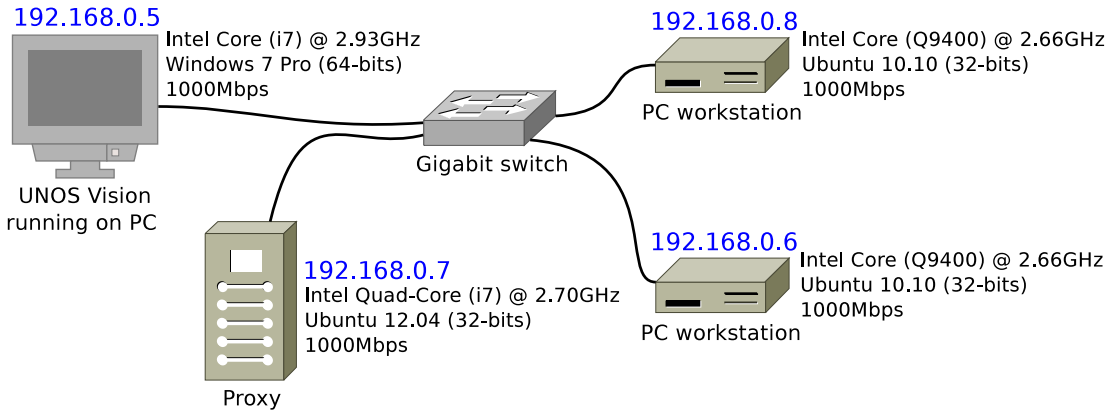
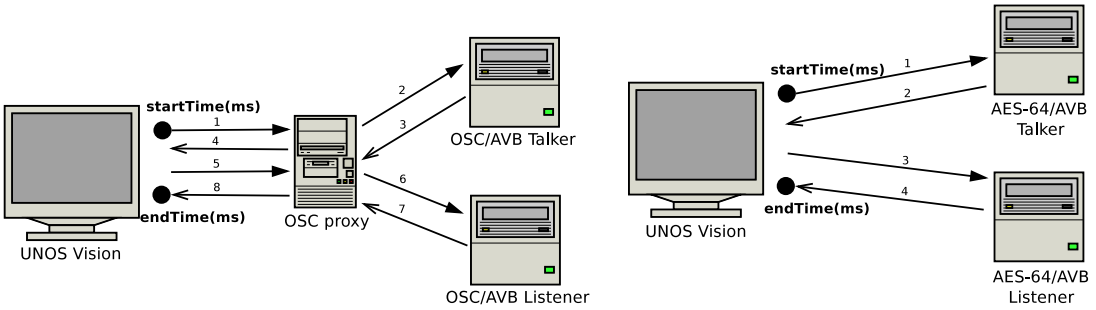


Figure 9.4: Test network topology

The OSC and AES-64 end stations used in this investigation were deployed on the PC workstations shown in Figure 9.4. The test scenarios are described in the following sections.

9.2.1 Scenario One: Connection between OSC end stations

In this scenario the latency added by an OSC proxy (described in chapter 6) was measured for thirty-five iterations of establishing and destroying audio stream connections between two OSC servers. Both OSC servers were Ethernet AVB end stations, and UNOS Vision was used for connection management. The network layout is shown in Figure 9.5a.



(a) Ethernet AVB network of OSC end stations

(b) Ethernet AVB network of AES-64 end stations

Figure 9.5: Timing connection management of layer 3 audio control protocols

Figure 9.5a shows the following steps for establishing a stream connection:

1. UNOS Vision sends an AES-64 command to obtain the stream ID of the OSC/AVB talker.
2. The OSC proxy receives the AES-64 command on behalf of the OSC talker, then it translates the received AES-64 command to a corresponding OSC command and it sends the OSC command to the OSC/AVB talker.
3. The OSC/AVB talker sends an OSC response to the proxy.
4. The proxy translates the OSC response to the appropriate AES-64 response, then it sends the AES-64 response to UNOS Vision.
5. UNOS Vision sends an AES-64 command to the OSC proxy, to set the stream ID of the OSC/AVB listener input.
6. The OSC proxy receives the AES-64 command on behalf of the OSC listener, then it translates the received AES-64 command to a corresponding OSC command and it sends the OSC command to the OSC/AVB listener.
7. The OSC/AVB listener sends an OSC response to the proxy.
8. The proxy translates the OSC response to the appropriate AES-64 response, then it sends the AES-64 response to UNOS Vision.

Wireshark, a network protocol analyzer, was used to note the *startTime* as the time when the first connection management command was transmitted by UNOS Vision, and the *endTime* as the time when the last connection management response was received by

UNOS Vision [162]. The latency is calculated as the difference between the *startTime* and *endTime*.

Wireshark utilizes *libpcap* software library (or *WinPcap* on Windows platform*)* to capture packets [163]. *libpcap* utilizes the operating system's time to time stamp each packet that it transfers to the wireshark application [164]. Thus, both operating system and *libpcap* library contribute to the time measurements. *libpcap* is able to ensure microseconds time resolution, which is sufficient for this investigation [164].

The time stamps associated with the wireshark packet analyzer are obtained from the operating system via *lipcap*

The results obtained from the setup shown in Figure 9.5a are compared with the results obtained from the network setup of Figure 9.5b. In Figure 9.5b, UNOS Vision is used to configure two AES-64 end stations so that an audio stream transmitted by the talker is received by the listener.

The results of the above timings for establishing (*CONNECT*) and destroying (*DISCONNECT*) audio stream connections are shown in Table 9.1. The values in the table are in milliseconds, and are the average values of thirty-five iterations.

	CONNECT (ms)	DISCONNECT (ms)
Network of OSC end stations	3.579	1.468
Network of AES-64 end stations	1.385	0.752
Difference	**2.194**	**0.716**

Table 9.1: Timing results of connection management between OSC end stations

The results shown in Table 9.1 reveal that:

- it takes an average of 3.579 milliseconds to establish an audio stream connection between the two OSC servers (Ethernet AVB end stations) when the proxy is used
- it takes an average of 1.468 milliseconds to destroy an audio stream connection between the two OSC servers when the proxy is used
- it takes an average of 1.385 milliseconds to establish an audio stream connection between two AES-64 end stations
- it takes an average of 0.752 milliseconds to destroy an audio stream connection between two AES-64 end stations
- 2.194 milliseconds is the time difference between establishing an audio stream on a network of OSC end stations, and a network of AES-64 end stations

- the time difference between destroying a stream connection on a network of OSC end stations and a network of AES-64 end stations is 0.716 millisecond.

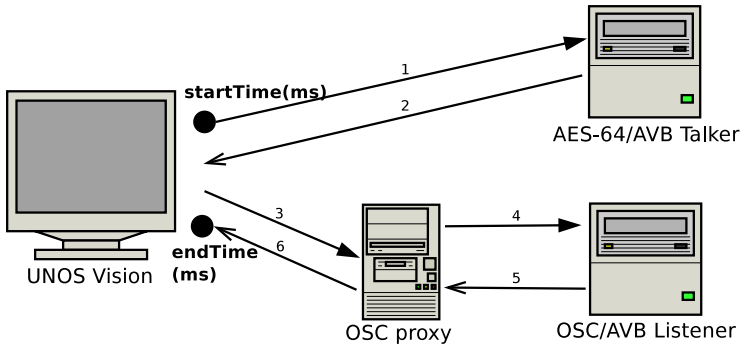
As Table 9.1 reveals, the time taken for a connection to be established when the proxy is used for the connection management procedure between two OSC servers is below the 10 milliseconds perception criterion described in section 9.1.3. Hence the proxy provides a time efficient solution for the connection management of OSC end stations by an AES-64 controller.

9.2.2 Scenario Two: Connection between OSC and AES-64 end stations

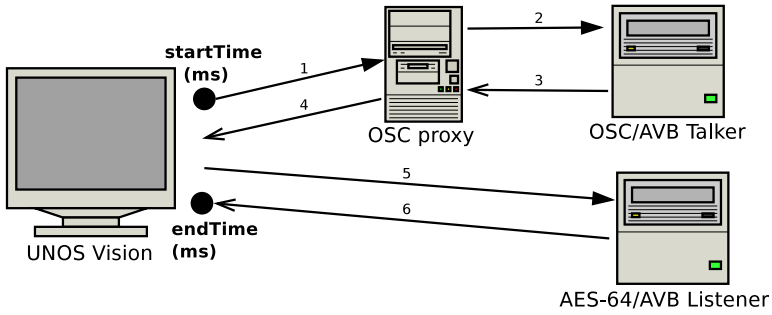
In this scenario UNOS Vision was used to establish and destroy audio stream connections between OSC and AES-64 end stations on an Ethernet AVB network. A timing investigation was conducted with:

- the OSC end station as AVB listener and the AES-64 end station as AVB talker. This setup is shown in Figure 9.6a.
- the OSC end station as AVB talker and the AES-64 end station as AVB listener. This setup is shown in Figure 9.6b.

In both cases, the *startTime* was noted as the time when the first connection management command was transmitted by UNOS Vision, and the *endTime* was noted as the time when the last connection management response was received by UNOS Vision. The times noted were from the packets captured by Wireshark [162].



(a) Ethernet AVB network with OSC end station as AVB listener



(b) Ethernet AVB network with OSC end station as AVB talker

Figure 9.6: Timing connection management of network with OSC and AVB end stations

Figure 9.6a shows the following steps for establishing a stream connection between an AES-64 talker and an OSC listener:

1. UNOS Vision sends an AES-64 command to obtain the stream ID of the AES-64 talker.
2. The AES-64 talker sends an AES-64 response to UNOS Vision.
3. UNOS Vision sends an AES-64 command to the OSC proxy in order to set the stream ID of the OSC/AVB listener's input.
4. The OSC proxy translates the received AES-64 command to a corresponding OSC command, then it sends the translated command to the OSC/AVB listener.
5. The OSC/AVB listener sends an OSC response to the proxy.
6. The proxy translates the OSC response to the appropriate AES-64 response, then it sends the AES-64 response to UNOS Vision.

Figure 9.6a shows the following steps for establishing a stream connection between an OSC talker and an AES-64 listener:

1. UNOS Vision sends an AES-64 command to obtain the stream ID of the OSC/AVB talker.
2. The OSC proxy receives the AES-64 command on behalf of the OSC talker, then it translates the received message to the corresponding OSC command and sends the translated command to the OSC/AVB talker.
3. The OSC/AVB talker sends an OSC response to the proxy.
4. The proxy translates the OSC response to the appropriate AES-64 response, then it sends the AES-64 response to UNOS Vision.
5. UNOS Vision sends an AES-64 command to set the stream ID of the AES-63 talker input.
6. The AES-64 listener sends an AES-64 response to UNOS Vision.

The *startTime* and *endTime* was noted for thirty-five iterations of establishing audio stream connections, and thirty-five iterations of destroying audio stream connections. The average of the thirty-five iterations for the setup in Figure 9.6 is shown in Table 9.2.

	CONNECT (ms)	DISCONNECT (ms)
AES-64 talker and OSC listener	3.242	0.856
AES-64 listener and OSC talker	3.138	0.966

Table 9.2: Timing results for Ethernet AVB network of OSC and AES-64 end stations

From the results in Table 9.2:

- it takes an average of 3.242 milliseconds to establish an audio stream connection when the AVB listener is the OSC end station and the AVB talker is the AES-64 end station.
- it takes an average of 0.856 milliseconds to destroy an audio stream connection when the AVB listener is the OSC end station and the AVB talker is the AES-64 end station.
- it takes an average of 3.138 milliseconds to establish an audio stream connection when the AVB listener is an AES-64 end station and the AVB talker is the OSC end station.
- it takes an average of 0.966 milliseconds to destroy an audio stream connection when the AVB listener is an AES-64 end station and the AVB talker is the OSC end station.

The values in Table 9.2 are below the 10 milliseconds acceptable limit for feedback that was described in section 9.1.3. This implies that the proxy is a time efficient solution for interoperability between AES-64 and OSC end stations since it does not add any noticeable delays to the connection management procedures.

9.2.3 Results analysis

In Table 9.3, the values in Table 9.2 are compared with timings for the connection management of AES-64 end stations when the proxy is not used. The timing values for the connection management of AES-64 end stations are taken from Table 9.1. All values shown in Table 9.3 are the average of thirty-five iterations.

	CONNECT (ms)	DISCONNECT (ms)
AES-64 (talker) and OSC (listener)	3.242	0.856
AES-64 end stations	1.385	0.752
Difference (OSC listener)	**1.857**	**0.104**
AES-64 (listener) and OSC (talker)	3.138	0.966
AES-64 end stations	1.385	0.752
Difference (OSC talker)	**1.753**	**0.214**

Table 9.3: Comparing results obtained from layer 3 connection management procedure

Table 9.3 reveals that when the proxy is used by UNOS Vision for connection management:

- an average of 1.857 milliseconds are added when UNOS Vision utilizes the proxy to establish an audio stream connection between AES-64 and OSC end stations, with the OSC end station as the listener.
- an average of 0.104 milliseconds are added when UNOS Vision utilizes the proxy to destroy an audio stream connection between AES-64 and OSC end stations, with the OSC end station as the listener.
- an average of 1.753 milliseconds are added when UNOS Vision utilizes the proxy to establish an audio stream connection between AES-64 and OSC end stations, with the OSC end station as the talker.
- an average of 0.214 milliseconds are added when UNOS Vision utilizes the proxy to destroy an audio stream connection between AES-64 and OSC end stations, with the OSC end station as the talker.

9.3 Quantitative analysis of Layer 2/Layer 3 Proxy

The scenarios in this section describe the use of a proxy to enable a network controller to establish and destroy audio stream connections between networked audio devices that utilize layer 2 and layer 3 messaging. Two scenarios were investigated, and they are:

- Ethernet AVB network with AVDECC (IEEE 1722.1) end stations
- Ethernet AVB network with AES-64 and AVDECC end stations

The results of these investigations are provided in the following sections.

9.3.1 Scenario One: Connection between AVDECC end stations

In this scenario, the connection management procedure between networked AVDECC end stations was timed. The aim was to determine the time added by the proxy when connection management was performed on two AVDECC end stations. A comparison was conducted between the time taken for:

- a test application (running on a PC) to establish and destroy an audio stream connection by sending AVDECC commands directly to the AVDECC listener (as illustrated in Figure 9.8a), and
- UNOS Vision to establish and destroy an audio stream connection when utilizing the AVDECC proxy. This is illustrated in Figure 9.8b.

This investigation was conducted using commercial Ethernet AVB end stations. The network topology (which is the same as in Figure 8.13) is shown in Figure 9.7.

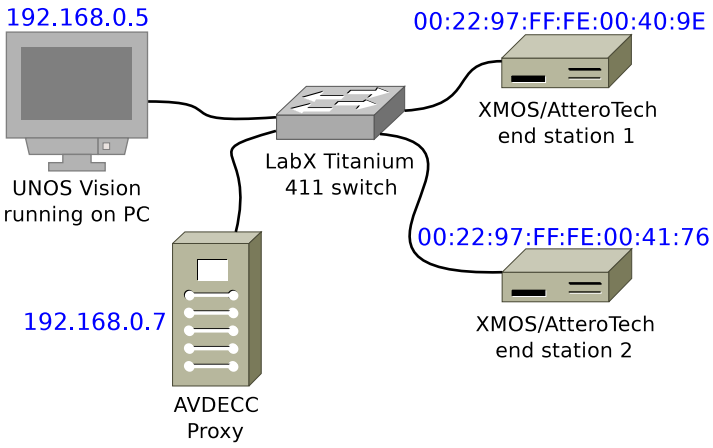


Figure 9.7: Test bed topology with commercially available Ethernet AVB end stations

In Figure 9.7, an Ethernet AVB switch ('*LabX Titanium 411*') is used to connect two XMOS/AtteroTech end stations ('*XMOS/AtteroTech end station 1*' and '*XMOS/AtteroTech end station 2*') that implement the IEEE 1722.1 standard. Also connected on the network is the '*AVDECC Proxy*' that was described in chapter 8, and UNOS Vision running on a PC workstation.

Wireshark was used to capture the packets transmitted on the network, and the times were noted [162]. In Figure 9.8a, the time at which an AVDECC command was issued by the test application was noted as the *startTime*. The time at which the test application received a response from the '*AVDECC listener*' was noted as the *endTime*.

Figure 9.8a shows the following steps for establishing a stream connection:

1. The test application sends an AVDECC command to the listener in order to establish a stream connection between the listener and talker AVDECC end stations.
2. The listener sends an AVDECC command to the talker. This command specifies which of the talker's source streams that the listener wishes to receive.
3. The talker sends an AVDECC response to the listener. If the stream is available, this response will also contain the multicast MAC address for the particular stream connection.
4. The listener sends an AVDECC response to the test application, indicating whether or not the connection was established.

Using the same network packet analyzer, the time at which UNOS Vision issued an AES-64 message to the AVDECC proxy (in Figure 9.8b) was noted as the *startTime*. The time at which a response was received by *'UNOS Vision'* from the '*AVDECC proxy*' was noted as the *endTime*.

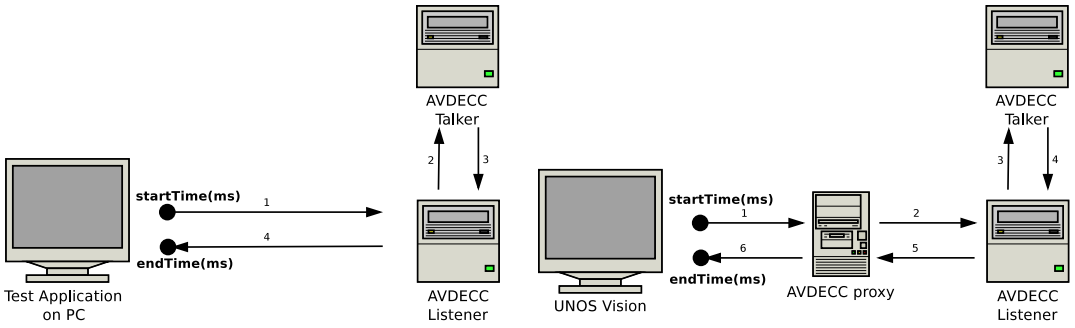
Figure 9.8b shows the following steps for establishing a stream connection with the aid of the proxy:

1. UNOS Vision sends an AES-64 command to the AVDECC proxy in order to obtain the stream ID of the AVDECC talker.
2. The AVDECC proxy translates the received AES-64 command to the corresponding AVDECC command, then it sends the translated command to the AVDECC listener.
3. The listener sends an AVDECC command to the talker. This command specifies which of the talker's source streams that the listener wishes to receive.

4. The talker sends an AVDECC response to the listener. If the stream is available, this response will also contain the multicast MAC address for the particular stream connection.

5. The listener sends an AVDECC response to the AVDECC proxy, indicating whether or not the connection was established.

6. The proxy translates the AVDECC response to the appropriate AES-64 response, then it sends the AES-64 response to UNOS Vision.



(a) Test application as network controller (b) UNOS Vision as network controller

Figure 9.8: Timing connection management between two AVDECC end stations

In the illustrations in Figure 9.8, the time taken (overhead) was calculated as the difference between *endTime* and *startTime* in milliseconds.

The test application in Figure 9.8a and the AVDECC proxy in Figure 9.8b used the same machine (PC workstation). This was to ensure that performance of the PC did not affect the results. The workstation had the following specification:

- Intel Quad-Core (*i7*) processor, with each core processing at a rate of 2.70GHz
- Linux (*Ubuntu 12.04*) operating system
- 1000Mbps (gigabit) Ethernet connector.

Thirty-five iterations of the connection management sequences depicted in Figure 9.8a and Figure 9.8b were observed. The averages of the time taken (in milliseconds) for the thirty-five iterations are shown in Table 9.4.

	CONNECT (ms)	DISCONNECT (ms)
UNOS via AVDECC proxy	9.549	4.344
Test application	6.248	1.072
Difference	**3.301**	**3.282**

Table 9.4: Timing results of connection management between networked AVDECC end stations

From the results of Table 9.4:

- it takes UNOS Vision an average of 9.549 milliseconds to establish a stream connection with the aid of the AVDECC proxy
- it takes UNOS Vision an average of 4.344 milliseconds to destroy a stream connection with the aid of the AVDECC proxy
- it takes the test application an average of 6.248 milliseconds to establish a stream connection
- it takes the test application an average of 1.072 milliseconds to destroy a stream connection
- the proxy adds 3.301 milliseconds delay when establishing a stream connection
- the proxy adds 3.282 milliseconds delay when destroying a stream connection

As described in section 9.1.3, 10 milliseconds has been chosen as the permissible maximum time for auditory feedback. Hence the 9.549 milliseconds interval between events when using UNOS Vision to establish an audio stream connection between two AVDECC end stations (with the aid of the AVDECC proxy) is acceptable. The 4.344 milliseconds interval when destroying an audio stream connection also falls within the permissible maximum time for auditory feedback.

9.3.2 Scenario Two: Connection between AVDECC and AES-64 end stations

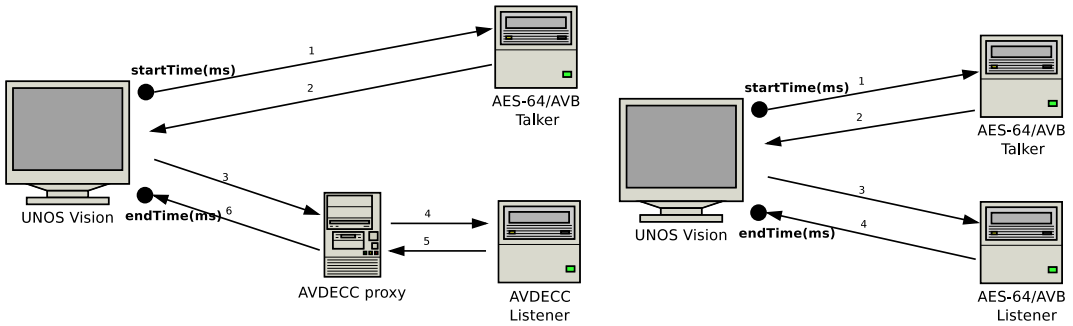
In this scenario, the connection management procedure for an Ethernet network that contains AES-64 and AVDECC end stations was timed. The intention was to determine the overhead added when a proxy is used for connection management when:

- the AVDECC end station assumes the role of AVB talker, and

- the AVDECC end station assumes the role of AVB listener.

The setup for this test is shown in Figure 9.9a. The connection times were noted in order to determine if the proxy added a noticeable delay to the connection management procedure.

The values obtained from this investigation were compared with the setup shown in Figure 9.9b, where the time taken for the connection management procedure between two AES-64 end stations (on the same Ethernet AVB network), was noted.



(a) AES-64 and AVDECC Ethernet AVB end stations (b) AES-64 Ethernet AVB end stations

Figure 9.9: Quantitative analysis of AES-64 and AVDECC connection management procedure

In Figure 9.9a, the *startTime* was noted as the time when '*UNOS Vision*' transmitted the first AES-64 command for the connection management procedure. The *endTime* was noted as the time when '*AVDECC proxy*' returned a final response to '*UNOS Vision*'. The time taken (overhead added) was calculated as the difference between *endTime* and *startTime* in milliseconds.

Similarly in Figure 9.9b, the *startTime* was noted as the time when '*UNOS Vision*' transmitted the first AES-64 command for the connection management procedure, and the *endTime* was noted as the time when '*AES-64 Listener*' returned a final response to '*UNOS Vision*'.

The test bed topology used for this test in Figure 9.9a is shown in Figure 9.10.

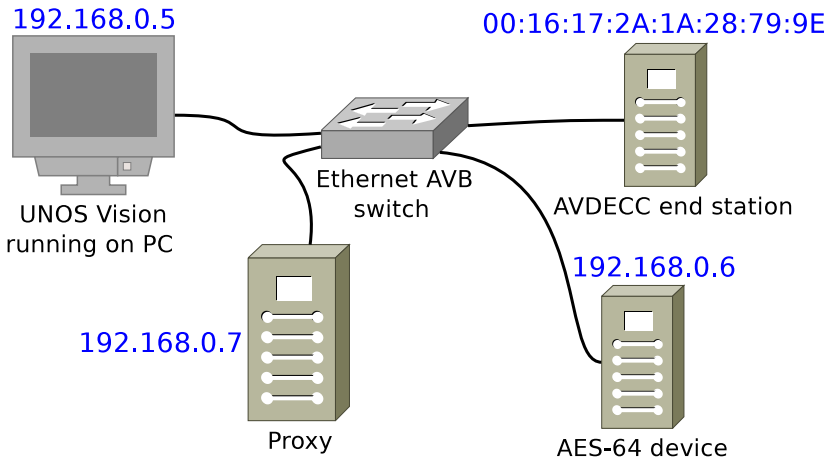


Figure 9.10: Test bed topology for Ethernet network with AES-64 and AVDECC end stations

Shown in Figure 9.10 is an Ethernet AVB network with a UNOS Vision controller, which is running on a PC workstation. The network consists of:

- UNOS Vision running on a PC,
- an '*Ethernet AVB switch*',
- an '*AVDECC end station*',
- an '*AES-64 device'*, and
- a '*Proxy*'.

The '*AVDECC end station*' and '*AES-64 device'* shown in Figure 9.10 are audio streaming devices, which run on PC workstations. When testing for connection management between two AES-64 end stations, the '*AVDECC device'* is replaced with an '*AES-64 end station*' so that only AVDECC end stations are on the network.

In these tests:

- UNOS Vision is used for device configuration and control. UNOS Vision runs on the same PC (workstation) for each of the scenarios. The PC specifications are:

Processor	Intel Core (i7) @ 2.93GHz
Operating System	Windows 7 Professional (64-bits)
Ethernet Speed	1000Mbps

- The AVDECC and AES-64 end stations are implemented as PC audio streaming devices. Identical PC specifications were used for both end stations. The specifications are:

Processor	Intel Core Quad (Q9400) @ 2.66GHz
Operating System	Linux Ubuntu 10.10 (32-bits)
Ethernet Speed	1000Mbps

- When a proxy was used, it ran on the same PC for each of the investigations. The specification of the PC is:

Processor	Intel Quad-Core (i7) @ 2.70GHz
Operating System	Linux Ubuntu 12.04 (32-bits)
Ethernet Speed	1000Mbps

Table 9.5 shows the results of the tests, when establishing (*CONNECT*) and destroying (*DISCONNECT*) an audio stream connection. The times noted in Table 9.5 are the average of thirty-five attempts for each of the connection management processes (that is establishing and destroying stream connections). In Table 9.5, the values for the connection management procedure between AES-64 end stations is the same as those in Table 9.1.

	CONNECT (ms)	DISCONNECT (ms)
AES-64 listener and AVDECC talker	3.370	3.404
AES-64 talker and AVDECC listener	3.449	3.540
AES-64 talker and AES-64 listener	1.385	0.752

Table 9.5: Timing results of connection management between networked AES-64 and AVDECC devices

From the results of Table 9.5:

- it takes 3.370 milliseconds to establish an audio stream connection between an AES-64 device (as Ethernet AVB listener) and an AVDECC end station (as Ethernet AVB talker)
- it takes 3.404 milliseconds to destroy an audio stream connection between an AES-64 device (as Ethernet AVB listener) and an AVDECC end station (as Ethernet AVB talker)
- it takes 3.449 milliseconds to establish an audio stream connection between an AVDECC end station (as Ethernet AVB listener) and an AES-64 device (as Ethernet AVB talker)

- it takes 3.540 milliseconds to destroy an audio stream connection between an AVDECC end station (as Ethernet AVB listener) and an AES-64 device (as Ethernet AVB talker)
- it takes 1.385 milliseconds to establish an audio stream connection between two AES-64 end stations where one of them is the Ethernet AVB listener and the other is Ethernet AVB talker
- it takes 0.752 millisecond to destroy an audio stream connection between two AES-64 end stations where one of them is the Ethernet AVB listener and the other is Ethernet AVB talker

These values (shown in Table 9.5) are below the 10 milliseconds perception time limit described in section 9.1.3. This demonstrates that the proxy is a viable solution for integrating layer 2 and layer 3 audio devices (in particular AES-64 and AVDECC devices), since the overhead is not noticeable.

9.3.3 Results analysis

When the values in Table 9.4 are compared with those in Table 9.5, there are clear differences in the time taken for connection management in the different tests. These differences are shown in Table 9.6.

		CONNECT (ms)	DISCONNECT (ms)
Commercial AVDECC end stations	UNOS Vision network controller	9.549	4.344
	Test application network controller	6.248	1.072
PC AVDECC	AVDECC end stations	4.292	1.663
	AES-64 listener and AVDECC talker	3.370	3.404
	AES-64 talker and AVDECC listener	3.449	3.540
	AES-64 talker and AES-64 listener	1.385	0.752

Table 9.6: Comparing results obtained from layer 2/layer 3 connection management procedure

From Table 9.6, the values obtained for connection management between AVDECC end stations varies. When the connections are created by UNOS Vision between the commercial (XMOS/AtteroTech) end stations, the average connect and disconnect values are 9.549 and 4.344, respectively. When the connections are created by UNOS Vision between the AVDECC PC end stations, the average connect and disconnect values are

4.292 and 1.663, respectively. The connections between commercial AVDECC end stations are delayed by 5.257 milliseconds beyond those of the PC AVDECC end stations. The PC AVDECC end stations perform better than the commercial AVDECC end stations by an average of 2.681 milliseconds when being disconnected by UNOS Vision.

The same proxy was used in both tests, so it became necessary to investigate the cause of the noted differences in the average values obtained.

Tests were done to determine the latency of the switches in order to establish whether the extra overhead was a result of the LabX Titanium 411 switch, or the D-Link Gigabit Ethernet switch (used in the commercial end stations or PC end stations scenarios respectively). The latency of a packet-switched network can be expressed as the round-trip time (RTT) [165]. Figure 9.11 shows a network in which the RTT is used to specify latency.

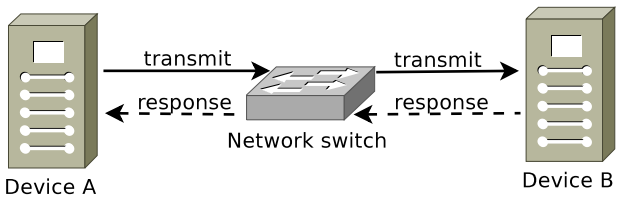


Figure 9.11: Switch latency investigation

A *ping* test can be used to determine RTT [166]. With regards to Figure 9.11, the ping test requires that '*Device A'* sends a *ping packet* to '*Device B'*, which is illustrated as '*transmit*'. In response to the *'transmit'* ping packet, '*Device B'* sends a ping response to '*Device A',* which is illustrated as '*response*' in Figure 9.11. The RTT is the time it takes between '*Device A'* sending a ping packet and it ('*Device A'*) receiving a response from '*Device B'*.

The average RTT for the network depicted in Figure 9.11 after fifty ping transmits and responses is shown in Table 9.7.

	Average RTT (ms)
LabX Titanium 411	0.2487
D-Link Gigabit Ethernet	0.3657

Table 9.7: Results for ping test to determine switch latency

The values shown in Table 9.7 reveal that the LabX switch has a RTT of 0.2487 milliseconds, and the D-Link switch has a RTT of 0.3657 milliseconds. These values do not sufficiently account for the differences observed in Table 9.6 in:

- connection management between commercial AVDECC end stations
- connection management between PC (workstation) AVDECC end stations.

Hence the difference in value has to be attributed to the performance of the end stations. The XMOS/AtteroTech end stations incorporate the XMOS *XS1-L2* chip, which has the following specification:

Number of cores	2
Instruction set	32-bit
MIPS	500 per core
Clock frequency	500 MHz per core

Table 9.8: XMOS XS1-L2 chip specification

Each core in Table 9.8 is analogous to a 'tile' on an XMOS board. An XMOS tile has 8 XMOS cores that each run at 63 MHz when all eight cores are in use [167].

The PC workstation utilizes an *Intel Core 2 Quad Q9400* processor with the following specification:

Number of cores	4
Instruction set	64-bit
Clock frequency	2.66 GHz per core

Table 9.9: Intel Core Quad Q9400 specification

When the XMOS processor is compared with the processor in the PC, it becomes clear that the PC workstations should outperform the XMOS end stations. However, the processing power (of a processor) alone does not provide sufficient indication of the performance of a computer. Other factors such as the cache size, processor loads, and operating system latency contribute to a computer's performance [168].

Table 9.6 also reveals that the connection times for AVDECC (PC) end stations are larger than those of the AES-64 end stations. In particular, the AES-64 end stations outperform the AVDECC end stations by:

- 2.907 milliseconds for connect
- 0.911 milliseconds for disconnect

Since the AVDECC and AES-64 end stations used the same PC hardware and were deployed on the same network setup, these delays can be attributed to the relative complexity of the AVDECC endpoint implementation. A number of state transitions are

involved in processing an AVDECC connection management packet (ACMP). The state transition diagram (Figure 7.3) illustrates how an AVDECC end station processes an AVDECC message. The AES-64 approach entails triggering an appropriate callback for the parameter addressed by a received message (described in subsection 3.4.2 on page 82).

9.4 Summary

In this chapter, visual and auditory stimuli were described as relevant forms of feedback to the user of an audio network control application. The perception time for visual feedback was described by the MHP model to be less than 100 milliseconds, and the perception time for auditory feedback was described to be less than 10 milliseconds. In terms of perception time, auditory stimuli were described as being more sensitive than visual stimuli.

In order to determine the acceptable latency of the proxy that was described in chapter 6 and 8, 10 milliseconds was used as the criterion for a quantitative analysis.

The quantitative analysis involved timing the connection management procedure for establishing and destroying an audio stream connection within different scenarios. The time at which a control command was transmitted by a controller was noted as the *startTime* and the time at which the controller received a response was noted as the *endTime*. The difference between the startTime and endTime was noted as the time taken for the particular transaction.

The average of 35 iterations of the time taken for each transaction was noted for different scenarios. They reveal the following:

- When the proxy was used for connection management between devices that implement layer 3 audio control protocols:
  - the proxy enabled the creation and termination of audio stream connections within the 10 milliseconds limit, when the network consisted of only OSC Ethernet AVB end stations.
  - the proxy enabled the creation and termination of audio stream connections between OSC and AES-64 Ethernet AVB end stations within the 10 milliseconds limit, irrespective of whether the OSC end station was AVB listener or AVB talker.

- When the proxy was used for connection management between devices that implement layer 2 and layer 3 audio control protocols:
  - the proxy enabled the creation and termination of audio stream connections within the 10 milliseconds limit, when the network consists of only AVDECC (IEEE 1722.1) end stations.
  - the proxy enabled the creation and termination of audio stream connections between AVDECC and AES-64 (Ethernet AVB) end stations within the 10 milliseconds limit, irrespective of whether the AVDECC end station was AVB listener or AVB talker.

The results obtained show that the overhead added by the proxy was not sufficiently noticeable to a user establishing or destroying stream connections from a graphical user interface, thereby making the proxy a viable solution for enabling control protocol interoperability, from a time efficiency point of view.

Chapter 10

Conclusion

Digital networks are becoming the preferred solution for interconnecting audio devices in large installations such as stadiums, casinos, theme parks, conference centers, air ports, shopping malls and places of worship. In these installations digital audio networks enable the distribution of multiple channels of audio between devices that are some distance apart.

The networking technology used to transport audio streams considers the time-sensitive nature of audio and endeavors to adequately provide for reliable transport of time-sensitive data. Currently, there exist a number of audio networking technologies. They include:

- IEEE 1394
- Ethernet AVB
- CobraNet
- RockNet
- EtherSound
- Q-LAN
- RAVENNA
- Livewire
- Dante

The above technologies have been described in chapter 2. Some of them have been published by a standards body, while others are proprietary solutions for audio transport. An audio networking technology should be able to ensure that:

- the necessary network resources (such as bandwidth, buffer queues, and channels) for the transmission of an audio stream are guaranteed for the duration of the stream connection.
- the networked devices are tightly synchronized by providing a mechanism for exchanging time information. This will avoid glitches and jitter since the audio sample rate of the transmitter will be the same as that of the receiver(s). It also ensures that there is 'lip-sync' between multiple receiving devices by providing them with a common presentation time.
- the network infrastructure does not interfere with the audio data transmission by adding significant delay.

Various techniques are used to ensure that the above three requirements are met by the available audio networking technologies. As a result of the different transport techniques used by these audio networking technologies, interoperability remains a challenge. Thus, devices that are compatible with one audio transport technology type cannot exchange audio data with devices that are designed to communicate on different audio transport technologies. Chapter 2 has described attempts to enable audio transport interoperability.

One of the benefits of audio networks is the ability to remotely configure, connect, control and monitor audio devices that are some distance away from a control station. An audio control protocol enables remote access to the controls within a device by the exchange of control messages. An audio control protocol defines:

- a message structure that formats the protocol commands and responses
- a device model that is used to organize the various controls (within an audio device) in a structured manner, and makes them remotely accessible
- a mechanism by which the networked devices can discover each other
- a procedure for establishing and destroying audio stream connections between the networked devices

There are a number of audio control protocols. Each audio control protocol achieves the above requirements in its own particular manner. Some of the available audio control protocols are:

- Open Sound Control (OSC)
- Architecture for Control Networks (ACN)
- Common Control Interface for Networked Audio and Video Products (IEC 62379)
- Audio Engineering Society standard for audio applications of networks - Command, control and connection management for integrated media (AES-64)
- Open Control Architecture (OCA)
- Audio Video Control (AV/C)
- IEEE 1722.1 (AVDECC)
- Music Local Area Network (mLAN)

Audio control protocols can be classified based on the OSI/ISO 7 layer that they utilize for control messaging. The two categories identified are:

- Layer 3 audio control protocols - includes audio control protocols that transport their messages within layer 3 (mostly IP) packets. Examples include the OSC, ACN, IEC 62379, AES-64 and OCA protocols.
- Layer 2 audio control protocols - includes audio control protocols that utilize layer 2 packets for messaging. For example AV/C, IEEE 1722.1 and mLAN.

Details about the audio control protocols mentioned here, can be found in chapter 3.

While the audio networking technology is designed to guarantee reliable audio data transport, it also provides transport for control messages. Figure 10.1 shows control messaging and audio stream transmission between two networked audio devices.

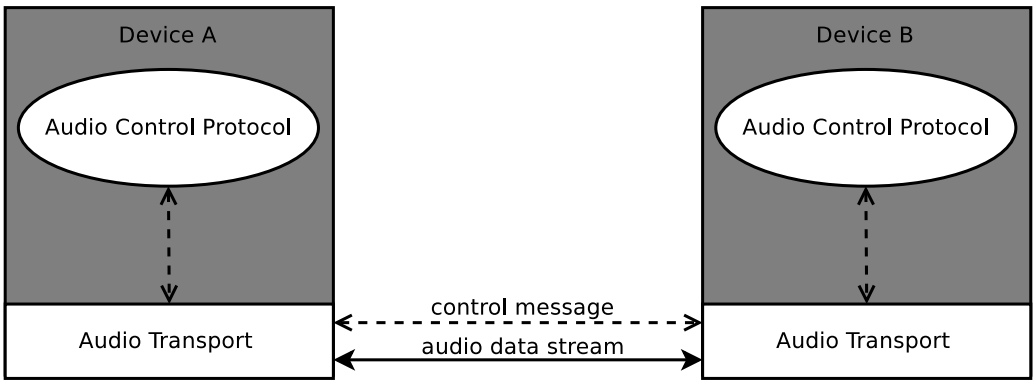


Figure 10.1: Control messaging and audio data transmission

The figure illustrates how the audio control protocol is used for communication between devices. Control messages for device monitoring and configuration are exchanged between *'Device A'* and *'Device B'*. When a device receives a control message at its '*Audio Transport'* layer, it sends the message to the '*Audio Control Protocol'* for processing. A successful connection management command to establish an audio stream connection will cause audio data to be transmitted between *'Device A'* (source device) and *'Device B'* (destination device).

A network controller can remotely establish and destroy audio stream connections between networked devices. This process is aided by the audio control protocol that the devices implement. A typical scenario for establishing an audio stream connection between two devices on the same audio networking technology is shown in Figure 10.2.

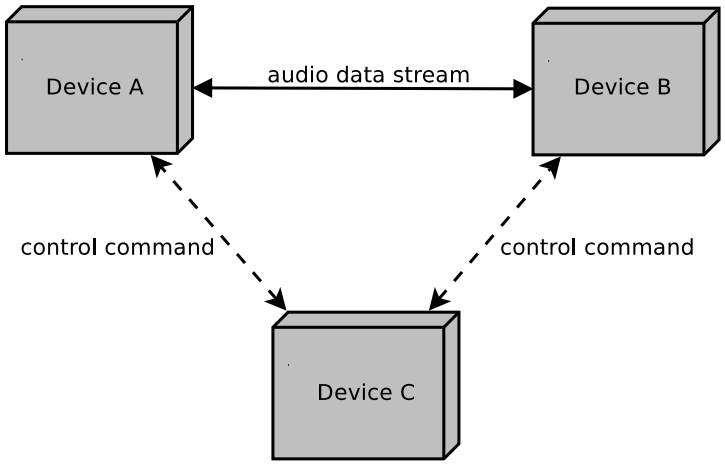


Figure 10.2: Controller configures networked devices

In Figure 10.2, '*Device A*', '*Device B*' and '*Device C*' implement the same audio control protocol. '*Device C*' is a network controller, and it sends control messages to '*Device A*' and '*Device B*' according to the procedure for connection management that is defined by their common control protocol. Following the exchange of connection management messages to establish an audio stream connection, audio data is transported between '*Device A*' and '*Device B*'.

Interoperability is said to exist when audio can be exchanged between the networked devices. Most commercial audio devices implement a single control protocol, thus they can only communicate with other networked devices that implement the same protocol. It is often desirable to network devices that implement different audio control protocols. For example a sound engineer may want audio from an AES-64 mixing console to be received by IEEE 1722.1 speakers within the same Ethernet AVB network. Currently interoperability between devices that implement different control protocols remains a

challenge.

This research project has proposed the use of a protocol command translator that implements multiple audio control protocols, to achieve interoperability between networked audio devices that implement different control protocols. Figure 10.3 depicts how a command translator can be used to achieve interoperability.

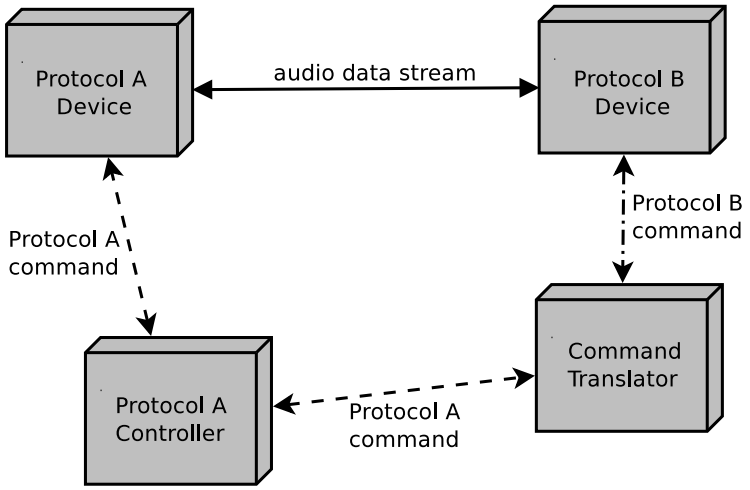


Figure 10.3: Command translator enables common control

In Figure 10.3, '*Protocol A Controller*' is a network controller that sends 'Protocol A' commands in order to establish an audio stream connection between '*Protocol A Device*' and '*Protocol B Device*'. The '*Command Translator*' receives all 'Protocol A' commands that are intended to configure '*Protocol B Device*'. Then it sends the corresponding 'Protocol B' command to '*Protocol B Device*'.

The functional process that occurs within a command translator is depicted in Figure 10.4.

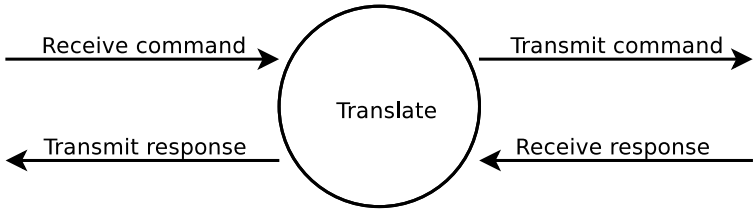


Figure 10.4: Command translation process

The process shown in Figure 10.4 is:

- a protocol command is received from a controller
- the received command is translated

- the translated command is sent to the intended target
- a response message is received from the target
- the response is translated
- the translated response is sent to the controller

Chapters 6 and 8 describe how a command translator, which was implemented as a proxy, is able to map control commands from one audio control protocol to another. An example of the mapping technique for three different audio control protocols is shown in Table 10.1.

Command index	Command description	Protocol A	Protocol B	Protocol C
1	Get device name	A_1	B_1	C_1
2	Get device address	A_2	B_2	C_2
3	Get number of AVB sources	A_3	B_3	C_3
4	Get number of AVB sinks	A_4	B_4	C_4
5	Set channel number	A_5	B_5	C_5 C_6

Table 10.1: Command message mapping

There are five command messages in Table 10.1, each uniquely identified by a command index. The three protocols (Protocol A, Protocol B and Protocol C) fulfill each of the commands by sending specific messages. If the proxy receives a Protocol B command (B_3) to obtain the number of Ethernet AVB streams from devices that implement Protocol A and Protocol C, the following occurs:

- the proxy determines the index of the received message, in this case the index is '3'
- the proxy sends Protocol A and Protocol C commands that correspond to index '3', that is A_3 and C_3 respectively

It is possible for the proxy to translates a received command to multiple commands of the target device. Also the proxy might translate multiple commands to a single command of a target device. For example in Table 10.1, a Protocol B command (B_5) to set the channel number is translated to a single command (A_5) for Protocol A, and two commands ($C_5$and C_6) for Protocol C.

By mapping commands in this manner, other audio control protocols can be incorporated into the proxy. In order to do this, the mapping table will have to be updated with the new protocol commands for each command index.

Several tests were conducted in order to investigate the effectiveness of the command translator approach. The tests involved utilizing a proxy for connection management between devices of different audio control protocols. Two proxies were investigated, and they are:

- a layer 3 proxy that enabled an AES-64 network controller to discover OSC servers, as well as enable connection management between AES-64 and OSC devices on an Ethernet AVB network.
- a layer 2/layer 3 proxy that enables an AES-64 network controller to discover IEEE 1722.1 (AVDECC) end stations, as well as enable connection management between AES-64 and AVDECC end stations on an Ethernet AVB network.

In both cases, the proxy was able to effectively allow a common controller to configure the networked Ethernet AVB end stations, irrespective of the audio control protocol that they implement.

It is possible to host the command translator within the same PC workstation as the network controller. An approach to achieving this for the AES-64 network controller application has been described in section 6.7. It entails implementing two AES-64 applications on the same PC (a controller application and a command translator application), each with its own AES-64 protocol stack. Each protocol stack is bound to a different IP address, and allocated a different AES-64 *Device ID*. Thus each application appears to the other as if it were hosted on a remote PC. This has the following advantages:

- avoiding a single point of failure that could result from the proxy going down.
- enabling the command translator functionality to take advantage of the processing power of the host PC workstation on which the control application is running.

A concern related to the proxy approach was the time required for protocol translation. To address this concern a number of quantitative tests were performed.

The Modeling Human Processor (MHP) is a conceptual model for human graphical user interface processing. MHP defines the time it takes an observer to perceive external visual stimuli (event) and respond to it as the *perceptual processor cycle time*. MHP defines the perceptual processor cycle time to be about 100 milliseconds. This means that any number of events that occur within a 100 milliseconds interval, appear as a single event to an observer.

In an investigation of gesture control systems, Wessel and Wright suggest that 10 milliseconds is an acceptable maximum time within which auditory feedback should be

received. In another paper, Moore indicates that humans are unable to distinguish between multiple auditory stimuli when they occur within 30 milliseconds.

Within the context of audio device control, the visual perception time is relevant when a graphical interface is used to control networked audio devices. The auditory perception time applies when connection management is performed on networked devices, and the receiving devices are connected to sound outputs.

In the quantitative analysis of the command translators (proxies), 10 milliseconds was chosen as the maximum acceptable latency that could be contributed by a command translator that is used for connection management.

When a PC workstation control application that has a graphical interface is used to establish an audio stream connection between two networked Ethernet AVB end stations, the control application may do the following:

- Determine the 'stream ID' of the audio stream from the AVB talker (source device).
- Specify the 'stream ID' of the audio stream that the AVB listener (destination device) should listen to.
- Instruct the AVB talker to advertise its stream on the (Ethernet AVB) network via MRP.
- Instruct the AVB listener to indicate to the network (via MRP) that it is prepared to receive the specified stream .

The graphical control application might provide a button that can be used to establish a stream connection according to the above steps. All four steps should appear as a single event to a sound engineer who is using the control application. That is, when the button is clicked by the sound engineer the connection should be visually and audibly determined as being instantaneous.

The command translators (proxies) implemented in this research were investigated to determine whether they added significant delay that would cause the connection management transaction to exceed the 10 milliseconds limit. The investigations involved setting up a network that utilized the proxy, and subsequently:

- Noting the time at which a control application transmitted the first command in the sequence of connection management commands necessary to fulfill a task.

- Noting the time at which the control application received the final response in the sequence of connection management commands necessary to fulfill the task.

An AES-64 connection management application was used to establish and destroy stream connections between Ethernet AVB end stations that implement the following audio control protocols:

- AES-64
- OSC
- IEEE 1722.1 (AVDECC)

The results of the quantitative tests were described in chapter 9.

The results revealed that the proxy enabled connection management efficiently, since it did not add a sufficient overhead (time delay) to the connection management procedure. The observed worst case for the CONNECT procedure was between two commercial AVDECC end stations, which took 9.549 milliseconds. The worst case for the DISCONNECT procedure was between two commercial AVDECC end stations. It took 4.344 milliseconds for the DISCONNECT procedure.

Since the values obtained for the worst case are less than 10 milliseconds, the addition of the proxy into the network did not cause an observable difference in the perception of a user. This served to demonstrate that the protocol command translation approach provides an efficient solution for protocol interoperability between networked audio devices.

References

[1] P.J. Foulkes. *An Integration into the Control of Audio Streaming across Diverse Quality of Service Networks*. Dissertation, Rhodes University, 2011.

[2] IEEE-SA. IEEE Standard for a High-Performance Serial Bus. *IEEE Std 1394-2008*, 2008.

[3] Institute of Electrical and Electronics Engineers (IEEE). *Audio/Video Bridging Task Group*. IEEE. `http://www.ieee802.org/1/pages/avbridges.html` [Accessed: 2011.04.28].

[4] Cirrus Logic. Cobranet faq. `http://www.cobranet.info/support/faq` [Accessed: 2012.09.10].

[5] Riedel. *RockNet - Performance Audio Networks*. `http://www.riedel.net/LinkClick.aspx?link=Downloads%2fBroschures%2fRiedel_RockNet_EN.pdf&portalid=0&mid=0&language=en-US&forcedownload=true` [Accessed: 2012.09.11].

[6] Digigram. EtherSound Overview - An Introduction to the technology Rev. 2.0c. October 2004.

[7] K. Gross. QSC White paper: Q-LAN. October 2009.

[8] ALC NetworX. *RAVENNA - Operating Principles Draft 1.0*. ALC NetworX GmbH, June 2011.

[9] Axia Audio. *Introduction to Livewire - IP Audio System Design Reference & Primer*. `http://axiaaudio.com/tech/introduction-to-livewire-systems-primer-v21/download` [Accessed: 2012.09.17].

[10] Audinate. Dante - Digital Audio Networking Just Got Easy. *Audinate Whitepaper*, 2009.

[11] Audio Engineering Society (AES). AES-X192 (SC-02-12-H) Draft 1.0 AES standard for audio applications of networks - High-performance streaming audio-over-IP interoperability. November 2012.

[12] Institute of Electrical and Electronics Engineers (IEEE). The IEEE website, 2012. `http://www.ieee.org/` [Accessed: 2012.08.10].

[13] Audio Engineering Society (AES). `http://www.aes.org/` [Accessed:2012.12.09].

[14] A. Schmeder, A. Freed, and D. Wessel. Best Practices for Open Sound Control. In *Linux Audio Conference*, Utrecht, NL, May 2010.

[15] American National Standard (ANSI). *Entertainment Technology - Architecture for Control Networks*, draft document ansi e1.17 edition, 2005.

[16] International Electrotechnical Commission (IEC). IEC 62379 Common Control Interface for networked audio and video equipment - Background, 2005. `http://www.iec62379.org/details.html` [Accessed: 2012.10.05].

[17] Audio Engineering Society (AES). *AES standard for audio applications of networks - Command, control, and connection management for integrated media*, AES64-2012 edition, January 2013.

[18] J. Berryman, G. van Beuningen, K. Dalbjurn, H. Hamamatsu, M. Lave, N. O'Neil, M. Renz, M. Smaak, D. Takahashi9, S. van Tienen, B. Tudor, and E. Wetzell. The Open Control Architecture. In *Audio Engineering Society Convention 133*, October 2012.

[19] Institute of Electrical and Electronics Engineers (IEEE). *IEEE P1722.1/D21: Draft Standard for Device Discovery, Connection Management and Control Protocol for IEEE 1722 Based Devices*, July 2012.

[20] R. Foss and J. Fujimori. mLAN - The Current Status and Future Directions. In *Audio Engineering Society Convention 113*, October 2002.

[21] J. Emmett. Panel Discussion: The View from Here. In *Audio Engineering Society Conference: UK 8th Conference: Digital Audio Interchange (DAI)*, May 1993.

[22] F. Rumsey. Audio Networking for the Pros. *Journal of the Audio Engineering Society*, 57(4):271–275, 2009.

[23] D. Jacobs and D.P. Anderson. Design Issues for Digital Audio Networks. In *Audio Engineering Society Conference: 13th International Conference: Computer-Controlled Sound Systems*, December 1994.

[24] B. Moses. Audio Applications of the IEEE 1394 High Performance Serial Bus. In *Audio Engineering Society Conference: UK 15th Conference: Moving Audio, Pro-Audio Networking and Transfer*, May 2000.

[25] N. Fonseca and E. Monteiro. Latency in Audio Ethernet Networks. In *Audio Engineering Society Convention 114*, March 2003.

[26] N. Olifer and V. Olifer. *Computer Networks: Principles,Technologies And Protocols For Network Design*. Wiley India Pvt. Limited, 2006.

[27] F. Rumsey. Audio in the Age of Digital Networks. *Journal of the Audio Engineering Society*, 59(4):244–253, 2011.

[28] IEEE-SA. The IEEE Standards Association website, 2012. `http://standards.ieee.org/` [Accessed: 2012.08.10].

[29] IEEE-SA. IEEE Standard for a High-Performance Serial Bus. *IEEE Std 1394-1995*, August 1996.

[30] 1394 Trade Association (1394TA). *IEEE1394: versatility, performance, security, flexibility*. `http://www.1394ta.org/press/WhitePapers/IEEE%201394%20Comparison.pdf` [Accessed: 2010.08.10].

[31] IEEE-SA. IEEE Standard for a High-Performance Serial Bus – Amendment 1. *IEEE Std 1394a-2000 (Amendment to IEEE Std 1394-1995)*, 2000.

[32] 1394 Trade Association (1394TA). *FireWireTM Reference Tutorial (An Informational Guide)*, January 2010.

[33] IEEE-SA. IEEE Standard for a High-Performance Serial Bus – Amendment 3. *IEEE Std 1394c-2006 (Amendment to IEEE Std 1394-1995)*, 2006.

[34] 1394 Trade Association (1394TA). The 1394 Trade Association website. `http://www.1394ta.org` [Accessed 2012.08.21].

[35] H.A. Okai-Tettey. *High Speed End-to-end Connection Management in a Bridged IEEE 1394 Network of Professional Audio Devices*. Dissertation, Rhodes University, 2005.

[36] R.H.J. Bloks. The IEEE-1394 high speed serial bus. *Philips Journal of Research*, 50(1-2):209–216, 1996.

[37] International Electrotechnical Commission (IEC). *Consumer audio/video equipment - Digital interface - Part 6: Audio and music data transmission protocol*, 2nd edition edition, October 2005.

[38] International Electrotechnical Commission (IEC). *Consumer audio/video equipment - Digital interface - Part 1: General*, 2nd edition edition, January 2003.

[39] O.P. Igumbor. A Proxy Approach to Protocol Interoperability within Digital Audio Networks. Masters Thesis, Rhodes University, 2009.

[40] D. Anderson. *FireWire system architecture (2nd ed.): IEEE 1394a*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[41] 1394 Trade Association (1394TA). FireWire Reaches 4 Gigabit/Second Speeds. `http://www.1394ta.org/press/TAPress/2012_0110.html` [Accessed 2012.08.21].

[42] M. J. Teener. AVnu Alliance Whitepaper: No-excuses Audio/Video Networking: the Technology Behind AVnu. August 2009.

[43] N. Parik. Ethernet and Multimedia Applications - The History and the Future - Part 1. November 2007.

[44] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks - Amendment: 9: Stream Reservation Protocol (SRP)*, 28 June 2010.

[45] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard for Local and Metropolitan Area Networks—Virtual Bridged Local Area Networks - Amendment: Forwarding and Queuing Enhancements for Time-Sensitive Streams*, 5 January 2010.

[46] Institute of Electrical and Electronics Engineers (IEEE). *Standard for Local and Metropolitan Area Networks - Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*, 30 March 2011.

[47] Institute of Electrical and Electronics Engineers (IEEE). *Standard for A Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, 2008.

[48] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard for Local and Metropolitan Area Networks: Audio Video Bridging (AVB) Systems*, 14 March 2011.

[49] Institute of Electrical and Electronics Engineers (IEEE). *IEEE Standard for Layer 2 Transport Protocol forTime-Sensitive Applications in Bridged Local Area Networks*, May 2011.

[50] Institute of Electrical and Electronics Engineers (IEEE). *Multiple Registration Protocol (MRP)*, June 2007.

[51] A. Holzinger and A. Hildebrand. Realtime Linear Audio Distribution Over Networks: A Comparison of Layer 2 and 3 Solutions Using the Example of Ethernet AVB and RAVENNA. In *Audio Engineering Society Conference: 44th International Conference: Audio Networking*, November 2011.

[52] IEEE-SA. IEEE Standard for for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. *IEEE Std 1588-2008*, 2008.

[53] J. Koftinoff. Audio Video Bridging and Linux. August 2011. Presented at LinuxCon Vancouver.

[54] K. Gross and D.J. Britton. Deploying Real-Time Ethernet Networks. In *Audio Engineering Society Conference: UK 15th Conference: Moving Audio, Pro-Audio Networking and Transfer*, May 2000.

[55] K. Bradley and F. Richard. A Comparative Study of mLAN and CobraNet Technologies and their use in the Sound Installation Industry. In *Audio Engineering Society Convention 114*, March 2003.

[56] J.D. Case, M. Fedor, M.L. Schoffstall, and J. Davin. Simple Network Management Protocol (SNMP). RFC 1157 (Historic), May 1990.

[57] Riedel. RockNet - Digital Audio Network, 2012. `http://www.riedel.net/en-us/products/signaltransportprocessing/rocknetdigitalaudionetwork/about.aspx` [Accessed: 2012.08.10].

[58] Digigram. Technology Overview, 2008. `http://www.ethersound.com/technology/overview.php` [Accessed: 2012.09.10].

[59] Digigram. The EtherSound Standard. `http://www.ethersound.com/download/files/EtherSoundTechnology.pdf` [Accessed: 2012.09.10].

[60] Digigram. Technology: Latency, 2008. `http://www.ethersound.com/technology/latency.php` [Accessed: 2012.09.10].

[61] B. Cain, S. Deering, I. Kouvelas, B. Fenner, and A. Thyagarajan. Internet Group Management Protocol, Version 3. RFC 3376 (Proposed Standard), October 2002. Updated by RFC 4604.

[62] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474 (Proposed Standard), December 1998. Updated by RFCs 3168, 3260.

[63] The QSC website. `http://www.qscaudio.com/` [Accesses: 2012.0.15].

[64] Q-Sys Network Audio Solution. `http://qsc.com/products/network/QSys/` [Accesses: 2012.0.15].

[65] IETF. User Datagram Protocol. 1980.

[66] Q-SYS Network Audio Solution. Q-Sys Core. `http://www.qscaudio.com/products/network/Qsys/Q-Sys_core.php` [Accessed:2012.12.22].

[67] J. Postel. Internet Protocol. RFC 791 (Standard), September 1981. Updated by RFCs 1349, 2474.

[68] S. Kalarchik. QSC Application Note: Q-Sys Networking Overview. January 2011.

[69] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003. Updated by RFCs 5506, 5761, 6051, 6222.

[70] H. Schulzrinne and S. Casner. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 3551 (Standard), July 2003. Updated by RFC 5761.

[71] J. Postel. User Datagram Protocol. RFC 768 (Standard), August 1980.

[72] K. Kobayashi, A. Ogawa, S. Casner, and C. Bormann. RTP Payload Format for 12-bit DAT Audio and 20- and 24-bit Linear Sampled Audio. RFC 3190 (Proposed Standard), January 2002.

[73] H. Schulzrinne, A. Rao, and R. Lanphier. Real Time Streaming Protocol (RTSP). RFC 2326 (Proposed Standard), April 1998.

[74] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[75] S. Church and S. Pizzi. *Audio Over IP - Building Pro AoIP Systems with Livewire*. Focal Press, 2010. ISBN 978-0-240-81244-1.

[76] Axia Audio. "Did you say 'networked audio?'". `http://axiaaudio.com/livewire` [Accessed; 2012.09.19].

[77] Axia Audio. Axia AES/EBU. `http://axiaaudio.com/system-components#AES` [Accessed: 2012.12.04].

[78] Axia Audio. Axia Router Selector Node. `http://axiaaudio.com/system-components#Audio_Nodes` [Accessed: 2012.12.04].

[79] N. Bouillot, E. Cohen, J.R. Cooperstock, A. Floros, N. Fonseca, R. Foss, M. Goodman, J. Grant, K. Gross, S. Harris, B. Harshbarger, J. Heyraud, L. Jonsson, J. Narus, M. Page, T. Snook, A. Tanaka, J. Trieger, U. Zanghieri. AES White Paper: Best Practices in Network Audio. *Journal of the Audio Engineering Society*, 57(9):729–741, 2009.

[80] Audinate. Dante Q&A, 2012. `http://www.audinate.com/index.php?option=com_content&view=article&id=99` [Accessed: 2012.09.17].

[81] Audinate. Whitepaper - Audio Networks Past, Present and Future [CobraNet and Dante]. `http://www.audinate.com/images/PDF/Audio%20Networks%20Past%20Present%20and%20Future.pdf` [Accessed: 2012.09.17].

[82] Audinate. Dante Virtual Soundcard, 2012. `http://www.audinate.com/index.php?option=com_content&view=article&id=235` [Accessed: 2012.09.17].

[83] Audinate. Whitepaper - Evolving networks to Audio Video Bridging (AVB). 2011.

[84] AES-X192 task group. AES-X192 - High-performance streaming audio-over-IP interoperability. `http://www.x192.org/` [Accessed: 2012.09.19].

[85] W. Fenner. Internet Group Management Protocol, Version 2. RFC 2236 (Proposed Standard), November 1997. Obsoleted by RFC 3376.

[86] H. Weibel. Technology Update on IEEE 1588: The Second Edition of the High Precision Clock Synchronization Protocol. Zurich University of Applied Sciences, 2009.

[87] H. Weibel and S. Heinzmann. Media Clock Synchronization Based on PTP. In *Audio Engineering Society Conference: 44th International Conference: Audio Networking*, November 2011.

[88] M. Wright. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10:193–200, December 2005.

[89] The Center for New Music and Audio Technology (CNMAT). OSC Application Areas. `http://opensoundcontrol.org/osc-application-areas` [Accessed: 2012.10.01].

[90] The Center for New Music and Audio Technology (CNMAT). Sensor/Gesture-Based Electronic Musical Instruments. `http://opensoundcontrol.org/sensor-gesture-based-electronic-musical-instruments` [Accessed: 2012.10.01].

[91] The Center for New Music and Audio Technology (CNMAT). Multiple-User Shared Musical Control. `http://opensoundcontrol.org/multiple-user-shared-musical-control` [Accessed: 2012.10.01].

[92] The Center for New Music and Audio Technology (CNMAT). Web Interfaces. `http://opensoundcontrol.org/web-interfaces` [Accessed: 2012.10.01].

[93] The Center for New Music and Audio Technology (CNMAT). Networked LAN Musical Performance. `http://opensoundcontrol.org/networked-lan-musical-performance` [Accessed: 2012.10.01].

[94] The Center for New Music and Audio Technology (CNMAT). WAN performance and Telepresence. `http://opensoundcontrol.org/wan-performance-and-telepresence` [Accessed: 2012.10.01].

[95] M. Wright, A. Freed, and A. Momeni. Open Sound Control: State of the Art 2003. pages 153–159, 2003. OpenSound Control.

[96] M. Wright. *The Open Sound Control 1.0 Specification*. CNMAT, version 1.0 edition, March 2002. `http://opensoundcontrol.org/spec-1_0` [Accessed: 2012.10.01].

[97] PLASA. The "PLASA Membership" website. `http://www.plasa.org/welcome/` [Accessed: 2012.10.05].

[98] P. Nye. ACN - A Protocol Suite for Entertainment Technology Networking. In *Audio Engineering Society Convention 111*, November 2001.

[99] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[100] American National Standard (ANSI). *Draft BSR E1.17 Architecture for Control Networks - Device Management Protocol*, 2009.

[101] American National Standard (ANSI). *Draft ANSI E1.17-2010 Architecture for Control Networks - Device Description Language (DDL)*, 2010.

[102] American National Standard (ANSI). *Draft BSR E1.17-20xx Architecture for Control Networks - Session Data Transport Protocol (SDT))*, 2009.

[103] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. RFC 2608 (Proposed Standard), June 1999. Updated by RFC 3224.

[104] K. McCloghrie and M. Rose. Management Information Base for Network Management of TCP/IP-based internets:MIB-II. RFC 1213 (Standard), March 1991. Updated by RFCs 2011, 2012, 2013.

[105] S. Turner. *IEC 62379 Common control interface for networked digital audio and video products*. AudioScience Inc, December 2009. IEC62379 Presentation for P1722.1.

[106] Institute of Electrical and Electronics Engineers Standards Association (IEEE-SA). *Guidelines for 64-bit Global Identifier (EUI-64TM) Registration Authority*, November 2012.

[107] R. Foss, R. Gurdan, B. Klinkradt, and N. Chigwamba. An Integrated Connection Management and Control Protocol for Audio Networks. In *Audio Engineering Society Convention 127*, October 2009.

[108] UMAN. Universal Media Access Network, 2012. `http://www.umannet.com/` [Accessed: 2012.11.08].

[109] N. Chigwamba, R. Foss, R. Gurdan, and B. Klinkradt. Parameter Relationships in High-Speed Audio Networks. *Journal of the Audio Engineering Society*, 60(3):132–146, 2012.

[110] R. Foss, R. Gurdan, B. Klinkradt, and N. Chigwamba. The XFN Connection Management and Control Protocol. In *Audio Engineering Society Conference: 44th International Conference: Audio Networking*, November 2011.

[111] OCA Alliance. *OCA Release 1.1*, September 2012.

[112] OCA Alliance. About the OCA Alliance. `http://www.oca-alliance.com/About/index.html` [Accessed: 2012.10.09].

[113] Audio Engineering Society (AES). *AES standard for sound system control - Application protocol for controlling and monitoring audio devices via digital data networks - Part 1: Principles, formats, and basic procedures*, AES24-1-1999 edition, 1999.

[114] OCA Alliance. *OCA Open Control Architecture Release 1.1 - OCF:Framework*, revision 11 edition, September 2012.

[115] OCA Alliance. *OCA Open Control Classes Overview*, revision 05 edition, September 2012.

[116] OCA Alliance. *OCA Open Control Architecture Release 1.1 - OCP.1 OCA Protocol for TCP / IP Networks*, revision 10 edition, September 2012.

[117] 1394 Trade Association (1394TA). *AV/C Digital Interface Command Set General Specification Version 4.2*, September 2004.

[118] A. Butterworth. AVB based AVB Device. Apple Inc, 2009.

[119] 1394 Trade Association (1394TA). *Specifications*. `http://www.1394ta.org/developers/Specifications.html` [Accessed: 2010.10.11].

[120] International Electrotechnical Commission (IEC). *Consumer audio/video equipment - Digital interface - Part 1: General*, 3rd edition edition, 2008.

[121] 1394 Trade Association (1394TA). *TA Document 1999008: AV/C Audio Subunit Specification 1.0*, 2000.

[122] 1394 Trade Association (1394TA). *TA Document 2004007: AV/C Music Subunit Specification 1.1*, 2005.

[123] 1394 Trade Association (1394TA). *TA Document 1999045: AV/C Information Block Types Specification Version 1.0*, 2001.

[124] 1394 Trade Association (1394TA). *TA Document 1999025: AV/C Descriptor Mechanism Specification Version 1.0*, 2001.

[125] R. Foss and J. Fujimori. mLAN - The Current Status and Future Directions. In *Audio Engineering Society Convention 113*, October 2002.

[126] Yamaha Corporation. *mLAN-NC1 PH1 Block Specification*, 2001. Confidential.

[127] Yamaha Corporation. *mLAN-PH2 (YTS440-F) Specification*, 2003. Confidential.

[128] R. Foss and J. Fujimori. A New Connection Management Architecture for the Next Generation of mLAN. In *Audio Engineering Society Convention 114*, March 2003.

[129] R. Foss, J. Fujimori, and H. Okai-Tettey. An Open Design and Implementation for the Enabler Component of the Plural Node Architecture of Professional Audio Devices. In *Audio Engineering Society Convention 119*, October 2005.

[130] R. Foss, J. Fujimori, K. Kounosu, and R. Laubscher. An Open Generic Transporter Specification for the Plural Node Architecture of Professional Audio Devices. In *Audio Engineering Society Convention 118*, May 2005.

[131] The Center for New Music and Audio Technology (CNMAT). OSC Implementations. `http://opensoundcontrol.org/implementations` [Accessed: 2012.10.01].

[132] Audio Engineering Society (AES). Aes-x170 initiation, May 2010. `http://www.aes.org/standards/meetings/init-projects/aes-x170-init.cfm` [Accessed: 2012.11.09].

[133] U. Franke. WOscLib: The Weiss OpenSound Control Library, 2005. `http://wosclib.sourceforge.net/` [Accessed: 2012.10.19].

[134] A. Freed and A. Schmeder. Features and Future of Open Sound Control version 1.1 for NIME. In *NIME*, June 2009.

[135] P.V. Mockapetris. Domain names - implementation and specification. RFC 1035 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 1995, 1996, 2065, 2136, 2181, 2137, 2308, 2535, 2845, 3425, 3658, 4033, 4034, 4035, 4343, 5936, 5966, 6604.

[136] S. Cheshire and M. Krochmal. DNS-Based Service Discovery. IETF, 2011. `http://files.dns-sd.org/draft-cheshire-dnsext-nbp.txt` [Accessed: 2012.09.20].

[137] A. Gulbrandsen, P. Vixie, and L. Esibov. A DNS RR for specifying the location of services (DNS SRV). RFC 2782 (Proposed Standard), February 2000. Updated by RFC 6335.

[138] Internet Assigned Numbers Authority (IANA). Service Name and Transport Protocol Port Number Registry. `http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xml` [Accessed: 2012.10.17].

[139] R. Foss, R. Gurdan, B. Klinkradt, and N. Chigwamba. The AES-3CIM Connection Management and Control Protocol. 2012.

[140] P1722.1 Working Group. Device Discovery, Enumeration, Connection Management & Control Protocol for AVTP devices. `http://grouper.ieee.org/groups/1722/1/AVB-DECC/IEEE-1722.1_Working_Group.html` [Accessed:2012.10.21].

[141] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320, 4916, 5393, 5621, 5626, 5630, 5922, 5954, 6026, 6141, 6665.

[142] G. Camarillo. *SIP Demystified*. McGraw-Hill Professional, 2001.

[143] M. Mealling and R. Denenberg. Report from the Joint W3C/IETF URI Planning Interest Group: Uniform Resource Identifiers (URIs), URLs, and Uniform Resource Names (URNs): Clarifications and Recommendations. RFC 3305 (Informational), August 2002.

[144] C. Holmberg. Session Initiation Protocol (SIP) Response Code for Indication of Terminated Dialog. RFC 6228 (Proposed Standard), May 2011.

[145] A. Niemi and D. Willis. An Extension to Session Initiation Protocol (SIP) Events for Conditional Event Notification. RFC 5839 (Proposed Standard), May 2010.

[146] G. Camarillo, W. Marshall, and J. Rosenberg. Integration of Resource Management and Session Initiation Protocol (SIP). RFC 3312 (Proposed Standard), October 2002. Updated by RFCs 4032, 5027.

[147] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093, 6528.

[148] O.P. Igumbor and R. Foss. A Proxy Approach for Interoperability and Common Control of Networked Digital Audio Devices. In *Audio Engineering Society Convention 128*, May 2010.

[149] The AVAHI Team. AVAHI, 2005. `http://www.avahi.org/` [Accessed: 2012.11.09].

[150] The AVAHI Team. avahi 0.6.31, February 2012. `http://avahi.org/download/doxygen/` [Accessed: 2012.11.09].

[151] ALSA Project. Advanced Linux Sound Architecture (ALSA) project homepage. `http://www.alsa-project.org/main/index.php/Main_Page` [Accessed: 2012.10.31].

[152] D. Steinberg and S. Cheshire. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., first edition, December 2005.

[153] Universal Media Access Networks (UMAN). UNOS Creator User Manual, February 2010. `http://www.unosnet.com/unosnet/index.php/unos-core.html` [Accessed: 2011.05.29].

[154] P.T. Ward and S.J. Mellor. *Structured Development for Real-Time Systems, Volume 1: Introduction and Tools*. Prentice Hall, June 1986.

[155] P. Foulkes. kmsrp for end station. `http://code.google.com/p/kmsrp/` [Accessed: 2012.10.31].

[156] Universal Media Access Networks (UMAN). UNOS Vision. `http://www.unosnet.com/unosnet/index.php/unos-vision.html` [Accessed: 2012.04.26].

[157] XMOS. Low-Cost AVB Audio Kit Built on XCore Processor, 2011. `http://www.xmos.com/resources/xkits?category=Low-cost+AVB+Audio+Endpoint+Kit` [Accessed: 2012.11.20].

[158] LabX Technologies, LLC. Titanium 411 Ruggedized AVB Ethernet Bridge. `http://www.labxtechnologies.com/connectivity/titanium-411-ruggedized-avb-ethernet-bridge/` [Accessed:2012.11.20].

[159] S. Card and T. Moran. User technology - from pointing to pondering. In *Proceedings of the ACM Conference on The history of personal workstations*, HPW '86, pages 183–198, New York, NY, USA, 1986. ACM.

[160] D. Wessel and M. Wright. Problems and Prospects for Intimate Musical Control of Computers. *Computer Music Journal*, 26(3):11–22, September 2002.

[161] R.F. Moore. The Dysfunctions of MIDI. *Computer Music Journal*, 12(1):19–28, March 1988.

[162] Wireshark Foundation. Wireshark the world's foremost network protocol analyzer. `http://www.wireshark.org/` [Accessed: 2012.02.12].

[163] WinPcap. WinPcap - The industry-standard windows packet capture library. http://www.winpcap.org/ [Accessed: 2012.09.21].

[164] U. Lamping, R. Sharpe, and E. Warnicke. *Wireshark User's Guide*, 2004.

[165] QLOGIC. Introduction to Ethernet Latency - An Explanation of Latency and Latency Measurement. August 2011.

[166] M. Muus. The Story of the PING Program. `http://ftp.arl.mil/mike/ping.html` [Accessed: 2012.11.21].

[167] XMOS. *XS1-L02A-QF124 Datasheet*. XMOS, 10 2012. Document Number: X118.

[168] John L. Hennessy and David A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.